

SDL-2000: A Language with a Formal Semantics

Authors:

- Joachim Fischer, Humboldt-University Berlin, Institut für Informatik, Rudower Chaussee 25, D-12489 Berlin, Germany, email: fischer@informatik.hu-berlin.de Tel: +49 30 2093 3109, Fax: +49 30 2093 3112
- Eckhardt Holz, Humboldt-University Berlin, Institut für Informatik, Rudower Chaussee 25, D-12489 Berlin, Germany, email: holz@informatik.hu-berlin.de Tel: +49 30 2093 3116, Fax: +49 30 2093 3112
- Martin v. Löwis, Humboldt-University Berlin, Institut für Informatik, Rudower Chaussee 25, D-12489 Berlin, Germany, email: loewis@informatik.hu-berlin.de Tel: +49 30 2093 3116, Fax: +49 30 2093 3112
- Andreas Prinz, Humboldt-University Berlin, Institut für Informatik, Rudower Chaussee 25, D-12489 Berlin, Germany, email: prinz@informatik.hu-berlin.de Tel: +49 30 2093 3831, Fax: +49 30 2093 3112

SDL-2000: A Language with a Formal Semantics

Joachim Fischer, Eckhardt Holz, Martin v. Löwis, Andreas Prinz

Humboldt-University Berlin, Institut für Informatik, Rudower Chaussee 25, D-12489 Berlin, Germany

{ fischer, holz, loewis, prinz }@informatik.hu-berlin.de

Abstract. A new version of SDL called SDL-2000 is currently reaching maturity, and is expected to pass the standardization bodies shortly. It will offer new features as object-oriented data types, simplify and unify concepts of previous language versions and provide an alignment with UML. Due to the significant changes to SDL it was necessary to define a new formal semantics for SDL. *Abstract State Machines (ASMs)* have been chosen as the underlying formalism. In this paper, after introducing the new SDL language features, it is shown, how these features get an executable formal semantics using ASMs.

1 Introduction

SDL-2000 is on the agenda for approval by ITU-T Study Group 10 in November 1999, after which it will come into power as an international standard as soon as the ballot procedure is completed.

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 the 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 slow 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.

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 large and overlapping number of concepts.

Furthermore, concepts for exception handling, improved representation of sequential algorithms and the possibility to define hierarchical state machines were required.

These latest trends have pushed SDL in two directions: combining it with object modeling techniques (in particular UML), and improving its use for the automatic generation of implementations. This is reflected by the new language version SDL-2000 which includes now features as:

- exceptions and exception handling;
- new data model, including object data and direct support of ASN.1 with SDL;
- a unified structuring concept for blocks, processes (agents);
- composite states and state machine decomposition by services;
- interfaces, class symbols as references, associations.

The complete description of SDL-2000 consists of Z.100(11/99) for the main text, Z.105(11/99) for ASN.1 in SDL modules, Z.107 for ASN.1 embedded in SDL, and Z.109 for SDL combined with UML. Moreover, there are several annexes to Z.100, in particular Annex D for the predefined data and Annex F for the formal semantics.

Work is not yet completed for SDL-2000. The formal definition and the Z.106 Common Interchange Format (CIF) standard are to be updated. These should be approved at a special SG10 meeting in spring 2000.

Since 1988, a formal semantics is part of the Z.100 SDL standard of ITU-T [15]. Along with the efforts to improve SDL, this semantics has been revised several times since then. Essentially, the previous Annex F defined a sequence of Meta IV programs that took an SDL specification as input, determined the correctness of its static semantics, performed a number of transformations to replace several SDL language constructs, and interpreted the specification. With the ongoing work on SDL-2000, it has become apparent that a new SDL formal semantics has to be defined.

The prime design objective for the formal semantics is the demand for *executability*. This calls for an operational formalism with readily available tool support. Subsequent investigations have shown that *Abstract State Machines* (ASMs) meet this, and therefore have been chosen as the underlying formalism. In this paper, we show how ASMs are applied to define the behavior model of an SDL specification formally. More specifically, we develop a behavior model that can be understood as ASM code generated from an SDL specification. This approach differs substantially from the interpreter view taken in previous work (see also [5], [6]), and will enable SDL-to-ASM compilers.

We define a distributed execution model for SDL based on the formalism of *multi-agent real-time ASM* [8]. The behavior of active SDL objects is described in terms of concurrently operating and asynchronously communicating ASM agents directly reflecting the intuitive understanding of Z.100. Due to its abstract operational view, one obtains a clear and concise formalization of the dynamic semantics of SDL at a natural level of detail and precision.

In section 2 we introduce the language SDL-2000 in more detail and section 3 shows an example of how to use SDL. Section 4 shows the SDL way to handle the large number of language constructs. Section 5 shows how the dynamic semantics is given to the language constructs, Section 6 shows executability and finally Section 7 contains the conclusions.

2 SDL-2000

An SDL specification [15, 17, 18] is a formal description of both the architecture and the behavior of a system. Systems are hierarchically structured into agents connected by channels. Agents can in turn be decomposed into sub-agents. Agents may be used for 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.

In this section we will show some important features of SDL that are needed in the rest of the paper.

2.1 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 characterized by features such as structure, behavior, or values. Classes are defined by SDL types, e.g. *block type*, *process type*, *service type*, *object type*, etc. SDL types can be specialized, leading to subtypes. The specialization of a type *A* can lead to additional features, i.e. structure, behavior, or data, as well as to the redefinition of existing features. SDL distinguishes between simple specialization (adding features) and advanced specialization (redefining features). Syntactically, specialization is supported by an inheritance mechanism. In the following, we consider the specialization of behavior only; for further details, especially on specializing structure, see [17], [18].

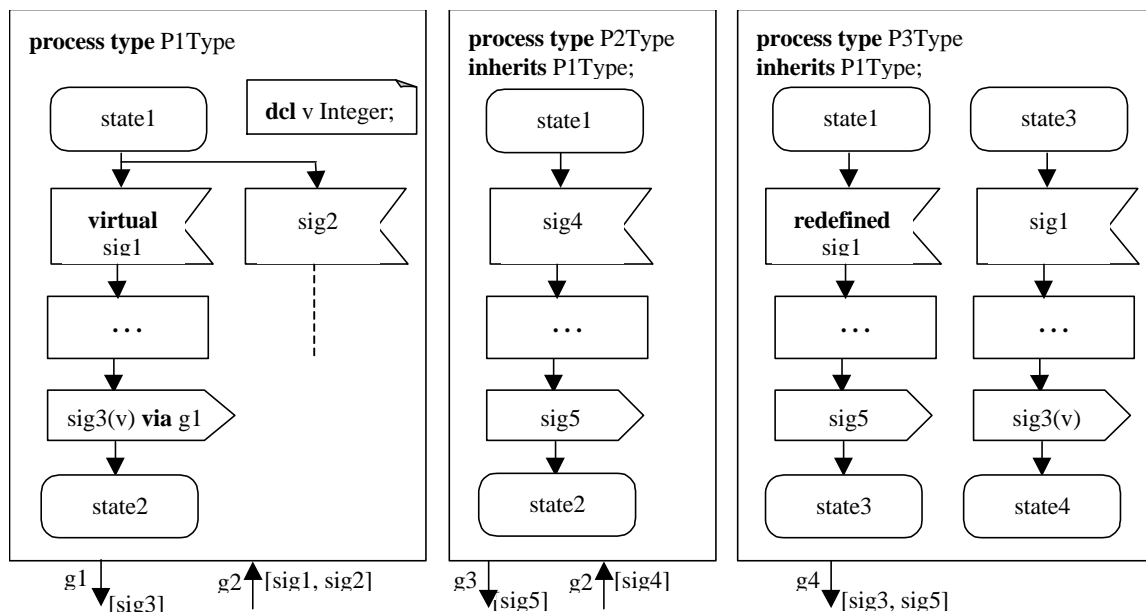


Fig. 1. Specialization of Behavior in SDL

Figure 1 above gives examples of specialization by adding and replacing behavior. By inheriting from P1Type, P2Type acquires all features (declarations (*v*), states (*state1*, *state2*), transitions (the behavior between the states), gates (*g1*, *g2*), etc.) from P1Type. Further elements extending the process behavior, in particular a new transition (and additional states) 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 1). 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 specialization is possible in SDL with respect to structure. For instance, a block type A specializing 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 specialization of the containing class specification. This option is necessary in cases where the interaction of added instances with inherited instances influences their behavior. Also, signals can inherit attributes of other signals while adding further signal parameters.

2.2 UML alignment

Great care has been taken to include new features into SDL to make it possible to use SDL together with UML. This is, because UML is a language that is good at describing classes/types and relationships between them in a simple and intuitive manner, but UML also has its weaknesses. However, SDL is strong exactly where UML is weak:

- it provides a clear structural decomposition concept (agents), which can be combined with inheritance;
- agents can have sequential behavior defined using composite states and contain passive objects (data) as well as concurrent or interleaved active objects (agents);
- it is a complete language with formal semantics based on communicating state machines that enable comprehensive tool support for simulation, verification, validation, testing and code generation. In this way SDL provides the formal semantics badly needed in UML and in particular when using UML for reactive real-time systems.

Therefore a combined use is seen as a very attractive approach for embedded reactive systems. In such a marriage UML contributes the description of classes/types and their external relationships, while SDL contributes typed object structures, and well-defined object behavior. In order to facilitate the marriage, the two languages have been aligned in SDL-2000:

- SDL now includes composite states similar to UML;
- Associations are introduced into SDL;
- UML Class symbols may be used to represent type references in SDL;
- A way to use UML with SDL and mapping to SDL has been defined in recommendation (Z.109).

This combination of SDL-2000 and UML is a two-way approach for a subset of common concepts. The direct introduction of UML concepts (class, association, composite states) makes them not only available for use in an SDL specification, but gives them also a formal semantics (see section 4). Whereas the static semantics corresponds to the meta-model based UML semantics (model elements and well-formedness rules), the dynamic semantics is a formalization of the plain english text given in the UML standard. The UMLSDL profile defined in Z.109 on the other hand reflects the static semantics of SDL-2000 concepts in UML terms like stereotypes and well-formedness rules.

SDL-92 supported structuring of types through containment (nested type definitions), but offered limited support to the usage of types in other relations. SDL-2000 overcomes this limitation by offering an alternative graphical representation for types that shows more information about the type, by adding a graphical symbol for

inheritance and by introducing the association concept. Associations show graphically the relationship between types that in the application may be instantiated or defined in different parts of the hierarchical structure. Using associations, it is possible to visualize complex relationships similar to those existing in the real life, which may be recursive and reflexive. As an example, consider the simple system described below. It contains a telephone operator that owns switchboards and has clients. Each client can be listed in several yellow pages. The relationships between different entities are designed using associations.

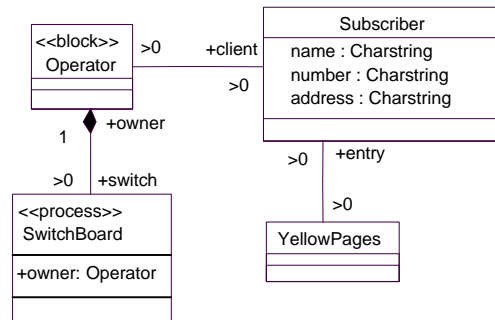


Fig. 2. Example of Associations

While adding new features to SDL the main concern was to keep its most important feature, which is the formality, unaffected. Associations can be used to carry informal and partial information, which is not suitable for formalization; as a consequence, the approach is to keep them as special annotations with no semantics in an SDL system.

2.3 Agents

Agents are the fundamental specification concept of SDL-2000. There are two kinds of agents, namely block agents and process agents. They differ in their concurrency scheme in that block agents have concurrent sub-agents while process agents have interleaving sub-agents. An SDL system is conceptually modeled as a set of asynchronously communicating extended finite state machines (EFSMs). An agent combines both behavior 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 behavior is achieved inside the agent.

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 behavior 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 possible interaction points (*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 interface specification is very similar to an interface in CORBA-IDL. It declares a set of operations (remote procedures) 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

```

Fig. 3. Example of an Interface

Like any other type definition agent types as well as interfaces can be organized in an inheritance hierarchy (interfaces even with multiple inheritance). This allows in a specialization to add new (structural or behavioral) elements or to redefine existing virtual elements.

Exactly one input port is associated with the agent. Signals sent to a 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. SDL-2000 supports three different styles of communication as built-in constructions:

- asynchronous messaging by *signals*
- 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 appropriated matching interfaces (required or supported) and a connection between these gates must exist. A connection is given by an explicit or implicit channel, which provides a reliable, but possibly delaying transmission of communication elements from a sender to a receiver agent.

The behavior of agents is specified by means of extended finite and communicating state machines in form of services. A service is a connected graph consisting of states and transitions. Transitions are triggered by external stimuli (signals, conditions, remote procedure calls). A service is always waiting for a stimulus in a state or performing a transition. During a transition a sequence of actions may be performed, including the sending of signals and the creation of agents. Two main ways to structure the behavior specification are provided by SDL-2000:

- Decomposition into sub-services: The behavior of such a decomposed service is the combination of the behaviors of its sub-services. The execution semantics is alternation on transition base (run-to-completion), i.e. exactly one sub-service in a service decomposition is performing a transition at any point in time.
- Decomposition of a state into sub-states: A large state space may be decomposed into a set of composite states, each of them containing a subset of the original state set as known from Harel's state charts or UML. Each of the composite states is reflecting a major activity component of the original state machine. In the

enclosing state machine, a composite state appears as an ordinary state (i.e. as a state symbol). Transitions applying to a composite state apply to all sub-states of the composite state. A separate state diagram defines the composite state. Also, states may be based on types with all the object-oriented capabilities of SDL.

2.4 Object-oriented data

With 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 in all details object-oriented. The new data type part is similar to 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; for example, an assignment of a value expression to an object variable automatically instantiates a new object, whose state is defined by the value. Object-orientation is introduced for the data with 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. Please note, that 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 checked to be type-safe. Moreover, it includes many features that are important for having support for data definitions based on ASN.1.

3 Application Example

A simple example of the application of SDL-2000 is the specification of a traffic light control system for an intersection of two streets, a highway and a farm road. As long as no cars are present on the farm road, the traffic light on the highway is switched to green and the one at the farm road to red. Sensors on both sides of the farm road detect the presence of cars. This shall result in an interrupt of the highway traffic flow and enable cars on the farm road to enter or cross the highway. The farm road light should remain green as long cars are on the road but not longer than a given time delay. When the highway lights turn green again, they should remain green for at least a given period of time.

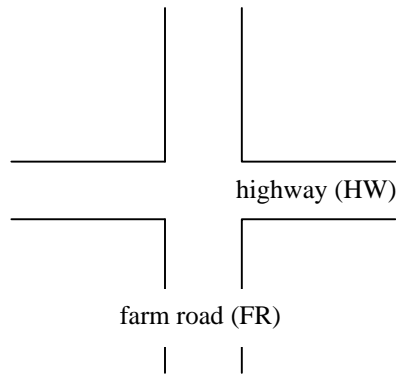


Fig. 4. Traffic Light Controller Example

In order to model this example with SDL the following steps have been performed:

1. Definition of the interfaces between the controller and its environment (traffic lights, sensor)
2. Definition of the internal structure of the controller
3. Definition of the overall behavior of the controller
4. Detailed specification of the behavior.

3.1 Interface definition

An interface defines the set of possible interactions. According to the example description the controller interacts with the traffic lights and with the sensor. Therefore two different interfaces are defined: The interface for the traffic lights contains methods to switch to red and to switch to green. The communication with the sensor is based on a continuously changing value, which is declared as an attribute (car).

```

interface TrafficLight;
    procedure SwitchRed;
    procedure SwitchGreen;
end interface;

interface sensor;
    dcl car Boolean;
end interface;

```

Fig. 5. The Interfaces of the Traffic Light Controller Example

3.2 Structure definition

The traffic light controller is an active object, therefore it is specified by a block agent type (see Figure 6). In order to control the traffic lights on both roads, two connection points (gates) are required: one for highway lights (hw) and one for the farm road lights (fr). The interface required by the traffic lights connected to these gates is in both cases the interface TrafficLight. A third gate serves for the communication with the sensor, the controller requires here an interface of type Sensor. Internally the traffic light controller is not further structured. All communication is handled directly by its service state machine ControlUnit. All gates are connected to the state machine by channels.

Two external constants have been defined in addition:

1. The length of the minimum time period the highway light must stay green (longT)
2. The maximum time the farm road light may remain green (shortT).

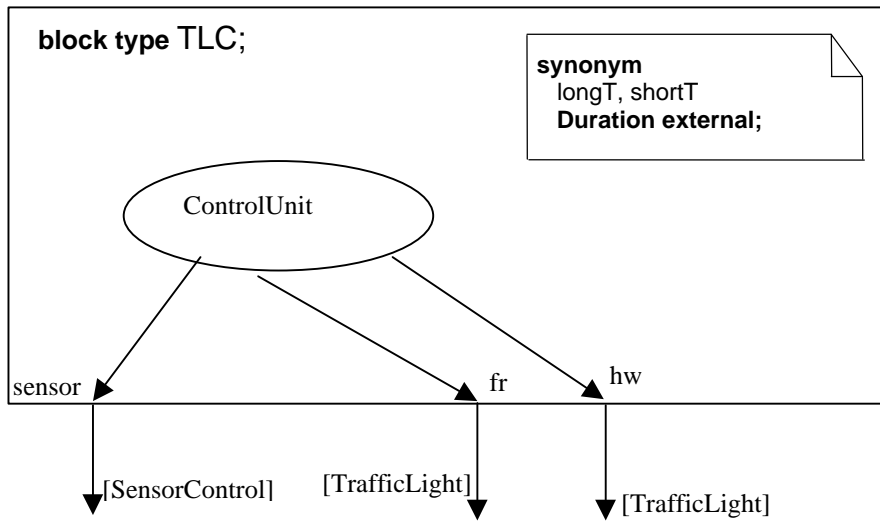


Fig. 6. Traffic Light Controller – Internal connections

3.3 Behavior Definition

The behaviour of the traffic light controller has been specified using composite states. On the top level the state machine consists of two main states, HWEnabled and FEnabled. This reflects the two main traffic flows given in the example. Initially the system starts in the state HWEnabled, what is visualised by the transition from the start symbol to HWEnabled. Two further transitions at the top level describe the switching between both states. The resulting behaviour of ControlUnit is that is either in state HWEnabled or in state FEnabled.

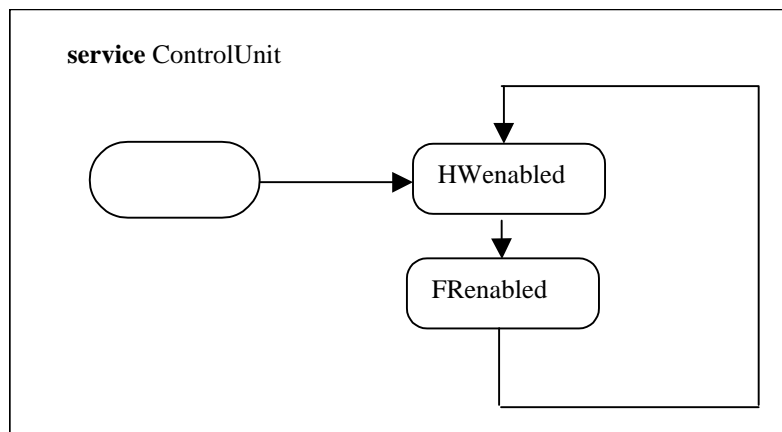


Fig. 7. Traffic Light Controller – Top-level state diagram

The internal structure of both states is then specified in more detail in the next diagrams. First, the behavior of the highway direction is given. Besides its state-transition graph this state has a special procedure called entry. This procedure is always executed when the state is entered. Its purpose is to switch the lights appropriately in

our case, i.e. the highway gets green light and the farmroad gets red light. After this entry procedure, the initial actions of the state (start transition) are performed. In this case, a timer *t* is set for the period which the highway at least should stay green. Another state *NoSwitching* is then entered. In *NoSwitching*, the controller waits until it receives the time-out signal. Then it proceeds to a state *MaySwitch*. Intuitively, the state *MaySwitch* marks the end of the waiting period of the controller and switching the highway to red is now allowed. There is exactly one transition going out from this state. This transition is triggered by the remote variable *car*. After this variable changes to the value *true*, the transition is executed. Subsequently the return is performed and the complete state *HWEnabled* is finished. As a result now also the transition from *HWEnabled* to *FREnabled* (at the top level) is executed and the controller enters the state *FREnabled*.

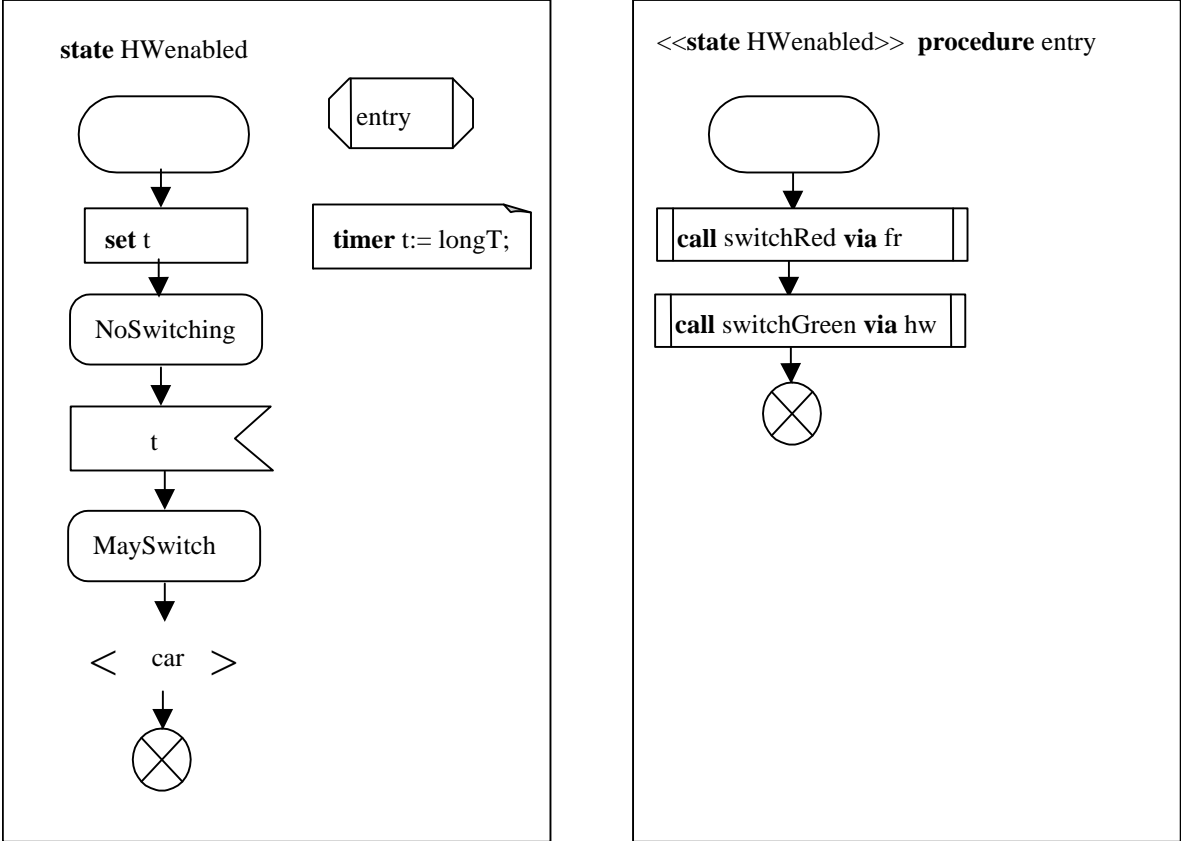


Fig. 8. Traffic Light Controller – Detailed state specification

Similar to *HWEnabled* the first activity on entering *FREnabled* is the execution of the entry procedure. However, in this case of course the freeway lights are set to red and the farmroad lights to green. After that the start transition is enabled. A local timer is set here before the substate *NoSwitching* is entered. This state is either left in case the timer is expired or in case the remote variable has changed its value to *true*.

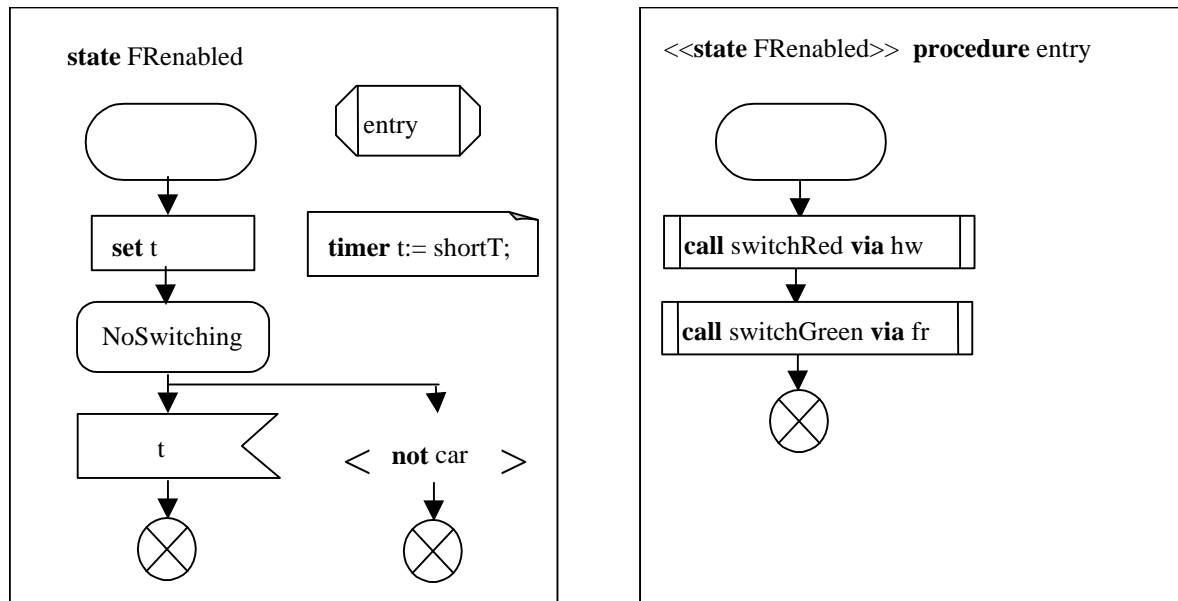


Fig. 9. Traffic Light Controller – Detailed state specification

In both cases a return from the superstate FEnabled is initiated. On the top level the Controller now switches back to the state HWEEnabled.

The detailed specification of the traffic lights and the sensor has been omitted here to limit space. Both would also be specified as block type agents. The type defining the traffic lights has to implement the interface TrafficLight. This implies that it has to export two functions (switchRed and switchGreen). The sensor agent on the other side has to implement the interface Sensor and therefore must export the variable car. As long as a vehicle is on the farm road the variable obtains the value true. Otherwise, if the farmroad is empty, the value will be set to false.

4 Abstraction and Static Semantics

For a language with a rich syntax it is essential to define the semantics only for the core concepts and transform the other constructs into this core. In figure 10, the general approach is shown. The language is defined with its concrete appearance using lexical and grammar rules. Consistency constraints are defined on this concrete grammar.

However, the meaning is given only to the core concepts that are within the Abstract Syntax. In order to be able to do so, a transformation is given mapping the concrete grammar to the abstract grammar. It is important here to correctly identify the core concepts in order to facilitate easy transformation. If there are too many concepts, giving a semantics is unnecessary complicated. If there are too few or the wrong concepts, the transformations tend to be very complex and their meaning is no longer easily understood. In the scope of SDL, object-orientation was introduced in SDL-92, but still the (formal) semantics definition and the abstract syntax relied on instances and had no notion of class. The result was a very cumbersome transformation. This is changed in the SDL-2000 version: the meaning of specialization and virtuality is given directly, rather than by means of “flattening”.

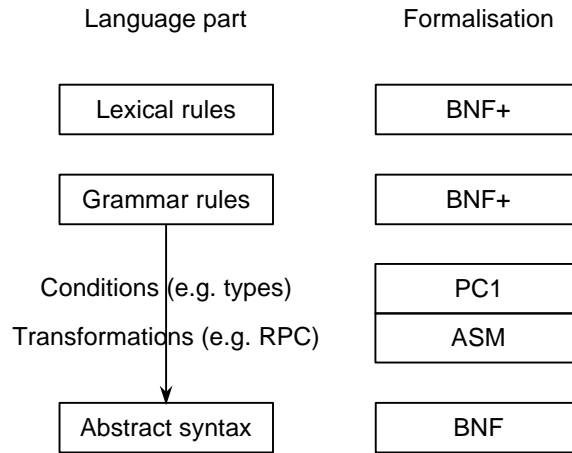


Fig. 10. Static Part of SDL

With a proper and small abstract syntax it is possible to define a formal semantics of the language, see also below. However, in order to have a valid formal semantics, it is also necessary to make the transformations formal. This is achieved using a suitable formalization for the transformation. In the case of SDL, rewrite rules were chosen for the transformation. These rules define patterns of the Abstract Syntax Tree (AST) which are to be replaced by other AST patterns. In fact several groups of such rewrite rules are defined that are applied in turn.

5 Dynamic Semantic Model

The dynamic semantics associates with each SDL specification, represented as an abstract syntax tree (AST), a particular multi-agent real-time ASM. Intuitively, an ASM consists of a set of *autonomous ASM agents* cooperatively performing *concurrent machine runs*. The behavior of ASM agents is determined by *ASM programs*, each consisting of a set of *transition rules*, which define the set of possible runs. Each ASM agent has its own *partial view* on a given global state, on which it fires the rules of its program. According to this view, ASM agents have a *local state* and a *shared state* through which they can interact.

It is pointed out that there are strong structural similarities between SDL systems and ASMs, which is a prerequisite for intelligibility and maintainability. These structural similarities are now exploited by identifying ASM agents with certain SDL units. For instance, ASM agents will be introduced for SDL agents, agent sets, and channel segments. ASM agents can be created during the system initialization phase as well as dynamically, which allows, e.g., to directly represent dynamic process creation in the underlying model.

The execution of a system starts with the creation of a single ASM agent for the SDL unit "system". This ASM agent then creates further ASM agents according to the substructure of the system as defined by the specification, and associates a view and an ASM program with each of them. ASM programs are determined by the kind of the SDL unit modeled by a given ASM agent.

Following an abstract operational view, behavior is expressed in terms of *SDL abstract machine runs*. The formal definition of the dynamic semantics is restricted to specifications that comply with the static semantics of SDL. Previous work on an ASM semantics for SDL (see [5], [6]) provides a conceptual framework which is partly reused here. The dynamic semantics consists of four parts as can be seen in figure 11.

These parts are:

- An SDL Abstract Machine (SAM) which is defined using ASMs (former work in this direction is reported in [13]). In the SAM there are basic features to express structure: ASM agents are enhanced by gates. These gates are the interfaces used by the SDL part. Moreover, connections are predefined here – they will represent SDL channels and other SDL connections later. A last part of the SDL Abstract Machine are behavior primitives, in a way the abstract machine instructions. The SDL abstract machine is independent of a particular SDL specification.
- An initialization which is necessary to handle static structural properties of the specification. The initialization does a recursive unfolding of all the static objects of the specification. In fact, the same process will be initiated at interpretation time also when new SDL agents are created. From this point of view, the initialization is merely the instantiation of the SDL system agent.
- A compilation function that maps behavior representations into the SDL abstract machine primitives. This function amounts to an abstract compiler taking the AST of the state machines as input and transforming it into the abstract machine instructions (see above).
- A data semantics, which is separated from the rest of the semantics by an interface. The use of an interface is intentional at this place. It will allow to exchange the data model, if for some domain another data model is more appropriate than the SDL built-in model. Moreover, also the SDL built-in model can be changed this way without affecting the rest of the semantics.

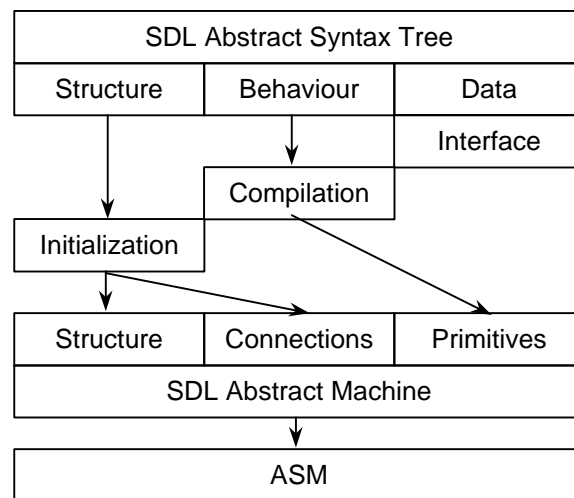


Fig. 11. Dynamic Semantics Overview

As in the past, the new formal semantics is defined starting from the abstract syntax of SDL, which is documented in Z.100. From this abstract syntax, a behavior 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.

The compilation step generates abstract code from a state machine description. Please find the code generated for the state machine of the block type TLC below.

startlabel("TLC") = l_0

```

behaviour("TLC") =
{
  mk-enter(l0, "HwEnabled", l1, l6), /* switch to label l1, and after return go to l6 */
  mk-call(l1, "HwEnabled.entry", l2),
  mk-set(l2, "HwEnabled.t", l3),
  mk-statenode(l3, {mk-input("HwEnabled.t", l4)}),
  mk-statenode(l4, {mk-condition("car", l5)}),
  mk-return(l5)
  mk-enter(l6, "FwEnabled", l7, l0),
  mk-call(l7, "FwEnabled.entry", l8),
  mk-set(l8, "FwEnabled.t", l9),
  mk-statenode(l9, {mk-input("FwEnabled.t", l10), mk-condition("not car", l11)}),
  mk-return(l10)
  mk-return(l11)
}

```

Fig. 12. Simplified Code Generated for the Example

Please note, that the abstract code is in fact a set of primitive instructions. The connections between the instructions is provided using labels. These labels are attached already to the nodes of the AST, and then to the code generated from these AST nodes.

Labels are abstractly represented by a static domain LABEL. A dynamic function *label*, defined on process agents, is used to model dynamic rule selection. That is, *label* identifies for each process agent the rule to be fired in a given machine state.

A special *labeling* of process graph nodes is used to model process specific control-flow information. Intuitively, node labels relate individual operations of an SDL process to transition rules in the resulting SDL machine model. The effect of state transitions of processes is then modeled by firing the related transition rules in an analogous order.

5.1 ASM behaviour

Behavioural properties of ASM agents are expressed in terms of transition rules defined by a finite collection of programs. A program is identified by its program name. A dynamic function *Mod* specifies for each ASM agent the program to be executed in the next step of the ASM.

An SDL Abstract Machine run splits into two basically different phases, an *initialization phase* followed by an *execution phase*. Accordingly, there are two kinds of programs, namely *initialization programs* and *execution programs*.

The ASM agents usually start with an initialization program. After having finished the initialization, they switch to the appropriate execution program. An exception are channel agents that start off immediately with the execution. There are only two kinds of initialization programs, namely agent initialization and agent set initialization. There are three kinds of execution programs, namely link programs, agent set programs and agent programs. Please note, that the execution programs for ASM agents of SDL agents are always the same, because

this is the ASM interpretation program of the abstract code. The difference of the agents is reflected by different behaviors that they get assigned when their state machine is compiled.

The compilation of SDL agents into abstract code handles control flow information through a *labeling* of state transition graph nodes. Resulting node labels relate individual SDL agent operations to transition rules in the machine model. The effect of state transitions of SDL agents is modeled by firing the related transition rules in an analogous order.

Starting from the initial state, the initialization rules describe a recursive *unfolding* of the specified system instance according to its hierarchical structure. Each initialization step may create several object instances simultaneously.

5.2 Support for Object-Oriented

In the scope of the SAM, behavior primitives have been defined in abstract operational terms by introducing a collection of ASM macros (with formal parameters). Building on these patterns, compilation of SDL processes into ASM transition rules essentially deals with parameter instantiation and control-flow aspects. For each occurrence of a behavior item in an SDL agent a corresponding macro and its actual parameters need to be determined. Furthermore, the missing control-flow has to be added ensuring that all operations are executed in the right order.

This special compilation function makes the handling of object orientation very easy. Recall that the abstract code of a specification is a set of behavior primitive applications connected with references to the appropriate labels (this has not been said before). To handle inheritance, it now suffices to get the compilation of the parent and to add the special parts of the child. Sometimes, it is also necessary to delete some elements of the parents behavior, but most of it will be present in the child.

6 Executable Semantics

From the beginning there was the desire to construct the semantics in a way to be readily executable. This came from the history of SDL as a language having many tools that provide code generation for SDL specifications. On the other hand it appeared that the previous formal semantics had deficiencies in exactly that area: it was not executable and hence one could not be sure that the semantics would really match what was intended.

There are several aspects of tool support for the formal language description.

- Usually it is assumed that the description of syntax and lexis of a language would allow easy transformation into a parser of the language. However, the grammar of SDL is intentionally made as a presentation grammar, i.e. it would not be unmistakable in all parts. Moreover, parts of the grammar information, especially some of the precedence rules, are formulated within the accompanying natural language text.
- The correctness of the formal description has to be checked compared to the language rules of the language chosen to formulate the semantics. This involves grammar as well as static correctness conditions, especially typing issues.
- Finally, the best thing to do would be to derive a reference compiler directly from the formal definition. This is done providing tools for all compiler phases.

6.1 An SDL reference compiler

Such a reference compiler could have the usual compiler structure as can be seen in figure 12.

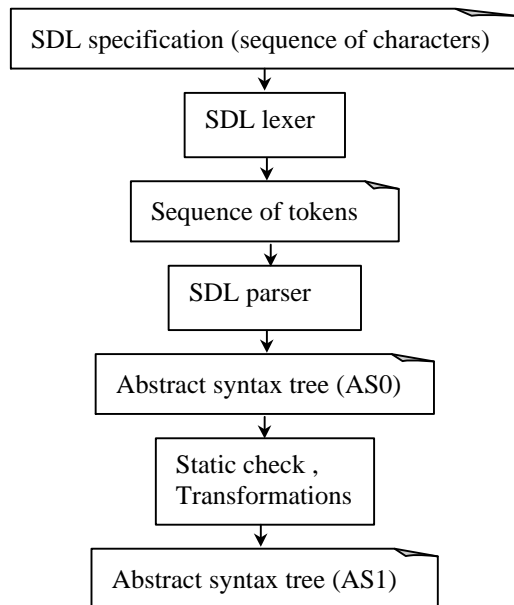


Fig. 13. Structure of the Reference Compiler

It starts with a specification as a sequence of characters. The lexer is able to produce from that a uniquely defined sequence of tokens. The token sequence is the input for the parser, which will analyze it and transform it to an abstract syntax tree which represents the concrete syntax structure (called AS0 in the SDL context). This abstract syntax tree is checked in the static semantics part and transformed into a more abstract tree representation called AS1. The AS1 is the abstract SDL specification. From this the interpretation starts.

At interpretation time we distinguish three parts of the AS, namely state machine representations (called behavior in the figure), structure representations and data representations. All data aspects are represented using functions that are predefined and work on the AS1 of the data. These do not need to be changed.

The state machines are compiled into abstract code which can then be interpreted by the ASM workbench. The structural aspects do only come into play at system initialization time and are therefore faithfully handled by an initialization function that works over the abstract syntax. When the initialization phase is finished, no reference to the abstract syntax is needed any more, only the abstract code for the machines and the abstract run-time representation of the data types.

The compilation part, initialization part and the data part are represented exactly that way in the SDL formal definition. It therefore suffices to extract those parts and to slightly adapt the syntax to make them fit into the ASM workbench.

It is far more difficult to handle the other parts.

The lexer is the easiest of the rest. It is represented as BNF and can be relatively easily transformed into lex format. In this transformation some side conditions have to be considered, which result in some predefined parts of the lex specification. Moreover, SDL has quite complicated rules about the handling of control characters which make it necessary to introduce a pre-processing step that handles those characters.

The next step is parsing. As already pointed out, the SDL grammar is a presentation grammar and not usable for e.g. generation of yacc input from it. So it has to be manually extended and changed to be free of conflicts. There is an abstract syntax AS0 given in the formal model of SDL which can be used as a guideline for the manual process. Moreover, it can be checked whether the presentation grammar matches the processing grammar. To make this process as exact as possible, a checking feature is provided that checks these both grammars against each other.

From the formal AS0 definition an abstract tree representation is generated. This is used to make yacc generate such a tree.

Starting from AS0, the static constraints are given using Boolean-valued functions, that operate on the tree. These are easily generated from the formal definition. Moreover, the transformations from AS0 to AS1 are intentionally given as rewrite rules such that as implementation as a rewrite system does not make problems.

7 Conclusions

In this paper, we presented SDL as a modern implementation-oriented system description language. Using an example we showed its suitability for the design of open distributed systems.

The executability of SDL does not only provide means for efficient implementations, but also for simulation in all stages of the design.

It is noted, that these new features of SDL do not harm its power of being a formal technique in having a formal semantics. The formal semantics of SDL is explained in an overview, taking especially care of those features of the formal semantics that make it easily executable.

The following benefits arise from this special style of defining the semantics:

- The SDL language designers get fast response about the semantics they have defined.
- Tool builders have a reference when semantic questions occur.
- The reference implementation itself might be a starting point for a real implementation.

With the UML-SDL alignment the formal semantics of SDL-2000 covers already parts of the UML semantics. Starting from this common subset of concepts a full formalization of the dynamic semantics of UML in a similar way as it has been done for SDL-2000 is possible.

Last, but not least, the approval of SDL-2000 as an international standard makes it possible to base upcoming standards for protocols and architectures in the area of distributed systems on SDL. Furthermore, it can be expected, that SDL tools use the same semantics of the language.

References

- [1] E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64:105--127, February 1998.
- [2] Egon Börger. Why use evolving algebras for hardware and software engineering? In *Proc. of SOFSEM'95*, volume 1012 of *LNCS*, pages 236--271. Springer-Verlag, 1995.

- [3] Egon Börger. High level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, LNCS, (to appear). Springer-Verlag, 1999.
- [4] Giuseppe Del Castillo. ASM-SL, a Specification Language based on Gurevich's Abstract State Machines: Introduction and Tutorial. Technical report, Department of Mathematics and Computer Science, Paderborn University. Technical Report (to appear).
- [5] U. Glässer. ASM semantics of SDL: Concepts, methods, tools. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (Berlin, June 29 - July 1, 1998)*, volume 2, pages 271-280, 1998.
- [6] U. Glässer and R. Karges. Abstract State Machine semantics of SDL. *Journal of Universal Computer Science*, 3(12):1382--1414, 1997.
- [7] R. Gotzhein, B. Geppert, F. Röbler, and P. Schaible. Towards a new formal SDL semantics. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (Berlin, June 29 - July 1, 1998)*, volume 1, pages 55-64, 1998.
- [8] Y. Gurevich and J. Huggins. The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions. In *Proceedings of CSL'95 (Computer Science Logic)*, volume 1092 of LNCS, pages 266--290. Springer, 1996.
- [9] Y. Gurevich and M. Spielmann. Recursive Abstract State Machines. *Journal of Universal Computer Science*, 3(4):233--246, 1997.
- [10] Yuri Gurevich. Evolving algebras - a tutorial introduction. *Bulletin of the EATCS*, (43):264--284, 1991.
- [11] Yuri Gurevich. Evolving Algebra 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9--36. Oxford University Press, 1995.
- [12] Yuri Gurevich. ASM Guide 97. CSE Technical Report CSE-TR-336-97, EECS Department, University of Michigan-Ann Arbor, 1997.
- [13] St. Lau and A. Prinz. BSDL: The Language -- Version 0.2. Department of Computer Science, Humboldt University Berlin, August 1995.
- [14] The CoFI Task Group on Language Design. CASL - The Common Algebraic Specification Language Technical report, BRICS, 1998. <http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/>.
- [15] ITU-T Recommendation Z.100. *Specification and Description Language (SDL)*. International Telecommunication Union (ITU), Geneva, 1994 + Addendum 1996.
- [16] U. Glässer, R. Gotzhein, A. Prinz: Towards a New Formal SDL Semantics based on Abstract State Machines, in R. Dssouli, G.v.Bochmann, Y. Lahav: *SDL'99: The Next Millenium*, Elsevier Science B.V., Amsterdam, 1999.
- [17] J. Ellsberger, D. Hogrefe, A. Sarma: *SDL – Formal Object-oriented Language for Communicating Systems*. Prentice Hall, 1997.
- [18] A. Olsen, O. Færgemand, B. MøllerPedersen, R. Reed, J.R.W. Smith: *Systems Engineering Using SDL-92*, Elsevier Science B.V., 1994.
- [19] N. Fischbeck, M. Born, A. Hoffmann, M. Winkler, G. Baudis, H. Böhme, J. Fischer: *SDL Enhancements and Application for the Design of Distributed Systems*, in R. Dssouli, G.v.Bochmann, Y. Lahav: *SDL'99: The Next Millenium*, Elsevier Science B.V., Amsterdam, 1999.

