THE FORMAL DESCRIPTIVE TECHNIQUE SDL IN SOFTWARE ENGINEERING

J.C. van der Walt*

**BSW Data South, Tokai, Western Cape, South Africa*

Abstract. Formal descriptive techniques provide a means to describe and manage the behaviour and structure of complex systems. In the last few years powerful commercial tool support for the formal descriptive technique SDL (Specification and Description Language) became available. This allows for the practical application of SDL in the software engineering process. A brief overview of the SDL language is given. The basic SDL concepts available to describe the structure and behaviour of systems are also explained. The typical capabilities of tools are listed and industrial experience with SDL is described. A comprehensive list of references identifies the available information on SDL.

Key Words. Complex systems; interactive systems; real time systems; formal descriptive techniques; SDL; software engineering; system specification; object orientation

1. INTRODUCTION

Modern software systems can be very complex. Typical examples are the software that control real time systems such as public and private telephone exchanges, urban traffic controllers, pay phones, assembly line controllers, safety systems on board aircraft, and automatic teller machine systems. Apart from functional requirements, these systems must normally satisfy severe safety, reliability and quality of service requirements which increase the complexity of the control software. The required behaviour of complex systems is difficult to specify, model and test. Time to market further requires that these systems be developed by large development teams which require accurate means of communicating concepts and requirements.

The very first step in engineering the software of a new system is the identification of all the requirements that the software must satisfy. This is typically done in a natural language which often results in ambiguous requirements[1,2]. It is also very difficult to determine the completeness and consistency of these requirements. From these requirements, expressed in a natural language, different people develop different conceptual models of the problem. This inevitably leads to misunderstanding and miscommunication, resulting in an inefficient software engineering process.

The use of a formal description technique (FDT) to capture the required behaviour and structure of complex systems can assist greatly in solving these problems. A formal language with formal semantics forces the software engineer to unambiguously define each and every concept used to describe the required behaviour of the complex parts of a system. This results in clear, precise and concise specifications [3].

Since specifications now have a formal basis, requirements can be analysed for completeness, correctness and consistency by executing the specification in a simulation environment. Completeness is verified by testing that the specification describes the required behaviour for all the possible combinations of external inputs to the system. Correctness and consistency is determined by exploring the state space for deadlock situations and for parts of the specification which are never executed.

Conceptual problems which would normally only show up late in the software engineering process during detailed design and implementation, can now be identified early when the cost of modifications is still small. FDTs focus the software engineering effort on the early phases of system development to ensure that the final implementation will be based on sound requirements.

A FDT has been likened to a mirror, not only showing the structural grace and functional consistency of a system design, but also its structural poverty and functional deficiency [4]. Although it may seem tedious at first, FDTs force the designer to consider and control all aspects of the complex parts of a system.

Using a FDT also provides a basis for determining conformance of implementations to specifications since the required behaviour is now well defined.

Since a FDT has a formal syntax and semantics, it allows for computer based tool support to create, maintain, analyse, simulate and implement specifications. Commercial tools have matured in the last few years and now enable the software engineer to practically apply FDTs in the development of complex software systems.

Three FDTs have been standardised by international bodies, namely ESTELLE (Extended Finite State Machine Language), LOTOS (Language of Temporal Ordering Specifications) and SDL (Specification and Description Language) [4]. ESTELLE and LOTOS were developed within the International Organisation for Standardisation (ISO) while SDL was developed by the International Telegraph and Telephone Consultative Committee (CCITT, now called the International Telecommunication Union or ITU). ESTELLE [5] and SDL are based on communicating finite state machines, while LOTOS is based on process algebraic methods [6,7]. After an extensive literature and Internet search it was found that SDL currently has the best available commercial tool support, making it the candidate FDT in a commercial software engineering environment. This has prompted BSW Data South to start using SDL in the software engineering of real time systems.

The purpose of this paper is to provide a very brief introduction to SDL and the capabilities of the tools that are available. Industrial experience with SDL at BSW Data South as well as at other companies reported in the literature, are summarised. A comprehensive list of references to text books and available literature will enable the interested reader to gain further knowledge of SDL. An SDL Newsletter is published approximately once a year under the auspices of the ITU. General information on SDL and SDL tools are also available on the Internet [8].

2. SDL

SDL is recommended by the ITU for the unambiguous specification and description of the behaviour of systems. Although SDL originated within the telecommunications field, it has a much broader application area, covering all concurrent, reactive, distributed systems. It is particularly suited to real time systems. It allows for a formal definition of the behaviour and structure of complex systems in a clear and concise way that can be understood, communicated and analysed independently from the implementation.

In this respect it differs from the visual language proposed by Mostert [9] which also contains constructs for physical implementation issues such as direct memory access (DMA) and interrupts. SDL allows the user to focus on required behaviour and structure and to postpone implementation decisions until requirements are clearly defined and understood.

The development of SDL started in 1968 when the ITU recognised the need for a new language to specify and describe the functional features of systems [10]. The language started from the well known model of finite state machines used by telecommunication engineers to design electromechanical exchanges, taking into account the evolving technology of computer science. The first version of the language was issued in 1976, followed by new standardised versions in 1980, 1984, 1988, 1992 and 1996 [11]. Its graphical and textual syntax and semantics are formally defined. Since the 1992 version, called SDL-92, the language also contains object oriented extensions. The overview of the language below is based on SDL-88 however, as described in [12] and [13], since the object oriented extensions do not add to the basic underlying concepts of the language. The object oriented features are summarised in Section 5.

3. OVERVIEW OF THE SDL LANGUAGE

SDL provides several classes of constructs to model the properties of complex systems. These constructs represent

- the dynamic behaviour of the different parts of the system
- the system structure identifying the co-operating parts
- the communication within the system and between the system and its environment
- the internal information affected and affecting the behaviour of the system

3.1. System Behaviour

The behaviour of a *system* is modelled by the combined behaviour of concurrent *processes* which communicate asynchronously through discrete messages called *signals*. More than one instance of a process may exist in a system. In what follows, the word process may also mean process instance, unless stated otherwise. Each process is an autonomous extended finite state machine (i.e. a finite state machine that can use and manipulate data stored in variables local to the machine). The *environment* of the system also communicates with

processes within the system using signals. The behaviour of a process is deterministic; it reacts to external stimuli according to its state machine description. Constructs to describe non deterministic behaviour have been added in SDL-92 [17].

A process has a memory of its own for the storage of variables. A process cannot write to the variables of another process. All variables are therefor local to the process to which they belong and can only be modified indirectly by another process using signals.

Each process has a unique address. A signal always contains the address of the sending process in addition to possible data values that one process may want to send to another process. The receiving process therefor always knows the address of the sending process.

Each process has an infinite first-in-first-out input queue, where incoming signals are queued. A process is either in a *state* waiting for an input signal, or performs a *transition* between two states. A transition is triggered by the first signal in the input queue. When a signal has initiated a transition, it is removed from the input queue (and is said to be consumed). In a transition, variables can be manipulated, decisions can be made, new processes can be created and signals can be sent (to other processes or to the process itself).

No assumption is made about the time that a signal will remain in a queue or about the duration of a state transition. The only assumption is that a signal can only be consumed after it has been output. The semantics of time in SDL therefor do not provide any facilities for performance modelling. Extensions to SDL for performance modelling are receiving research attention however. An extension of the SDL syntax to attach performance sub-models to SDL-specifications is proposed in [14]. Bütow et al. on the other hand describes the introduction of time semantics for performance modelling without affecting the syntax of SDL [15].

3.2. System Structure

Structuring of a system provides a means to deal with complexity. The main structuring element is the *block*. A system can be divided into blocks and a block can again be partitioned into blocks, resulting in a block tree structure with the system as the root block. Leaf blocks are not partitioned and contain only processes. Within a block signals are conveyed between processes on *signal routes*. Signal routes also connect the processes in a block to the boundary of the block. Between blocks, as well as between blocks and the environment of the system, signals are conveyed on *channels*.

3.3. Internal Information

In SDL the abstract data type approach has been chosen to represent internal information in the system. All data types (predefined as well as user defined) are defined in an implementation independent way in terms of their properties only. The definition of an abstract data type has three components:

- a set of values
- a set of associated operations
- a set of definitions defining these operations

3.4. Representation Forms

SDL has both a graphical and a textual representation form. The graphical form, called SDL/GR uses a graphical syntax to give an overview, combined with a textual syntax for some concepts where graphical symbols are not suitable (e.g. abstract data types). The textual phrase representation SDL/PR uses only a textual syntax which overlaps the textual syntax of SDL/GR.

4. BASIC SYSTEM STRUCTURE

4.1. System

The top level of the system structure is called the *system diagram*. An example of a system diagram for a simplified traffic signal control system is shown in Fig. 1. It shows a system that consists of vehicle detectors, a manual control panel, an intersection stager and signal lamps. The intersection stager controls the signal lamps and can react to inputs from vehicle detectors and the manual control panel.

The system diagram usually contains:

- a frame, representing the boundary between the system and its environment (which is not described). Similar frame symbols are used in all SDL diagrams to delimit the entity being described from its environment. Channels may also have an associated but unspecified delay - an implicit first-in-first-out queue in each direction which delays a signal for an arbitrary time
- a heading containing the system name
- descriptions of the blocks of the system
- descriptions of the channels connecting the blocks of the system and between the blocks and the environment. A channel has a name, a list of the signals that it can convey in each direction,

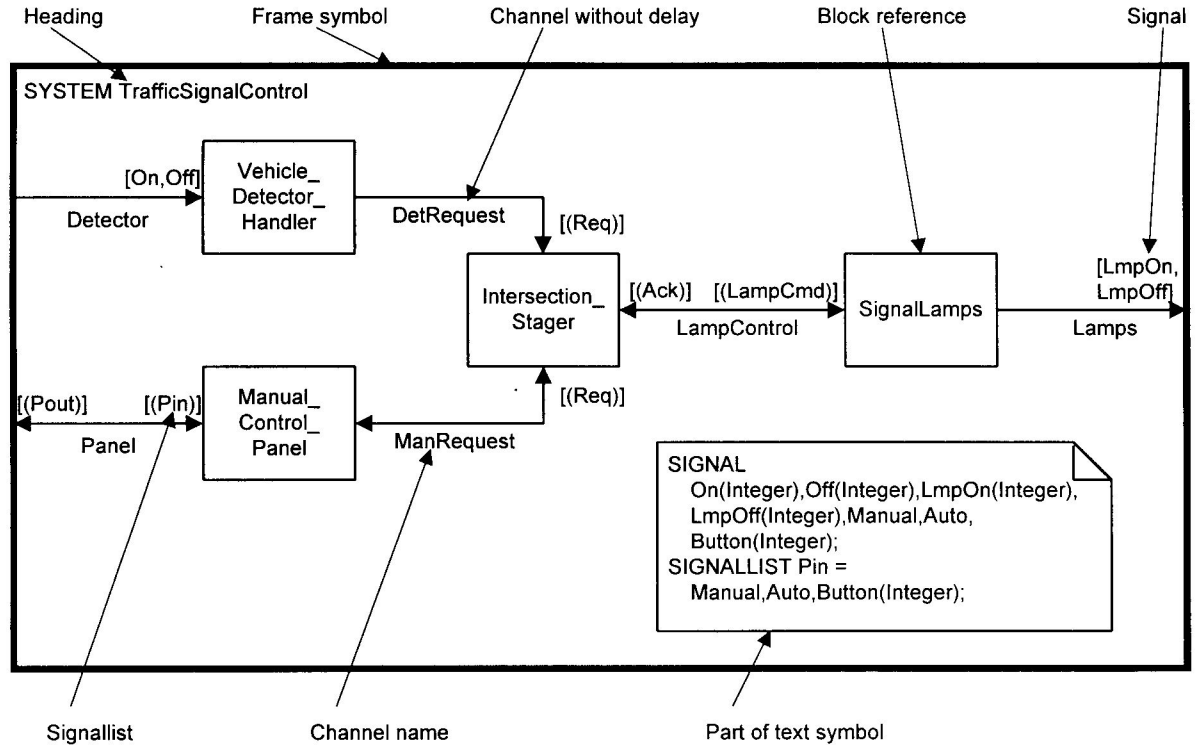

Fig. 1. Example of a system diagram.

and the identification of the endpoints of the channel

- descriptions of the signals visible on the channels at this level. A signal has a name, and the types of values conveyed by the signal
- descriptions of the user defined data types visible in the whole system and its environment

Textual descriptions, such as descriptions of signals and abstract data types, are placed in a *text symbol* in SDL/GR.

SDL provides a referencing mechanism enabling the separation of the use of a structuring construct, e.g. a block or a process, from its actual description. In Fig. 1, the actual description of block *SignalLamps* is referenced by a rectangle containing the name of the block. This mechanism is used by computer aided tools to allow the user to work on a specific part of the system description, referencing parts in the next lower level and being referenced by the next higher level in the system hierarchy.

4.2. Block

A block is described by a *block diagram*. An example of the leaf block *SignalLamps* of the *TrafficSignalControl* system is shown in Fig. 2. It consists of a process *ConflictChecker* and a process *Lamp*. An instance of the *Lamp* process controls the colours of a set of signal lights working in unison. Commands to change the colour (i.e. state) of a *Lamp* process instance are passed through the *ConflictChecker* process to ensure that conflicting signal lights are not green simultaneously. These commands are acknowledged by the *Lamp* process.

A block diagram usually contains:

- a heading containing the block name
- descriptions of the *blocks* or *processes* contained in the block
- descriptions of the *channels* between blocks or *signal routes* between processes and to the environment of the block
- channel to channel connections or channel to route connections specifying the connection between channels external to the block and internal channels or signal routes to the environment of the block

The block diagram may also contain signal and data type descriptions similar to the system diagram. These descriptions are then only visible within the block.

4.3. Process

A process description defines a process type. During interpretation of the system description more than one instance of a process may exist. These instances can either be created automatically at system start-up or dynamically by another parent process during system interpretation. Processes can also terminate

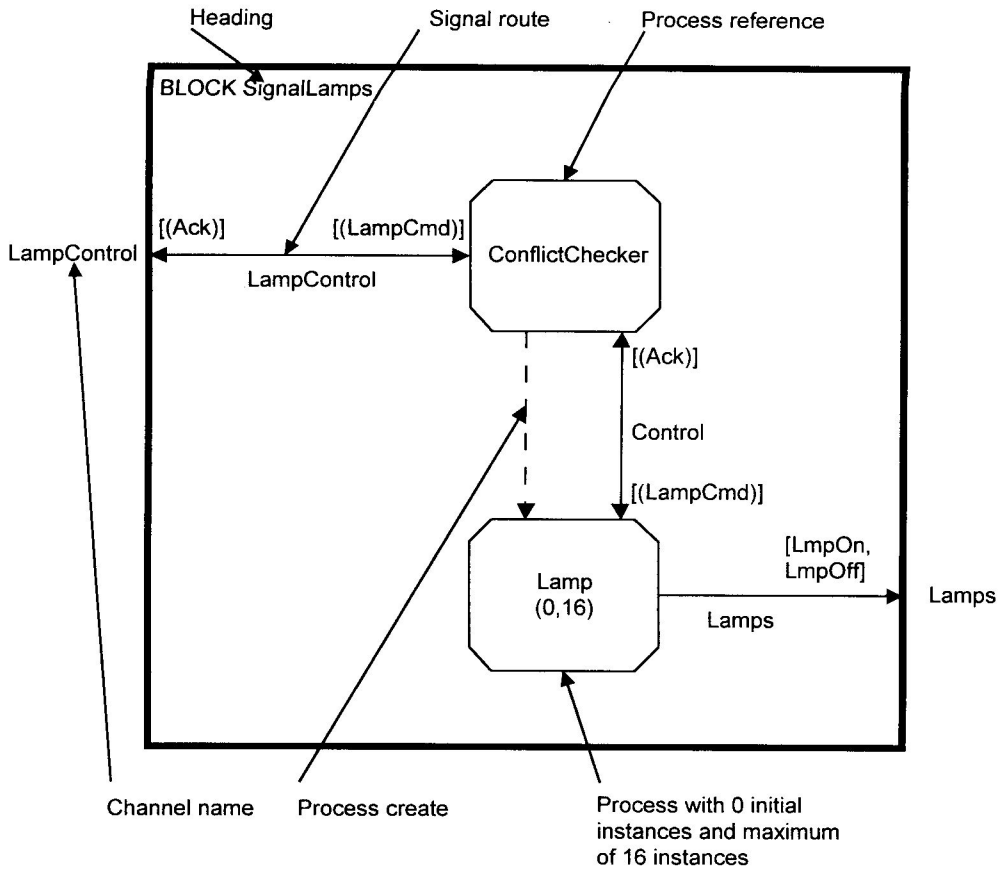

Fig. 2. Example of a block diagram.

themselves, i.e. a transition can result in process termination instead of entering a state.

A process description is called a *process diagram*. It contains:

- the process name
- formal parameters, used to pass data from a parent process during dynamic process instantiation
- descriptions of process variables and timers
- descriptions of procedures
- the process graph, which describes the finite state machine behaviour of the process

5. PROCESS BEHAVIOUR

The behaviour of a process is described in a process diagram. An example of a process diagram for the *Lamp* process of the *TrafficSignalControl* system is shown in Fig. 3. There are five basic constructs for the description of the behaviour of a process:

start - the starting point of process behaviour at process creation

state - to wait in a specified state for input

input - to start a transition on consumption of a specified input signal. The input construct contains the name of a signal and a list of variables which will assume the values of the actual parameters conveyed by the signal

output - to send a specified signal to a process. It contains the name of a signal and a list of actual parameters. The addressing of the target process can either be explicit by specifying the address of the receiving process instance, or implicit by specifying the signal route that the signal must follow or by letting the interpreter determine the target address. In the latter case, if more than one process instance have the signal defined as a valid input, the selection of the receiving process is arbitrary

nextstate - indicating the end of a transition, the process is again in a state waiting for input

These are augmented by the following constructs:

timer - to describe time dependent behaviour. Timers can be set and reset during transitions, and when a timer expires, a timer signal is generated and queued in the input queue of the process

decision - to describe alternative behaviour based on the values of process variables

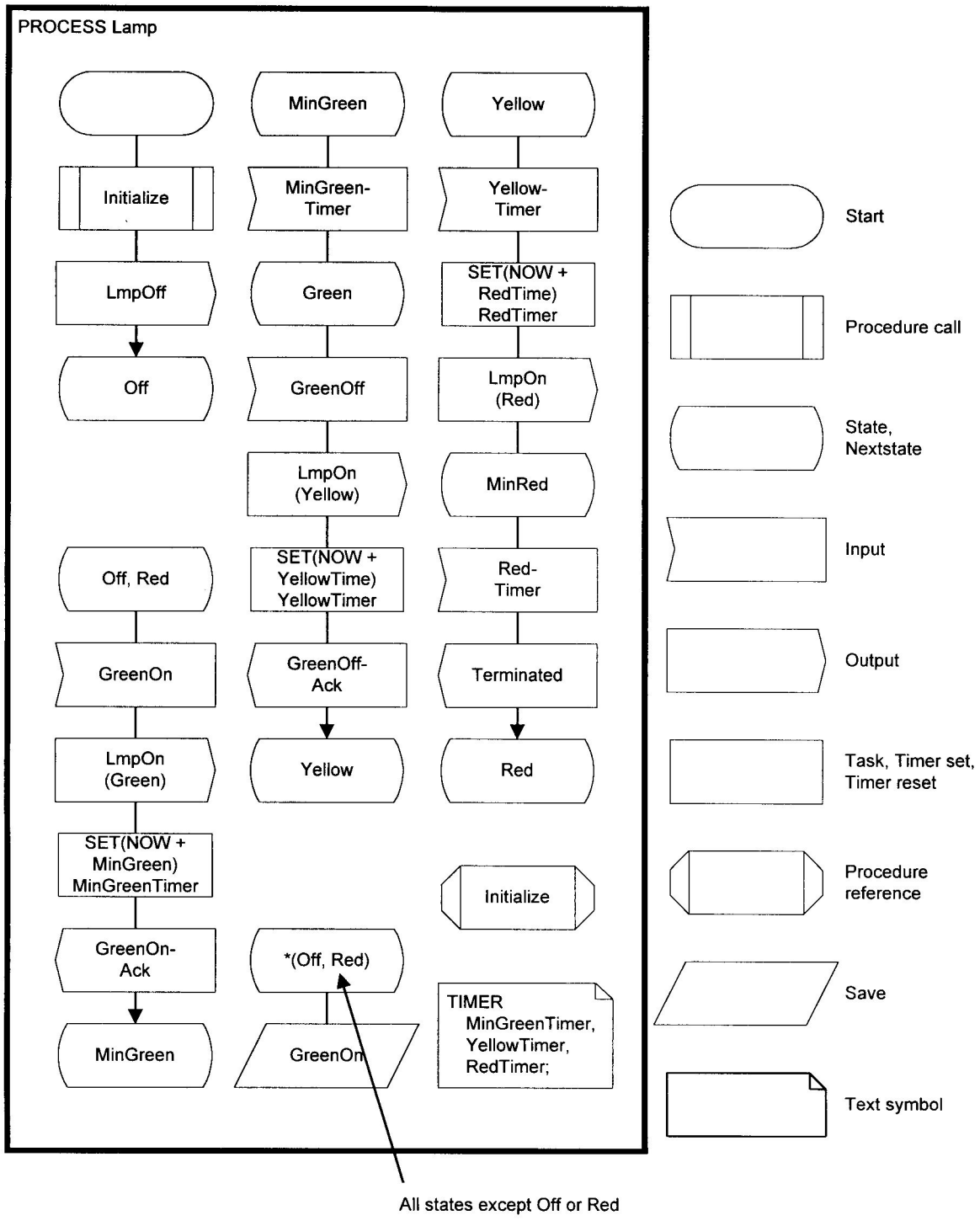

Fig. 3. Example of a process diagram.

task - an assignment of the result of a data type operator to a process variable

create - to dynamically create an instance of a process. The creating and created process must belong to the same block

stop - process termination, at the end of a transition

procedure - to call procedures from within a process or from another procedure. A procedure belongs to a process. It can have its own states, but it shares the same valid input signal set as the enclosing process. Procedure descriptions can also be referenced

save - to keep a signal in the input queue and consume the next signal in the queue

label and *join* - to jump from one part of a state graph to a label at another part of the state graph

enabling condition - to enable signal consumption and state transitions only when a given condition is satisfied

remote procedures - introduced in SDL-92 as an alternative to ordinary signal communication

shared variables - apart from the explicit inter-object communication mechanism of normal signal exchange to gain access to the variables of a process, a variable declared as *revealed* in one process may be *viewed* (but not modified) by another process in the same block. This provides an implicit inter-object communication mechanism in SDL, which overcomes the shortcoming of OMT as discussed by van den Heever and Kourie elsewhere in this issue [16].

To convey information between processes in different blocks, a variable may be *exported* during a transition by a process in one block, and then later *imported* during a transition by a process in another block. This is really a shorthand construct for inter-process communication with signals. A shorthand construct is defined in terms of other constructs and ultimately by the primitive constructs.

Additional shorthand constructs are available. It simplifies the use of the language and reduces the size of a system description. An example of the 'asterisk state' shorthand is shown in Fig. 3. This is a useful shorthand when a given input should cause the same transition in all states.

6. OBJECT ORIENTATION

The SDL model of concurrent communicating processes with encapsulated data and process types that are instantiated is inherently object based. However, the re-use of parts of descriptions is not directly supported by this model. Object oriented structuring features were added to the language in the 1992 version to support re-use [17]. These features will be briefly summarised here. For an introduction to SDL-92, the reader is referred to [18], while a complete treatment can be found in [19] or [20].

In SDL-92 there is a clear distinction between types and instances, not only of processes, signals and data, but also of blocks and complete systems. Instances are the entities in an interpreted system, while types define the properties of the instances in an interpreted system. The types themselves are not part of the interpreted system instance [18].

Since block types or process types can be defined as standalone entities, to be used later in different system descriptions, the notion of a *gate* is introduced. A gate provides a connection point or interface on the boundary of a block or a process, to which channels or routes can be connected when the type is used in a specific system description. Just as for channels or signal routes, the signal types that may be conveyed by a gate must be described.

Powerful object oriented concepts are available for handling types. A type may be a *specialisation* (i.e. an extension) of another type. The extended type is called a *subtype* and the original type a *supertype*. A subtype may need to modify the supertype for specific needs, by redefining some of the locally defined types of the supertype. The supertype determines which types the subtype may redefine. These redefinable types are called *virtual types*. For types which define state machines, such as process types or procedure types, the state transitions may also be redefined by a subtype. These redefinable transitions are called *virtual transitions*.

Types that can be specialised may also be *generic*. A generic type is incomplete as it refers to entities which are not bound to a definition of the type. These entities are called *formal context parameters*. When a generic type with formal context parameters are used, *actual context parameters* must be supplied to complete the description of the generic type. Several similar types may thus be built from a generic type, by supplying different context parameters. It is also possible to place formal constraints on the formal context parameters, which must be obeyed by the actual context parameters.

A collection of types can be grouped into a *package*, as defined by a *package diagram*. This allows for types only to be defined once for subsequent use in the definition of different system instances or other packages. By using a package in a system diagram, for example, all the types contained in the package are then available to be used in the system diagram.

7. COMPUTER AIDED TOOLS

The formal and standardised definition of SDL made the development of sophisticated tools possible to aid the use of SDL during all phases of software development. Two of the commercially available tools are SDT from Telelogic in Sweden and ObjectGeode from Verilog in France [8]. These tools can run on different platforms ranging from PC's to workstations. Both of these tools allow:

- Graphical editing with on-line syntax analysis and semantic help to describe the structure and behaviour of systems
- Translation between SDL/GR and SDL/PR
- Static semantic analysis of a description
- Simulation of a specification to analyse the dynamic semantics of a description

- Exhaustive simulation to check all possible behaviours of a description
- Automatic code generation
- Test suite generation for automatic test execution

An SDL tool with similar functionality, except code generation, has also been developed at the DNA laboratory of the Department of Computer Science, University of Cape Town.

A common interchange format (CIF) to allow the interchange of SDL descriptions between tools from different vendors is currently being standardised by the ITU.

8. INDUSTRIAL EXPERIENCE

BSW Data South recently started using SDL after evaluating and purchasing a commercial tool. The first application was a pilot project involving the embedded software for a traffic signal controller. A team of three software engineers developed the software for the controller using conventional techniques and coding in C. In parallel the software was developed by one engineer using a commercial SDL tool set, which generates C code automatically. After a period of about two months both efforts could demonstrate controller software with the same functionality. The SDL effort resulted in a significant increase in productivity. On the other hand, the automatically generated code required significantly more memory space and computer processing power. This is considered a small price to pay, however, as the cost of computer resources is continually decreasing. SDL is now being used to engineer the software of a much larger real time application.

It was found that software engineers who are used to conventional languages such as C need to make a paradigm shift when starting to use SDL. This sometimes causes a reluctance to grasp the benefits of a formal descriptive technique. The focus of the problem now shifts away from the implementation domain to the requirements domain. Requirements need to be thoroughly analysed before it can be specified formally. Effort needs to be spent on defining the protocols of communication between processes and between the system and its environment before the behaviour of each individual process is described.

It became clear that to use SDL effectively it must be applied within a methodology which emphasises a proper requirements analysis phase. To this end, commercial SDL tool vendors are now providing special analysis and modelling tools as front ends to their SDL tool chains. These tools are based on the Object Modelling Technique (OMT) of Rumbauch et al. [21]. Braek and Haugen also proposes a methodology emphasising requirements analysis and conceptual modelling before specifying a system in SDL [19].

To apply SDL successfully requires a commitment from management to invest in the training of people and to develop and adopt a methodology for the effective use of SDL and associated tools.

Industrial experience with SDL has been reported widely in the literature. The use of SDL in an ISDN terminal design was already reported in 1989 when only limited tool support existed [22]. Chung et. al. describes the use of SDL in the development of the software for a public telephone switching system and emphasises the improvement that SDL achieves in the communication among development teams [23]. Koono et. al. describes their experiences in using SDL for a telephone switching system, a voice response system, a digital telephone, a fax machine and a test bench for a switching system [24]. They conclude that the use of SDL has significantly improved their software engineering process.

Klick et al. also describes the use of SDL in enhancing the feature set of an existing telephone switch [25]. They show how SDL supports the extensive use of iteration in the development process including requirements definition, design, implementation and testing. They state that the graphical representation of SDL made it easy to describe, review, update and restructure the evolving requirements and design.

The introduction of SDL into GEC Plessey Telecommunications (GPT) in the United Kingdom to improve their software development practices is described by Sandhu [26]. He states that the adoption of SDL can only be successful if there is commitment from management as well as users, as the use of SDL requires the acquisition of expensive tools, the development of a methodology and the training of people by means of pilot projects. He emphasises the benefits of the formal use of SDL, namely simulation and code generation with appropriate tools. He concludes that "if they had the opportunity to turn the clock back and were given another chance, they would still choose SDL for their applications". A specific example of the use of SDL within GPT, a call processing subsystem for a telephone exchange, is reported by Wiggins [27]. He found that during development SDL helped to identify errors in both the requirements and the coding.

The application of SDL to system level design of hardware in the context of hardware/software co-design is described by Glunz [28]. An example of a

processor communicating with RAM is shown and the translation of SDL to VHDL is discussed.

Carracedo et. al. describes the use of SDL tools to facilitate incremental prototyping in the development of a gateway between two ISDN signalling systems [29]. They note a shift in effort from the implementation phase to the specification phase when compared with conventional methods.

Experience in the use of SDL in the specification and implementation of a trunking mobile radio system is reported by Zaim et. al. [30]. They report a 100% improvement in productivity by using SDL tools and have decided to increase the use of SDL in future projects.

The use of formal SDL for the specification and validation of Inmarsat Aeronautical system protocols for global communication between aircraft and ground stations is described by Mitchell and Lu [31]. They note that the formal use of SDL to obtain unambiguous specifications results in a large volume of specifications which makes it difficult to present to a large user community.

The development of the software for a public branch exchange (PBX) using SDL is reported by Robnik et. al. [32]. They describe the use of a commercial SDL tool from prototype to product, including automatic code generation. They also emphasise the need for a well defined methodology to guide the use of SDL during system development.

Experience of introducing rigorous use of SDL at Siemens is described by Amsjø and Nyeng [33]. They state that using SDL resulted in a real improvement in terms of calendar time to complete the software for a military data terminal.

The use of SDL to specify future Airbus air navigation systems at Aerospatiale is described by Goudenove and Doldi [34]. They have found that the early test of system specifications allowed by commercial tools detected errors which would not have been discovered with conventional methods.

9. CONCLUSION

With sophisticated computer tools available, the practical application of a formal description technique such as SDL in the engineering of complex software systems has become possible. The graphical syntax makes it user friendly. The effort in the development of systems shifts from the traditional implementation and testing phases to the specification phase. Formal testing and simulation of specifications allows detection and correction of errors early in a project at minimum cost. Automatic code generation makes implementation a simple task and provides continual traceability between implementation and specification during the total life cycle of a product.

The effective use of SDL requires a methodology emphasising object oriented requirements analysis and modelling during the early phases of software engineering.

Industrial experience shows significant improvements in quality and productivity when SDL is used in the engineering of complex software systems. This is particularly true in the case of real time systems.

REFERENCES

[1] B. Meyer, "On formalism in specifications", *IEEE Software*, pp. 6-26, Jan. 1985.

[2] J. Wing, "A Specifiers introduction to formal methods", *IEEE Computer*, pp. 8-24, Sept. 1990.

[3] F. Belina *et al.*, *SDL with applications from protocol specification*, London: Prentice Hall, 1991.

[4] K Turner, *Using formal description techniques, An introduction to ESTELLE, LOTOS and SDL*, John Wiley & Sons, 1993.

[5] S. Budkowski and P. Dembinski, "An introduction to ESTELLE: A specification language for distributed systems", *Computer Networks and ISDN Systems*, vol 14, pp. 3-23, 1987.

[6] T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language LOTOS", *Computer Networks and ISDN Systems*, vol 14, pp. 25-59, 1987.

[7] L. Logrippo *et al.*, "An introduction to LOTOS: learning by examples", *Computer Networks and ISDN Systems*, vol 23, pp. 325-342, 1992.

[8] World Wide Web address for general SDL information: http://www.tdr.dk/public/SDL.

[9] S. Mostert, "A visual method for real-time software engineering", *ibid.*

[10] R. Saracco and P.A.J. Tilanus, "CCITT SDL: Overview of the language and its applications", *Computer Networks and ISDN Systems*, vol 13, 1987.

[11] "CCITT Recommendation Z.100: Specification and Description Language SDL", *Blue Book*, Volume X.1-X.5, Geneva: ITU, 1988.

[12] F. Belina and D. Hogrefe, "The CCITT-Specification and description language SDL",

Computer Networks and ISDN Systems, vol 16, pp. 311-341, 1988/89.

[13] R. Braek, "SDL Basics", tutorial presented at 7th SDL Forum, Oslo, Sept. 1995.

[14] M. Diefenbruch *et al.*, "Performance evaluation of SDL systems adjunct by queueing models", *SDL'95 with MSC in CASE*, Proceedings of the 7th SDL Forum, Amsterdam: Elsevier Science, 1995, pp. 231-242.

[15] M. Bütow *et al.*, "Performance modelling with the formal specification language SDL", to be presented at FORTE/PSTV'96, Joint Int. Conf. on FDTs for distributed systems and communication protocols and protocol specification, testing and verification, Kaiserslautern, Oct. 1996.

[16] R. van den Heever and D. Kourie, "Explicit and implicit inter-object communication", *ibid.*

[17] "CCITT Specification and Description Language (SDL) , Revised recommendation Z.100", COM X-R 26, Geneva: ITU, 1992.

[18] O. Faergemand and A. Olsen, "Introduction to SDL-92", *Computer Networks and ISDN Systems*, vol 26, pp. 1143-1167, 1994.

[19] R. Braek and Ø. Haugen, *Engineering real time systems. An object-oriented methodology using SDL*, Hemel Hempstead: Prentice Hall, 1993.

[20] A. Olsen *et al.*, *Systems engineering using SDL-92*, Amsterdam: Elsevier Science, 1994.

[21] J. Rumbauch *et al.*, *Object-oriented modeling and design*, Englewood Cliffs: Prentice-Hall, 1991.

[22] D. Carl, "The use of SDL in an ISDN terminal design", *SDL'89: The language at work*, Proceedings of the 4th SDL Forum, Amsterdam: Elsevier Science, 1989, pp. 367-376.

[23] C.J. Chung *et al.*, "Using SDL in switching system development", *SDL'89: The language at work*, Proceedings of the 4th SDL Forum, Amsterdam: Elsevier Science, 1989, pp. 377-386.

[24] Z. Koono *et al.*, "Experiences in applying SDL", *SDL'89: The language at work*, Proceedings of the 4th SDL Forum, Amsterdam: Elsevier Science, 1989, pp. 395-404.

[25] V.B. Klick *et al.* "Experiences in the use of SDL/GR in the software development process", *SDL'91: Evolving Methods*, Proceedings of the 5th SDL Forum, Amsterdam: Elsevier Science, 1991, pp. 449-458.

[26] K.K. Sandhu, "The introduction of CCITT SDL into GPT", *SDL'91: Evolving Methods*, Proceedings of the 5th SDL Forum, Amsterdam: Elsevier Science, 1991, pp. 471-482.

[27] M.A. Wiggins, "An example of the use of SDL within GPT", *SDL'91: Evolving Methods*, Proceedings of the 5th SDL Forum, Amsterdam: Elsevier Science, 1991, pp. 499-508.

[28] W. Glunz *et al.*, "System-level hardware design with SDL", *SDL'93: Using Objects*, Proceedings of the 6th SDL Forum, Amsterdam: Elsevier Science, 1993, pp. 17-28.

[29] J. Carracedo *et al.*, "An industrial experience on SDL introduction in a conventional development life cycle", *SDL'93: Using Objects*, Proceedings of the 6th SDL Forum, Amsterdam: Elsevier Science, 1993, pp. 29-40.

[30] A. Zaim *et al.*, "Using SDL in a commercially available wide area coverage trunking mobile radio system development", *SDL'93: Using Objects*, Proceedings of the 6th SDL Forum, Amsterdam: Elsevier Science, 1993, pp. 41-50.

[31] L. Mitchell and S. Lu, "Specifications and validations of Inmarsat aeronautical system protocols", *SDL'93: Using Objects*, Proceedings of the 6th SDL Forum, Amsterdam: Elsevier Science, 1993, pp. 51-64.

[32] A. Robnik *et al.*, "Industrial experience of using SDL in IskraTel", *SDL'95 with MSC in CASE*, Proceedings of the 7th SDL Forum, Amsterdam: Elsevier Science, 1995, pp. 3-14.

[33] G. Amsjø and A. Nyeng, "SDL-based software development in Siemens A/S", *SDL'95 with MSC in CASE*, Proceedings of the 7th SDL Forum, Amsterdam: Elsevier Science, 1995, pp. 339-348.

[34] F. Goudenove and L. Doldi, "Use of SDL to specify Airbus future air navigation systems", *SDL'95 with MSC in CASE*, Proceedings of the 7th SDL Forum, Amsterdam: Elsevier Science, 1995, pp. 359-370.