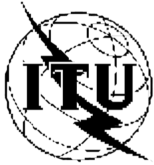


INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.100
Annex F1
(11/2000)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

CAUTION !

PREPUBLISHED RECOMMENDATION

This prepublication is an unedited version of a recently approved Recommendation. It will be replaced by the published version after editing. Therefore, there will be differences between this prepublication and the published version.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications. 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.

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/had not] received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

© ITU 2000

All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the ITU.

1 Preface ... 2
1.1 Motivation ... 2
1.2 Main Objectives ... 2
1.3 References ... 2
1.4 Bibliography ... 3
2 Overview of the Semantics ... 3
2.1 Grammar ... 4
2.2 Well-formedness Conditions ... 4
2.3 Transformation Rules ... 4
2.4 Dynamic Semantics ... 5
3 Abstract State Machines ... 6
3.1 Basic ASM Model ... 6
3.1.1 Vocabulary ... 6
3.1.2 States ... 8
3.1.3 Derived Names ... 8
3.1.4 Initial States ... 8
3.1.5 State Transitions and Runs ... 9
3.1.6 Transition Rules ... 9
3.1.7 Abbreviations ... 11
3.1.8 ASM Programs ... 12
3.2 Distributed ASM ... 13
3.2.1 Vocabulary ... 13
3.2.2 Agents and Runs ... 14
3.2.3 Distributed ASM Programs ... 15
3.3 The External World ... 16
3.4 Real-time Behaviour ... 18
3.5 Example: The System *RMS* ... 18
3.6 Predefined Names ... 20

1 PREFACE

This formal definition of SDL provides a precise language definition supplementing the definition given in the recommendation text. It is for use by those requiring a very precise definition of SDL such as maintainers of the SDL language, designers of SDL tools, and users of the SDL language.

The formal definition consists of the three volumes:

Annex F.1 (this volume)

This part provides motivation, 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 semantics.

Annex F.2 This part describes the static semantic constraints, and transformations as identified by the Model sections of Z.100.

Annex F.3 This part defines the dynamic semantics of SDL.

1.1 Motivation

Specifications in natural languages are ambiguous, that is, more than one interpretation can be given to them. A specification is formal if its meaning (semantics) is unambiguous. Special languages, known as formal description techniques (FDTs), have been developed for this purpose. FDTs are distinguished from formal languages in general by the fact that they have a formal syntax *and* a formal semantics. This is different from most formal languages such as Java or C++, which have a formal syntax only.

The formal semantics of a language is defined in terms of an underlying mathematical formalism, e.g. axiomatically or operationally. The choice of an appropriate formalism is influenced by the expressiveness of the FDT as well as by the objectives of the semantics beyond unambiguity. The formal semantics defines, for each specification, a corresponding mathematical model that captures its meaning precisely and completely.

This annex defines the semantics of SDL formally. If there is an inconsistency between the main body of Z.100 and Annex F, then there is an error that needs correction. Neither the main body of Z.100 nor the Annex F take precedence in this case.

1.2 Main Objectives

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

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

SDL can be classified as a model-oriented FDT for the specification of distributed and concurrent systems, which means that an SDL 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 to improve its intelligibility. In addition, an operational semantics lends itself naturally to executability, given the availability of tools, which is another explicit objective.

1.3 References

ITU-T Recommendation Z.100. *Languages for Telecommunication Applications – Specification and Description Language*. International Telecommunication Union (ITU), Geneva, 1999.

1.4 Bibliography

[1] http://www.uni-paderborn.de/cs/asm/ and http://www.eecs.umich.edu/gasm/.

[2] R. Eschbach, U. Glässer, R. Gotzhein and A. Prinz. On the Formal Semantics of SDL-2000: a Compilation Approach Based on an Abstract SDL Machine. In Abstract State Machines - Theory and Applications. Y. Gurevich, P.W. Kutter, M. Odersky and L. Thiele , editors, vol. 1912 of LNCS, Springer-Verlag, 2000.

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

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

2 OVERVIEW OF THE SEMANTICS

In order to define the formal semantics of SDL, the language definition is decomposed into several parts:

- the grammar,
- the well-formedness conditions,
- the transformation rules, and
- the dynamic semantics.

Starting point for defining the formal semantics of SDL is a syntactically correct SDL 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 (see Figure 1).

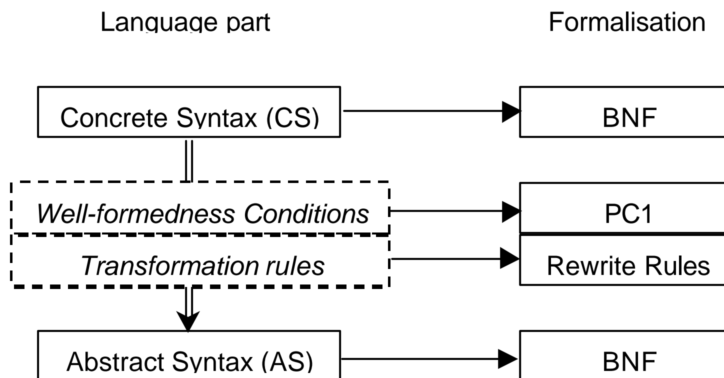


Figure 1: Static aspects of SDL

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 extensions to capture the graphical language constructs. 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 *well-formedness conditions* 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, or which kind of values it is allowed to assign to variable. 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 Z.100, and are formally expressed 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

The *grammar* of SDL is formalised using a grammar representation in BNF (Backus-Naur-Form). However, the grammar in Z.100 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 can not 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 representation to the abstract syntax representation of Z.100 (called AS1) consists of two steps. The first step from the concrete textual SDL representation to the abstract syntax AS0 is not formally defined, but is derived from the correspondence between the two grammars, which is almost one-to-one, and removes irrelevant details such as separators and lexical rules. The second step, translating AS0 to AS1, is formally captured by a set of transformation rules (see Annex F.2).

2.2 Well-formedness Conditions

The *well-formedness conditions* define additional constraints that a specification has to satisfy. These constraints can not be expressed in BNF, but are static, i.e., they can be defined and checked independent of the dynamic semantics definition (see Annex F.2). An SDL specification is *valid* if and only if it satisfies the syntactical rules and the static conditions of SDL. In fact, the well-formedness conditions refer to the syntax, but they have not been stated in the concrete syntax because they are not expressible in a context free grammar.

There are basically five kinds of well-formedness conditions:

- *Scope/visibility rules*: the definition of an entity introduces an identifier that may be used as the reference to the entity. Only visible identifiers must be used. 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 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.
- *Special rules*: there are some rules applicable to specific entities. For example, there be local blocks or a graph within a block.
- *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 name at the beginning and at the end of a definition have to match.

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 that are introduced for convenience, but do not add to the expressiveness of the language (such as shorthand notations), can be replaced using these core concepts. Since replacements, which are described by transformation rules, can be formalised, it suffices to define the dynamic semantics only for the core concepts, which adds to its conciseness and intelligibility.

For the formal semantics definition, the choice of the “right” core concepts is crucial. If there are too many core concepts, the dynamic semantics will be less concise and intelligible. If there are too few or the wrong concepts, the transformations tend to be very complex. In the scope of SDL, object-orientation was introduced in 1992, but still the (formal) semantics definition and the abstract syntax relied on instances and had no notion of class. The result was a very cumbersome transformation. This is rectified in this formal semantics definition.

Z.100 prescribes the transformation of SDL specifications by a sequence of *transformation steps*. Each transformation step consists of a set of single transformations as stated in the Model sections, 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 formalise the transformation rules of Z.100, rewrite rules are used. A single transformation is realised 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 F.2).

2.4 Dynamic Semantics

The *dynamic semantics* (Annex F.3) consists of the following parts (see Figure 2):

- The *SDL Abstract Machine (SAM)*, which is defined using ASM. The definition of the SAM is divided into three parts, corresponding to the abstract syntax: (1) basic signal flow concepts (signals, timers, exceptions, gates, channels), (2) various types of ASM agents (modelling corresponding SDL agents), and (3) behaviour primitives (SAM instructions).
- The *compilation*, defined as a function over the abstract syntax tree of an SDL specification. Result of the compilation are sets of behaviour primitives, modelling the actions of the SDL agents.
- The *initialisation*, defining a pre-initial state of a system and several initialisation programs. The initial system state is then reached by creating the SDL system agent, and by activating this agent in the pre-initial state. The initialisation recursively unfolds the static structure of the system, creating further SDL agents as specified. In fact, the same process will be initiated in the subsequent execution phase, whenever SDL agents are created. From this point of view, the initialisation merely describes the instantiation of the SDL system agent.
- The *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 built-in model. Moreover, also the built-in model can be changed this way without affecting the rest of the semantics.

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

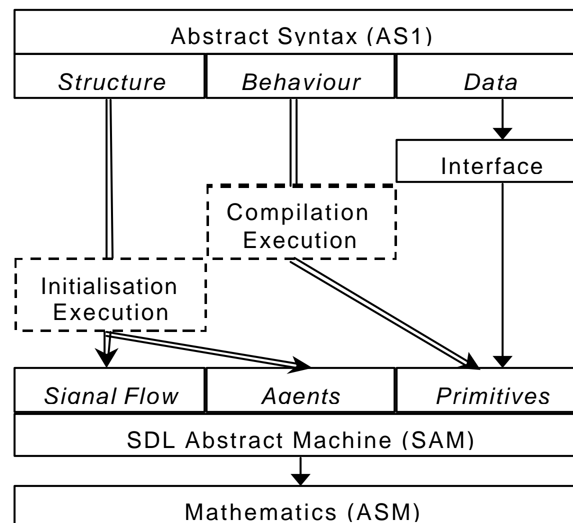


Figure 2: Overview of the Dynamic Semantics

The *dynamic semantics* associates, with each SDL specification, a particular distributed real-time ASM. Intuitively, this consists of a set of autonomous agents cooperatively performing concurrent machine runs. The behaviour of an agent is determined by an ASM program consisting of a transition rule. Collectively, these rules define the set of possible machine 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 modelled. An introduction to the ASM model, and the notation used in Annex F, is given in Section 3.

3 ABSTRACT STATE MACHINES

This section explains basic notions and concepts of *Abstract State Machines* (*ASM*) as well as the notation used in this document to define the SDL 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, the reader is referred to [3] and [4]. A discussion and motivation of the appropriateness of the semantic framework used here is given in [2]. Further references on ASM-related material can also be found on the ASM Web Pages [1].

The ASM model used to define the dynamic semantics of SDL is explained in several steps. Firstly, the *basic ASM model* with a single agent is treated (Section 3.1). Next, this model is extended to cover *multi-agent systems* (Section 3.2). Then, *open systems*, i.e. systems interacting with an environment they cannot control, are addressed by adding the notion of *external world* (Section 3.3). Finally, the model is extended by introducing a notion of *real-time behaviour* (Section 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 summarised in Section 3.5. Additional notation used to define the dynamic semantics of SDL is explained in Section 3.6.

EXAMPLE (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, some device or service) is defined. Informally, this system is characterised 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* 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 summarised in Section 3.5.

3.1 Basic ASM Model

An *Abstract State Machine M* is defined over a given *vocabulary V* by its *states S*, its *initial states* $S_0 \subseteq S$, and its *program P*. These items will be explained in the following subsections.

3.1.1 Vocabulary

The vocabulary (or signature) *V* denotes a finite set of *function names*, *predicate names*, and *domain names*, each of a fixed arity. Names in *V* are classified as *basic* or *derived*, and further distinguished into *static* or *dynamic* (see Figure 3). The meaning associated with these classifications will be explained in subsequent subsections.

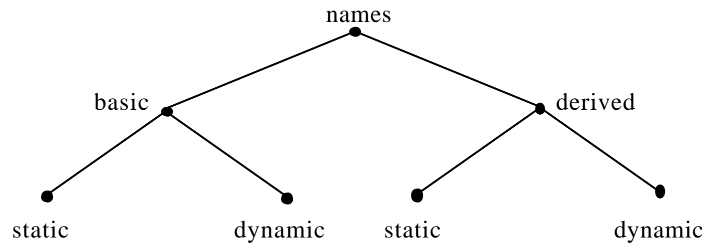


Figure 3: Classification of ASM Names

V is declared when defining an ASM, except for a subset of *predefined names*. This subset includes, for instance, the equality sign, the 0-ary predicate names *True*, *False*, the 0-ary function name *undefined*, the domain names *BOOLEAN*, *NAT* and *REAL*, as well as the names of frequently used standard functions (such as Boolean operations $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$, and set operations $\subseteq$, $\cup$, $\cap$, $\in$, $\notin$, etc.). Predefined names are listed in Section 3.6.

To *declare* names when defining a concrete ASM, we use the following notational conventions:

- *Domain names* are written in capitalised italics (as in *AGENT*), except when denoting a non-terminal of the abstract grammar. Here, domain names are written as the non-terminals, i.e. in italics, hyphenated, and starting with a capital (as in *AgentDefinition*). A domain name *D* is declared by **domain** *D*.
- *Function names* are written in italics starting with a small letter (as in *mode*). A function name *f* is declared by *f*: $D_1 \times D_2 \times ... \times D_n \rightarrow D_0$, where *n* is the arity of *f*, and $D_0, D_1, D_2, ..., D_n$ are domain names.
- *Predicate names* are also written in italics, but starting with a capital letter (as in *Available*). A predicate name *P* is declared by *P*: $D_1 \times D_2 \times ... \times D_n \rightarrow$ *BOOLEAN*.
- *Basic static names* are qualified by the keyword **static**, when they are declared (see Figure 3).
- *Basic dynamic names* are qualified by one of the keywords **controlled**, **shared**, or **monitored**, when they are declared (as will be explained in Section 3.3).
- Names without a preceding keyword are *derived names* by default (see Figure 3).

EXAMPLE (Vocabulary):

To define an ASM model of the system *RMS*, assume a vocabulary *V* including the following names:

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

shared *mode*: *AGENT* $\rightarrow$ *MODE*
controlled *owner*: *TOKEN* $\rightarrow$ *AGENT*
static *ag*: $\rightarrow$ *AGENT*

Idle: *AGENT* $\rightarrow$ *BOOLEAN*
Waiting: *AGENT* $\rightarrow$ *BOOLEAN*
Busy: *AGENT* $\rightarrow$ *BOOLEAN*
Available: *TOKEN* $\rightarrow$ *BOOLEAN*

monitored *Stop*: *AGENT* $\rightarrow$ *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.

3.1.2 States

A state $s \in S$ is given by assigning a meaning, also called *interpretation*, to the names in *V* over an infinite set, called the *base set of M* (to which we refer by the predefined domain name *X*).[1] That is, to each domain name, function name and predicate name in *V* a basic domain, function or predicate is to be associated , respectively. The interpretation of derived names follows from the interpretation of basic names. Note that the base set is the same for all states of *M*. It is required that *True*, *False* and *undefined* denote distinct elements of the base set. Predefined operations have their usual interpretation.

Recall that names are classified as static or dynamic. If classified as static, names are required to have the same interpretation in all states of *M*. Otherwise, they may have different interpretations in different states of *M*. Thus, the states *S* of *M* are given by the set of all interpretations of the names in *V* over the base set of *M* that comply with these and other explicitly stated constraints.

Strictly speaking, all functions are *total* functions on the base set of *M*. To imitate *partial functions*, "undefined" function values are marked by the distinguished element *undefined*. Predicates only yield one of the values *True* or *False*, i.e., they must not be partial.

Every state has a potentially infinite number of *reserve elements* allowing to extend domains dynamically (see Section 3.1.6). By definition, the reserve elements of a state are all those elements of the base set that are neither identified by a function nor contained in one of the domains.

3.1.3 Derived Names

The meaning of derived names follows from the interpretation of basic names, and is defined in terms of *formulae* (see Section 3.6); derived names may therefore be understood as abbreviations. 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 (Definitions):

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

For an improved readability, we use a *"."-notation* for unary functions and predicates. For instance, we write *a.mode*, which is equivalent to writing *mode*(*a*).

3.1.4 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 section 3.6. Initial constraints have the following general form:

initially *ClosedFormula*

[1] Formally speaking, ASM states are (*many-sorted*) *first-order structures*.

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.

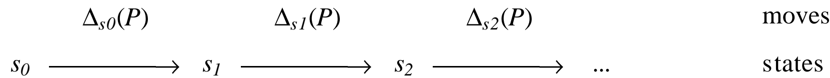
3.1.5 State Transitions and Runs

Recall that 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 $loc_s = \langle f, s(x)\rangle$, 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 $\delta_s = \langle loc_s, s(y)\rangle$, 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 Sections 3.1.6 and 3.1.8). 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 $\delta_s = \langle loc_s, s(y)\rangle$, $\delta_s = \langle loc_s, s(y')\rangle$, 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 $loc_s = \langle f,s(x)\rangle$ of *s*, $f_{s'}(s(x)) = s(y)$, if $\langle\langle f,s(x)\rangle,s(y)\rangle \in \Delta_s(P)$, and $f_{s'}(s(x)) = f_s(s(x))$ otherwise, and is called *state transition*. Firing an inconsistent update set[2] has no effect, i.e., $s' = s$.

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



$$s_0 \xrightarrow{\Delta_{s0}(P)} s_1 \xrightarrow{\Delta_{s1}(P)} s_2 \xrightarrow{\Delta_{s2}(P)} \ldots$$

moves
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 Section 3.1.8). 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.*

3.1.6 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

  $Rule ::= \quad f(t_1,...,t_n) := t_0 \qquad (n \geq 0)$

  Here, *f* is a non-static name of *V* denoting either a controlled or a shared function, predicate or domain, and $t_0,t_1,...,t_n$ are terms over *V* identifying, for a given state *s*, the location $loc = \langle f,\langle s(t_1),..., s(t_n)\rangle\rangle$ 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 $\{\langle\langle f,\langle s(t_1),..., s(t_n)\rangle\rangle, s(t_0)\rangle\}$, 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.

---

[2] In the context of the SDL 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.

EXAMPLE (Update Instruction):

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

$t.owner := ag$ specifies the update set {<<*owner*, <*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 semantics, six different constructors 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
              Rule1
          [else
              Rule2]
          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]
              Rule1
              ...
              Rulen
          [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)
              Rule0(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, $\Delta_s(Rule)$ is the union of all update sets $\Delta_s(Rule_0(v))$ 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)
              Rule0(v)
          endchoose
```

The effect of *Rule* is that $Rule_0$ is fired for some 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, $\Delta_s(Rule)$ is some update set $\Delta_s(Rule_0(v))$ such that $g(v)$ holds in state s, or the empty update set if no such v exists.

- *extend*-constructor[3]

  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 Section 3.1.2) are imported into the dynamic domain *D* (while being removed from the reserve), that $v_1,...,v_n$ become bound to one of the imported elements each, and then $Rule_0(v_1,...,v_n)$ 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 initialised.

- *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 (Transition Rule):

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 $ag.mode = shared \wedge ag.Waiting$ **then**
 choose t: $t \in TOKEN \wedge t.Available$
  $t.owner := ag$
 endchoose
endif

The precise meaning of the rule is given by its update set with respect to a state *s*, which is either {<<*owner*, <*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.

3.1.7 Abbreviations

Rules can be structured using *abbreviations*, consisting of *rule macros* and *derived names*, that 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 Sections 3.1.1 and 3.1.3, 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 $v_1,...,v_n$ of domains $D_1,...,D_n$, $n \geq 0$. The general form of a rule macro definition is:

  RuleMacroDefinition ::= $RuleMacroName(v_1{:}D_1,...,v_n{:}D_n) \equiv$
  $Rule_0(v_1,...,v_n)$

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

- *where*-part

[3] Strictly speaking, *extend* can be defined in terms of the *import* constructor (not shown here); however, the *import* constructor is not used in this Recommendation.

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 parameterised 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 $ag.mode = shared \wedge ag.Waiting$ **then**
GETTOKEN(*ag*)
endif
where
GETTOKEN(*a*:*AGENT*) ≡
choose t: $t \in TOKEN \wedge t.Available$
$t.owner := a$
endchoose
endwhere

3.1.8 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. In the next section, we will allow to define several ASM programs, and associate them 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.

3.2 Distributed ASM

Mathematical modelling of concurrent and reactive systems requires to extend the basic ASM model. In this section, the concept of *distributed ASM*, which generalises the basic ASM model presented in Section 3.1, is explained.

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 subsections, as far as they differ from the basic ASM model.

3.2.1 Vocabulary

The 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. Furthermore, *V* includes a distinguished function name

controlled *program*: $AGENT \rightarrow PROGRAM$

and a special 0-ary function *Self* (see Section 3.2.2).

3.2.2 Agents and Runs

A distributed ASM *M* may have any finite number of agents, where this number may vary dynamically depending on the given state. The behaviour of each agent is determined by some program of *M*, defined by a transition rule like in the basic ASM model. Agents operate concurrently by running their programs, and interact asynchronously through globally shared locations of a state, i.e. two or more agents may read and write the same location. Concurrent execution steps of the distributed ASM model are restricted to independent operations, where the admissible behaviour is defined in terms of *partially ordered runs* (see [3]). Intuitively, this notion of concurrency allows for true concurrency instead of approximating concurrency by an interleaving model.

To assign a behaviour to an agent of *M*, the distinguished function *program* (see Section 3.2.1) yields, for each agent *a* of *M*, the program of *P* to be executed by *a*. The function *program* thus allows to define (or to redefine) 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 (as defined by some program of *P*) to be associated with *a*.

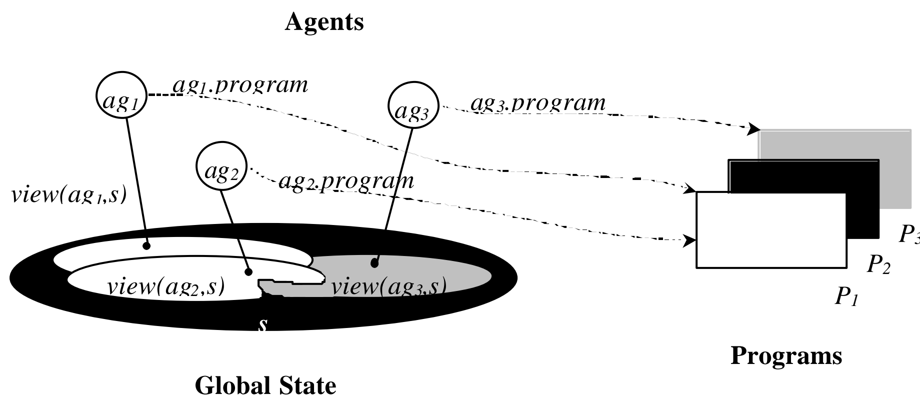
A special 0-ary function *Self* serves as a *self reference* identifying the respective agent calling *Self*:

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 distibuted 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 which

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 (adopted from [3]).

A run ρ of a distributed ASM *M* is given by a triple (Λ,*A*,σ) satisfying the following four conditions:

1. Λ is a partially ordered set of moves, where each move has only finitely many predecessors;
2. *A* is a function on Λ associating agents to moves such that the moves of any single agent of *M* are linearly ordered;
3. σ assigns a state of *M* to each initial segment *Y* of Λ, where σ(*Y*) is the result of performing all moves in *Y*; if *Y* is empty, then $\sigma(Y) \in S_0$;
4. if *y* is a maximal element in a finite initial segment *Y* of Λ and *Z* = *Y* – { *y* }, then *A*(*y*) is an agent in σ(*Z*) and σ(*Y*) is obtained from σ(*Z*) by firing *A*(*y*) at σ(*Z*) (*coherence condition*).

Implications

Partially ordered runs have certain characteristic properties that can be stated in terms of *linearisations* of partially ordered sets. A linearisation of a partially ordered set Λ is a linearly ordered set Λ' with the same elements such that if *y* < *z* in Λ then *y* < *z* in Λ'. Accordingly, the semantic model of concurrency as implied by the notion of partially ordered run can further be characterised as follows [3]:

- All linearisations 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 linearisation of ρ.

3.2.3 Distributed ASM Programs

A distributed *ASM M* has a finite set *P* of programs. Each program *p* **Î** *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[4]. Programs are stated in the following form:

ASM-PROGRAM:

Rule

Program names are, by convention, hyphenated and written in small capitals, with a leading capital 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.

[4] Strictly speaking, the program names of *M* are represented by a distinguished set of elements from the base set.

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 to associate the program with different agents, while accessing the local state of these agents.

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 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.

- *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.

- *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.

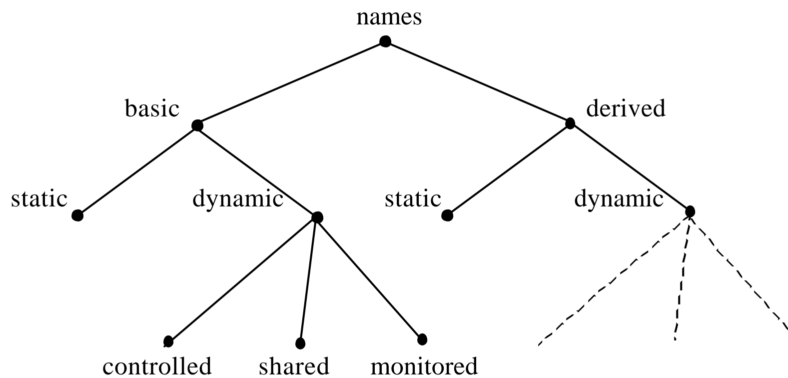


Figure 4: Extended classification of ASM names

EXAMPLE (External World):

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

shared *mode*: $AGENT \rightarrow MODE$
controlled *owner*: $TOKEN \rightarrow AGENT$

monitored *Stop*: $AGENT \rightarrow BOOLEAN$

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 Section *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 (Integrity Constraints):

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

constraint $\forall a \in AGENT: (a.Stop \Rightarrow a.Busy)$

3.4 Real-time Behaviour

By introducing a notion of *real time* and imposing additional constraints on runs, we obtain a specialised 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*

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*, where $\sigma(t)$ denotes the reduct[5] of $R(t)$ to V:

1. for every $t \geq 0$, *currentTime* evaluates to t at state $R(t)$;
2. 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 $\sigma(\alpha) = \sigma(\beta)$.

Exploiting the discreteness property, one effectively obtains some finite representation (*history*) for every finite (sub-) run by abstracting from those states which are not considered as significant such that they contribute any relevant information to a behaviour description. In particular, one can simply ignore all states which 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.

3.5 Example: The System *RMS*

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

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, some device or service, is defined. Informally, this system is characterised 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.

[5] That is, for a given value t, we obtain $\sigma(t)$ from $R(t)$ by ignoring the interpretation of the function name *currentTime*.

Vocabulary

static domain *TOKEN*

shared *mode*: *AGENT* ® *MODE*
controlled *owner*: $TOKEN \rightarrow AGENT$

monitored *Stop*: $AGENT \rightarrow BOOLEAN$

Derived Names

$MODE =_{\text{def}} \{exclusive, shared\}$

$Idle(a{:}AGENT){:}\ BOOLEAN$	$=_{\text{def}}$	$a.mode = undefined \wedge \forall t \in TOKEN{:}\ t.owner \neq a$
$Waiting(a{:}AGENT){:}\ BOOLEAN$	$=_{\text{def}}$	$a.mode \neq undefined \wedge \forall t \in TOKEN{:}\ t.owner \neq a$
$Busy(a{:}AGENT){:}\ BOOLEAN$	$=_{\text{def}}$	$a.mode \neq undefined \wedge \exists t \in TOKEN{:}\ t.owner = a$
$Available(t{:}TOKEN){:}\ BOOLEAN$	$=_{\text{def}}$	$t.owner = undefined$

Integrity Constraints

constraint $\forall a \in AGENT{:}\ (a.Stop \Rightarrow a.Busy)$

Initial Constraints

initially $|AGENT| > 1$
initially $|TOKEN| < |AGENT|$
initially $\forall a \in AGENT{:}\ a.program =$ RESOURCE-MANAGEMENT-PROGRAM
initially $\forall a \in AGENT{:}\ a.Idle \wedge \forall t \in TOKEN{:}\ t.Available$

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
```

3.6 Predefined Names

To define an ASM model, in particular the ASM model capturing the semantics of SDL, certain names and their intended interpretation are predefined. These names are grouped and listed in this section (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.

ASM-specific Domains	
static domain *X*	ASM base set (meta domain)
static domain *BOOLEAN*	Boolean values
static domain *NAT*	Integer values
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

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.

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

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-**$Fun(x) =_{def} take(\{ a \in D: Fun(a) = x \})$

Functions and Relations on Integers	
_ > _, _ ≥ _, _ <_ , _ ≤ _ prec(4)	Comparison operators
_ + _ prec(6)	Addition
_ - _ prec(6)	Subtraction
_ *_ prec(7)	Multiplication
_ / _ prec(7)	Division
0, *1*, ...	Integer literals

Functions on Sequences	
static *empty*: → D^*	Empty sequence
static *head*: D^* → D	Head of the sequence (*undefined* when empty)
static *tail*: D^* → D^*	Tail of the sequence (*undefined* when empty)
static *last*: D^* → D	Last element of a sequence (*undefined* when empty)
static *length*: D^* → *NAT*	Length of a sequence
static < >: D^n → D^*	Sequence constructor; arguments are listed inside the brackets, separated by commas
_ ⁀ _ prec(6)	Concatenation of sequences
toSet: D^* → *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?
< <*result*> $\mid$ <*var*> **in** <*seq*> : <*cond*> >	Sequence comprehension; acts like a filter on <seq>, i.e. order-preserving
< <*var*> **in** <*seq*> : <*cond*> > $=_{def}$ < <*var*> $\mid$ <*var*> **in** <*seq*> : <*cond*> >	Abbreviated sequence comprehension
< <*result*> $\mid$ <*var*> **in** <*seq*> > $=_{def}$ < <*result*> $\mid$ <*var*> **in** <*seq*> : *True* >	Abbreviated sequence comprehension

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?
$\mid$ _ $\mid$	Size of a set
U _	Big union: union of all sets included within the argument set
∅	Empty set
static { }: D^n → *D*-**set**	Set constructor; comma-separated list of arguments in the brackets
take: *D*-**set** → *D*	Select an arbitrary element from the set, or *undefined* for an empty set
_ .. _ prec(5)	Integer range from the first value to the second. Empty set when the second expression is smaller than the first one.
{ <*result*> $\mid$ <*var*> ∈ <*set*> : <*cond*> }	Set comprehension, acts like a filter on <set>
{ <*var*> ∈ <*set*> : <*cond*> } $=_{def}$ { <*var*> $\mid$ <*var*> ∈ <*set*>: <*cond*> }	Abbreviated set comprehension
{ <*result*> $\mid$ <*var*> ∈ <*set*> } $=_{def}$ { <*result*> $\mid$ <*var*> ∈ <*set*>: *True* }	Abbreviated set comprehension

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 a 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**
| $Pattern_1$: $Term_1$
| $Pattern_2$: $Term_2$
...
[**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*.

Union Domains

Union domains simply contain the values of their constituent domains.

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

Tuples

For every declared tuple domain, several implied constructor and selector functions are defined. A definition

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

also defines the following functions:

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

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

Abstract Syntax Rules

Abstract syntax rules from the language definition are directly translated to the ASM notation, using certain conventions that will be explained by examples. Basically, an abstract syntax rule can be understood as declaring one or more (tuple) domains, and defining functions to construct and select values of the component domains. However, syntax nodes have an identity as opposed to ordinary tuples. There are syntax rules introducing named constructors as well as named and unnamed unions. Rules introducing constructors are composed of terminal and non-terminal symbols, they have the form

$Symbol :: Symbol_1\ \ Symbol_2^+\ \ Symbol_3\textbf{-set}\ \ [Symbol_4]$

which is translated to

$Symbol\text{-}aux =_{def} Symbol_1 \times Symbol_2^* \times Symbol_3\textbf{-set} \times Symbol_4$
controlled domain *Symbol*
controlled *contents-Symbol*: *Symbol* → *Symbol-aux*
$\textbf{s-}Symbol_1(x: Symbol): Symbol_1 =_{def} \textbf{s-}Symbol_1(x.contents\text{-}Symbol)$
$\textbf{s-}Symbol_2\textbf{-seq}(x: Symbol): Symbol_2^* =_{def} \textbf{s-}Symbol_2\textbf{-seq}(x.contents\text{-}Symbol)$
$\textbf{s-}Symbol_3\textbf{-set}(x: Symbol): Symbol_3\textbf{-set} =_{def} \textbf{s-}Symbol_3\textbf{-set}(x.contents\text{-}Symbol)$
$\textbf{s-}Symbol_4(x: Symbol): Symbol_4 =_{def} \textbf{s-}Symbol_4(x.contents\text{-}Symbol)$

Moreover, there is an abbreviation **mk-***Symbol*. This abbreviation amounts to creating a new object of domain *Symbol* using the **extend** primitive and to set the *contents-Symbol* value of the newly produced object to the result of **mk-***Symbol-aux*. Note that this kind of abbreviation is not a function, but in fact a rule item. Therefore, it must be used only within rules. The fact that the optional $Symbol_4$ is not present is expressed in the ASM model by leaving the corresponding value *undefined*.

An empty sequence of symbols (constructor with no parts) is denoted by ( ).

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

The syntax rules introducing named unions, i.e., synonyms, have the form

$$Symbol = Symbol_1 \mid Symbol_2 \mid ... \mid Symbol_n \qquad (n \geq 1)$$

which is translated to

$$Symbol =_{def} Symbol_1 \cup Symbol_2 \cup ... \cup Symbol_n$$

Note that since *Symbol* is a *union* domain, the expansion yields a domain definition, but no functions **mk-** or **s-**.

In some cases, it is not necessary to refer to synonyms. Here, unnamed unions may be introduced by

$$Symbol :: Symbol_1 \ \{ \ Symbol_{21} \mid ... \mid Symbol_{2n} \ \}$$

instead of introducing synonyms:

$Symbol :: Symbol_1 \ Symbol_2$
$Symbol_2 = Symbol_{21} \mid ... \mid Symbol_{2n}$

For each SDL 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:

$$DefinitionAS1 =_{def} Symbol_1 \cup Symbol_2 \cup ... \cup Symbol_n$$

where $Symbol_1, Symbol_2, ..., Symbol_n$ is the list of *all* terminal and non-terminal symbols of the abstract grammar.

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

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*

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

$parentAS0ofKind(from: DefinitionAS0, x: DefinitionAS0\textbf{-set}): DefinitionAS0 =_{def}$
if $from = undefined$ **then** $undefined$
elseif $from \in x$ **then** $from$
else $parentAS0ofKind(from.parentAS0, x)$
endif
$parentAS1ofKind(from: DefinitionAS1, x: DefinitionAS1\textbf{-set}): DefinitionAS1 =_{def}$
if $from = undefined$ **then** $undefined$
elseif $from \in x$ **then** $from$
else $parentAS1ofKind(from.parentAS1, x)$
endif

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

$isAncestorAS1(n: DefinitionAS1 ,n': DefinitionAS1): \textit{BOOLEAN} =_{def}$
$n = n'.parentAS1 \vee isAncestorAS1(n, n'.parentAS1)$

$isAncestorAS0(n: DefinitionAS0 ,n': DefinitionAS0): \textit{BOOLEAN} =_{def}$
$n = n'. parentAS0 \vee isAncestorAS0(n, n'.parentAS0)$

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

controlled $rootNodeAS1: \rightarrow DefinitionAS1$
controlled $rootNodeAS0: \rightarrow DefinitionAS0$

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

$replaceInSyntaxTree: DefinitionAS0 \times DefinitionAS0 \times DefinitionAS0 \rightarrow DefinitionAS0$

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.