An Executable Formal Semantics for SDL-2000

Andreas Prinz[1], Robert Eschbach[2], Reinhard Gotzhein[3]

[1]Humboldt-University Berlin, Department of Computer Science, Rudower Chaussee 25, D-12489 Berlin, Germany
prinz@informatik.hu-berlin.de
[2]Foundations of Computing Science Group, University of Kaiserslautern, Dept. of Computing Science,
D-67653 Kaiserslautern, Germany
eschbach@informatik.uni-kl.de
[3]Computer Networks Group, University of Kaiserslautern, Dept. of Computing Science,
D-67653 Kaiserslautern, Germany
gotzhein@informatik.uni-kl.de

Abstract. A new version of the Specification and Description Language (SDL) called SDL-2000 was approved as international standard in November 1999. SDL-2000 offers important new features, including object-oriented data type definitions, a unified concept for blocks, processes and services, composite states, and exception handling. To capture the meaning of SDL-2000 formally, a new formal semantics has been devised and is currently being worked out in detail. Design objectives are intelligibility, maintainability, conciseness, and in particular executability, which has become a key issue. Executability calls for an operational formalism with readily available tool support. For this reason, Abstract State Machines (ASMs) have been selected as the underlying formalism. To support executability, the formal semantics defines an SDL Abstract Machine (SAM) and, for each SDL specification, code running on that machine. This differs substantially from the interpreter view taken in previous work, and enables SDL-to-ASM-compilers. Thus, given the availability of an SDL-to-ASM-compiler, it will be straightforward to generate reference implementations. This paper focuses on the execution-related aspects of the new SDL semantics, and introduces appropriate compiler technology.

Keywords: semantic foundations, executability, compiler, SDL

1 Introduction

SDL (Specification and Description Language) is an ITU standardized language for the description of distributed systems. Along with the work on SDL-2000, it has become apparent that a new SDL semantics has to be defined, superseding the former semantics which dates back to SDL-88. In several meetings of the SDL language definition group (ITU-T Study Group 10, Question 6: SG10/Q6), 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 (*traceability*), 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. Because of the language extensions and modifications of SDL-2000, the semantic model has to be sufficiently rich and flexible.

During the discussions within ITU-T SG10/Q6, 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, and a suitable style for defining the semantic model. Subsequent investigations have shown that Abstract State Machines (ASMs, cf. [1], [6], [7]) meet this and all other design objectives, and therefore have been chosen as the underlying formalism.

The interpreter defining the semantics of SDL-88 and SDL-92 is written in Meta IV, and consists of two parts. Starting point is an SDL specification in an abstract syntax called AS0. The translation step from the concrete textual SDL syntax to AS0 is not formally defined, but is derived from the correspondence between names in the two syntaxes.[1] In the first step, the AS0 representation is checked for well-formedness conditions, also referred to as *static semantics*, and is transformed into a representation in another abstract syntax called AS1 by replacing several SDL language constructs. In the second step, the AS1 representation is interpreted, i.e., used as a data structure that determines the set of computations.

[1] However, this transformation is almost one-to-one, such that no further formality is needed here.

While this operational approach based on an interpreter is feasible and appealing in general, there have been three problems in this particular case:

- At no point in time has there been tool support for executing Meta IV programs, although type checks have been performed.
- The interpreter has not been written in pure Meta IV, but makes use of CSP-like communication. Although it should be straightforward to define a precise semantics for this variant of Meta IV, this has not been done so far.
- There are several places where SDL systems depend on environmental conditions, such as channel delays or spontaneous transitions. These places are rather hidden in the Meta IV interpreter.

For the SDL-2000 formal semantics, the approach was changed. In practice, processing specifications into an abstract syntax representation is simpler and faster than compiling it into some kind of executable code. On the other hand, compilation into code directly enables executability. There is one well-known technique combining the benefits of interpretation and of compilation, which is the technique of *abstract code*. With this technique, a specification is transformed into an abstract code representation that is tailored to the needs of the language and also fairly low-level. More specifically, the new formal SDL semantics defines the ASM code that an SDL-to-ASM-compiler is supposed to generate, thus enabling the application of compiler techniques. Such a compiler is currently being devised. Due to the nature of ASMs and the semantics specification, the compiler generated from the formal semantics is intended to be a *reference compiler*. This means, the compiler is correct, because it reflects the formal semantics, but it is not intended to be particularly effective. Note that once an SDL-to-ASM-compiler is available, it should be feasible to adapt it to another target language by modifying just the code generation phase.

The paper is structured as follows: In Section 2, the formal semantics of SDL is surveyed. Section 3 is devoted to present a basic implementation methodology, which is then used for the implementation of the SDL formal semantics. Conclusions are drawn in Section 4.

2 Formal Semantics for SDL-2000: Overview

In order to define the formal semantics of SDL, the language definition is decomposed into several parts (see also Figure 1):

- grammar
- well-formedness conditions
- transformation rules, and
- dynamic semantics based on an abstract syntax.

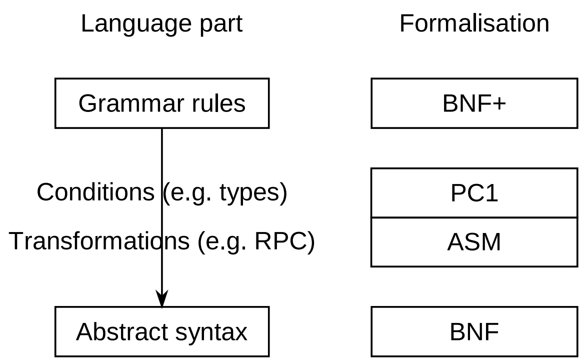

Figure 1. Static Semantics of SDL

The first three parts of the formal semantics are collectively referred to as *static semantics* in the context of SDL.

The *grammar* defines the set of syntactically correct SDL specifications. In Z.100, a concrete textual, a concrete graphical, and an abstract grammar are defined formally using the Backus-Naur-Form (BNF) with some extensions for capturing the graphical part. The abstract grammar is obtained from the concrete grammars by removing irrelevant details such as separators and lexical rules.

The *well-formedness* conditions (static semantics) define which specifications that are correct with respect to the grammar are also correct with respect to context information, such as which names it is allowed to use at a given place, which kind of values it is allowed to assign to variables etc.

Furthermore, some language constructs appearing in the concrete grammars (shorthand notations) are replaced by other language elements in the abstract grammar using *transformation rules* to obtain a normal form, thereby using a set of semantic core concepts only. These transformations are described in the model paragraphs of Z.100. Formally they are represented as rewrite rules.

The *dynamic semantics* is given only to syntactically correct SDL specifications that satisfy the well-formedness conditions. The dynamic semantics defines the set of computations associated with a specification.

2.1 Grammar

Formalizing a grammar is a well-known task. As usual, this is done in SDL using a grammar representation in BNF (Backus-Naur-Form). However, the grammar in [9] is designed to be a presentation grammar, i.e. it is not made to generate a parser automatically. Moreover, some restrictions that finally guarantee uniqueness of the semantics could not be stated formally in BNF and have been stated in the text instead. Therefore, the grammar is defined using BNF and some text (mostly for the precedence rules).

2.2 Well-formedness Conditions

There are some static conditions that must be satisfied by a well-formed SDL specification and that can be checked without interpretation of an instance. An SDL specification is valid if and only if it satisfies the syntactic rules and the static conditions of SDL. In fact, the static conditions refer to the grammar, but for conciseness reasons they have not been stated in the grammar because they are not expressible in a context free grammar.

There are basically five kinds of well-formedness conditions:

1. *Scope/visibility rules*: the definition of an entity introduces an identifier that may be used as the reference to the entity. Only visible identifiers may be used. The scope/visibility rules are applied to determine whether the corresponding definition of an identifier is visible or not.
2. *Disambiguation rules*: Sometimes a name might refer to several identifiers. Rules are applied to find out the correct one.

3. *Data type consistency rules*: these rules guarantee that at interpretation time no operation is applied to operands that do not match their argument types. More specifically, the data type of an actual parameter must be compatible with that of the corresponding formal parameter; the data type of an expression must be compatible with that of the variable to which the expression is assigned.
4. *Special rules*: there are some rules applicable to specific entities. For example, there must exist at least one block definition or a process definition in a system definition.
5. *Plain syntax rules*: There are some rules that refer to the correctness of the plain syntax and that do not get transformed into the abstract syntax, e.g. that the name at the beginning and at the end of a definition match.

The static semantics of SDL is defined in terms of first order predicate calculus.

2.3 Transformation Rules

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 1, the general approach is shown. The language is defined with its concrete grammar 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 surface grammar to the abstract grammar. It is important here to correctly identify the core concepts in order to facilitate transformation. If there are too many concepts, giving a semantics is unnecessarily 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. For instance, 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, which required a very cumbersome transformation. This is improved in the SDL-2000 version.

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. In the case of SDL, term rewriting was chosen to formalize the transformation. A single transformation is realized by the application of a rewrite rule to the AST, which essentially means to replace parts of the AST by new parts as defined by the rule. Moreover, the SDL informal semantics defines transformation steps that are given by sets of rewrite rules. Each of those steps defines how to handle one special class of shorthand notations. The result of one step is used as input for the next step. A suitable reduction ordering within each step ensures the termination of the underlying reduction relation, i.e. transformation always terminates with a completely reduced AST. A reduction strategy (leftmost-innermost) guarantees that transforming the AST always leads to a unique completely reduced AST.

2.4 Dynamic Semantics

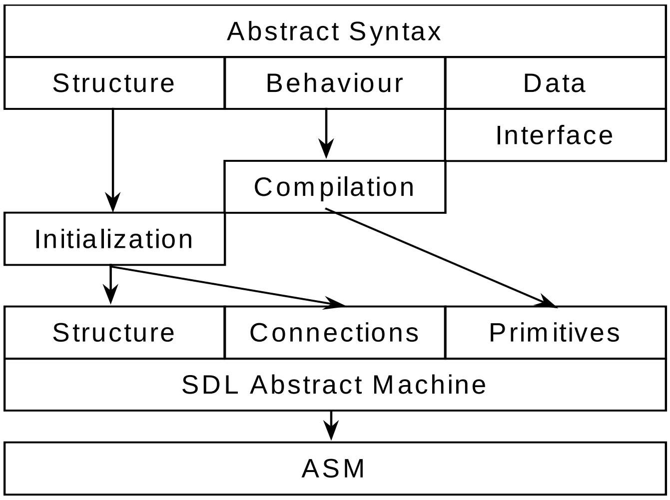

Figure 2. Structure of the Dynamic Semantics

The dynamic semantics consists of four parts as shown in Figure 2.

- An *SDL Abstract Machine (SAM)* which is a special multi-agent, real-time ASM (cf. [5]) tailored to the needs of SDL. In the SAM there are basic features to express *structure* in that ASM agents are enhanced by gates that are the interfaces to use for the SDL part. Moreover, *connections* are predefined here – they will represent SDL channels and other (implicit) SDL connections later. A last part of the SDL Abstract Machine are *behavior primitives*, in a way the abstract machine instructions of the SAM. Early work on the SAM can be found in [8].

- An *initialization* which is necessary to handle static structural properties of the specification. The initialization recursively unfolds all initial objects of the specified system. In fact, the same kind of unfolding occurs during execution whenever SDL agents are created. From this point of view, the initialization is merely the creation of the SDL system agent.
- A *compilation* function that maps behavior representations to SAM 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.

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 is derived. The approach chosen here has the characteristics of all three basic semantics definition styles. The SAM is an extension of ASM, and both of them have a mathematical definition (axiomatic semantics). The compilation defines an abstract compiler mapping the behavior parts of SDL to abstract code (denotational semantics). Finally, the initialization describes an interpretation of the abstract syntax tree to build the initial system structure (operational semantics). Furthermore, the new Annex F includes the static semantics constraints and the data model.

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 behavior of agents is determined by ASM programs, each consisting of a transition rule, which defines the set of possible runs. Each agent has its own partial view on a global state, which is defined by a set of static and dynamic functions and domains. By having non-empty intersections of partial views, interaction among agents can be modeled.

3 Implementation of the Formal Semantics

Right from the beginning, there was the desire to implement the formal semantics, i.e. to produce an executable version of the semantics. This is easily understandable when one takes into account that the complete formal semantics definition will span more than 200 pages. Despite all efforts to have a good structuring and an intelligible presentation of the semantics, it is still a substantial effort to understand the semantics in every single detail. Furthermore, the executable version can be used for the validation of the formal semantics against the informal language definition in recommendation Z.100 [9]. This can be done, for example, by means of simulations and evaluation of test cases.

This was taken into account when designing the formal semantics. The following characteristics make an implementation possible.

- ASMs are an operational technique: the basic state change primitive defined in ASMs is an update, which is mostly the same as an assignment. All the other constructors are restricted to finite sets of updates.
- There is tool support for ASMs: there are actually some implementations for ASMs, which are now available. We use the ASM workbench from Paderborn [2]. A close contact to the developers of the ASM workbench ensures that the SDL semantics can be processed by this tool.
- A special style of using ASMs is applied for the definition of the SDL semantics: although ASMs are executable in principle, there are some instructions that could cause problems. Especially sets can be constructed using all the features of first order predicate calculus. In the SDL semantics, these pitfalls have been avoided by ensuring that all set constructions are computable, i.e. they use only computable functions over finite domains.

To make the implementation feasible, existing tools are used as far as possible. The formal semantics is implemented according to the technology given in the next section.

3.1 Implementation Technology Based on Abstract Syntax Trees

The tooling for the formal semantics implementation is based on an abstract syntax tree representation of the input. It has been policy from the beginning to use readily available tools to assist in making an implementation of the semantics. So the compiler-related parts are handled with the standard tools `lex` and `yacc`. To go on after the parsing, the back end tools work over an abstract syntax tree representation of the input. This representation is generated by the tool `kimwitu` [10].

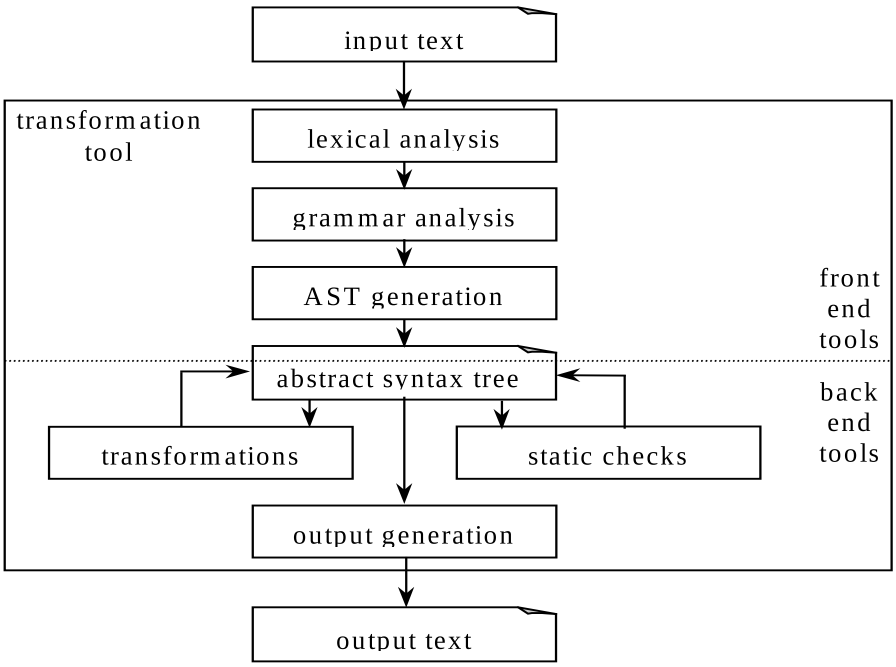

Figure 3. Overview of the Implementation Technology

The compiler technology with `kimwitu` is centered around the abstract representation of the input (see Figure 3 for an overview of the technology). The input is first analyzed using compiler tools (`lex` and `yacc` in our case), and then an abstract representation is generated using `kimwitu`. In fact, `kimwitu` just generates the constructor functions for building the AST; these functions are called from `yacc`.

Once the AST is constructed, the back end tools can work on it. There are several tasks for back end tools. These tools may have to check the AST. For doing this, they may check for the appearance of some special patterns and handle them. They may also check names for being defined. All of these tasks can be handled by `kimwitu` *unparse rules*. Unparse rules allow to specify what is to be done in special situations in the tree, with special situations being defined using patterns over the tree definition. Moreover, `kimwitu` allows the definition of functions over the tree in a simple way. This makes it possible to implement static checks: one only has to define what the static check of one entity is with respect to its sub entities within the tree.

Another back end task is to make certain transformations on the tree in order to bring it into "normal form". Some constructs might only be abbreviations of other constructs, and are transformed away. This is possible using the `kimwitu` feature of *rewrite rules*. Here it is possible to define rewrite rules for the transformation of constructs. The simple form of rewrite rules is to just map some tree pattern to another tree pattern. However, sometimes it is necessary to describe more complex transformations, that involve e.g. more than one place in the tree or that compute some primitive elements out of other elements. For this case it is possible with `kimwitu` to define transformation rules that resort the arbitrary programming language functions.

Finally, if the input is analyzed and transformed into normal form, one would like to process it or to generate code out of it. This can be achieved using the `kimwitu` unparse rules again. Here it is possible to define in a very simple way how to generate strings from the tree nodes. This can be used to generate code or to even generate another tree from the original one.

3.2 Implementing the Static Formal Semantics

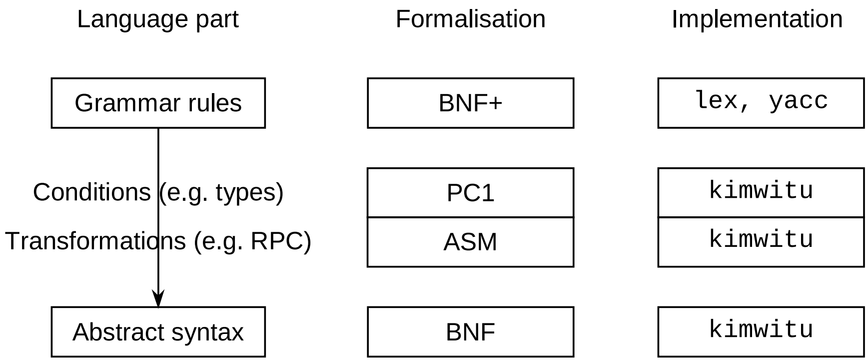

Figure 4. Implementation of the Static Semantics

Please find in Figure 4 an overview of the implementation of the static aspects of the formal semantics of SDL. The formal aspects of the lexis, the grammar, and the abstract grammar are already defined within the main part of the SDL language description. These are implemented using `lex`, `yacc` and `kimwitu`. As already noted, the grammar in Z.100 is a presentation grammar and not immediately usable for an automatic parser generator. Instead, it has to be made unique before, which is done manually. Afterwards, the consistency with the original grammar is checked.

After this step, an abstract syntax tree representation of the input is available. Now the static semantics part starts. As already said, the static semantics conditions are defined as predicate calculus formulae over the AST. These formulae are implemented as Boolean functions over the tree. Then the first step in the analysis of the AST is to call all the static condition formulae on all the places where they are applicable. If any of these functions yields the value *False*, the specification is statically ill-formed.

In the case of a well-formed specification, the next step is to transform the input into an abstract syntax tree of the abstract SDL grammar. For this, several transformations are necessary, that are all defined within the formal SDL definition as rewrite rules. Of course, these rewrite rules are then transformed into rewrite rules of `kimwitu` and performed within `kimwitu`. Please note, that in fact there are several sets of rewrite rules to be performed one after the other. This is implemented in `kimwitu` with several rewrite rule sets which are activated in turn.

The result of this transformation is an abstract syntax tree of the abstract SDL grammar. The SDL dynamic semantics is defined over this tree and the static semantics part is finished here.

3.3 Implementing the Dynamic Formal Semantics

The dynamic formal semantics consists of four different parts as explained in the previous section. The implementation of each of those is discussed in turn.

SDL Abstract Machine The SDL Abstract Machine consists of a number of ASM declarations and definitions. These have to be extracted from the formal definition and then to be transformed into a format for a tool. We use the ASM workbench here for the implementation.

The transformation of the ASM text into input for the ASM workbench is again done using the implementation methodology as explained in Section 3. This means, first an abstract syntax tree of the ASM definitions is generated and then the input for the ASM workbench is generated from that using `kimwitu` unparse rules. Please note that these parts need to be generated only once, as they are independent of any particular specification.

Compilation The compilation is defined as a function over the abstract syntax tree for SDL behavior, generating behavior of the SDL Abstract Machine. It is straightforward to define a matching function for the implementation, as this means to define a function on the abstract syntax tree, and this is well supported by `kimwitu`. The SAM behavior representation that is generated from this function must have an ASM format. This is achieved by unparsing the abstract tree.

Initialization The initialization consists of two parts. There is a static part of the initialization, namely the initialization programs for the SDL agents and for the SDL agent sets as well as the pre-initial system state. These are handled like the SAM parts, i.e. they are transformed into the ASM workbench.

The second part is the abstract syntax tree of the SDL specification. This tree has to be transformed into the ASM initial conditions. This is done using the unparsing features of `kimwitu`, i.e., it is then another output format of the tree.

Data Semantics The semantics of the data part involves a special data interface. This interface is designed such that it only provides functional aspects of the data. All dynamic aspects are already covered by the compilation step. Therefore, the data semantics is handled as part of the SAM.

3.4 Current Status

This paper presents an ongoing project (see also [3], [4], [11], [12], and [13]). With the SDL-2000 language being fixed since November 1999, its formal semantics is expected to be ready by November 2000. Currently, there exists a draft version of the dynamic semantics, with only very few issues unresolved. The implementation already works for small examples of a restricted SDL, proving the feasibility of the approach presented in this paper.

4 Conclusions

In this paper, we have applied ASMs to define the formal semantics of SDL. ASMs meet the design objectives of the formal SDL semantics and have therefore been chosen as the underlying formalism.

A particularity of the dynamic semantics is the flexible use of different techniques (interpretation, compilation, direct definition), which – in conjunction with ASMs – yields an extremely concise behavior model. The semantics is written in a style that enables SDL-to-ASM compilers. Existing ASM-tools can be used to execute the semantic model of any given SDL specification.

The SDL-to-ASM compiler is derived from the formal semantics using standard compiler technology. This allows to extract the relevant parts from the formal semantics and to transform it into an appropriate implementation language. The use of meta tools (`lex`, `yacc`, `kimwitu`) makes the implementation easy.

There are two stages for the application of the compiler technology.

- Transformation of the formal SDL semantics into a real compiler is done using several small compilers from the extracted formal definition into the input formats of the tools.
- The SDL compiler generated does also work according to the same principle.

The use of ASMs for the semantics definition together with the chosen compiler technology makes the semantics executable.

References

[1] E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64:105-127, February 1998

[2] G. 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)

[3] 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

[4] R. Gotzhein, B. Geppert, F. Rößler, 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

[5] 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

[6] Y. Gurevich. Evolving Algebra 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9-36. Oxford University Press, 1995

[7] Y. Gurevich. ASM Guide 97. CSE Technical Report CSE-TR-336-97, EECS Department, University of Michigan-Ann Arbor, 1997

[8] S. Lau and A. Prinz. BSDL: The Language - Version 0.2. Department of Computer Science, Humboldt University Berlin, August 1995

[9] ITU-T Recommendation Z.100. *Specification and Description Language (SDL)*. International Telecommunication Union (ITU), Geneva, 1999
[10] P. van Eijk, A. Belinfante: The term processor Kimwitu - Manual and cookbook (version 4.0). Technical report, University of Twente, Enschede, Netherlands, 1996
[11] 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 Millennium, Elsevier Science B.V., Amsterdam, 1999.
[12] A. Bo, R. Eschbach, U. Glässer, R. Gotzhein, A. Prinz, W. Ying, Z. Yuhong, Z. Weilei. SDL Formal Semantics, Technical Report, Beijing University of Posts & Telecommunications, Beijing China, November, 1999
[13] R. Eschbach, R. Gotzhein, A. Prinz. On the Formal Semantics of SDL-2000: A Compilation Approach using Abstract State Machines. In Proc. of the International Workshop on Abstract State Machines ASM2000, Monte Verita, Switzerland, March 2000