©PTT/Dr. Neher Laboratories, 1987

COMPARISON OF SDL AND LSD

W M Beynon† and M T Norris*

†Department of Computer Science
University of Warwick
Coventry
ENGLAND

*British Telecom Research Laboratories
Systems and Software Engineering Division
Martlesham Heath
Ipswich
ENGLAND

Well-documented standard languages for system specification such as CCITT SDL provide essential support for the modern software designer. This in itself is not enough, however. To be fully effective, such languages require a sound mathematical semantics which will permit formal reasoning about specifications, and enable more comprehensive support tools and techniques to be developed. This paper proposes a language, LSD, within which many of the most important concepts in SDL can be interpreted, and suggests methods which may be useful in the design and analysis of SDL specifications.

1. INTRODUCTION

As information technology systems become increasingly complex, more powerful techniques are needed to specify, design and implement them. The problems are particularly acute in the telecommunications area, where large systems of concurrent processes are involved. In response to this challenging problem, formal specification techniques based on a mathematically rigorous semantics have been developed [1]. For such techniques to be effectively used in practice, it is arguably important to balance the need for rigour against a complementary requirement for ease of use. It is in this respect that many of the evolving formal description methods fall down [2,3].

The specification and description language (SDL) has been developed by the CCITT over the last 16 years very much with the user in mind. The result is a rich synthesis of constructs and concepts, most of which admit a common informal interpretation, and which together address a wide spectrum of significant aspects of system design. Indeed, one of the main reasons why SDL is now so widely used for the specification of interactive systems is its acceptance as an intuitive vehicle for expressing system designs.

Despite the fact that SDL has a well-defined syntax and uses many concepts which are easily informally understood, it does not inherently possess a secure semantic foundation, and incorporates a range of features which it is hard to describe formally in a consistent manner. Remedying this deficiency would have many important benefits: Specifications in the language would be open to formal analysis, teaching SDL would be simpler and less reliant on shared

understanding, and requirements for support tools would be easier to generate and validate.

This problem has been recognised for some time, and several attempts have been made to alleviate its effects. It is possible, for instance, to "borrow" a partial semantics for SDL from an appropriate source such as Petri nets [4]. This proves very valuable in the analysis of specifications but, in common with other approaches, only captures the semantics of SDL at the lower levels of abstraction. Ideally, the development of a semantic model which more fully reflects the scope and nature of SDL is required. The work described in this paper has been carried out with this objective in mind.

Our approach is based upon the development of an alternative notation for concurrent systems LSD [7], closely resembling SDL in many important respects, which can be used to give a proper semantic framework for the study of SDL processes. Unlike SDL in its most extended forms, the LSD notation has a small set of primitive constructs and concepts. It is essentially based upon a shared variable paradigm for communication, but has sufficient expressive power to enable the whole variety of communication mechanisms in SDL to be simulated. In common with SDL, LSD incorporates mechanisms for process creation and termination, and offers the prospect of semantic analysis of SDL specifications at a higher-level of abstraction than hitherto achieved. As yet, the possibility of models for even higher-level SDL abstractions such as blocks and channels is unexplored, but will be the focus for further research. The idea of ascribing a semantic model to an appropriate subset of SDL according to particular needs in this fashion should be familiar to existing users. After all, expressing SDL in a variety of syntactic forms has long been accepted!

In the next two sections, we develop the central notions of our model, and show how it can be applied. An example is used for further illustration of both the application and the usefulness of this approach. Following this, we briefly take stock of where we are at the moment, and what may develop in the future.

2. BACKGROUND

In this section, we give a brief introduction to the key ideas underlying our semantic model. For more details, see [5,6,7].

Spreadsheets have proved to be a very effective basis for human-computer interaction. In interaction using a spreadsheet, the state of the dialogue is represented by a combination of scalar variables with explicit values and variables which are implicitly defined by formulae. A *definitive notation* [5,6] is a natural generalisation of a spreadsheet in which the variables may be of many different sorts. In effect, a dialogue state is represented partly by procedural abstractions (variables whose values can be updated) and partly by functional abstractions (variables whose values are functionally defined). Such notations offer 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.

Consider a system of sequential processes acting concurrently, 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. As in an SDL system, it will be assumed that processes within the system are in general created, live for some period of time, then 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 others having known definitions in terms of other variables and constants. In principle, the values represented by these variables may be of

a number of different sorts (eg *boolean* , *integer, 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, and in this context most variables can be assumed to have simple scalar, typically integer and boolean, values. A variable whose value is implicitly defined by a formula in terms of other variables known to P will be called a *derivate* of P. 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 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. As a simple illustration, when you are driving a car, the validity of your insurance can be viewed as a derivate - a function of whether you have paid your insurance premium, and how much time has elapsed, the registration number of the car as a constant, the state of the traffic lights as an oracle, and the condition of your headlights as a state. There is of course some degree of arbitrariness in this classification: over an extended period of time, as in a school record, "age" might be viewed as a derivate: at any particular time, as in registering for a competition, as a constant: in a play, as a state variable under the control of an actor: to a barman, as an oracle. (See §4 for further illustration.)

In the context of the above model, the natural way to describe communication is via shared variables. A variable which is a state or derivate for one process and an oracle for another can then be viewed as communicating a value. A full discussion of system behaviour within this model is beyond the scope of this short paper, but some key points should be noted. In the semantic model proposed, the active process instances are participants in a dialogue, and the state of the system as viewed by each process, and by an observing "system analyst", is represented by a dialogue state (see Figure 1). 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 coincide, so that, in effect, communication is instantaneous, and all the derivates are instantaneously updated. This will not always be a realistic supposition, but in describing systems informally at a reasonably high level of abstraction, the model of communication is often artificially simplified, and our notion of "idealised behaviour" reflects this. In practice, the oracles of a process cannot be assumed 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 licensing system should operate. To study the behaviour under more realistic assumptions, it will in general be necessary to refine the model of a system to incorporate explicit processes to handle signals and data transfer. Some integrity constraints will also have to be imposed upon a specification; for instance, it will be important to ensure that there is no possibility of two independent processes simultaneously manipulating a common state variable in a contradictory fashion.

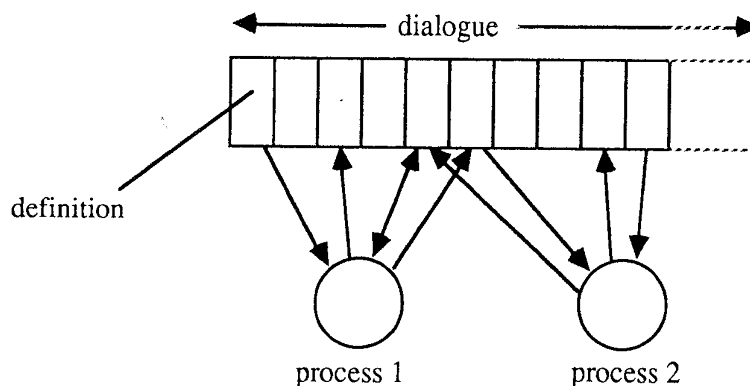

FIGURE 1 System View

The formal notation to describe processes, called "LSD" because of its strong affinities with SDL, reflects the ideas above. To define a process type, the states, oracles and derivates are given, together with a protocol describing the exact nature of the control exercised over state. The binding of names to variables is determined from the context when a process instance is created, and names of variables bound to process instances of a particular type will be preceded by a hash symbol in the process type definition. 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. Termination of a process is controlled as if by adjoining a clause "LIVE and ..." to each guard in the protocol, where LIVE is a special private boolean state or derivate. Informally, the behaviour of a process instance is described by a combination of declarative elements - the functional definitions of its derivates - and procedural elements - the actions within its protocol (see Figure 2).

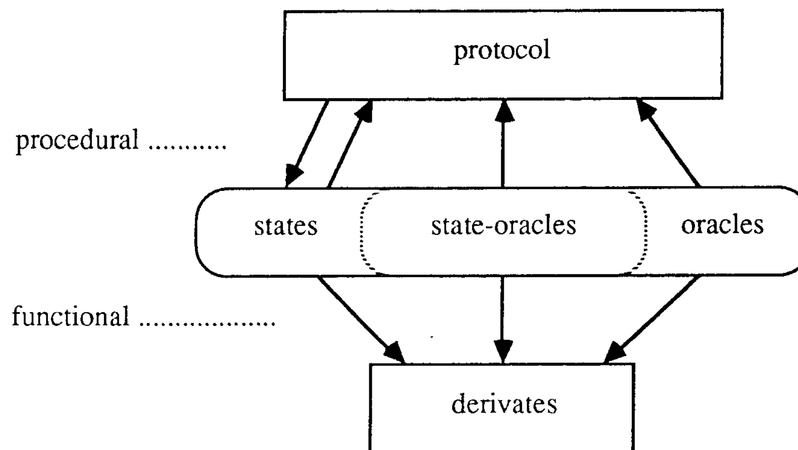

FIGURE 2 Process View

3. LSD AS A MODEL FOR SDL

LSD can be viewed in two ways: as a notation for concurrent systems in its own right, or as a medium which can be used for the interpretation of SDL programs. Our interest here is primarily in the latter. In this section, we outline an LSD semantics for a rich spectrum of techniques associated with SDL processes. An SDL afficianado might wish to view LSD, with its more limited range of constructs for describing data communication and state representation, as a form of intermediate code.

Strong basic similarities between SDL and LSD are readily apparent. Both include a process creation mechanism, and incorporate variables bound to process instances. Superficially, LSD has

a more limited range of possible actions, but we can simulate all the required features of SDL within LSD. It is helpful at this point to summarise the basic SDL actions (cf Z101 [8]), and indicate how each action has a counterpart in LSD.

Assignment - The simple SDL action of assigning to a variable can clearly be simulated in an LSD protocol. Indeed, an LSD process can conditionally assign values to any its state variables, including variables bound to other process instances.

State change - An SDL process is essentially an extended finite state machine. Like an SDL process, an LSD process can either be busy in the execution of an action ("carrying out a transition"), or in a state of readiness pending commitment to an action ("in a state"). The state of an LSD process is effectively determined by the current values of its state and oracle variables, and the set of permissible transitions by the set of guards which is true in that state. In SDL, the current state of a process can change only in response to a signal from another process. LSD is similar, in that a change of state occurs only when the value of an oracle is changed, and the set of possible transitions is affected only when this affects the truth value of a guard. It is possible to model state changes directly, by using a private variable to record state. In practice, as illustrated in §4, an LSD specification would normally be developed in a different fashion - there is a role for derivates, and for state information which is less cryptic than that in SDL.

Signal input/output - Primary communication in LSD is modelled using shared variables. In effect, a process has received a signal when the value of one of its oracles has been changed. As explained above, it is appropriate to use shared variables to model communication in this fashion only at certain levels of abstraction, and there is a role for explicit signalling processes. LSD specifications of a variety of signalling mechanisms are quite easy to derive, and these can be adapted to the appropriate SDL conventions. For instance, it is a straightforward exercise to describe an LSD protocol whereby a signal is consumed on receipt if it gives rise to a transition, and otherwise queued or discarded according to prescribed rules. SDL signals in which a data value is transferred can similarly be described implicitly using a shared variable, or embedded in a data transfer protocol. It is possible to go yet further, and model a physical transfer of a form, such as the circulation of an attendance record at a meeting, by creating a process to represent the form, and a oracle within the form process to designate its current holder.

Procedure call - Semantically both SDL and LSD processes implicitly comprise a procedure and a processor. In general, invocation of a process is understood to entail "initiating a new processor". A procedure call within one process differs from a process call only insofar as no new processor is involved. The semantics of procedures can thus be derived.

Decision step - An SDL action which involves branching on a boolean condition can clearly be described within an LSD protocol. Indeed, an LSD protocol provides for non-deterministic behaviour when two or more guards are true.

Process creation - Process invocation in LSD is essentially the same as that in SDL, and in general leads to process termination via a guarded command or via a terminating action.

Another basic feature of SDL is the notion of absolute time. This can be described in LSD by means of a state variable for the environment which is an oracle for all processes. It is also easy to model timers within LSD (cf the example below).

We have sketched the way in which the basic subset of SDL can be described in LSD. SDL offers other features at higher levels of abstraction, and it remains to consider how certain of these may also be related.

Enabling conditions in SDL bear a strong resemblance to the guarded commands in LSD. A significant, and perhaps crucial, difference is the way in which an LSD specification permits a process instance to have direct control over variables bound to another. This single feature

contributes significantly to the relatively simple form of the LSD description given below. Data abstraction [8] is another feature of SDL that should be reasonably easy to accommodate within the LSD framework. The chief problems here probably relate to the parallel access to components of complex data types, and resemble well-recognised difficulties in generalising spreadsheets to deal with complex data types [5,6].

The blocks and channels introduced in partitioning of systems represent the single most important aspect of SDL yet to be interpreted in the LSD model. The problems here reflect the exceedingly broad range of possible abstract views of a system, and are closely connected with the development of appropriate ways for composing LSD processes, and hiding internal features. This area is the focus of our current research.

4. AN EXTENDED EXAMPLE

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 simplified telephone system, in which the role of a user, and 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, where X,Y,M and N are to be interpreted as integer indices for distinguishing the process instances.) 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 which the user is associated with. (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. A significant feature of the final description is the use of an *exchange*() process to resolve the problems which concurrent access to a single phone can present. In effect, with the appropriate semantic model, the *exchange*() process expresses the fact that at any one time at most one *call*(*, *N*) instance can be responsible for phone N ringing. (See [7] for a fuller discussion.)

```
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) is ringing[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
              and not dialled_number is valid ;
           (bool) #LIVE  =  dialling[M] and not onhook[M]}

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 = LIVE
protocol   not engaged[M] and calling[M] and not onhook[N]
           -> connected[M] := true; < connect M and N  > }
```

```
process     exchange() {
oracle      (bool) #ring[*,*], onhook[*]
derivate    (bool) #isringing[*] = ring [?,*] and onhook[*],
            (time) #T_RTD = < max dialling delay > ,
            (time) #T_call = < max time for calling  > }
```

5. STATUS AND PROSPECTS

So far, we have presented a soundly based notation which supports some of the key notions of SDL, and SDL processes in particular. The characteristic notions of oracles, states and derivates within an LSD process suggest a distinctive way of thinking about systems of concurrent processes which is helpful in developing and documenting a specification. To justify our approach formally, appropriate proof techniques for reasoning about the behaviour of SDL and LSD systems have now to be developed. Our model already offers good prospects for simulation and animation of system behaviour - activities which have proved difficult to carry out using the traditional approach.

One of the primary aims behind this work is the provision of support tools for system designers. Our objective is to allow the system designer to specify a system at a high level of abstraction, using concepts which have close counterparts in practical experience, but in such a way as to allow further refinement and formal verification of this intuitive specification. The LSD notation was conceived as a convenient medium for representing processes, as a first step towards describing the complex relationships involved in composing and viewing systems. Further research will investigate methods of interpreting yet higher level SDL concepts, such as blocks, channels and subprocesses, centrally concerned with the fundamental problems of partitioning systems, and viewing systems of processes at different levels of abstraction. In this way, we hope to identify a suitable family of algebraic operations on systems which can be used as the basis of an SDL design tool resembling a generalised spreadsheet. In this context, it should be noted that LSD already incorporates declarative features which can to some extent be used to describe low-level structure abstractly.

The teaching of SDL is another important area which should benefit from this work. Experience has shown that newcomers to the language readily acquire superficial familiarity with the central concepts of SDL, but feel the need for a formal and consistent interpretation to support their intuition. We can now begin to explain concepts and interrelations precisely rather than waving hands and relying on an informal shared understanding.

6. CONCLUSIONS

The work presented in this paper gives a foundation for the development of a formal method for using SDL. This approach offers a more objective basis for evaluating specifications, and pays dividends in those areas where complex concepts have to be used - teaching and tool building.

ACKNOWLEDGEMENTS

The authors of this paper would like to thank the director of British Telecom research Laboratories for permission to publish this paper. Thanks are also due to the many friends and colleagues in the System and Software Engineering Division who contributed to this work.

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

REFERENCES

[1] C.A.R.Hoare 'Programming - sorcery or science?'
IEEE Software, April 1984

[2] Economist Informatics Report 'Formal methods in Software Engineering'
Alvey News, June 1985 p12

[3] M T Norris, 'The application of formal methods in system design',
British Telecom Technol J Vol 3 No 4 October 1985

[4] T. Agerwala, 'Putting Petri nets to work'
Computer 12 , no 12 pp 85-94, 1979

[5] W M Beynon 'Definitive notations for interaction'
Proc hci'85 "People and Computers: designing the interface", CUP 1985

[6] W M Beynon 'A programming paradigm based on definitions'
University of Warwick Computer Science Research Report in preparation

[7] W M Beynon 'The LSD notation for communicating systems'
University of Warwick Computer Science Research Report #87, 1986

[8] Functional Specification and Description Language (SDL),
Recommendations Z100-Z104, CCITT 8th Plenary Assembly 1984