Tutorial: SDL 2000

Instructors

Prof. J. Fischer, Humboldt-University Berlin, Humboldt-University Berlin, Department of Computer Science, Rudower Chaussee 25, D-12489 Berlin, Germany, email `fischer@informatik.hu-berlin.de`
Professor Fischer has contributed much to the current shape of SDL. Many of the new features were developed within his research team, and discussed within ITU until fitting into the overall language conception. In his work at Humboldt-University he has given many lectures about design and specification in the scope of SDL. He is also involved in many national and international projects using SDL.

Dr. Andreas Prinz, DResearch Digital Media Systems GmbH, Otto-Schmirgal-Straße 3, D-10319 Berlin, Germany, email `prinz@dresearch.de`
Dr. Prinz has led the activities to define a new formal semantics for SDL 2000. In this position he is among those with the most intimate knowledge of the language SDL and its peculiarities and the meaning of the constructs of the language. In his work with DResearch he is applying SDL in telecommunication projects with Siemens and other companies.

Dr. Eckhardt Holz, Humboldt-University Berlin, Humboldt-University Berlin, Department of Computer Science, Rudower Chaussee 25, D-12489 Berlin, Germany, email `holz@informatik.hu-berlin.de`
Dr. Holz was responsible for the harmonization of the concepts BLOCK and PROCESS in SDL. He was also involved into the alignment of SDL to UML. He has led several projects for the application of SDL.

Summarised Contents of the Tutorial

A new version of the Specification and Description Language (SDL) has been standardized in November 1999 by the International Telecommunication Union (ITU). This version is a major revision of SDL taking into account many modern concepts of the design of distributed and object-oriented systems.
In the tutorial, the following areas will be covered.

- **SDL History**: SDL is a living language, which means it is used frequently with new applications in new areas. Although SDL was developed already 1968 for the description of signalling and switching systems, its current version is used today for the description and the implementation of distributed systems in general.
- **SDL-2000 Language**: The latest version of SDL will be covered by the tutorial. Simple examples will illustrate the power of the language. The development of a complete specification of a system over various refinement steps is one goal of the tutorial.
  - **Agents with Behaviour and Structure**: Agents are the fundamental specification concept of SDL. An agent combines both behaviour and structure in a single entity and corresponds thus to the concept of an object in other object-oriented techniques and languages.
  - **Communication Principles**: On the base of the asynchronous messaging by signals SDL-2000 supports specialised styles of communication.
  - **Object-Orientation in SDL**: Object oriented features of the SDL are not only defined for structural and data concepts. For behaviour descriptions on the base of extended final state machines inheritance and virtuality are important new concepts.
  - **Composite States**: In SDL-2000 the traditional scalable transition oriented state machine concept of SDL is combined with the benefits of Harels well-known statecharts.
- **SDL and Other Technologies**: The language SDL integrates very well with other languages and technologies used for the specification and design of distributed systems. The span of such technologies ranges from the data description language ASN.1 over Message Sequence Charts to the object-oriented language UML.
- **SDL Formal Semantics**: SDL-2000 comes with a formal semantics definition as all versions of SDL since 1988. The SDL-2000 formal semantics is defined on the base of Abstract State Machines (ASM). Great care has been taken to make the semantics intelligible, maintainable and executable. It covers all parts of the SDL language that are semantically relevant.
- **SDL Tool Support**: By using convenient tools, the SDL developers are able to reduce the complexity of the language. Existing tools for the previous language version support different areas, as specification and design, verification and validation, code generation and interface support, documentation and change management

A subset of the complete tutorial can be found at
http://www.informatik.hu-berlin.de/~holz/SDLTutorial/SAMTutorialFinal.htm

Justification of the Relevance for FME 2001

The current trend in software development is going towards higher levels of abstraction. For this reason, specification languages, object-orientation and design notations play an increasing role. SDL has a long standing tradition as a design and specification language, it has a long tradition with its formal semantics and it is also object-oriented. Moreover, in its latest release it is aligned with UML making many important constructs of UML available in the scope of a semantically well-founded language.
The study and application of SDL helps in understanding many of today's problems in software construction. Moreover, for many of those problems, SDL is actually the solution, as it not only comes with a *formal* but also an *executable* semantics making automatic code generation possible. In fact there are a variety of tools available that provide code generation for large subsets of SDL.

Duration of the Tutorial

The tutorial will last a full-day. A restricted version could be provided as a half-day tutorial.

Key learning objectives for the participants

The participants can expect to learn:

- Knowledge about SDL as a specification and design language.
- Knowledge about modern software paradigms and their smooth integration.
- Knowledge about assigning semantics to a real-world language and the benefits thereof.

Intended audience and required background

The intended audience is software engineers up to managers.
While learning about SDL they will also learn about modern approaches to the design, specification and implementation of distributed systems.
The tutorial will be self-contained, such that no background knowledge is required (apart from basic computer and software knowledge that can be presupposed in a conference about software).

Detailed Contents of the Tutorial

Apart from the usual language maintenance, SDL-2000 does offer new features as object-oriented data types, it simplifies and unifies concepts of previous language versions and provides an alignment with UML. These new language features allow to use SDL in combination with UML, but also to use SDL for object-oriented analysis and design directly. However, unlike UML SDL is also used as a high-level implementation language with a formal semantics. Due to the significant changes to SDL it was necessary to define a new formal semantics for SDL.

The History of SDL

Although SDL today is a language applicable to the specification and implementation of distributed systems in general, it has its origins in telecommunications. The development of SDL arose out of a study of the appropriate way to handle stored program control switching systems raised in the ITU in 1968. The result of this study was to agree in 1972 that languages were needed for specification, programming and human machine interaction. The first, small SDL standard was produced in 1976 as the language for specification. Things changed significantly around 1984 as the first tools were being introduced. The tools forced both, users and designers of SDL, to be more formal. This required more work, but the benefits were the identification of errors and the ability to animate models, so "what-if" questions could be answered. 1988 saw the introduction of a formal definition for SDL in VDM (alias Meta IV) to underpin the natural language description. There was another significant update to SDL in 1992 by the addition of **type** constructs for an object oriented version of SDL. This version of SDL is called SDL-92.
Uptake of tools was initially slow even within the industry, because the graphical tools were weak and expensive. The situation today is that SDL tools have high functionality, and a proven track record. The SDL tool market has expanded and changed significantly in the last two years. The reason is, that it has become practical to use SDL for the (semi-) automatic generation of implementations. SDL tools can produce source code in programming languages (usually C/C++) directly from an SDL specification and this code can be linked with a run time system to make products. The generated C++ is treated as an intermediate language in much the same way as compilers treat assembly language. Of course, SDL can still be used in an abstract way with informal text, so that SDL is a broad-spectrum language that can be used from requirements capture to implementation.

Being a standardised language, a major user of SDL is the ITU itself. Many telecommunication protocol standards come with a formal description of the protocol in SDL. Currently, it is even mandatory to have such formal SDL parts (in ETSI standards even with normative character, see Q.2931).
As a consequence of the increased application of SDL for the description, design and implementation of a multitude of distributed systems some deficiencies of the language have been identified by the users. Major points concerning the object-oriented features were:

- lack of a strong and syntactically clean distinction between classes/types and interfaces;
- lack of an object-oriented data model with polymorphic properties and reference signal parameters;
- complexity of the language due to many and overlapping concepts.

Furthermore, concepts for exception handling, improved representation of sequential algorithms and the possibility to define hierarchical state machines were required.
All of these issues have been addressed with SDL-2000.

The Language SDL-2000

An SDL specification of a distributed system is a formal description of both its architecture and its behaviour. Systems are hierarchically structured into agents connected by communication channels. Agents can in turn be decomposed into sub-agents. Agents may be used of structuring purposes and may also have a dynamic behavior consisting of internal actions and interactions by asynchronous signal exchange with other agents or the systems environment.

Agents with behaviour and structure

Agents are the fundamental specification concept of SDL. An agent combines both behaviour and structure in a single entity and corresponds thus to the concept of an object in other object-oriented techniques and languages. What distinguishes the agent from the object is the property that an agent is always active, i.e. it owns a thread of control and may initiate control activity. An agent reinforces the principles of abstraction and encapsulation by capturing the outside view on an agent in a set of interfaces the agent supports and by hiding the details on how this behaviour is achieved inside the agent.

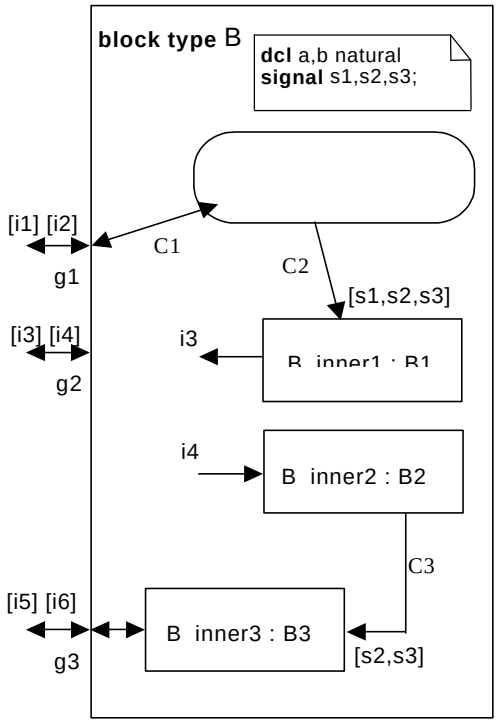

Figure 1 Sample Agent Type Definition

A class of agents is defined by an agent type definition, whereas interaction diagrams give a view on a concrete configuration of agent instance sets and the communication infrastructure between these sets. In general an agent type definition is split into four parts:

- A behaviour specification given by the agents *service*. A service is an extended finite and communicating state machine.
- An interaction diagram detailing the internal structure of the agent, i.e. the sets of contained agents and the internal communication infrastructure (*channels*).

- The internal data of the agent (*variables*).
- The outside view on the agent consisting of the connection points for channels (*gates*) and the *interfaces* supported or required at those gates.

Besides that an agent type definition may also contain other type definitions (e.g. data type definitions, agent type definitions), which are then local to the containing definition. An example of an agent type definition is given in figure 1.
The diagram in figure 1 defines an agent type B. Using the three gates (g1, g2, g3) the interfaces i2, i4 and i6 are specified as supported by the agent, whereas the interfaces i1, i3 and i5 are required from the environment. Internally the agent contains three other agents (B_inner1, B_inner2 and B_inner3) and some explicit channels (C1, C2, C3). The unlabelled oval circle is a reference to the agent´s behaviour specification.
SDL-2000 distinguishes the three agent kinds block, processes and system.
A block is the most common kind. The agent instances which are contained within a block are interpreted concurrently and asynchronously with each other and with the state machine of the containing block instance.
A system is just a special kind of block, it is the outermost agent and contains also the definitions of all signals, channels, data types etc., used in the interface with the environment.
Processes differ from blocks by the execution semantics of their contained agents; an interleaving interpretation is applied here using a full transition as granularity.
An interface specification is very similar to an interface in CORBA-IDL. It declares a set of operation and signal signatures as well as attributes (remote variables).

```
interface i1;
        signal s1 (natural, charstring);
        procedure p (in boolean) boolean;
        dcl v integer;
end interface;
```

Figure 2 Interface Example

As any other type definition agent types as well as interfaces can be organized in an inheritance hierarchy (interfaces also with multiple inheritance). This allows in a specialization to add new elements or to redefine existing virtual elements.
The behaviour part of an agent is given by a service specification in form of a state-transition-graph. The finite state machine of an agent is either waiting in a state or active, performing a transition. A transition is initiated by a trigger, which is usually the result of a communication (signal, remote procedure call). During a transition an agent may perform a series of actions ranging from sending signals to the creation of new agent instances. A transition is terminated either by entering a new state or by stopping the agent. Exactly one input port is associated with the agent. Signals sent to an agent will be delivered to the input port of the agent. Signals are consumed in the order of their arrival either as a trigger of a transition or by being discarded in case there is no transition defined for the signal. The consumption may be deferred to some other state by the special save construct.

Communication Principles

Based on the asynchronous messaging by signals SDL-2000 supports specialised styles of communication:

- asynchronous message exchange (basic communication pattern),
- client-server style by remote procedures, and
- read-only access to remote variables.

In order for two agents to communicate successfully both have to provide gates with matching interfaces (required and supported) and a connection between these gates must exist. A connection is given by an explicit or implicit channel, which provides a secure, but possibly delaying transmission of communication elements from a sender to a receiver agent.

Object-orientation in SDL

Basic object-oriented concepts are *object* and *class*. For historical reasons, these concepts are named differently in SDL, the corresponding notions are the terms *instance* and *type*.
Examples of SDL instances are *system*, *block*, *process*, *service*, *procedure*, and *signal*. SDL instances have an identity and are characterised by features such as structure, behaviour, or values.

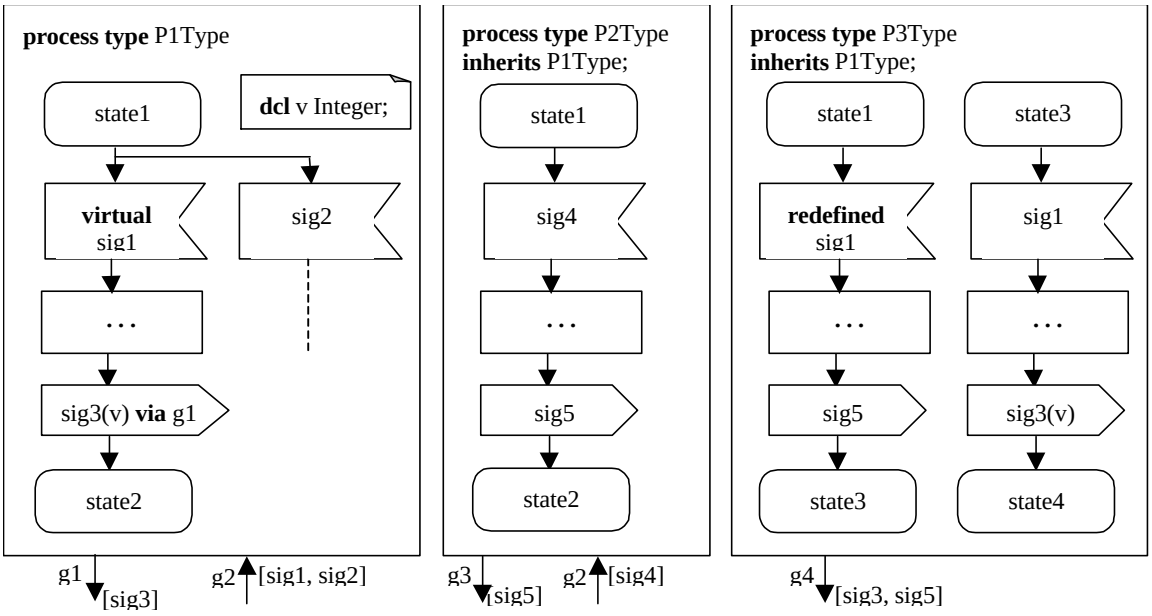

Figure 3 Specialisation of Behaviour in SDL

Classes are defined by SDL types, e.g. *system type*, *block type*, *process type*, *service type*, *object type*, etc. SDL types can be specialised, leading to subtypes. The specialisation of a type *A* can lead to additional features, i.e. structure, behaviour, or data, as well as to the redefinition of existing features. SDL distinguishes between simple specialisation (adding features) and advanced specialisation (redefining features). Syntactically, specialisation is supported by an inheritance mechanism.
Figure 4 above gives examples of specialisation by adding and replacing behaviour of an agent process type `P1Type`. By inheriting from `P1Type`, `P2Type` acquires all features (declarations (`v`), states (`state1`, `state2`), transitions (the behaviour between the states), gates (`g1`, `g2`), etc.) from `P1Type`. Further elements extending the process behaviour, in particular a new transition and an extension of gate g2, are added. Note that in the definition of `P1Type`, the input transition accepting `sig1` is declared as *virtual*. This allows the redefinition of this transition in process types that inherit `P1Type` as shown in `P3Type` (see again Figure 4). If a virtual transition is redefined, it is replaced by another transition. A redefined transition is again virtual by default unless it is marked as *finalized*.
Further specialisation is possible in SDL with respect to structure. For instance, a block type A specialising block type B inherits all features, i.e. all declarations, instantiations, connections, of B and can add further features. As in case of transitions, block types or process types may be declared as virtual, which allows one to replace this block type in a specialisation of the containing class specification. This option is necessary in cases where the interaction of added instances with inherited instances influences their behaviour. Also, signals can inherit attributes of other signals while adding further signal parameters.

Composite States

In SDL-2000 the traditional and scalable transition oriented state machine concept of SDL is combined with the benefits of Harels Statecharts offering the following expressive power:

- Separate *state diagrams* for composite states, with *state connection points* facilitating scalability and encapsulation;
- *Types and subtypes of states* for states in general, and not only for the outermost state, inheritance of states and transitions;
- *Virtual states/transitions* that can be *redefined* in a subtype, and states that can *not* be redefined (in the spirit of Java's final methods);
- State types with *type parameters*.

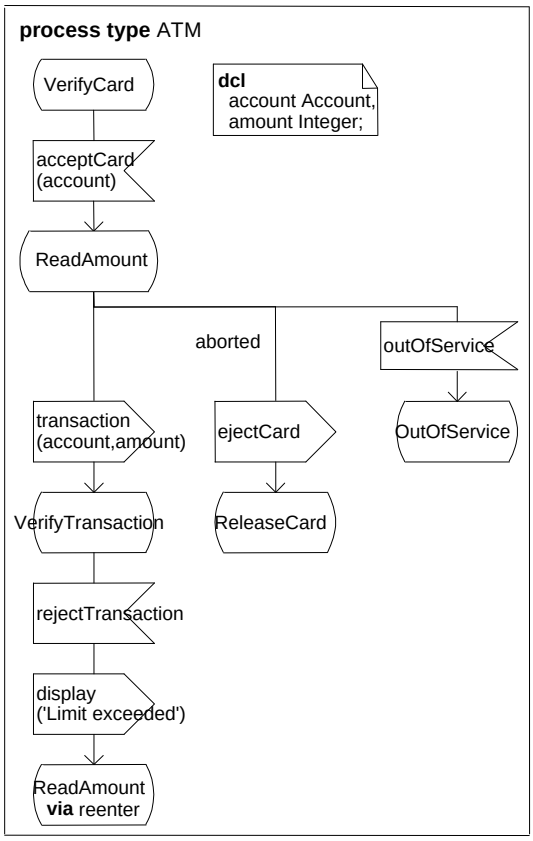

Figure 4 Sample State Machine with Composite State

An SDL agent may have a state machine with a set of states and transitions. A composite state is a state that in turn consists of states and transitions. In the enclosing state machine, a composite state appears as an ordinary state (i.e. as a state symbol); with the exception that *composite states* are entered through *entry points* and *exited through exit points*. Transitions applying to a composite state apply to all substates of the composite state. A separate state diagram defines the composite state.

As an example, a state machine for an ATM agent (process) with a composite state `ReadAmount` (to enter the desired amount) is described in Figures 5 and 6.
If the composite state `ReadAmount` exits (normally), the state machine makes a transition to `VerifyTransaction`. If it exits via the *exit point* `aborted`, then it makes a transition to `ReleaseCard`. An input of the signal `outOfService` applies to *all substates* of `ReadAmount`, and in case of an `outOfService` signal, the next state is `OutOfService`.
The `ReadAmount` composite state is described in a *separate state diagram* (Figure 6). This defines the substates (`SelectAmount` and `EnterAmount`) and the transitions associated with them. The state diagram defines the entry and exit connection points (`reenter` and `aborted`). A designer of the composite state is therefore free to decompose the composite state in whatever sub states, without any implications for the enclosing state machine. The state with the `*` illustrates an SDL feature (asterisk state): as specified here, the implication is that in all states the reception of signal abort will exit the composite state through the connection point aborted.

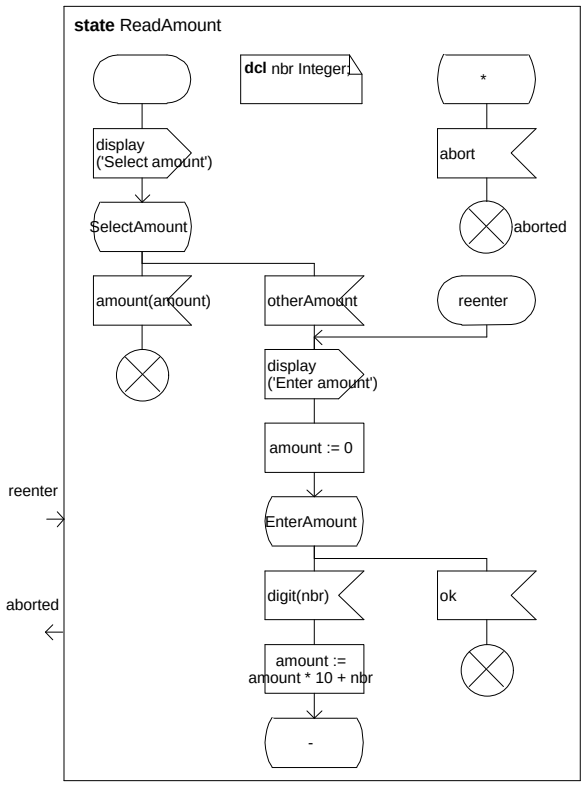

Figure 5 Sample Composite State Diagram

An exit symbol without any label indicates a normal exit of the composite state; in the enclosing state machine there can only be one normal exit transition from a composite state, but as many labelled exit transitions as there are exit points.
An entry point is matched by a corresponding labelled start symbol, indicating where the composite state shall be (re)started when entered via the connection point.
State types are useful in cases where the properties of a composite state are applicable in many systems or even in different parts of the same system. Several larger state machines may use composite state types describing the states and transitions needed e.g. for starting/stopping, for providing status or for the handling of general management signals.
A state type may be defined as a *specialisation* of another (super) state type, *inheriting states and transitions* of the supertype and *redefining virtual states* and *transitions*. States of the supertype cannot be deleted, but virtual states can be redefined. States and transitions can be added in a subtype.
Entry/exit procedures (that are executed on enter to/exit from a state) may also be defined as virtual procedures that can be redefined.
In order for state types to be defined in *packages* and used in different systems with different local definitions, state types can have *type parameters*, in the same way as provided for classes (template parameters) in many other languages.

Object-oriented Data

In the new version of SDL, also the data type part of the language has object-oriented features. With this extension, it is now possible to design systems object-oriented in all details. The new data type part is similar to that of the Eiffel language. The basic concepts of the data are object types and value types. Object types refer to instances in the usual sense, while value types refer to value instances with no identity, that are not allowed to have references to them. Therefore, value types have a value semantics as it was before in SDL, while object types have reference semantics as usual in object-oriented languages. It is always possible to change the mode of a data definition by giving it a *value* or *object* prefix. Moreover, it is also easy to switch between object and value instances at run time. Object-orientation is introduced for the data by means of inheritance, i.e. it is possible to base data type definitions on other data type definitions. Currently, only single inheritance is possible. The derived data type has the same features as its parent. It may, however, change these features as well. For this to be possible the corresponding methods of the parent have to be marked as virtual. For inherited methods, method lookup is done dynamically, i.e. the method to be called is computed from the actual arguments rather than from the formal arguments. The new SDL data have multiple dispatch, which means that the method to be called is computed taking into account many arguments. Moreover, new methods can be added in the scope of inheritance. Finally it has to be noted that the new data type part of SDL is carefully designed to be backwards

compatible to the old one which had value semantics (even the syntax was kept as far as possible), and it is designed to be type-safe. Moreover, it includes many features that are important for having support for data definitions based on ASN.1.

SDL and Other Technologies

The language SDL integrates very well with other languages and technologies used for the specification and design of distributed systems. The span of such technologies ranges from the data description language ASN.1 over Message Sequence Charts to the object-oriented language UML. Due to its importance for the development of large software systems, the tutorial will put the main emphasis on the latter one.
Rather than being competitors, the two languages UML and SDL complement each other for object-oriented analysis, specification and design. UML has its strengths in domain analysis, early system analysis and data object modelling, while SDL is strong on system structure and reactive behaviour design. By combining the languages a larger part of the life cycle and a wider application area can be covered than by each of the languages alone. The SDL language definition facilitates the combination with UML on two levels:

- Use of UML in the early phases of the development and use of SDL for the detailed design up to the code generation.
- Use of UML within or together with an SDL specification in the same development phase.

The first kind of combination allows the user to apply the full variety of concepts and notations of UML for the requirements capture and the analysis. Special rules to restrict the number of concepts and to eventually transform the model into an SDL specification are applied subsequently.

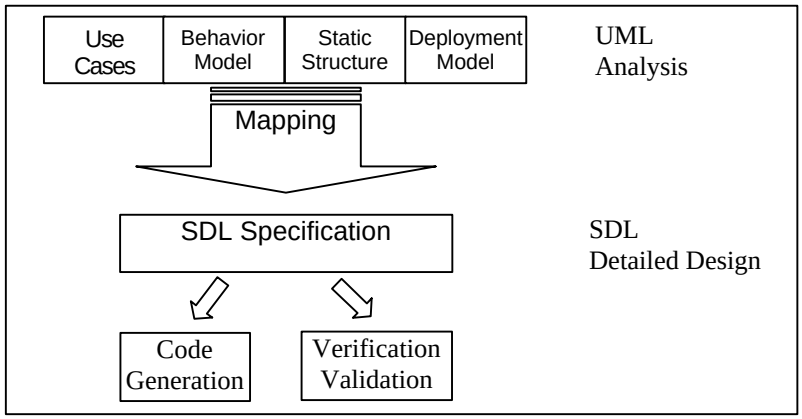

Figure 6 Combination of SDL with Other Technologies

The resulting specification serves as the starting point for a further detailed design as well as for the application of validation and code generation tools. The rules for the transition to a design specification in SDL are defined in the ITU-T Recommendation Z.109 " *UML Specialisation and Restrictions for combination with SDL* (an *SDL UML Profile*)". It ensures a well-defined mapping between a UML model and an SDL model. For each of the UML model elements that are included in *SDL UML*, there is a *one-to-one* mapping to the corresponding SDL concept is. A tool that implements *SDL UML* supports these specialisations and restrictions accordingly and is able to provide this one-to-one mapping. UML specialisations and restrictions are defined in terms of the UML meta-model and the abstract grammar of SDL, that is independent of notation. The Recommendation also gives notation guidelines for *SDL UML*, in the same way as notation guidelines are given as part of the UML standard. The transformation rules from UML to SDL do not cover the full language UML. The starting point for a mapping are the static structure diagrams (classes, objects, interfaces and relations) and the behaviour diagrams (state charts, activity diagrams). The main task a user has to undertake in order to make his UML model apt for the mapping, is to assign the stereotypes defined in Z.109 to the elements of his model (see also Figure 8).

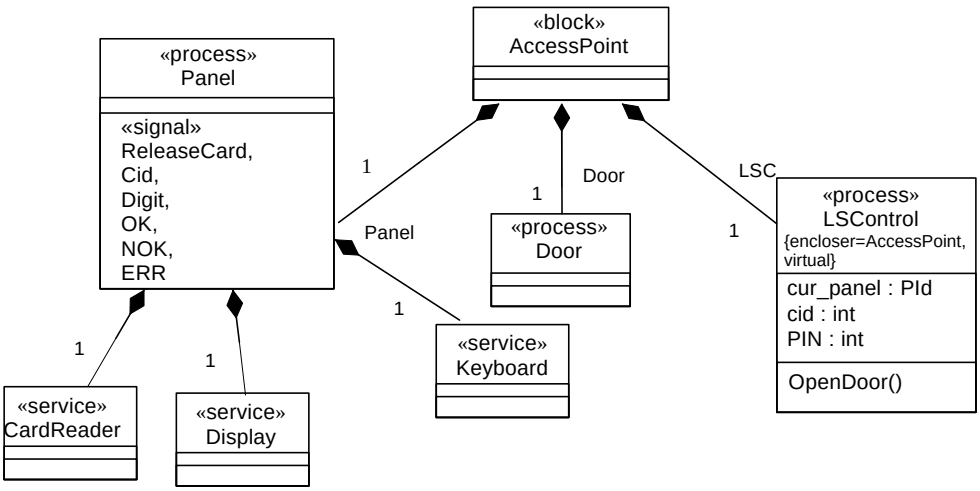

Figure 7 An SDL UML Example

For the use of UML and SDL at the same stage of the development (especially within the SDL specification) it is now possible to use a subset of the UML notation elements in an SDL-2000 specification. Furthermore some notational conventions of SDL have been adapted to the UML style. The main additions and changes in this area are:

- use of UML class symbols to visualise SDL type definitions in addition to the existing type symbols;
- use of UML inheritance symbols to visualise SDL specialisation relations between types;
- use of UML association symbols to reflect relationships between SDL types;
- use of the UML package symbol.

These new graphic symbols can be used freely in combination with already existing SDL symbols. The main contribution – besides the notation of UML sub-models in an SDL specification – lies in the reflection of information otherwise distributed over and deeply hidden in an SDL specification. Moreover, the use of a common subset of notation elements also eases the understanding of specifications obtained from UML by a mapping according to the Z.109 rules.
Besides these structural concepts SDL-2000 now also provides new concepts for the specification of behaviour in a similar way as UML. The major new features here are the separation between interfaces and implementation and the possibility to define hierarchical state machines (composite states) as already mentioned.

A practical formal semantics

In 1988, a formal semantics for SDL has been added to the Z.100 SDL standard as Annex F. Along with the efforts to improve SDL, the semantics has been revised several times since then. Essentially, the SDL-92 Annex F defines a sequence of Meta IV programs that take an SDL specification as input, determine the correctness of its static semantics, perform a number of transformations to replace several SDL language constructs, and interpret the specification. It has been argued that this style of defining the formal semantics is particularly suitable for tool builders.
With the ongoing work to improve SDL, and the new version SDL-2000, the formal SDL semantics has to be adapted. Based on the assessment that the existing Meta IV programs will be too difficult to maintain, it has been decided to define a new formal semantics. In several discussions with the SDL language definition group (ITU-T study group 10, question 6), the essential design objectives of such a semantics have been clarified. Among the primary design objectives is *intelligibility*, which is to be achieved by building on well-known mathematical models and notation, a close correspondence between specification and underlying model, and by a concise and well-structured semantics document. Of similar importance and causally dependent on intelligibility is *maintainability*, since SDL is an evolving language. Language modifications currently under consideration include exception handling, hierarchical states, dynamic block creation, object-oriented data types, real time expressiveness beyond timers, and the removal of some obsolete language features. Therefore, the semantic model has to be sufficiently rich and flexible in order to capture the SDL-2000 semantics and to be able to also include the modifications to come.
During the discussions, it has turned out that a prime design objective is the demand for an *executable semantics*. This calls for an operational formalism with readily available tool support. Subsequent investigations have shown that *Abstract State Machines* (ASMs) meet this and all other design objectives, and therefore have been chosen as the underlying formalism. The dynamic semantics associates, with each SDL specification, a particular multi-agent, real-time ASM. Intuitively, an ASM consists of a set of autonomous agents cooperatively

performing concurrent machine runs. The behaviour of agents is determined by ASM programs, each consisting of a set of transition rules, which define the set of possible runs. Each agent has its own partial view on a given global state, on which it fires the rules of its program. According to this view, agents have a local state, and a shared state through which they can interact.
As in the past, the formal semantics is defined starting from the abstract syntax of SDL, which is documented in Z.100. From this abstract syntax, a behaviour model that can be understood as ASM code generated from an SDL specification is derived. This approach differs substantially from the interpreter view taken in previous work, and will enable SDL-to-ASM compilers. Furthermore, the new Annex F includes the static semantics constraints and the data model.

SDL Tool Support

SDL is a complex language and the addition of object oriented concepts has increased the complexity even more. However by using convenient tools, both the occasional SDL user and the expert SDL developer are able to reduce this complexity. Typical SDL tools can be classified into the following groups.

- Specification and Design:
  - graphical and textual editors that incrementally check for syntactical correctness
  - navigation facilities
- Verification and Validation in the following sense
  1. correctness of SDL syntax and static semantics
  2. analysis of dynamic semantics and reachability
  3. model equivalence checking (requirements against specifications)
     - simulators, MSC-generators
     - debugging facilities
     - test case generators
- Implementation
  - code generator
  - environment and interface support
- Document and change management
  - maintenance facilities
  - online help documentation
- Environment and interface support
  - translators between different syntactical representations of SDL
  - common interchange format (CIF)
  - tool integration across multiple environments
  - exchange of drawings with commercial available text processing systems

Whereas commercially available tools usually tend to cover all these areas, specialized proprietary tools concentrate on the areas of verification /validation and implementation. The language standard currently covered by the tools is SDL-92. Major work is going on to adapt them to the new version SDL-2000 and to integrate other technologies. Two strong tendencies which can be observed are the combination with UML tools and the extension of the code generation possibilities in the domain of CORBA.