International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.100

Annex F1

(11/2018)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

Formal description techniques (FDT) — Specification and Description Language (SDL)

Specification and Description Language – Overview of SDL-2010

SDL-2010 formal definition: General overview

Recommendation Annex T Z.100 — Annex F1

Recommendation Annex ITU-T Z.100

Specification and Description Language – Overview of SDL-2010

Summary

Annex F1 provides the motivation for and the main objectives of a formal semantics definition for SDL-2010. It gives an overview of the structure of the formal semantics, and it also contains an introduction to the Abstract State Machine (ASM) formalism, which is used to define the SDL-2010 semantics.

History

Edition	Recommendation	Approval	Study Group	Unique ID[a]
1.0	ITU-T Z.100	1984-10-19		11.1002/1000/2222
1.1	ITU-T Z.100 Annex A	1984-10-19		11.1002/1000/6664
1.2	ITU-T Z.100 Annex B	1984-10-19		11.1002/1000/6665
1.3	ITU-T Z.100 Annex C1	1984-10-19		11.1002/1000/6666
1.4	ITU-T Z.100 Annex C2	1984-10-19		11.1002/1000/6667
1.5	ITU-T Z.100 Annex D	1984-10-19		11.1002/1000/6668
2.0	ITU-T Z.100	1987-09-30	X	11.1002/1000/10954
2.1	ITU-T Z.100 Annex A	1988-11-25		11.1002/1000/6669
2.2	ITU-T Z.100 Annex B	1988-11-25		11.1002/1000/6670
2.3	ITU-T Z.100 Annex C1	1988-11-25		11.1002/1000/6671
2.4	ITU-T Z.100 Annex C2	1988-11-25		11.1002/1000/6672
2.5	ITU-T Z.100 Annex D	1988-11-25	X	11.1002/1000/3646
2.6	ITU-T Z.100 Annex E	1988-11-25		11.1002/1000/6673
2.7	ITU-T Z.100 Annex F1	1988-11-25	X	11.1002/1000/3647
2.8	ITU-T Z.100 Annex F2	1988-11-25	X	11.1002/1000/3648
2.9	ITU-T Z.100 Annex F3	1988-11-25	X	11.1002/1000/3649
3.0	ITU-T Z.100	1988-11-25		11.1002/1000/3153
3.1	ITU-T Z.100 Annex C	1993-03-12	X	11.1002/1000/3155
3.2	ITU-T Z.100 Annex D	1993-03-12	X	11.1002/1000/3156
3.3	ITU-T Z.100 Annex F1	1993-03-12	X	11.1002/1000/3157
3.4	ITU-T Z.100 Annex F2	1993-03-12	X	11.1002/1000/3158
3.5	ITU-T Z.100 Annex F3	1993-03-12	X	11.1002/1000/3159
3.6	ITU-T Z.100 App. I	1993-03-12	X	11.1002/1000/3160
3.7	ITU-T Z.100 App. II	1993-03-12	X	11.1002/1000/3161
4.0	ITU-T Z.100	1993-03-12	X	11.1002/1000/3154
4.1	ITU-T Z.100 (1993) Add. 1	1996-10-18	10	11.1002/1000/3917
5.0	ITU-T Z.100	1999-11-19	10	11.1002/1000/4764
5.1	ITU-T Z.100 (1999) Cor. 1	2001-10-29	17	11.1002/1000/5567
6.0	ITU-T Z.100	2002-08-06	17	11.1002/1000/6029
6.1	ITU-T Z.100 (2002) Amd. 1	2003-10-29	17	11.1002/1000/7091
6.2	ITU-T Z.100 (2002) Cor. 1	2004-08-29	17	11.1002/1000/356

7.0	ITU-T Z.100	2007-11-13	17	11.1002/1000/9262
8.0	ITU-T Z.100	2011-12-22	17	11.1002/1000/11387
8.1	ITU-T Z.100 Annex F1	2000-11-24	10	11.1002/1000/5239
8.2	ITU-T Z.100 Annex F2	2000-11-24	10	11.1002/1000/5576
8.3	ITU-T Z.100 Annex F3	2000-11-24	10	11.1002/1000/5577
8.4	ITU-T Z.100 Annex F1	2015-01-13	17	11.1002/1000/12354
8.5	ITU-T Z.100 Annex F2	2015-01-13	17	11.1002/1000/12355
8.6	ITU-T Z.100 Annex F3	2015-01-13	17	11.1002/1000/12356
9.0	ITU-T Z.100	2016-04-29	17	11.1002/1000/12846
9.1	ITU-T Z.100 Annex F1	2016-10-29	17	11.1002/1000/13040
9.2	ITU-T Z.100 Annex F2	2016-10-29	17	11.1002/1000/13041
9.3	ITU-T Z.100 Annex F3	2016-10-29	17	11.1002/1000/13042
9.4	ITU-T Z.100 Annex F1	2018-11-13	17	11.1002/1000/13732
9.5	ITU-T Z.100 Annex F2	2018-11-13	17	11.1002/1000/13733
9.6	ITU-T Z.100 Annex F3	2018-11-13	17	11.1002/1000/13734

[a] To access the Recommendation, type the URL http://handle.itu.int/ in the address field of your web browser, followed by the Recommendation's unique ID. For example, http://handle.itu.int/11.1002/1000/11830-en.

Keywords

abstract state machines, ASM, formal definition, overview, overview of semantics, SDL-2010, specification and description language.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications , information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1 .

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency .

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party .

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at http://www.itu.int/ITU-T/ipr/.

© ITU 2018

All rights reserved. No part of this publication may be reproduced, by any means whatsoever, without the prior written permission of ITU.

Table of Contents

Page

Recommendation Annex ITU-T Z.100

Specification and Description Language – Overview of SDL-2010

Annex F1

SDL 2010 formal definition: General overview

(This annex forms an integral part of this Recommendation.)

F1.1. Preface

The formal definition of SDL-2010 provided in this annex is a precise language definition, which supplements the definition given in the Recommendation text. It is for use by those requiring a very precise definition of SDL-2010, such as maintainers of the SDL-2010 language, designers of SDL-2010 tools and users of the SDL-2010 language.

The formal definition consists of three annexes:

Annex F1 This annex provides the motivation for and the main objectives of a formal semantics definition for SDL-2010. It gives an overview of the structure of the formal semantics, and contains an introduction to the Abstract State Machine (ASM) formalism, which is used to define the SDL-2010 semantics.

Annex F2 This annex describes the static semantic constraints of SDL-2010, as well as the transformations identified by the 'Model' sections of the ITU-T Z.100 series for SDL-2010.

Annex F3 This annex defines the dynamic semantics of SDL-2010.

F1.1.1. Motivation

SDL-2010 has both a formal syntax and a formal semantics. Annexes F1, F2 and F3 define the formal semantics of SDL-2010. If there is any inconsistency between Annexes F1, F2 and F3 and other parts of the ITU-T Z.100 series for SDL-2010, then there is an error that needs correcting. Neither the other parts of the ITU-T Z.100 series for SDL-2010 nor Annexes F1, F2 and F3 take precedence in this case.

F1.1.2. Main objectives

A primary objective of a formal SDL-2010 semantics is intelligibility, a prerequisite for correctness, acceptance and maintainability. Intelligibility is supported by building on well-known mathematical formalisms and notations, by a close correspondence between the specification technique and semantics to be formalized, and by concise and well-structured documentation.

Maintainability is another important objective because SDL-2010 is an evolving technical standard. Apart from the language extensions that are incorporated into this Recommendation, further language features under consideration. Therefore, the mathematical formalism has to be sufficiently rich and flexible so that the formal semantics can be adapted and extended with a reasonable effort.

SDL-2010 can be classified as a model-oriented formal description technique (FDT) for the specification of distributed and concurrent systems, which means that an SDL-2010 specification explicitly defines a set of computations. This calls for an operational semantics in order to achieve a close correspondence with the specification, and thus improve its intelligibility. In addition, operational semantics lends itself naturally to executability ([b-Eschbach], [b-Eschbach 2001] and [b-Glässer]), which, given the availability of tools, fulfils another explicit objective.

F1.1.3. References and definitions

The references and definitions of the main body of Recommendation ITU-T Z.100 apply throughout Annexes F1, F2 and F3.

F1.1.4. Bibliographical references (for this annex only)

[b-ASM]http://www.eecs.umich.edu/gasm/ (accessed 20 March 2018).

[b-Blass 2008]Blass, A. and Gurevich, Y. (2008), *Abstract State machines capture parallel algorithms: Correction and extension*. ACM Transactions on Computational Logic, Vol. 9, No. 3, ACM.
NOTE: The postulates presented in [b-Blass] do not allow proclets to be created on the fly. On the fly creation of proclets is required to correct one of the flaws identified in the examples in Section 8 of [b-Blass]. Other flaws in the earlier article have also been corrected.

[b-Blass]Blass, A. and Gurevich, Y. (2003), *Abstract State Machines capture parallel algorithms*, ACM Transactions on Computational Logic, Vol. 4, No. 4, ACM.
NOTE: The axiomatic definition of abstract state machines for sequential algorithms is modified to capture parallel algorithms. Specifically, Bounded Exploration is replaced by Background, Proclet (sub-process of a parallel algorithm that contains no unbounded parallelism) and Bounded Sequentiality to ensure that the number of state elements involved in a given computation step is bounded, with the bound depending only on the algorithm and not on the state.

[b-Börger Stärk]Börger, E., and Stärk, R. S. (2003), *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer-Verlag.
NOTE: Design and analysis for multi-agent as well as single agent abstract state machines. Used here to clarify the coherence condition.

[b-Börger]Börger, E. (2003), *The ASM Refinement Method*, Formal Aspects of Computing Vol. 15, pp. 237-257, BCS.

[b-Eschbach 2001]Eschbach, R., Glässer, U., Gotzhein, R., von Löwis, M., and Prinz, A. (2001), *Formal Definition of SDL-2000: Compiling and Running SDL Specifications as ASM Models*, Journal of Universal Computer Science Vol. 7, No. 11, pp. 1024-1049, Springer.

[b-Eschbach]Eschbach, R., Glässer, U., Gotzhein, R., and Prinz, A. (2000), *On the Formal Semantics of SDL-2000: A Compilation Approach Based on an Abstract SDL Machine*, in: Y. Gurevich, M. Odersky, P. Kutter, L. Thiele (Eds.), Abstract State Machines – Theory and Applications, Lecture Notes in Computer Science, Vol. 1912, Springer-Verlag.

[b-Glausch]Glausch, A. and Reisig, W. (2007), *A Semantic Characterization of Unbounded-Nondeterministic Abstract State Machines*, in T Mossakowski et al. (Eds.) CALCO 2007, LNCS 4624, Springer-Verlag.
NOTE: The axiomatic definition given by Gurevich for the sequential algorithms captured by ASMs is extended to nondeterministic ASMs. Unbounded nondeterminism means that there may be uncountably many update sets that could be produced by an algorithm in a given state. However, so long as each of these is bounded in size, the fact that only one of them is applied means that the number of state elements involved in a computation step is bounded.

[b-Glässer 2007]Glässer, U., Gurevich, Y., and Veanes, M. (2007), *Abstract Communication Model for Distributed Systems*, IEEE Transactions on Software Engineering, Vol. 30, No. 7, pp. 458-472.
NOTE: A high level abstract model for message based communication networks is presented. The model is based on distributed abstract state machines, has been implemented in AsmL and has been used for testing distributed systems.

[b-Glässer]Glässer, U., Gotzhein, R., Prinz, A. (2003), *The formal semantics of SDL-2000 – Status and perspectives*, Computer Networks, Vol. 42, No. 3, pp. 343-358, Elsevier Sciences.
NOTE: The design objectives of the SDL semantics include executability, intelligibility, conciseness and flexibility as well as the ideals of correctness and completeness (for which indisputable evidence cannot be inferred from the SDL grammars and textual description). The decision to base the SDL formal semantics on ASM is documented, and the ITU-Tapproach is described. This approach entails analysis of an SDL model and synthesis of an ASM program that defines the behaviour of SDL agents. Execution is defined in terms of the SDL virtual machine, which provides operating system functionality that controls the execution of ASM programs on the logical hardware of the SDL abstract machine (SAM).

[b-Gurevich 2000]Gurevich, Y. (2000), *Sequential Abstract State Machines Capture Sequential Algorithms*, Microsoft Research.

[b-Gurevich]Gurevich, Y. (1995), *Evolving algebras 1993: Lipari guide,* in Specification and validation methods, Börger, E. (ed.), pp. 9-36, Oxford University Press.

F1.1.5. Status of Annex F1 (this annex)

The (01/2015) edition was an improvement on the previous edition, because it updated the description of ASM and the list of references. The (10/2016) edition clarified a number of items in the (01/2015) edition and added descriptions of items used in F2 and/or F3 that were previously missing or newly added to F2 and/or F3. This edition is essentially a maintenance update further clarifying existing content and adding additional features used in F2 and/or F3.

F1.2. Overview of the semantics

In order to define the formal semantics of SDL-2010, the language definition is decomposed into several parts:
– grammar
– well-formedness conditions
– transformation rules
– dynamic semantics.

The starting point for defining the formal semantics of SDL-2010 is a syntactically correct SDL-2010 specification, represented as an abstract syntax tree (AST).

The first three parts of the formal semantics are collectively referred to as *static semantics* or *static aspects* in the context of SDL-2010 (see Figure F1.1), and are described in Part 2 of the formal definition, i.e., Annex F2.

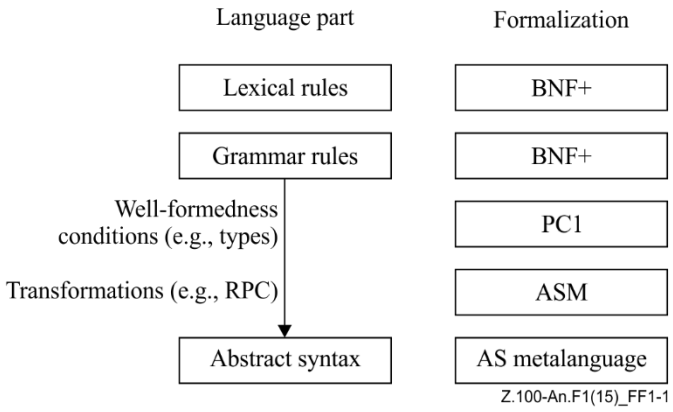

Figure F1.1 — Static aspects of SDL-2010

The *grammar* defines the set of syntactically correct SDL-2010 specifications. The ITU-T Z.100 series for SDL-2010 defines a concrete graphical grammar, a concrete textual grammar, and an abstract grammar. The syntax of the concrete grammars is defined formally using the Backus-Naur form (BNF) with some extensions to capture the graphical language constructs (see clause 5.4.1 of [ITU-T Z.111]). The abstract grammar is obtained from the concrete grammars by removing irrelevant details such as separators and lexical rules, and by applying transformation rules (see below). The syntax of the abstract grammar is defined in the textual presentation metalanguage for abstract grammar (see clause 5.4.1.1 of [ITU-T Z.111]).

From the specifications that are correct with respect to the grammar syntax, the *well-formedness conditions* define the specifications that are also correct with respect to context information. The well-formedness conditions address matters of scope, visibility and type. The well-formedness conditions are defined in terms of first order predicate calculus (PC1).

Furthermore, some language constructs appearing in the concrete grammars are replaced by other language elements in the abstract grammar using *transformation rules* to keep the set of core concepts small. These transformations are described in the 'Model' paragraphs of the ITU-T Z.100 series for SDL-2010, and are formally expressed as rewrite rules.

The *dynamic semantics* applies only to syntactically correct SDL-2010 specifications that satisfy the well-formedness conditions. The dynamic semantics defines the set of computations associated with a specification, and are described in Part 3 of the formal definition, i.e., Annex F3.

F1.2.1. Grammar

The *grammar* of SDL-2010 is formalized as described above. The primary concrete grammar is given for SDL-GR. Most of the grammar of SDL-GR is textual, but it has some graphical elements. To enable formalisation of SDL-2010 specifications into ASM, any SDL-GR graphical element is changed to the equivalent concrete textual representation (SDL-PR) defined in [ITU-T Z.106]. The grammar in the ITU-T Z.100 series for SDL-2010 is designed to be a presentation grammar: it is not adapted to automatic parser generation. Moreover, some restrictions that finally guarantee uniqueness of the semantics cannot be expressed 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). The translation from the concrete textual SDL-2010 representation to the abstract syntax representation of SDL-2010 (called AS1) consists of two steps. The first step from the concrete textual SDL-2010 representation to AS0 (the concrete syntax with details such as separators and lexical rules removed) is not formally defined, but is derived from the correspondence between the two grammars, which is almost one-to-one. The second step, translating AS0 to AS1, is formally captured by a set of transformation rules (see Annex F2).

F1.2.2. Well-formedness conditions

The *well-formedness conditions* define additional constraints that a well-formed SDL-2010 specification has to satisfy. These constraints cannot be expressed using context-free grammar rules, but they are static, and can be defined and checked independently of the dynamic semantics of SDL-2010 (see Annex F2). An SDL-2010 specification is *valid* if and only if it satisfies the syntactical rules and the static conditions of SDL-2010. The well-formedness conditions are context-dependent conditions on elements of a context free grammar.

There are five kinds of well-formedness conditions:

– *Scope/visibility rules*: The definition of an entity introduces an identifier used as the reference to the entity. Only the use of visible identifiers is allowed. The scope/visibility rules are applied to determine whether the corresponding definition of an identifier is visible or not.

– *Disambiguation rules*: Sometimes a name might refer to several identifiers. Rules are applied to find out the correct one.
– *Data type consistency rules*: These rules ensure that dynamically, no operation is applied to operands that do not match its argument types. More specifically, the data type of an actual parameter has to be compatible with that of the corresponding formal parameter; and the data type of an expression has to be compatible with that of the variable to which the expression is assigned.
– *Special rules*: There are some rules applicable to specific entities. For example, it is not allowed to export a procedure variable (that is, a variable defined within a procedure).
– *Plain syntax rules*: There are some rules that refer to the correctness of the concrete syntax, and that have no counterpart in the abstract syntax. For instance, the names at the beginning and at the end of a definition in SDL-PR have to match.

F1.2.3. Transformation rules

For a language with a rich syntax, it is important to identify the core concepts matching the intentions of the language designer. Further language constructs, such as shorthand notations, that are introduced for convenience, but do not add to the expressiveness of the language, can be replaced using these core concepts. Since replacements, which are described by transformation rules, can be formalized, it suffices to define the dynamic semantics only for the core concepts, which adds to its conciseness and intelligibility. Figure F1.1 illustrates the general approach. The language is defined with its concrete grammar using lexical and syntax rules. Consistency constraints are defined on this concrete grammar.

The ITU-T Z.100 series for SDL-2010 prescribes the transformation of SDL-2010 specifications by a sequence of *transformation steps*. Each transformation step consists of a set of single transformations as stated in the *Model* clauses, and determines how to handle one special class of shorthand notations. The result of one step is used as input for the next step.

To formalize the transformation rules of SDL-2010, the rewrite rules in PC1 are used. These rules define patterns of the 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. A single transformation is realized by the application of a rewrite rule to the concrete specification, which essentially means to replace parts of the specification by other parts as defined by the rule (see Annex F2).

F1.2.4. Dynamic semantics

The *dynamic semantics* (clauses F3.2 and F3.3) consists of the following parts (see Figure F1.2):

a) The *SDL-2010 Abstract Machine (SAM)*: this is defined using ASM. The definition of the SAM is divided into three parts, corresponding to the abstract syntax:
   1) basic signal flow concepts (such as signals, timers, gates, channels) defined in terms of an ASM model in clause F3.2.1.1;
   2) various types of ASM agents to model corresponding SDL-2010 agents in clause F3.2.1.2; and
   3) signal processing and behaviour primitives (the abstract machine instructions of the SAM) in clause F3.2.1.4 (that uses the interface to the data type part in clause F3.2.1.3).
b) The *compilation function* (clause F3.2.2): this maps the AST of an SDL-2010 specification to *SAM* behaviour primitives that model the actions of the SDL-2010 agents. The compilation function amounts to an abstract compiler taking the AST of the state machines as input and transforming it to *SAM* instructions.
c) The *SAM Programs* (clause F3.2.3): these define the set of computations. These programs consist of an initialization phase and an execution phase. SAM programs have fixed parts that

are the same for all SDL-2010 specifications, and variable parts that are generated from the abstract syntax representation of a given SDL-2010 specification.

1) The *initialization* (clause F3.2.3.1) phase handles static structural properties of the specification. The pre-initial state of a system is defined followed by several initialization programs. The initial system state is then reached by creating the SDL-2010 system agent, and by activating this agent in the pre-initial state. The initialization recursively unfolds the static structure of the system, creating further SDL-2010 agents as specified so that all the initial objects are created. The same process is initiated in the subsequent execution phase, whenever SDL-2010 agents are created. From this point of view, the initialization merely describes the instantiation of the SDL-2010 system agent.
2) The *execution* (clause F3.2.3.2) phase is modelled by distinguishing two alternating phases, namely the selection and the firing of transitions.

d) The *data semantics* (clause F3.3): this is separated from the rest of the semantics by an interface (clause F3.2.1.3). The use of an interface is intentional at this place. It allows the data model to be exchanged, if for some application area another data model is more appropriate than the SDL-2010 built-in model. Moreover, the SDL-2010 built-in model can be changed this way without affecting the rest of the semantics.

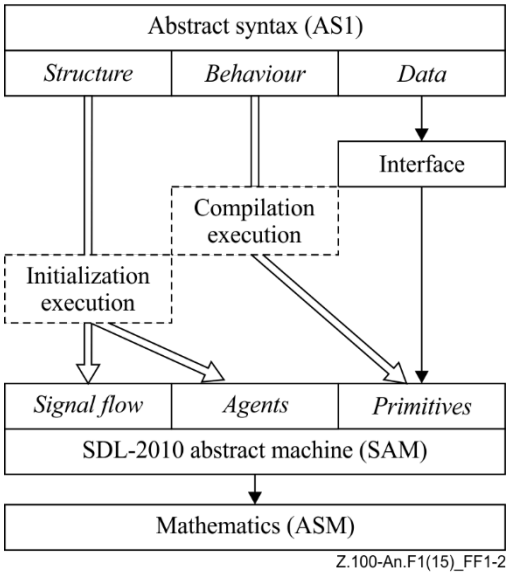

Figure F1.2 — Overview of the dynamic semantics

The formal semantics is formalized starting from the abstract syntax AS1 of SDL-2010. From this abstract syntax, a behaviour model for SDL-2010 specifications is derived that can be understood as abstract code generated from an SDL-2010 specification. The approach chosen here is based on an abstract operational view using the ASM formalism as the underlying mathematical framework for a rigorous semantic definition of the SAM model. The compilation defines an abstract compiler mapping the behaviour parts of SDL-2010 to abstract code (denotational semantics). Finally, the initialization describes an interpretation of the abstract syntax tree to build the initial system structure (operational semantics).

The *dynamic semantics* associates a particular distributed, real-time ASM with each SDL-2010 specification. Intuitively, an ASM consists of a set of autonomous agents cooperatively performing concurrent machine runs. The behaviour of an agent is determined by an ASM program, each consisting of a transition rule that defines the set of possible computations (called "runs" in the context of ASM). 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 modelled. An introduction to the ASM model, and the notation used in Annexes F1, F2 and F3, is given in clause F1.3.

F1.3. Abstract State Machines

This clause explains the basic notions and concepts of *Abstract State Machines* (*ASM*) as well as the notation used in these Annexes F1, F2 and F3 to define the SDL-2010 abstract machine model. The objective here is to provide an intuitive understanding of the formalism; for a rigorous definition of the mathematical foundations of ASM and its application, the reader is referred to [b-Gurevich], [b-Blass], [b-Blass 2008], [b-Glausch], [b-Börger Stärk] and [b-Börger]. A discussion and motivation of the appropriateness of the semantic framework used here is given in [b-Eschbach], [b-Eschbach 2001] and [b-Glässer]. Further references on ASM-related material can also be found on the ASM webpages [b-ASM].

The ASM model used to define the dynamic semantics of SDL-2010 is explained in several steps. Firstly, the *basic ASM model* with a single agent is treated (see clause F1.3.1). Next, this model is extended to cover *multi-agent systems* (see clause F1.3.2). Then, *open systems*, i.e., systems interacting with an environment they cannot control, are addressed by adding the notion of *external world* (see clause F1.3.3). Finally, the model is extended by introducing a notion of *real-time behaviour* (see clause F1.3.4). To illustrate these steps, an ASM model for a simple system is developed, step-by-step. The final ASM model of this system is summarized in clause F1.3.5. Additional notation used to define the dynamic semantics of SDL-2010 is explained in clause F1.3.6.

EXAMPLE — A simple resource management system (*RMS*):

In order to illustrate the ASM model, a simple resource management system (*RMS*) consisting of *a group of* $n > 1$ *agents* competing for a *resource* (for instance, a device or service) is defined. Informally, this system is characterized as follows:

- There is a set of m *tokens*, $m < n$, used to grant *exclusive* or *non-exclusive* (*shared*) access to the resource.
- Depending on whether the desired access mode is exclusive or shared, an agent must own all tokens or one token, respectively, before the agent may access the resource.
- An agent is idle when not competing for a resource, waiting when trying to obtain access to the resource, or busy while owning the right to access the resource.
- Once an agent is waiting, it remains so until it obtains access to the resource.
- A busy agent releases the resource when it is no longer needed, as indicated by a stop condition for that agent that is externally set. On releasing the resource, all tokens owned by the agent are returned.
- Stop conditions are only indicated when an agent is busy. This is an integrity constraint on the behaviour of the external world.
- Initially, all agents are idle, and all tokens are available.

The system will be defined step by step, as the explanations of the ASM model proceed, starting with the basic ASM model with a single agent. The final ASM model of this system is summarized in clause F1.3.5.

F1.3.1. Basic ASM model

F1.3.1.1. Overview

An abstract state machine (ASM) is a model of computation that treats first-order structures as dynamic entities whose states can change during a computation.

An abstract state machine has a set of states S, a subset of initial states $S_0 \subseteq S$ and a function $\tau : S \to S$, called the one-step transformation. Every state is a first-order structure. All the states of an ASM have the same signature, which is also called the signature of the ASM, and all the states have the same base set, called the base set of the ASM. τ does not change the base set, but it does, in general, change the equivalences that hold between terms of the signature.

The behaviour of an abstract state machine is modelled as a run or sequence of states. A run starts with an initial state, and each subsequent state is derived from its predecessor by application of the one-step transformation. Each application of the one-step transformation is called a move.

	τ		τ		τ		*moves*
$s_0 \in S_0$	$\rightarrow$	s_1	$\rightarrow$	s_2	$\rightarrow$	...	*states*

F1.3.1.2. States

The base set of the abstract state machine, which is the base set of every state of the ASM, contains three distinct elements: *true*, *false* and *undefined*. The base set also contains an infinite number of *reserve elements*. A state also has functions and predicates that are defined over the base set. All functions are total, with *undefined* being used to mimic partial functions. Predicates are functions whose only possible values are *true* or *false*. Special unary predicates called *domain names* identify members of the base set as belonging to particular *domains*. This allows states to be viewed as many-sorted structures.

Embedded within a state are certain substructures or *background classes* [b-Blass] and [b-Blass 2008]. One such background includes *true*, *false*, the domain name *BOOLEAN* and the Boolean operators. The backgrounds used in modelling SDL-2010 include numbers, sets and sequences. A minimal background is defined in [b-Blass] and [b-Blass 2008] to characterize the ASMs that model parallel algorithms.

F1.3.1.3. One-step transformation

The one-step transformation τ updates the values of functions. In general, for $s \in S$, some equivalences hold in s but not in $\tau(s)$ and vice versa. In some cases, τhas no effect, so s and $\tau(s)$ are the same.

To express the relationship between s and $\tau(s)$ more precisely, the changes effected by τare described in terms of an *update set*. An update set Δ is a set of triples $< f, [a], b >$ where f is a function symbol, $[a]$ is a tuple of elements of the base set of the abstract state machine that respects the arity of f, and b is an element of the base set. The relationship between s and τ is expressed by stating that Δ contains all information of the form $f[a] = b$ that is not true in the state s but is true in $\tau(s)$ [b-Blass].

The pair $< f, [a] >$ is also called a *location* of a state s. This captures the idea of updating the value of a variable in a conventional imperative programming language. A member $< f, [a], b >$ of an update set is also known as an *update* and may be written $f[a] := b$ to reinforce the idea of variable assignment.

An abstract state machine provides a model of computation for an algorithm that is expressed as a program using a programming-style syntax. The algorithm defines the update set for every state of the ASM. To facilitate the study of the kinds of algorithms captured by different kinds of ASM, axioms [b-Blass] have been developed to define classes of ASMs in a syntax-independent way [b-Glausch]. As well as capturing the ideas of state and one-step transformation outlined above (the Sequential Time and Abstract State postulates, [b-Blass] and [b-Gurevich 2000]), axioms aim to capture the idea that the amount of computation required to move from s to $\tau(s)$ must be bounded, where the bound depends only on the algorithm and not on the state.

F1.3.1.4. Specifying an abstract state machine

An ASM specification consists of a set of declarations that define its vocabulary (signature), its initial state(s) and a transition rule that defines the one-step transformation $\tau : S \rightarrow S$. The transition rule, called the ASM program, is defined using a pseudo-code-like syntax based on terms of the signature.

F1.3.1.5. Specifying the signature (vocabulary)

An ASM signature comprises function names, predicate names and domain names. Names in the signature have a specified arity, and are interpreted over the base set of the states of an ASM. The interpretation of names respects their arity.

The following notational conventions are used when declaring names.

– *Domain names* start with an uppercase letter and presented in small-capitalized italics (as in *AGENT*), except when denoting a non-terminal of the SDL-2010 abstract grammar. In that case, domain names are written as the SDL-2010 non-terminals, i.e., in italics, hyphenated, and starting with an uppercase letter (as in *Agent-definition*). A domain name *D* is declared by **domain** *D*. A domain name is interpreted as a unary predicate which yields *true* for the members of the base set that belong to the domain.

– *Function names* are written in italics starting with a lowercase letter (as in *mode*). A function name *f* is declared by $f : D_1 \times D_2 \times \ldots \times D_n \rightarrow D_0$, where *n* is the arity of *f*, and D_0, D_1, $D_2 \ldots D_n$ are domain names. A function name is interpreted as a function over the base set. The interpretation respects the arity of *f*.

– *Predicate names* that are not *Domain names* are also written in italics (as in *available*). A predicate name *p* is declared by $p : D_1 \times D_2 \times \ldots \times D_n \rightarrow$ *BOOLEAN*. A predicate is interpreted as a function whose value is either *true* or *false*.

Declarations also include qualifiers, which specify further restrictions on their interpretation. Qualifiers on name declarations constrain interpretation of the one-step transformation.

– *Static names* are qualified by the keyword **static**. A name that is declared static has the same interpretation in every state of an ASM. This means that the one-step transformation cannot update a static name. So the interpretation of a static domain name yields *true* for the same elements in every state, and a static function name yields the same value for a given argument tuple in every state of an ASM.

– Dynamic names are qualified by one of the keywords **controlled**, **shared** or **monitored**. The one-step transformation can change the value of the interpretation of a dynamic name. A dynamic function can yield different values for a given set of arguments in different states of an ASM. An ASM can be subject to external environmental influences. ASM agents can communicate [b-Glässer 2007]. When any of these situations is specified, the keywords **controlled**, **shared** and **monitored** constrain visibility and updates of domains, functions and predicates.

EXAMPLE 1 — The RMS Signature:

static domain *AGENT*
static domain *TOKEN*
domain *MODE*

shared *mode*: *AGENT* → *MODE*
controlled *owner*: *TOKEN* → *AGENT*
static *ag*: → *AGENT*

idle: *AGENT* → *BOOLEAN*
waiting: *AGENT* → *BOOLEAN*
busy: *AGENT* → *BOOLEAN*
available: *TOKEN* → *BOOLEAN*

monitored *stop*: *AGENT* → *BOOLEAN*

The static domain names *AGENT*, *TOKEN* and *MODE* are introduced to represent the (single) agent of the system, the set of tokens, and the different access modes (exclusive, shared), respectively. The

names *mode* and *owner* denote dynamic functions; they are used to model the current access mode of an agent and the current owner of a token, respectively. The 0-ary function name *ag* refers to a value of the domain *AGENT*. *idle*, *waiting*, *busy*, and *available* are names of derived, dynamic predicates. *stop* denotes a monitored predicate, which will be explained later.

The vocabularies we will consider also include *predefined names*, which include all the items in the background classes of the ASM. These include the equality sign, the 0-ary predicate names *true*, *false* and *undefined*, the domain names *BOOLEAN*, *NAT* and *REAL*, as well as the names of frequently used standard functions (such as Boolean operations $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftarrow$, and set operations $\subseteq$, $\cup$, $\cap$, $\in$, $\notin$, etc.). The full collection of predefined names is listed in clause F1.3.6. Interpretation of predefined names is constrained to the usual meanings of those names.

The notational conventions described above enable declaration of *basic names* that are interpreted directly over the ASM base set. As well as *basic names*, the signature of an ASM may include *derived names*, whose interpretation depends on the interpretation of the basic names. Derived names are defined using logical formulae involving other names. The interpretation of a derived name is determined by the interpretations of those other names, and ultimately by the interpretation of base names.

Let *derivedName* be an n-ary name, and let *formula*(v_1, …, v_n) denote a formula of the domain D with free variables v_1, …, v_n of domains D_1, …, D_N, $n \geq 0$. The general form of a *derived name definition* is:

derivedNameDefinition ::= derivedName(v_1: D_1, …, v_n: D_n)*:D* $=_{\text{def}}$ *formula* (v_1,…, v_n)

The result domain D is omitted in case of a derived domain definition.

EXAMPLE 2 — Derived names for the *RMS* specification

The following derived predicates are defined to refer to the status of an agent/token in a given state.

MODE $=_{\text{def}}$ {*exclusive*, *shared*}
idle(*a*: *AGENT*): *BOOLEAN* $=_{\text{def}}$ *a.mode* = *undefined* $\wedge \forall$ *t* $\in$ *TOKEN*: *t.owner* $\neq$ *a*
waiting(*a*: *AGENT*): *BOOLEAN* $=_{\text{def}}$ *a.mode* $\neq$ *undefined* $\wedge \forall$ *t* $\in$ *TOKEN*: *t.owner* $\neq$ *a*
busy(*a*: *AGENT*): *BOOLEAN* $=_{\text{def}}$ *a.mode* $\neq$ *undefined* $\wedge \exists$ *t* $\in$ *TOKEN*: *t.owner* = *a*
available (*t*: *TOKEN*): *BOOLEAN* $=_{\text{def}}$ *t.owner* = *undefined*

An agent *a* is, for instance, idle if the function *mode* yields the value *undefined* for that agent, and *a* does not hold any token. A token *t* is available if no agent is holding *t*.

For an improved readability, use of the "." *-notation* is allowed for unary functions and predicates. For instance, *a.mode* is equivalent to *mode*(*a*).

F1.3.1.6. Initial states

The set of *initial states* $S_0 \subseteq S$ is defined by constraints imposed on domains, functions and predicates as associated with the names in V. The initial constraints for predefined domains and operations are given implicitly (see clause F1.3.6). Initial constraints have the following general form:

initially *ClosedFormula*

EXAMPLE — Initial states:

The following constraints define the set of initial states of the system *RMS*.

initially *AGENT* = {*ag*}
initially $\forall$ *a* $\in$ *AGENT*: *a.idle* $\wedge \forall$ *t* $\in$ *TOKEN*: *t.available*

The first constraint defines the initial set *AGENT* to consist of a single element *ag*. The second constraint expresses that initially, the agent of *RMS* is idle *(a.mode = undefined)*, and all tokens are available *(t.owner = undefined)*. Note that no constraint on *stop* is defined.

F1.3.1.7. State transitions and runs

A (global) state $s \in S$ is given by an interpretation of the names in V over the base set of M. State transitions can be defined in terms of partial reinterpretations of dynamic domains, functions and predicates. This gives rise to the notions of *location* as a conceptual means to refer to parts of global states, and of *update* to describe state changes.

A *location of a state s of M* is a pair $loc_s = < f, s(x) >$, where f is a dynamic name in V, and $s(x)$ is a sequence of elements of the base set according to the arity of f. An *update of s* is a pair $\delta_s = < loc_s, s(y) >$, where $s(y)$ identifies an element of the base set as the new value to be associated with the location loc_s. To *fire* δ_s means to transform s into a state s' of M such that $f_{s'}(s(x)) = s(y)$, while all other locations loc_s' of s, $loc_s' \neq loc_s$, remain unaffected. In other words, firing an update modifies the interpretation of a state in a well-defined way.

The potential behaviour of a basic ASM is captured by a *program P*, which is defined by a *transition rule* (see clause F1.3.1.8 and clause F1.3.1.10). For each state $s \in S$, a program P of M defines an *update set* $\Delta_s(P)$ as a finite set of updates of s. $\Delta_s(P)$ is *consistent*, if and only if it does not contain any two updates δ_s, δ'_s such that $\delta_s = < loc_s, s(y) >$, $\delta'_s = < loc_s, s(y') >$, and $s(y) \neq s(y')$. The *firing of a consistent update set* $\Delta_s(P)$ in state s means to fire all its members simultaneously, i.e., to produce (in one atomic step) a new state s' such that for all locations $loc_s = < f, s(x) >$ of s, $f_{s'}(s(x)) = s(y)$, if $< < f, s(x) >, s(y) > \in \Delta_s(P)$, and $f_{s'}(s(x)) = f_s(s(x))$ otherwise, and is called *state transition*. Firing an inconsistent update set has no effect, i.e., $s' = s$.

NOTE – In the context of the SDL-2010 semantics, an inconsistent update set indicates an error in the semantic model. The ASM semantics ensures that such errors do not destroy the notion of state.

The behaviour of a single-agent ASM M is modelled through (finite or infinite) *runs of M*, where a *run* is a sequence of state transitions of the form:

	$\Delta_{s0}(P)$		$\Delta_{s1}(P)$		$\Delta_{s2}(P)$		*moves*
s_0	→	s_1	→	s_2	→	…	*states*

such that $s_0 \in S_0$, and s_{i+1} is obtained from s_i, for $i \geq 0$, by firing $\Delta_{si}(P)$ on s_i, where $\Delta_{si}(P)$ denotes an update set defined by the program P of M on s_i (see clause F1.3.1.10). The meaning of an ASM is defined to be the set of all its runs. In the sequel, we restrict attention to runs starting in an initial state, also called *regular runs*.

F1.3.1.8. Transition rules

Transition rules specify update sets over ASM states. Complex rules are formed from elementary rules using various rule constructors. The elementary form of transition rule is called *update instruction*.

– *update* instruction

Rule ::= f(t1,...,tn) := t0 $(n \geq 0)$

Here, f is a non-static name of V denoting either a controlled or a shared function, predicate or domain, and $t_0, t_1, \ldots, t_n$ are terms over V identifying, for a given state s, the location $loc = < f, < s(t_1), \ldots, s(t_n) > >$ to be changed and the new value $s(t_0)$ to be assigned, respectively. In other words, the above update instruction specifies the update set

$\{ < < f, < s(t_1), \ldots, s(t_n) > >, s(t_0) > \}$, consisting of a single update. Note that only locations related to (non-static) basic names may occur at the left-hand side of an update instruction.

EXAMPLE 1 — Update instruction:

Let t be a variable denoting a token and *ag* be an agent.

t.owner := ag specifies the update set $\{ < < ow \neq r, < s(t) > >, s(ag) > \}$
ag.mode := undefined specifies the update set $\{ < <$ mode, $< s(ag) > >$, *s (undefined)* $>\}$

The construction of complex transition rules out of elementary update instructions is recursively defined by means of *ASM rule constructors*. For the ASM model applied to define the SDL-2010 semantics, six different constructors (**if-then**, **do-in-parallel**, **do-forall**, **choose**, **extend**, **let**) are used. These constructors are listed below, with an informal description of their meaning. Here, *Rule*, $Rule_i$ denote transition rules, *g* denotes a Boolean term, and $v, v_1, \ldots, v_n$ denote free variables over the base set of *M*. The scope of a rule constructor is expressed by appropriate keywords, and can additionally be indicated by indentation. The closing keywords can be omitted, if no confusion arises. If closing keywords are omitted, the corresponding constructor extends as much as possible, but not over the next **where**-clause.

– *if-then-constructor*

Rule ::= **if** *g* **then**
  $Rule_1$
  [**else**
  $Rule_2$]
  endif

The update set specified by *Rule* in a given state *s* is defined to be the update set of $Rule_1$ or $Rule_2$, depending on the value of *g* in state *s*. Without the optional **else**-part, the update set defined by *Rule* is the update set of $Rule_1$ or the empty update set. Sometimes, **elseif** is used as abbreviation for **else if**.

– *do-in-parallel-constructor*

Rule ::= **do in-parallel**
  $Rule_1$
  ...
  $Rule_n$
  [**enddo**]

The update set defined by *Rule* in state *s* is defined to be the union of the update sets of $Rule_1$ through $Rule_n$. In other words, the order in which transition rules belonging to the same block are stated is irrelevant. For brevity, the keywords **do in-parallel** and **enddo** may be omitted, where no confusion arises. Hence, an ASM program often appears as a collection of rules rather than a monolithic block rule.

– *do-forall-constructor*

Rule ::= **do forall** *v: g(v)*
    $Rule_0$ *(v)*
  enddo

The effect of *Rule* is that $Rule_0$ is fired simultaneously for all elements *v* of the base set of *M* for which the Boolean condition $g(v)$ holds in state *s*, where *v* is a free variable *in* $Rule_0$. *More precisely*, $\Delta_s(\text{Rule})$ is the union of all update sets Δ_s *(*$Rule_0(v)$*)* such that $g(v)$ holds in state *s*. Recall that update sets are required to be finite; therefore, $g(v)$ must hold for a finite number of values only.

– *choose-constructor*

Rule ::= **choose** *v: g(v)*

Rule$_0$ (v)
endchoose

The effect of *Rule* is that *Rule$_0$* is fired for an element *v* of the base set of *M* for which the condition $g(v)$ holds in state *s*, where *v* is a free variable *in Rule$_0$. More precisely*, $\Delta_s(\text{Rule})$ *is some update set* Δ_s *(Rule$_0$(v)) such that* $g(v)$ holds in state *s*, or the empty update set if no such *v* exists.

– *extend-constructor*

Rule ::= **extend** *D* **with** *v$_1$,...,v$_n$*
Rule$_0$ (v$_1$,..., v$_n$)
endextend

The effect of *Rule* when fired at state *s* is that *n* reserve elements of *s* (see clause F1.3.1.2) are imported into the dynamic domain *D* (while being removed from the reserve), that $v_1, \ldots, v_n$ become bound to one of the imported elements each, and then *Rule$_0$* $(v_1, \ldots, v_n)$ *is fired*.

The *extend* constructor can be used to mimic object-based ASM definitions, where objects are dynamically created. Thus, for each object to be created, an element from the reserve is assigned to the corresponding domain, and initialized.

NOTE – *extend* can be defined in terms of the *import* constructor (not shown here); however, the *import* constructor is not used in the formal definition of SDL-2010.

– *let-constructor*

Rule ::= **let** *v* = *expression* **in**
Rule$_0$ (v)
endlet

The effect of *Rule* when fired in some state *s* is that *v* is bound to the value of *expression*, and that *Rule$_0$* is fired with this value.

EXAMPLE 2 — Transition rule with if-then and choose:

The following transition rule defines the behaviour of agent *ag* when requesting shared access, i.e., when *ag.mode* = *shared*. The rule applies the if-then-constructor, the choose-constructor, and an update instruction.

```
if ag.mode = shared ∧ ag.waiting then
   choose t: t ∈ TOKEN ∧ t.available
      t.owner:= ag
   endchoose
endif
```

The precise meaning of the rule is given by its update set with respect to a state *s*, which is either $\{ <<ow \neq r, <s(t)>>, s(ag)> \}$ for some token $s(t)$ available in *s*, if all further predicates stated in the if-then-constructor hold in *s*, or the empty update set otherwise.

F1.3.1.9. Abbreviations

Rules can be structured using *abbreviations*, consisting of *rule macros* and *derived names*, which may have parameters. This allows for hierarchical definitions, and the stepwise refinement of complex rules, which supports the understanding of ASM model definitions.

Derived names are introduced (as explained in clause F1.3.1.5 and clause F1.3.5.3), i.e., by declaration and definition, or alternatively, in the compact form, by combining declaration and definition.

– *rule-macro*-definition

Let *Rule$_0$* denote a transition rule with free variables $v_1, \ldots, v_n$ *of domains* $D_1, \ldots, D_N$, $n \geq 0$. The general form of a rule macro definition is:

RuleMacroDefinition::= *RuleMacroName(v$_1$: D$_1$,...,v$_n$: D$_N$)* ≡

$Rule_0$ (v_1,...,v_n)

Rule macro names are, by convention, written in small capitals, with a leading capital letter (as in SHAREDACCESS).

– *where*-part

By default, *rule macros* and *derived names* have a global scope. However, their scope can also be restricted to a particular transition rule *Rule* by using the where-part.

Rule ::= $Rule_0$
where
(RuleMacroDefinition | DerivedNameDefinition)$^+$
endwhere

– *rule-macro*-constructor

Rule macros are applied in transition rules as follows:

Rule ::= RuleMacroName (t_1,...,t_n)

Formally, rule macros are syntactical abbreviations, i.e., each occurrence of a macro in a rule is to be replaced textually by the related macro definition (replacing formal parameters by actual parameters).

EXAMPLE — Rule macro:

The transition rule from the previous example can be stated using rule macros, and be defined as a macro itself. Here, SHAREDACCESS is a macro definition with global scope that can be used in other places of the ASM model definition. GETTOKEN is a parameterized macro definition with a local scope restricted to the rule SHAREDACCESS, with formal parameter *a*. When GETTOKEN is applied in SHAREDACCESS, *a* is replaced by the actual parameter *ag*.

```
SHAREDACCESS ≡
  if ag.mode = shared ∧ ag.waiting then
    GETTOKEN(ag)
  endif
  where
    GETTOKEN (a: AGENT) ≡
    choose t: t ∈ TOKEN ∧ t.available
      t.owner:= a
    endchoose
  endwhere
```

F1.3.1.10. ASM programs

An *ASM program P* is given by a framed *transition rule* (or *rule* for short) of the following form:

Rule

As already mentioned, rule macro definitions may either have a local or a global scope. To have a global scope, the macro definitions can be given outside the ASM program and can thus also be applied in the ASM program.

In the basic ASM model there is just one ASM program, which is statically associated with an implicitly defined agent executing this program. The next clause allows several ASM programs to be defined and associated with different agents that are introduced dynamically during abstract machine runs.

EXAMPLE — ASM program:

The ASM program *P* of the system *RMS* is defined as follows:

```
do in-parallel
  SHAREDACCESS
```

```
   ExclusiveAccess
   ReleaseAccess
enddo
where

SharedAccess ≡
   if ag.mode = shared ∧ ag.waiting then
      choose t: t ∈ Token ∧ t.available
         t.owner:= ag
      endchoose
   endif
ExclusiveAccess ≡
   if ag.mode = exclusive ∧∀ t ∈ Token: t.available then
      do forall t: t ∈ Token
         t.owner:= ag
      enddo
      endif
ReleaseAccess ≡
   if ag.busy ∧ ag.stop then
      do in-parallel
         ag.mode:= undefined
         do forall t: t ∈ Token ∧ t.owner = ag
            t.owner:= undefined
         enddo
      enddo
   endif
endwhere
```

The ASM program is defined by a single transition rule as shown in the frame. The transition rule uses the do-in-parallel-constructor and 3 rule macros, which results in a hierarchical rule definition.

F1.3.2. Distributed multi-agent ASM

Mathematical modelling of concurrent and reactive systems requires more than the basic ASM model described above. This section presents the concept of a *distributed ASM* operating in *parallel* with its external environment, where the environment behaves as one or more ASMs.

A *distributed Abstract State Machine M* is defined over a given *vocabulary V* by its *states S*, its *initial states* $S_0 \subseteq S$, its agents *A*, and its *programs P*. These items will be explained in the following subclauses insofar as they differ from the basic ASM model.

F1.3.2.1. Signature

The signature (vocabulary) *V* of a multi-agent ASM *M* includes distinguished domain names:

```
controlled domain Agent
static domain Program
```

representing *a dynamic* set *A* of agents and an invariant set *P* of ASM programs, respectively. *Agent*, *Program* and *program*: *Agent* → *Program* constitutes further *background classes*.

Furthermore, *V* includes a distinguished function name:

```
controlled program: Agent → Program
```

and a special 0-ary function *Self* (see clause F1.3.2.2), whose interpretation is different for each agent.

F1.3.2.2. Agents and runs

A multi-agent, distributed ASM has a finite number of agents. Agents can be created and destroyed dynamically. Each agent executes its own basic ASM. The behaviour of each agent is determined by a program, which is defined by a transition rule. The association between agents and their behaviour is specified by the background function *program*: *Agent* → *Program*. This function can be updated,

allowing agents' behaviour to be modified dynamically, and allowing behaviour to be assigned to newly created agents.

Agents operate concurrently and interact by sending messages to one another (see [b-Blass] and [ITU-T Z.101]). More precisely, agents interact by updating locations that are accessible to other agents. Agents can act as *COMMUNICATORS* [b-Glässer], whose behaviour is to read input locations, transform the values read, and update output locations that can be read by other agents. In this way messages can be passed asynchronously between agents without the original source or the final destination necessarily having a commonly accessible location. Agents also interact with an external environment, which can be viewed as an agent in its own right, or as a collection of agents.

A multi-agent distributed ASM is formed by combining its constituent single-agent ASMs. Like a single agent ASM, it has a set states S, a subset of initial states $S_0 \subseteq S$ and a function $\tau : S \to S$, called the one-step transformation.

To assign a behaviour to an agent of M, the distinguished function *program* (see clause F1.3.2.1) yields (for each agent a of M) the program of P to be executed by a. The function *program* thus allows the definition (or redefinition) of the behaviour of agents dynamically; it is thereby possible to create new agents at run time. In a given state s of M, the agents of M are all those elements a of s such that *a.program identifies* a behaviour (asdefined by some program of P) to be associated with a.

A special 0-ary function *Self* serves as a *self-reference* identifying the respective agent calling *Self*:

monitored *Self*: → *AGENT*

For every agent, *Self* has a different interpretation. By using *Self* as an additional function argument, each agent a can have its own partial view of a given global state of M on which it fires the rule in *a.program*.

EXAMPLE — Scheme of a distributed ASM

In the following figure, a particular distributed ASM M, consisting of three agents ag_1, ag_2, and ag_3 is illustrated. The function *program* associates, with each agent, one of the ASM programs P_1, P_2, and P_3. Here, ag_1 and ag_2 are assigned the same program. Program P_2 is currently not associated with any agent; however, this may change during execution, as *program* is a dynamic function. Each agent has its own *partial view* on a given global state s of M, in which it fires the rule of its current program. In the figure, this view is illustrated by the function *view*, which yields, for each agent, its local and its shared state. In fact, the current view of each agent is determined implicitly by the ASM model definition, including the ASM programs.

The semantic model of concurrency underlying the distributed ASM model defines behaviour in terms of partially ordered runs. A *partially ordered run* represents a certain class of (admissible) machine runs by restricting non-determinism with respect to the order in which the individual agents may perform their computation steps, so-called *moves*. To avoid that agents interfere with each other, moves of different agents need only be ordered if they are causally dependent (as detailed below).

Partially ordered runs

Regarding the moves of an individual agent, these are linearly ordered, whereas moves of different agents need only be ordered in case they are not *independent* of each other. Intuitively, independent moves model concurrent actions that are incomparable with regard to their order of execution. The precise meaning of independence is implied by the coherence condition in the formal definition of partially ordered runs [b-Gurevich].

A run ρ of a distributed ASM M is given by a triple $(\wedge, A, \sigma)$ satisfying the following four conditions:

a) $\wedge$ is a partially ordered set of moves, where each move has only a finite number of predecessors;
b) A is a function on Λ associating agents to moves such that the moves of any single agent of M are linearly ordered;

c) σ assigns a state of M to each initial segment Y of Λ, where $\sigma(Y)$ is the result of performing all moves in Y; if Y is empty, then $\sigma(Y) \in S_0$;

d) if y is a maximal element in a finite initial segment Y of $\wedge$ and $Z = Y - \{y\}$, then $A(y)$ is an agent in $\sigma(Z)$ and $\sigma(Y)$ is obtained from $\sigma(Z)$ by firing $A(y)$ at $\sigma(Z)$ (*coherence condition*).

Implications

Partially ordered runs have certain characteristic properties that can be stated in terms of *linearizations* of partially ordered sets. A linearization of a partially ordered set $\wedge$ is a linearly ordered set $\wedge'$ with the same elements such that if $y < z$ in $\wedge$ then $y < z$ in $\wedge'$. Accordingly, the semantic model of concurrency (as implied by the notion of a partially ordered run) can further be characterized as follows [b-Gurevich]:

– All *linearizations* of the same finite initial segment of a run of M have the same final state.
– A property holds in every reachable state of a run ρ of M if and only if it holds in every reachable state of every linearization of ρ.

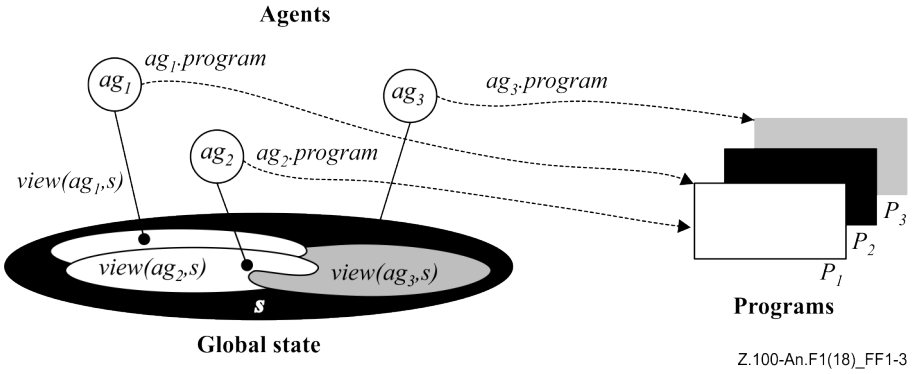

Figure F1.3 — Each agent has a partial view of the global ASM state

F1.3.2.3. Distributed ASM programs

A distributed *ASM M* has a finite set *P* of programs. Eachprogram $p \in P$ is given by a *program name* and a *transition rule* (or *rule* for short). The program name uniquely identifies *p* within *P*, and is represented by a unary static function. Programs are stated in the following form:

ASM-PROGRAM:

Rule

Program names are, by convention, hyphenated and written in small capitals, with a leading uppercase letter (as in RESOURCE-MANAGEMENT-PROGRAM).

By default, the following implicit constraint applies:

initially *PROGRAM* = {PROGRAM$_1$,..., PROGRAM$_n$ }

where PROGRAM$_1$,…, PROGRAM$_N$ are the names of the programs that are defined in the ASM model.

EXAMPLE — ASM program

The distributed ASM program of the system *RMS* defines a single program as follows:

```
RESOURCE-MANAGEMENT-PROGRAM:
do in-parallel
   SHAREDACCESS
   EXCLUSIVEACCESS
   RELEASEACCESS
enddo
where
      SHAREDACCESS ≡
        if Self.mode = shared ∧ Self.waiting then
```

```
        choose t: t ∈ TOKEN ∧ t.available
          t.owner:= Self
        endchoose
      endif
    EXCLUSIVEACCESS ≡
      if Self.mode = exclusive ∧ ∀ t ∈ TOKEN: t.available then
        do forall t: t ∈ TOKEN
          t.owner:= Self
        enddo
      endif
    RELEASEACCESS ≡
      if Self.busy ∧ Self.stop then
        do in-parallel
          Self.mode:= undefined
          do forall t: t ∈ TOKEN ∧ t.owner = Self
            t.owner:= undefined
          enddo
        enddo
      endif
endwhere
```

The program of the distributed ASM has the name Resource-Management-Program, and is defined as the single-agent ASM program before, with one difference: all occurrences of *ag* have been replaced by calls of the function *Self*. This allows the association of the program with different agents, while accessing the local state of these agents.

F1.3.3. The external world

Following an *open system view*, interactions between a system and the external world, e.g., the environment into which the system is embedded, are modelled in terms of various interface mechanisms. Regarding the reactive nature of distributed systems, it is important to clearly identify and precisely state:

– preconditions on the expected behaviour of the external world; and
– how external conditions and events affect the behaviour of an ASM model.

This is achieved through a classification of *dynamic* ASM names into three basic categories of names, which extends the classification of names shown in Figure F1.3:

– *controlled names*

These domains, functions or predicates can only be modified by agents of the ASM model, according to the executed ASM programs. Controlled names are preceded by the keyword **controlled** at their point of declaration, and are visible to the environment. See Figure F1.3.

– *monitored names*

These domains, functions or predicates can only be modified by the environment, but are visible to ASM agents. Thus, a monitored domain, function or predicate may change its values from state to state in an unpredictable way, unless this is restricted by *integrity constraints* (see below). Monitored names are preceded by the keyword **monitored** at their point of declaration. See Figure F1.3.

– *shared names*

These domains, functions or predicates are visible to and may be altered by the environment as well as by the ASM agents. Therefore, an *integrity constraint* on shared domains, functions or predicates is that no interference with respect to mutually updated locations must occur. Hence, it is required that the environment itself acts like an ASM agent (or a collection of ASM agents). Shared names are preceded by the keyword **shared** at their point of declaration. See Example 1.

EXAMPLE 1 — Extended classification of ASM names

External world

The vocabulary V of the system *RMS* is extended by a classification of dynamic functions and predicates:

shared *mode*: *AGENT* → *MODE*
controlled *owner*: *TOKEN* → *AGENT*
monitored *stop*: *AGENT* → *BOOLEAN*

The function *mode*, which determines the current access mode, is shared. It may be affected by externally controlled 'set' operations, switching it to one of the values *exclusive* or *shared*. Furthermore, it is reset internally when the resource is released (see clause F1.3.2.3).

The predicate *stop* represents an external stop request, such as an interrupt, and therefore is monitored.

In general, the influence of the environment on the system through shared and monitored names may be completely unpredictable. However, preconditions on the expected environment behaviour may be expressed by stating *integrity constraints*, which are required to hold in *all* states and runs of *M*. Note that integrity constraints merely express preconditions on the environment behaviour, but *not* properties the system is supposed to have.

Integrity constraints are stated in the following form:

IntegrityConstraint ::= **constraint** *ClosedFormula*

EXAMPLE 2 — Integrity constraints:

The following integrity constraint states that stop requests are only generated for busy agents:

constraint ∀ *a* ∈ *AGENT*: (*a.stop* ⇒ *a.busy*)

F1.3.4. Real-time behaviour

By introducing a notion of *real time* and imposing additional constraints on runs, we obtain a specialized class of ASMs, called *distributed real-time ASM*, with agents performing *instantaneous* actions in *continuous* time. Essentially, that means that agents fire their rules at the moment they are enabled.

To incorporate real-time behaviour into the underlying ASM execution model, we introduce a 0-ary monitored real-valued function *currentTime*. Intuitively, *currentTime* refers to the physical time. As an integrity constraint on the nature of physical time, it is assumed that *currentTime* changes its values monotonically increasing over ASM runs.

monitored *currentTime*: → *REAL*

Figure F1.4

Consider a given vocabulary V containing *REAL* (but not *currentTime*) and let V^+ be the extension of V with the function symbol *currentTime*. Restrict attention to V^+ -states where *currentTime* evaluates to a real number. One can then define a run R of the resulting machine model as a mapping from the interval $[0,\infty)$ to states of vocabulary V^+ satisfying the following *discreteness requirement*:

a) for every $t \geq 0$, *currentTime* evaluates to t at state $R(t)$;
b) for every $\tau > 0$, there is a finite sequence $0 = t_0 < t_1 < \ldots < t_n = \tau$ such that if $t_i < \alpha < \beta < t_{i+1}$ then $\sigma(\alpha) = \sigma(\beta)$.

where the reduct of $R(t)$ to V is denoted by $\sigma(t)$ such that for a given value t, $\sigma(t)$ is derived from $R(t)$ by ignoring the interpretation of the function name *currentTime*.

Exploiting the discreteness property, one effectively obtains some finite representation (*history*) for every finite (sub-) run by abstracting from those states that do not differ in any significant way from their neighbouring states. In particular, one can simply ignore all states that are identical to their preceding state except that *currentTime* has increased. From the above definition of run it follows that only finitely many states are left.

F1.3.5. Example: The system *RMS*

In this clause, we assemble the pieces of the ASM model definition of the system *RMS* into their final version. For completeness, we also repeat the informal description.

F1.3.5.1. Informal description

In order to illustrate the ASM model, a simple resource management system *RMS* consisting of a group of $n > 1$ *agents* competing for a *resource*, for instance, a device or service, is defined. Informally, this system is characterized as follows:

– There is a set of m *tokens*, $m < n$, used to grant *exclusive* or *non-exclusive* (*shared*) access to the resource.
– Depending on whether the desired access mode is exclusive or shared, an agent must own all tokens or one token, respectively, before he may access the resource.
– An agent is *idle* when not competing for a resource, *waiting* when trying to obtain access to the resource, or *busy* when owning the right to access the resource.
– Once an agent is *waiting*, it remains so until it obtains access to the resource.
– A busy agent releases the resource when it is no longer needed, as indicated by a *stop condition* for that agent that is externally set. On releasing the resource, all tokens owned by the agent are returned.
– Stop conditions are only indicated when an agent is busy.
– Initially, all agents are idle, and all tokens are available.

F1.3.5.2. Vocabulary

static domain *TOKEN*
shared *mode*: *AGENT* → *MODE*
controlled *owner*: *TOKEN* → *AGENT*
monitored *stop*: *AGENT* → *BOOLEAN*

F1.3.5.3. Derived names

MODE =$_{\text{def}}$ {*exclusive*, *shared*}
idle (a: AGENT): BOOLEAN =$_{\text{def}}$ *a.mode = undefined* ∧ ∀ *t* ∈ *TOKEN*: *t.owner* ≠ *a*
waiting (a: AGENT): BOOLEAN =$_{\text{def}}$ *a.mode* ≠ *undefined* ∧ ∀ *t* ∈ *TOKEN*: *t.owner* ≠ *a*
busy (a: AGENT): BOOLEAN =$_{\text{def}}$ *a.mode* ≠ *undefined* ∧∃ *t* ∈ *TOKEN*: *t.owner = a*
available (t: TOKEN): BOOLEAN =$_{\text{def}}$ *t.owner = undefined*

F1.3.5.4. Integrity constraints

constraint ∀ *a* ∈ *AGENT*: (*a.stop* ⇒ *a.busy*)

F1.3.5.5. Initial constraints

initially |*AGENT*| > 1
initially |*TOKEN*| < |*AGENT*|
initially ∀ *a* ∈ *AGENT*: *a.program* = RESOURCE-MANAGEMENT-PROGRAM
initially ∀ *a* ∈ *AGENT*: *a.idle* ∧ ∀ *t* ∈ *TOKEN*: *t.available*

F1.3.5.6. ASM programs

RESOURCE-MANAGEMENT-PROGRAM

do in-parallel

```
   SHAREDACCESS
   EXCLUSIVEACCESS
   RELEASEACCESS
enddo
where
   SHAREDACCESS ≡
      if Self.mode = shared ∧ Self.waiting then
         choose t: t ∈ TOKEN ∧ t.available
            t.owner:= Self
         endchoose
      endif
   EXCLUSIVEACCESS ≡
      if Self.mode = exclusive ∧∀ t ∈ TOKEN: t.available then
         do forall t: t ∈ TOKEN
            t.owner:= Self
         enddo
      endif
   RELEASEACCESS ≡
      if Self.stop then
         Self.mode:= undefined
            do forall t: t ∈ TOKEN ∧ t.owner = Self
               t.owner:= undefined
            enddo
      endif
endwhere
```

F1.3.6. Predefined names and special symbols

To define an ASM model, in particular the ASM model capturing the semantics of SDL-2010, certain names and their intended interpretation are predefined. These names are grouped and listed in this clause (where *D* refers to the syntactic category of domains). For prefix, infix and postfix operators, an underline ("_") is used to indicate the position of their arguments. Moreover, the precedence of the operators is indicated by prec(n), where n is a number. Higher numbers mean tighter binding. Monadic operators have a tighter binding than binary ones. Binary operators are associative to the left.

F1.3.6.1. ASM-specific domains

static domain *X*	ASM base set (meta domain)
static domain *Boolean*	Boolean values
static domain *Nat*	Natural values greater than or equal to zero
static domain *Real*	Real values
shared domain *Agent*	ASM agents
static domain *Program*	ASM programs
static domain *Token*	Syntax tokens (character strings)
_ *****	Domain constructor: finite sequences of
_ +	Domain constructor: non-empty, finite sequences of
_ **-set**	Domain constructor: finite sets of
_ × _ prec(7)	Tuple domain constructor
_ ∪ _ prec(6)	Union domain constructor

F1.3.6.2. ASM-specific functions

static *undefined*: → *X*	Indicator for undefined values
monitored *Self*: → *Agent*	Self reference for ASM agents
controlled *program*: *Agent* → *Program*	Program of an ASM agent

monitored *currentTime*: → *Real*	The current system time

F1.3.6.3. Boolean functions and predicates

static *true*: → *Boolean*	Predefined literal
static *false*: → *Boolean*	Predefined literal
_ = _ prec(4)	Equality
_ ≠ _ prec(4)	Inequality
_ ∧ _ prec(3)	Logical and
_ ∨ _ prec(2)	Logical or
_ → _ prec(1)	Implication
_ ⇔ _ prec(1)	Logical equivalence
¬ _	Negation
$\exists x \in D : P(x)$ prec(0)	Existential quantification (at least one element)
$\exists !x \in D : P(x)$ prec(0)	Unique existential quantification (exactly one element)
$\forall x \in D : P(x)$ prec(0)	Universal quantification

F1.3.6.4. Terms

X	0-ary function application
$f(t_1, ..., t_n)$	Function application with n argument expressions
if *Formula* **then** *Term* **else** *Term* **endif**	Conditional expression; again we use **elseif** instead of **else if**
s- _ (_)	Tuple selection function (see Tuples below)
mk- _ (…)	Tuple construction (see Tuples below)
inv- _ (…)	The inverse of a function or map, **inv-** $Fun(x) =_{def} take(\{ a \in D:Fun(a) = x \})$

F1.3.6.5. Functions and relations on syntax tokens (character strings)

_ + _ prec(6)	String concatenation
" _ "	Domain constructor – QUOTATION MARKS around character string
chr: *Nat* → *Token*	Single character *Token* for character with value given by *Nat*
head: *Token* → *Token*	First character of string representing the *Token*
last: *Token* → *Token*	Last character of string representing the *Token*
length: *Token* → *Nat*	Length of the string representing the *Token*
num: *Token* → *Nat*	*Nat* value of first character of string representing the *Token*
substring: *Token* × *Nat* × *Nat* → *Token*	substring of string representing the *Token* from element i length j.
tail: *Token* → *Token*	Tail of string representing the *Token*.

F1.3.6.6. Functions and relations on integers

_ > _, _ ≥ _, _ < _ , _ ≤ _ prec(4)	Comparison operators
_ + _ prec(6)	Addition
_ - _ prec(6)	Subtraction
_ * _ prec(7)	Multiplication
_ / _ prec(7)	Division
_ ^ _ prec(8)	Raise x to the power y

0, 1, ...	Integer literals

F1.3.6.7. Functions on sequences

static *empty*: $\rightarrow D^*$	Empty sequence
static *head*: $D^* \rightarrow D$	Head of the sequence (*undefined* when empty)
static *tail*: $D^* \rightarrow D^*$	Tail of the sequence (*undefined* when empty)
static *last*: $D^* \rightarrow D$	Last element of a sequence (*undefined* when empty)
static *length*: $D^* \rightarrow$ *Nat*	Length of a sequence
static < >: $D_1 \times D_2 \times \ldots \times D_n \rightarrow (D_1 \cup D_2 \cup \ldots \cup D_n)^*$	Sequence constructor; arguments are listed inside the brackets, separated by commas
1 ... n	Sequence constructor for the sequence $<1, 2, ..n>$; can be used so long as it is clear that this constructs a sequence and not a set.
_ ⌢ _ prec(6)	Concatenation of sequences
toSet: $D^* \rightarrow D$ **-set**	Conversion of the elements of a sequence into a set
_ [_]	Access an element of a list; the index within the brackets must be of type Nat
_ **in** _ prec(4)	Element of?
<*expression*> | <*var*> **in** <*seq*>*:* <*cond*> >	Sequence comprehension; acts like a filter on <*seq*>, i. e., order-preserving; <*seq*> is an input sequence whose values are bound to <*var*>; <*var*> is a free variable in <*expression*>; evaluating <*expression*> with <*var*> bound to a value of <*seq*> yields a value; <*cond*> is a condition which determines whether or not that value will be included in the final sequence.
< <*var*> **in** <*seq*>: <*cond*> > $=_{def}$ < <*var*> | <*var*> **in** <*seq*>: <*cond*> >	Abbreviated sequence comprehension; <*cond*> acts as a filter on <*seq*>
< <*expression*> | <*var*> **in** <*seq*> > $=_{def}$ < <*expression*> | <*var*> **in** <*seq*>: *true* >	Abbreviated sequence comprehension
NOTE – The character ⌢ is the Unicode character with the decimal value 8256 (use find u^8256 in MSWord to locate).	

F1.3.6.8. Functions on sets

_ ∪ _ prec(6)	Set union
_ ∩ _ prec(7)	Intersection
_ \ _ prec(6)	Set subtraction
_ ∈ _ prec(4)	Element of?
_ ∉ _ prec(4)	Not element of?
_ ⊆ _ prec(4)	Subset of?
_ ⊂ _ prec(4)	Proper subset of?
| _ |	Cardinality of a set
$\bigcup$ _	Big union: union of all sets included within the argument set
∅	Empty set
static { }: $D_1 \times D_2 \times \ldots \times D_n \rightarrow (D_1 \cup D_2 \cup \ldots \cup D_n)$ **-set**	Set constructor; comma-separated list of arguments in the brackets
1..n	Set constructor for the set $\{1, 2, ..n\}$; can be used where there is no ambiguity with sequence constructor.

take: *D*-**set** → *D*	Select an arbitrary element from the set, or *undefined* for an empty set
_ .. _ prec(5)	Natural range from the first value to the second. Empty set when the second expression is smaller than the first one
{ <*expression*> | <*var*> ∈ <*set*>: <*cond*>}	Set comprehension, acts like a filter on <*set*>; <*set*> is an input set; <*expression*> is evaluated for each binding of <*var*> to a value of <*set*>, yielding a candidate for inclusion in the new set; <*cond*> determines whether or not the candidate is included in the final set.
{<*var*> ∈ <*set*>: <*cond*> } =$_{\text{def}}$ {<*var*> | *var* ∈ <*set*>: <*cond*> }	Abbreviated set comprehension
{ <*expression*> | <*var*> ∈ <*set*> } =$_{\text{def}}$ { <*expression*> | <*var*> ∈ <*set*>: *true* }	Abbreviated set comprehension

F1.3.6.9. Patterns and case-expressions

Patterns provide a means to easily access the structure of values. The following patterns are provided:

– Variables: A variable matches any value. However, if the variable is already bound, it only matches itself.
– Anonymous variables: Anonymous variables are denoted by "*". They are shorthand for introducing an unused variable.
– Constructor: A constructor is given by its name and the arguments that are again patterns. It matches any value that is constructed using that constructor and with the arguments matching their corresponding pattern.

Named pattern: The notation Variable = Pattern introduces a name for (the value matching) the pattern.

Patterns are used to describe functions on the syntax tree. The non-terminal names of the grammar are used as the constructor functions.

A case expression is used to determine a value depending on pattern matching.

CaseExpression ::= **case** *Term* **of**
| *Pattern*$_1$ **then** *Term*$_1$
| *Pattern*$_2$ **then** *Term*$_2$
...
[**otherwise** *Term*$_0$]
endcase

If the value of *Term* matches at least one *Pattern*$_i$, then the result of the case expression is given by the *Term*$_i$. If no pattern matches, the result is *Term*$_0$ (if present). Otherwise, the result is *undefined*.

F1.3.6.10. Union domains

Union domains contain the values of their constituent domains. They are used below as models for named and unnamed unions and alternative right sides to abstract syntax rules.

$D =_{def} D_1 \cup D_2$

F1.3.6.11. Sequence domains

Sequence domains contain sequences of the values of their constituent domains. They are used below as models for named and unnamed sequences, and sequences defined as (parts of) the right sides of abstract syntax rules expressed using extended BNF.

$D =_{def} D_1^*$

F1.3.6.12. Tuples

Tuple domains contain elements of the Cartesian product of their constituent domains. They are used below as models for the tuples specified by the right sides of abstract syntax rules. For every declared tuple domain, several implied constructor and selector functions are defined. For example, the tuple definition:

$D =_{def} D_1 \times D_2^* \times D_3$ ***-set*** $\times D_1 \times (D_1 \cup D_2) \times (D_1 \times D_2)^* \times (D_1 \times D_2)$ ***-set*** $\times (D_1 \cup D_2)$ **-set**

also defines the following constructor and selector functions:

mk- D: $D_1 \times D_2^* \times D_3$ ***-set*** $\times D_1 \times (D_1 \cup D_2) \rightarrow D$
s- D_1: $D \rightarrow D_1$
s- D_2 ***-seq***: $D \rightarrow D_2^*$
s- D_3 **-set**: $D \rightarrow D_3$ **-set**
s2- D_1: $D \rightarrow D_1$
s-implicit: $D \rightarrow D_1 \cup D_2$
s-implicit-seq: $D \rightarrow (D_1 \times D_2)^*$
s-implicit-set: $D \rightarrow (D_1 \times D_2)$ **-set**

When the tuple includes the same domain more than once, selector functions similar to **s2-** D_1 are defined. For union, the special selector function s-**implicit** is defined.

If a domain on the right hand side is optional (written as [*D*]), the selector function **s-***D* is still valid and gives the result *undefined* if *D* is absent and the value of selecting *D* otherwise. Similarly **s-***D***-seq** is valid for [D^*], and **s-***D***-set** is valid for [*D*-**set**].

An invalid selector returns *undefined*.

F1.3.6.13. Abstract syntax rules

Abstract syntax rules from the language definition are directly translated to the ASM notation, using the metalanguage for the concrete grammar as defined in [ITU-T Z.111] with the modified representation of semantic subcategories used in Annex F2. This allows rules to be expressed using the usual BNF conventions extended to include square brackets to indicate optional items, suffixes to indicate various kinds of list, and curly brackets to group related items.

An abstract syntax rule defined with "::" declares a domain of syntax nodes. Each syntax node has an identity and a value. The value is a member of an auxiliary domain of tuples of syntax nodes. For example, the rule

Symbol : : $Symbol_1$ $Symbol_2$

declares

controlled domain *Symbol*
Symbol-aux $=_{def}$ $Symbol_1 \times Symbol_2$
controlled *contents-Symbol: Symbol* $\rightarrow$ *Symbol-aux*

The abbreviation **mk-***Symbol* creates new elements of the domain *Symbol*,

mk-*Symbol* ($s1$: $Symbol_1$, $s2$: $Symbol_2$) $\equiv$
extend *Symbol* **with** *v*
  contents-Symbol(*v*) := ($s1$, $s2$)
endextend

Note that **mk-***Symbol* is not an ASM function, but an abbreviation for an ASM rule.

Selector functions, s-, select the components of the tuple that is the value of a syntax node.

s-*Symbol*$_1$: *Symbol* $\rightarrow$ $Symbol_1$
s-*Symbol*$_2$: *Symbol* $\rightarrow$ $Symbol_2$

s-*Symbol*$_1$*(x: Symbol): Symbol*$_1$ =$_{\text{def}}$ **s-***Symbol*$_1$ *(x.contents-Symbol)*
s-*Symbol*$_2$*(x: Symbol): Symbol*$_2$ =$_{\text{def}}$ **s-***Symbol*$_2$ *(x.contents-Symbol)*

Here is a more elaborate example, illustrating the definitions introduced by a variety of extended BNF constructs.

Symbol : : *Symbol*$_1$ *Symbol*$_2^+$ *Symbol*$_3$ **-set***[Symbol*$_4$*]*
{ Symbol$_1$ *| Symbol*$_2$ *} {Symbol*$_1$ *Symbol*$_2$*}** *{ Symbol*$_1$ *| Symbol*$_2$ *}* **-set**

which is translated to:

Symbol-aux =$_{def}$ *Symbol*$_1$ × *Symbol*$_2^*$ × *Symbol*$_3$ **-set** × *Symbol*$_4$ × *(Symbol*$_1$ ∪ *Symbol*$_2$*)* × *(Symbol*$_1$ × *Symbol*$_2$*)** × *(Symbol*$_1$ ∪ *Symbol*$_2$*)* **-set**
controlled domain *Symbol*
controlled *contents-Symbol*: *Symbol* → *Symbol-aux*
s- *Symbol*$_1$ *(x: Symbol): Symbol*$_1$ =$_{\text{def}}$ **s-** *Symbol*$_1$ *(x.contents-Symbol)*
s- *Symbol*$_2$ **-seq** *(x: Symbol): Symbol*$_2^*$ =$_{\text{def}}$ **s-** *Symbol*$_2$ **-seq** *(x.contents-Symbol)*
s- *Symbol*$_3$ **-set** *(x: Symbol): Symbol*$_3$ **-set** =$_{\text{def}}$ **s-** *Symbol*$_3$ **-set** *(x.contents-Symbol)*
s- *Symbol*$_4$ *(x: Symbol): Symbol*$_4$ =$_{\text{def}}$ **s-** *Symbol*$_4$ *(x.contents-Symbol)*
s-implicit *(x: Symbol): (Symbol*$_1$ ∪ *Symbol*$_2$*)* =$_{\text{def}}$ **s-implicit** *(x.contents-Symbol)*
s-implicit-seq *(x: Symbol): (Symbol*$_1$ × *Symbol*$_2$*)** =$_{\text{def}}$ **s-implicit-seq** *(x.contents-Symbol)*
s-implicit-set *(x: Symbol): (Symbol*$_1$ ∪ *Symbol*$_2$ *)* **-set** =$_{\text{def}}$ **s-implicit-set** *(x.contents-Symbol)*

As with the previous example, an abbreviation **mk-***Symbol* is also introduced. This abbreviation creates a new object of domain *Symbol* using the **extend** primitive and sets the *contents-Symbol* value of the newly produced object to the corresponding element of *Symbol-aux*.

mk- *Symbol(s1: Symbol*$_1$*, s2seq: Symbol*$_2^*$*, s3set: Symbol*$_3$ **-set**, *s4: Symbol*$_4$*, s: (Symbol*$_1$ ∪ *Symbol*$_2$*),*
s1s2seq: (Symbol$_1$ × *Symbol*$_2$*)***, sset: (Symbol*$_1$ ∪ *Symbol*$_2$*)* **-set**) ≡
extend *Symbol* **with** *v*
contents-Symbol(v) := (s1, s2seq, s3set, s4, s, s1s2seq, sset)
endextend

The abbreviation **mk-***Symbol* is not a function, but in fact an ASM rule item. Therefore, it must be used only within ASM rules and not as an ASM location.

If for a given *Symbol*, *sym*, the optional $Symbol_4$ is not present, that fact is expressed in the ASM model by leaving the corresponding value *undefined*: *sym.s-Symbol*$_4$ = *undefined*. An empty sequence of symbols (constructor with no parts) is denoted by (). For example, the abstract syntax rule

Symbol : : ()

declares a singleton domain, *Symbol*, whose only element is the empty tuple.

The equality for syntax nodes is always a structural equality, i.e., the contents of the symbols are compared instead of the symbols themselves.

A rule of the form

Symbol : : *Symbol*$_1$ | *Symbol*$_2$ | ... | *Symbol*$_n$ $(n \geq 1)$

where each alternative right side consists of a single symbol, is equivalent to

Symbol : : *{Symbol*$_1$ | *Symbol*$_2$ | ... | *Symbol*$_n$*}* $(n \geq 1)$

In this case, there are selectors **.s-** $Symbol_i$, $(i = 1, \ldots, n)$ and if $x \in$ *Symbol*, then all but one of those selectors will have the value *undefined*. Moreover, *x.* **s-implicit** = *x.contents-Symbol*.

More usually, rules like this will be represented as named unions. The syntax rules introducing named unions have the form:

Symbol = *Symbol*$_1$ | *Symbol*$_2$ | ... | *Symbol*$_n$ ($n \geq 1$)

which is modelled as:

Symbol =$_{def}$ *Symbol*$_1$ ∪ *Symbol*$_2$ ∪ ... ∪ *Symbol*$_n$

Note that since *Symbol* is a *union* domain, the expansion yields a domain definition, but no abbreviation **mk-***Symbol* and no selector functions **s-** *Symbol*$_i$ ($i = 1, ..., n$) that can be applied to elements of *Symbol*.

Of course, it is still possible to construct elements of *Symbol* using any constructor that generates an element any of the domains *Symbol*$_1$, *Symbol*$_2$, ..., *Symbol*$_n$, and given $x \in$ *Symbol*, an expression of the form $x \in$ *Symbol*$_i$ ($i = 1, ..., n$) will to test whether or not x belongs to one of the constituent domains of *Symbol*.

If a name is not required, unnamed unions may be introduced by rules like:

Symbol : : *Symbol*$_1$ { *Symbol*$_{21}$ | ... | *Symbol*$_{2N}$ }

instead of introducing a name for the union:

Symbol : : *Symbol*$_1$ *Symbol*$_2$
Symbol$_2$ = *Symbol*$_{21}$ | ... | *Symbol*$_{2N}$

Named sequences can be introduced using rules like this:

Symbol = *Symbol*$_1^*$

which defines the domain of sequences of elements of *Symbol*$_1$. Similarly, a rule like

Symbol = { *Symbol*$_1$ *Symbol*$_2$}*

defines the domain of sequences of tuples *(a,b)* where $a \in$ *Symbol*$_1$ and $b \in$ *Symbol*$_2$

As with named unions, *Symbol* yields a domain definition, but not functions **mk-** nor **s-**. Elements of *Symbol* are constructed using the sequence constructor, $< >$, and components of *Symbol* can be accessed using the functions on sequences.

It is not always necessary to name a sequence explicitly. Unnamed sequences may be introduced by rules like:

Symbol : : *Symbol*$_1$ { *Symbol*$_{21}$ ... *Symbol*$_{2N}$ }*

or like:

Symbol : : *Symbol*$_1$ { *Symbol*$_{21}$ | ... | *Symbol*$_{2N}$}*

Named and unnamed sets can be introduced in a similar way.

For each SDL-2010 keyword **KEYWORD**, there is an associated keyword domain *Keyword* with just one value:

static domain *Keyword*

It is required that all keyword domains are mutually disjoint.

Given the abstract grammar, there is a derived domain called *DefinitionAS1*, which is composed of all abstract syntax symbol domains as follows:

DefinitionAS1 =$_{def}$ *Symbol*$_1$ ∪ *Symbol*$_2$ ∪ ... ∪ *Symbol*$_n$

where *Symbol*$_1$, *Symbol*$_2$, ..., *Symbol*$_n$ are *all* the symbols (terminal and non-terminal) of the abstract grammar. This includes unnamed non-terminals that appear in the syntax such as *Graph-node** and

{Terminator | Decision-node} in the AS1 rule *Transition*, and *{Open-range | Closed-range}** in the AS1 rule *Size-constraint*, which in ASM are denoted (respectively) as *Graph-node* **-seq**, *(Terminator ∪ Decision-node)* and *(Open-range ∪ Closed-range)* **-seq**.

There is a similar domain *DefinitionAS0* for the concrete grammar (AS0).

F1.3.6.14. Abstract syntax tree

The syntax nodes declared by abstract syntax rules can be used to construct an abstract syntax tree.

To navigate downward in a given abstract syntax tree, the functions **s-** can be used. To navigate upward, two parent functions are defined.

controlled *parentAS1: DefinitionAS1 → DefinitionAS1*
controlled *parentAS0: DefinitionAS0 → DefinitionAS0*

Suppose *node* is an abstract syntax node and a member of the domain *Symbol*, one of the constituent domains of *DefinitionAS1*:

node ∈ Symbol ⊂ DefinitionAS1

The parent of *node*, if it exists, includes *node*, in its *contents-Symbol*. That is,

parentAS1 (node: DefinitionAS1): DefinitionAS1 $=_{\text{def}}$
  if *node = undefined* **then** *undefined*
  elseif *node = rootNodeAS1* **then** *undefined*
  else *take* ({ *p ∈ Symbol ⊂ DefinitionAS1: node* **in** *p.contents-Symbol* })
  endif

Similarly, for *node ∈ Symbol ⊂ DefinitionAS0*:

parentAS0 (node: DefinitionAS0): DefinitionAS0 $=_{\text{def}}$
  if *node = undefined* **then** *undefined*
  elseif *node = rootNodeAS0* **then** *undefined*
  else *take* ({ *p ∈ Symbol ⊂ DefinitionAS0*: *node* **in** *p.contents-Symbol* })
  endif

Moreover, two functions are defined to find the parent of a particular kind.

parentAS0ofKind (from: DefinitionAS0, x: DefinitionAS0 **-set**): *DefinitionAS0* $=_{\text{def}}$
  if *from = undefined* **then** *undefined*
  elseif *from ∈ x* **then** *from*
  else *parentAS0ofKind (from.parentAS0, x*)
  endif

parentAS1ofKind (from: DefinitionAS1, x: DefinitionAS1 **-set**): *DefinitionAS1* $=_{\text{def}}$
  if *from = undefined* **then** *undefined*
  elseif *from ∈ x* **then** *from*
  else *parentAS1ofKind (from.parentAS1, x*)
  endif

The top node of the current abstract or concrete syntax tree is denoted by the following 0-ary functions:

controlled *rootNodeAS1: → DefinitionAS1*
controlled *rootNodeAS0: → DefinitionAS0*

The value of *rootNodeAS1.parentAS1* is *undefined*.

The value of *rootNodeAS0.parentAS0* is *undefined*.

The functions *isAncestorAS1* and *isAncestorAS0* determine if the first node is an ancestor of the second one:

isAncestorAS1 (n1: DefinitionAS1, n2: DefinitionAS1): BOOLEAN $=_{\text{def}}$
  n1 = n2.parentAS1 ∨ (n2 ≠ rootNodeAS1 ∧ isAncestorAS1 (n1 , n2.parentAS1))

isAncestorAS0 (n1: DefinitionAS0, n2: DefinitionAS0): BOOLEAN $=_{\text{def}}$
n1 = n2.parentAS0 ∨ (n2 ≠ rootNodeAS0 ∧ isAncestorAS0 (n1, n2.parentAS0))

The functions *isSameNodeAS1* and *isSameNodeAS0* determine if the first node is the same node as the second one, that is they are equal and in the same position in the syntax tree:

isSameNode1 (n1: DefinitionAS1, n2: DefinitionAS1): BOOLEAN $=_{\text{def}}$
n1 = n2 ∧ (n1 = rootNodeAS1 ∨ isSameNode1 (n1.parentAS1, n2.parentAS1))

isSameNode0 (n1: DefinitionAS0, n2: DefinitionAS0): BOOLEAN $=_{\text{def}}$
n1 = n2 ∧ (n1 = rootNodeAS0 ∨ isSameNode0 (n1.parentAS0, n2.parentAS0))

The abstract syntax tree AS0 can be modified using the following derived functions:

replaceInSyntaxTree0: DefinitionAS0 × DefinitionAS0 × DefinitionAS0 → DefinitionAS0

The first parameter of the function is the old sub-tree, the second one is the new sub-tree and the third parameter is the old tree. The function returns the new tree, where all old sub-trees are replaced by the new sub-tree.

replaceInSyntaxTree0: DefinitionAS0 × DefinitionAS0 × DefinitionAS0 → DefinitionAS0

The first parameter of the function is the old sub-tree, the second one is the new sub-tree and the third parameter is the old tree. The function returns the new tree, where the first old sub-tree is replaced by the new sub-tree, and each subsequent old sub-tree occurrence is unchanged.