TAM'97:

the Trace Assertion Method

of Module Interface Specification.

Reference Manual

Michal Iglewski[a], Marcin Kubica[b], Jan Madey[b], Janina Mincer-Daszkiewicz[b], Krzysztof Stencel[b]

a. Département d'informatique, Université du Québec à Hull, Hull, Québec, Canada J8X 3X7
Email: iglewski@uqah.uquebec.ca Tel: (819) 773 1602 Fax: (819) 773 1638

b. Institute of Informatics, Warsaw University, Banacha 2, 02-097 Warsaw, Poland
Email: Firstname.Lastname@mimuw.edu.pl Tel: (48-22) 658 3165 Fax: (48-22) 658 3164

ABSTRACT

A software module may be described precisely and completely by a set of related documents: interface specification of the module providing a "black-box" description of its behavior, internal design of the module containing its "clear-box" description, and the code itself. A special formalism is needed in each of these documents. We use the Trace Assertion Method for specification of module interfaces. This paper contains a description of the Trace Assertion Method: we present the syntax of trace specifications and define their semantics using the natural language.

Chapter 1 Introduction

Formal specification techniques are increasingly being recognized as essential means for the development of reliable software. Numerous projects have demonstrated that formal methods can be successfully applied in practice (see e.g. [4]). However, we are still a long way from their systematic use in commercial applications. What is needed is an overall software methodology which would integrate methods of software engineering generally accepted by practitioners with formal specification techniques advocated by theoreticians. What is further needed is a set of integrated tools to support the systematic development of software from specifications to code.

The foundations for such a methodology were laid nearly 20 years ago [18]. According to criteria enunciated in the early 1970s and now widely accepted, software should be hierarchically structured and consist of a set of information-hiding modules [17]. A module implements objects which can be manipulated from outside the module by means of its access-programs. The description of each module consists of three documents. A *module interface specification* provides a "black-box" view of the module. A *module internal design* is prepared for every implementation of the module interface specification. It presents the module's internal data structures and the effect of its access-programs on the state of that structure, i.e., it provides a "clear-box" description. The third document is the *code of the module*. In a multi-module project an additional document is needed, a *project guide*, which gathers data concerning all modules within the project. All documents should be precise, complete, and consistent. They constitute a series of specifications starting at a general level and successively introducing more details. They should be formal enough that each specification can be verified to ensure it meets the requirements of its predecessor. The whole documentation and specification process should be embedded within a sound and practically verified software engineering framework [8, 19].

The purpose of the project undertaken by the Université du Québec à Hull and Warsaw University was to implement an integrated set of tools supporting this methodology [6, 21]. We chose the *Trace Assertion Method* [1, 20] as the formalism for specifying module interfaces. Since its very first application in the A-7E Project [3], the Trace Assertion Method has undergone many modifications, aiming at laying sound mathematical foundations [7, 10, 14, 15, 22], improving notation [9], and making it more practically-oriented [2, 9, 11]. However, a complete formal description of the Trace Assertion Method has not been presented yet.

In this paper we have attempted to describe formally the Trace Assertion Method. For that purpose we had to clarify many issues and to introduce some new modifications to the method. To distinguish the resulted version of the trace assertion method from its predecessors we call it TAM'97 (in short: TAM).

Preparing a formal description of a specification method is a complex, difficult, time-consuming, and error-prone task. Though we did our best to make the description of TAM as complete, uniform, and correct, as possible, we are sure that a careful reader might find some remaining flaws. We would be very grateful for any comments and/or corrections.

The structure of this paper is as follows. In Chapter 2 we explain the main concepts of TAM. Chapter 3 describes notational conventions used in this paper and the basic notions. Chapter 4 details the expressions used in TAM, while Chapter 5 determines the structure of a trace specification. Chapter 6 sketches properties of basic types. Their specifications are given in Appendices A, B and C. Appendix D contains a sample specification.

Acknowledgments

This work was partly supported by the NATO Linkage grant (HTECH. LG. 941314), the State Committee for Scientific Research in Poland (KBN, grant 8 S503 040 04), and by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Bartussek, W., Parnas, D.L., “Using Traces to Write Abstract Specifications for Software Modules”, in *Proc. 2nd Conference of European Cooperation in Informatics*, Springer-Verlag, LNCS 65, 1978, pp. 211-236; Reprinted in Gehani, N., McGettrick, A.D. (Eds.), *Software Specification Techniques,* AT&T Bell Telephone Laboratories, 1985, pp. 111-130.
2. Bojanowski, J., Iglewski, M., Madey, J., Obaid, A., “Functional Approach to Protocols Specification”, in *Protocol Specification, Testing and Verification XIV*, Vuong, S.T., Chanson, S.T. (Eds.), Chapman & Hall, 1995, pp. 395-402.
3. Britton, K.H., Clements, P.C., Parnas, D.L., Weiss, D.M., “Interface Specifications for the SCR (A-7E) Extended Computer Module”, U.S. Naval Res. Lab., Washington D.C., *NRL Memorandum Rep. 5502*, 1984; p. 129.
4. Craigen, D., Gerhart, S., Ralston, T.J., “An International Survey of Industrial Applications of Formal Methods”, *NISTGCR 93/626*, it can be obtained by ftp from hissa.ncsl.nist.gov.
5. Desrosiers, B., Iglewski, M., Obaid, A., “Utilisation de la méthode de traces pour la définition formelle d'un protocole de communication”, *Electronic Journal on Networks and Distributed Processing*, No. 2, September 1995, pp. 57-73.
6. Iglewski, M., Kubica, M., Madey, J., “An Editor for the Trace Assertion Method”, in *Proceedings of the 10th International Conference of CAD/CAM, Robotics and Factories of the Future: CARs & FOF’94,* M.Zaremba (Ed.), OCRI, Ottawa, Ontario, Canada,1994, pp.876-881.
7. Iglewski, M., Kubica, M., Madey, J., “Trace Specifications of Non-deterministic Multi-object Modules”, in *Algorithms, Concurrency and Knowledge,* K.Kanchacasut, J-J.Levy (Eds.), *Proceedings of the 1995 Asian Computer Science Conference,* Pathumthani, Thailand, December 1995, Springer-Verlag LNCS 1023, 1995, pp.381-395.
8. Iglewski, M., Kubica, M., Madey, J., Mincer-Daszkiewicz, J., Stencel, K., “The Fun-Project: From Requirements Specification to Program Presentation”, Warsaw University, Institute of Informatics, Warsaw, Poland, *Technical Report* TR 95-18 (218), 1995, p. 27.
9. Iglewski, M., Madey, J., Parnas, D.L., Kelly, P.C., “Documentation Paradigms”, *CRL Report* No. 270, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1993.
10. Iglewski, M., Madey, J., Stencel, K., “On Fundamentals of the Trace Assertion Method”, *Technical Report* RR 94/09-6, Université du Québec à Hull, Hull, Québec, Canada, 1994.
11. Iglewski, M., Mincer-Daszkiewicz, J., Stencel, K., “Case Study in Trace Specification of Non-deterministic Modules”, in *Proceedings of the CS&P’95 Workshop*, Warsaw, Poland, October 11-13, 1995.
12. Janicki, R., “On Foundations of the Trace Assertion Method”, *Technical Report* TR 95-04, McMaster University, Dep. of Computer Science and Systems, Hamilton, Ont. 1995, p.35.
13. Kolodziejski, P., Majewska, M., “Specification of UNIX File System in the Trace Assertion Method” [in Polish], MSc Thesis, Warsaw University, Institute of Informatics, 1996, p.196.
14. Lasota, S., “Semantics of specifications in the Trace Assertion Method” [in Polish], MSc Thesis, Warsaw University, Institute of Informatics, 1995, p.44.
15. McLean, J.D., “A Formal Foundation for the Abstract Specification of Software”, *Journal of the ACM*, Vol. 31, No. 3, July 1984, pp. 600-627.
16. Norvell, T.S., “On Trace Specifications”, *CRL Report* 305, McMaster University, Communications Research Laboratory (CRL), July 1995, p.45.
17. Parnas, D.L, “On the Criteria To Be Used in Decomposing Systems Into Modules”, *Communnication of the ACM*, **15**(12), 1972, pp. 1053–1058.
18. Parnas, D.L., “The Use of Precise Specifications in the Development of Software”, *Proceedings of the IFIP Congress*, 1977, North Holland Publishing Company, pp. 861–867.
19. Parnas, D.L., Madey, J., “Functional Documents for Computer Systems”, *Science of Computer Programming,* Vol. 25, 1995, pp. 41-61.

20. Parnas, D.L., Wang, Y., "The Trace Assertion Method of Module Interface Specification", *Technical Report* 89-261, Queen's University, C&IS, Telecommunication Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, 1989.

21. Stencel, K., "Refined Simulation Techniques for the Trace Assertion Method", Warsaw University, Institute of Informatics, Warsaw, Poland, *Technical Report* TR 95-17 (217), 1995, p. 11.

22. Wang, Y., "Specifying and Simulating the Externally Observable Behavior of Modules", (Ph.D. Thesis), *CRL Report* No. 292, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1994.

Chapter 2 Foundations of TAM

2.1 Basic concepts

The Trace Assertion Method describes *software modules* as observed by *external observers*. Each such module implements a set of *homogeneous* and *independent objects*. From a programmer's point of view such modules deliver *abstract data types*. Objects correspond to *state machines* and they are homogeneous in that sense that their behavior is indistinguishable for the observer of the module. We adopt basic concepts of the automata theory, such as *states*, *events,* and *outputs*. In particular, an object is defined as an entity with the following properties:

- it has states, can be affected by events, and produces outputs,
- it is in one of its states at every instant of time; initially it is in the *initial* state,
- it may change its state only as the result of an event,
- outputs may be produced only in response to events,
- at most one event may occur at any given instant of time,
- the set of possible pairs consisting of the next state and the produced output depends exclusively on:
  - the present state of this entity,
  - the event affecting this entity.

In the remainder of this chapter we assume that the object under observation is settled. The only observable aspects of the object's behavior are events affecting the object and the outputs produced in response to these events. A history of the object can thus be represented by a finite sequence of event-output pairs. Such sequences are called *traces*. More formally, let E denote the set of events that can affect the object and O denote the set of produced outputs. The *set of all traces of the object* is equal to $(E \times O)^*$, where X^* denotes the set of finite sequences of elements from X, including the empty one. Subsequent pairs are separated by the dot, "**.**", i.e. a trace has the form $(e_1, o_1)\mathbf{.}(e_2, o_2)\mathbf{.} \ldots \mathbf{.}(e_n, o_n)$, where n is a non-negative integer and for each $i \in \{1,2,...,n\}$, $e_i \in E$ and $o_i \in O$. The empty sequence is called the *empty trace* and denoted by the underscore, "_"; it represents the history of the object affected by no events. The dot is also used as an operator defined on traces. If T_1 and T_2 are traces, then $T_1\mathbf{.}T_2$ is the trace obtained by concatenation of T_1 and T_2. The empty trace is the neutral element of the dot operation.

Traces are intended to represent the externally visible behavior of the object. We limit our observations to sequences of events representing "proper" usage of the object. The fact whether an event is legal is defined by the legality function described in Section 2.2. The subset of the set of all traces corresponding to proper usage is called the set of *proper* traces. The behavior of the object is only specified for proper traces. To fully specify the observable aspects of the object's behavior it is sufficient to define the subset of the set of proper traces which corresponds to possible histories of the object — it is called the set of *feasible* traces. The sets of proper and feasible traces can be recursively characterized in the following way:

- the *empty trace* is both proper and feasible,
- the trace $T\mathbf{.}(e,o)$ is proper if the trace T is feasible and the event e is legal for the object whose trace is T,
- the trace $T\mathbf{.}(e,o)$ is feasible if the trace $T\mathbf{.}(e,o)$ is proper and $o \in O$ is one of the possible outputs produced by the object, whose trace is T, when affected by the event $e \in E$.

If T and $T\mathbf{.}S$ are both feasible traces, then S is called a *feasible extension* of T. The set of all feasible extensions of T is called the object's *behavior after T*.

Two feasible traces are *observationally equivalent* iff the object's behaviors after these traces are the same. In other words, an equivalence relation (denoted by "$\overset{o}{\equiv}$") is defined on the set of all feasible traces, such that $T_1 \overset{o}{\equiv} T_2$ iff T_1 and T_2 have the same sets of feasible extensions. This relation is called the *observational equivalence relation*.

Note that if we know the relation "$\overset{o}{\equiv}$", then the description of the object's behavior after a feasible trace T also specifies the object's behavior after each trace from the equivalence class of T. We make use of this observation in trace specifications of modules.

2.2 A trace specification of a module

According to our basic assumption (cf. Section 2.1), all objects in a module are homogeneous and the state of each object in a module is independent of the states of other objects in this module. We assume that in case of an event concerning more than one object, state changes of these objects may be described independently. In this section we focus on events affecting only one, *generic*, object of the module (state changes of more than one object are considered in Section 2.4).

Since we are interested in software modules, it is reasonable to distinguish the following means of communication of the object with the outside world:

- a set of programs that can be used by objects from other modules to provide information to, and/or receive information from the object — they are called *access-programs*,
- a vector of external variables that affect the object's behavior — they are called *input variables*,
- a vector of variables whose values are computed by the object and can be observed externally — they are called *output variables*.

Thus, the following events can affect an object:

- access-program invocations,
- changes of values of the input variables, called *input variable events*;

and the following outputs can be produced by an object:

- values returned by access-program invocations,
- values of output variables.

If two or more events arrive at the same time, then the external environment of the module arranges them into a sequence and delivers to the objects of the module one by one.

In TAM we describe objects from an external observer's point of view, specifying which traces can be observed, i.e., which are feasible for a given set of proper traces. Usually this is not a simple task. TAM provides a technique which is based on the observation made at the end of the previous section. It consists in specifying a state machine which has the same set of proper and feasible traces as the described object. We proceed in the following steps:

- We choose a subset, C, of feasible traces, which we call *canonical traces*. At least one element of C must belong to each abstraction class of the observational equivalence relation. Canonical traces are states of the specified state machine (*abstract states* of the object). We describe the object's behavior only after traces from C.
- Since the initial state of the object is represented by the empty trace, either C must contain the empty trace, or we have to specify which canonical trace is observationally equivalent to it (otherwise, we could not predict the object's reaction after the very first event). This canonical trace is the initial state of the specified state machine.
- We define the *legality* function: its domain is $C \times E$ and its values are strings enclosed in '%' characters (called *legality tokens*). This function specifies which events are legal after a given canonical trace. If *legality*(c, e) = %legal% then for any $o \in O$ and for any feasible trace T leading to the state c, the trace $T.(e, o)$ is proper. Otherwise,

it is improper and the value of the legality function can bear some descriptive information why it is improper. We will denote the set of pairs (c, e) for which $legality(c, e)$ = %legal% by L.

- We define the *output relation*, *out*; it is a subset of the product $L \times O$. This relation states which outputs can be produced in response to events after a canonical trace — $(T_C, e, o) \in out$ iff the output o can be produced in response to the event e after the canonical trace T_C.
- We define the *extension function*, *ef*, which maps from *out* into the set of canonical traces. If $ef(T_C, e, o) = S_C$, then we know that $T_C.(e, o)$ is observationally equivalent to S_C and thus we may further assume that the propriety of traces and the object's behavior is described by S_C. The extension function is the state transition function of the specified state machine. It must be true that if $T_C.(e, o) \in C$, then $ef(T_C, e, o) = T_C.(e, o)$.

The extension function and the definition of the canonical trace equivalent to the empty trace allows us to reduce each feasible trace to an observationally equivalent canonical trace (this is discussed in the next section). Using the legality function we can decide which extensions of a feasible trace are proper. Using the output relation we can predict outputs produced in response to events by the object being in a state represented by a canonical trace. Therefore, the technique described above completely specifies the externally observable behavior of the object.

A *trace specification of a module* (often called in short a *module*) consists of a description of the generic object from the module, and introduces:

- access-programs (cf. Section 5.3),
- input variable events (cf. Section 5.7),
- output variables (cf. Section 5.8),
- a predicate "*canonical*" and possible auxiliary functions (cf. Section 5.4),
- a legality function (cf. 5.6.4.1),
- an extension function (cf. Section 5.5),
- an output relation (cf. Section 5.5).

2.3 The specification equivalence

A trace specification defines explicitly the set of canonical traces, the legality function, the extension function, *ef*, and the output relation, *out*. It also defines implicitly the propriety and feasibility of traces.

Let us introduce the *reduction function*, *reduce*, that maps the set of feasible traces onto the set of canonical traces. We define it, together with the propriety (predicate *proper*) and feasibility (predicate *feasible*) of traces, by a mutual induction (U is the canonical trace equivalent to the empty trace, or the empty trace, if it is canonical itself):

$$proper(\_) \wedge proper(U)$$

$$feasible(\_) \wedge feasible(U)$$

$$reduce(\_) = U$$

$$proper(T.(e, o)) \Leftrightarrow feasible(T) \wedge (legality(reduce(T), e) = \%\text{legal}\%)$$

$$feasible(T.(e, o)) \Leftrightarrow proper(T.(e, o)) \wedge (reduce(T), e, o) \in out$$

$$reduce(T.(e, o)) = ef(reduce(T), e, o)$$

The reduction function defines an equivalence relation on feasible traces, denoted by "$\overset{S}{\equiv}$":

$$T_1 \overset{s}{\equiv} T_2 \overset{df}{=} reduce(T_1) = reduce(T_2)$$

For each pair of feasible traces, T_1 and T_2, it is true that

$$T_1 \overset{s}{\equiv} T_2 \Rightarrow T_1 \overset{o}{\equiv} T_2$$

The predicate canonical defines unique representatives of equivalence classes of "$\overset{s}{\equiv}$" (otherwise, there would exist a canonical trace U such that $reduce(U) \neq U$). We might expect that it also defines unique representatives of equivalence classes of "$\overset{o}{\equiv}$". Sometimes, however, "$\overset{s}{\equiv}$" may partition the set of feasible traces into smaller equivalence classes than "$\overset{o}{\equiv}$" does. Thus, there may exist different canonical traces that are observationally equivalent.

Usually the above implication holds in both directions, i.e., the two relations on traces, "$\overset{s}{\equiv}$" and "$\overset{o}{\equiv}$", are identical (in such case they are called the *trace equivalence relation*).

Let us illustrate the possible differences between the two relations using a simple example — a stack of non-negative integers with the following access-programs:

PUSH(a)	pushes a onto the top of the stack,
POP	removes the top element from the stack,
TOP	returns the value of the top element of the stack.

It seems natural to define canonical traces as finite sequences of PUSH(a_i). Such sequences represent current contents of the stack, which constitutes the only relevant information. It also seems natural to define PUSH to be always legal, and to define POP and TOP to be legal for non-empty stacks. Both equivalence relations would be the same in this case.

Let us now slightly modify our example and require that the TOP access-program returns the value taken modulo 256. The new specification may differ only in the description of the output relation where the value returned by TOP is defined. Although the two traces PUSH(5) and PUSH(261) are canonical, they are observationally equivalent — the only way to retrieve information from the stack is to call the program TOP which would return 5 in both cases. In other words, if we do not change the definition of the predicate *canonical* in the specification of our modified stack, then the two equivalence relations ("$\overset{s}{\equiv}$" and "$\overset{o}{\equiv}$") are not identical anymore.

In this particular case we could easily change the second specification by modifying the canonical traces to be sequences of PUSH(x), where $0 \le x \le 255$, resulting in the two equivalence relations being the same. However, such a modification might be not so straightforward and/or reasonable in the case of more complex examples.

2.4 Events affecting more than one object in a module

In Section 2.2 we have made the assumption, that one event may alter one object only. Here we show how to deal with events affecting more than one object.

If an event affects more than one object, the outcome of this event may depend on the states of the affected objects. Thus traces representing those states should be a part of the event description — we add the names and traces of the affected objects to the argument lists of access-program invocations.

Since a trace can contain names and states of more than one object, it is generally no longer possible to identify the object under observation. We need a way to distinguish it from other objects. We replace the state and the name of the observed object in all events with the asterisk symbol "*". Note that for the event e_i from the trace (e_1, o_1)**.** ... **.**(e_n, o_n), the state of the observed object is determined by the prefix (e_1, o_1)**.** ... **.**$(e_{i\text{-}1}, o_{i\text{-}1})$. Since all objects implemented by the module are homogeneous, the actual name of the observed object is irrelevant — we assume that it is different from all names of objects given explicitly. We call the observed object the *subject* of the trace.

All steps in the construction of the trace specification remain the same as described in Section 2.2.

2.5 Notational conventions

In the previous sections we defined traces to be finite sequences of pairs (e, o) composed of events e and outputs o. An output o is a vector of output values. Each of these values depends either deterministically or non-deterministically on the preceding events and outputs. From now on we will omit deterministic values since they can be deduced from the preceding part of the trace. Non-deterministic values from o will be incorporated into the event description e.

Chapter 3 Basic concepts

3.1 Syntax and semantics

Syntactical aspects of TAM are formally described with the help of the abstract and presentation syntax. The meaning of each syntactical entity of the abstract syntax will be defined in the natural language.

3.1.1 Abstract syntax

The *abstract syntax* of trace specifications is presented in the standard way, e.g.

non_terminal ::= OperatorName1($arg_{1,1}$, ..., $arg_{1,m}$)

| OperatorName2($arg_{2,1}$, ..., $arg_{2,n}$)

| ...

where each $arg_{i,j}$ is a non-terminal symbol. All operator names are different. The operator is present even if there is only one production for a non-terminal symbol. The characters "|", "[[", "]]*", and "]]$^+$" have been adopted as meta-symbols, as follows:

x | *y*	means	*x* or *y*	(used to separate subsequent productions)
[[*x*]]*	means	zero or more occurrences of *x*	(used to denote possibly empty lists of *x*)
[[*x*]]$^+$	means	one or more occurrences of *x*	(used to denote non-empty lists of *x*)

The operators "[[ ]]$^+$" and "[[ ]]*" are formally defined as follows:

- the expression

  [[some_element]]*

  is an abbreviation of

  SomeElement0() | SomeElement2(some_element, some_element_list),

- the expression

  [[some_element]]$^+$

  is an abbreviation of

  SomeElement1(some_element) | SomeElement2(some_element, some_element_list)

Example

The production

trace_expr_list ::= [[trace_expr]]$^+$

is equivalent to

trace_expr_list ::= TraceExpr1(trace_expr) | TraceExpr2(trace_expr, trace_expr_list)

3.1.2 Presentation syntax

The *presentation syntax*, being a function of the abstract one, states how terms built in accordance with the abstract syntax are translated to the form they appear in a trace specification. However, in trace specifications we use a notation that cannot be described by an ordinary context-free grammar (i.e. tabular notation, indentations, justification of lines, specific type/size of fonts). For this purpose we adopted a “shaded box” convention — if a production specifies some graphical features, its right-hand side is written on a shaded background. In a very few cases we decided to desist from the rigorous formality to gain readability and conciseness - such situations are, however, always explained later on in the text (cf. e.g. the presentation syntax of the access-programs table in 5.3.2 and the corresponding comments in 5.3.5).

Usually, the mapping between the abstract syntax definition of a non-terminal symbol and the corresponding presentation syntax definition is straightforward (there is the same number of productions for the non-terminal symbol in both cases, the order of productions does not change, and the non-terminal symbols on right-hand sides of corresponding productions are the same). If, however, the mapping is not one-to-one and requires an explanation, then this is done in a separate section entitled “Comments on the presentation syntax”.

Non-terminal symbols are written in Sans Serif; remaining symbols are terminal, including spaces and newlines. If the vertical bar, “|”, is needed as a terminal symbol, it must be surrounded by single quotes. The empty word is denoted by “ε”.

We use the following meta-symbols in the presentation syntax:

$x \mid y$	means	x or y	(used to separate subsequent productions)
$[[\ x\ ]]^{+}(y)$	means	$x \mid x\ y\ [[\ x\ ]]^{+}(y)$	(used to denote non-empty lists of x separated by y)
$[[\ x\ ]]^{*}(y)$	means	$\varepsilon \mid [[\ x\ ]]^{+}(y)$	(used to denote possibly empty lists of x separated by y)

Example

The following production:

parameters ::= [[type_parameter | [[var_parameter]]$^{+}$(,) : type]]*(;)

defines a list (possibly empty, separated by semicolons) of two kinds of parameters: type parameters, and variable parameters, gathered on (non-empty, separated by commas) lists of variables of the same type.

3.1.3 Referenced non-terminal symbols

There are many inter-dependencies among sections describing the syntax — in a particular section we may refer to a non-terminal symbol which is defined in another section. All such non-terminals are enumerated together with the numbers of the sections containing the definitions of these symbols.

3.2 Basic non-terminal symbols

There are some commonly used, *basic non-terminal symbols*, for which we give no formal syntactical productions. Their definition is as follows:

- ident — is an *identifier*, i.e., a sequence of letters, digits and underscore characters beginning with a letter or an underscore, and having at least one letter. If it begins with one or more underscores, then the first character different than underscore must be a letter. Upper- and lower-case letters are considered distinct characters.
- index — is a *sequence of digits*. The first one must not be zero.
- string — is a *sequence of arbitrary characters* different from newlines.
- text — is a *sequence of arbitrary characters*. In particular, it can contain newlines.

token is a string enclosed in characters '%' and not containing this character.

3.3 Scopes of identifiers

Identifiers are used to denote entities such as: types, variables, or access-programs. One of all occurrences of a given identifier inside a specification is distinguished and it is called the *introduction of the identifier*; the other occurrences of the same identifier are called its *uses*. The introduction of an identifier is also called the *introduction of the entity* denoted by it.

Each identifier (and the entity denoted by it) have a *scope* which is to be understood as the portion of the specification where this identifier (and the entity denoted by it) can be used. The scope is described for every syntactic entity introducing an identifier. A specification can introduce the same identifier more than once provided that the scopes of the identical identifiers are disjoint. A specification can also introduce identifiers introduced in other specifications (cf. 5.2.4).

If an identifier introduces an entity *B* inside the definition of an entity *A* and there are no other occurrences of identifiers in *A*, then we may also say that *the entity A introduces the entity B*.

3.4 Trace sets and types

3.4.1 Abstract syntax

trace_set ::=

AllTraces(type) | CanTraces(type)

type ::=

Type(ident)

3.4.2 Presentation syntax

trace_set ::=

<<type>> | <type>

type ::=

ident

3.4.3 Referenced non-terminal symbols

ident	Section 3.2

3.4.4 Semantics

A *type* is a basic notion defined by a trace specification of a module (in short: *specified by a module*). Each type is identified by the argument of the operator Type and has two sets of traces associated with it. The first one, described by the operator AllTraces, is the set of all traces of the module which specifies the type being the argument of this operator. The second one, described by the operator CanTraces, the set of *canonical traces*, is a subset of the first set, and is defined by the predicate "*canonical*" (cf. Section 5.4). Canonical traces are also called *reduced traces* (cf. Section 2.3). A trace belonging to either of the two sets associated with a type *x* is said to be *of type x*.

Canonical traces represent states (called also *values*) of a given type. Note, however, that a value of a trace expression does not have to be a canonical trace (cf. Section 4.5).

The type being specified by a given module is called *domestic*. All other types used within this module are called *foreign*.

3.4.5 Predefined types

The four commonly used types: bool, char, int, and real, are known as *predefined types*. Their sets of values are as follows:

bool is a set of *boolean* (*logical*) *values*,

char is a set of *characters*,

int is a set of *integer numbers*,

real is a set of *real numbers*.

Both, the canonical traces of the predefined types, and the conventional operations, can be written in standard notation. Details about these types are to be found in Section 6.1.

Chapter 4 Expressions

4.1 Auxiliary notions

4.1.1 Abstract syntax

constraint ::=

ConstraintNo() | ConstraintYes(log_expr)

qualifier ::=

QualNo() | QualYes(type)

4.1.2 Presentation syntax

constraint ::=

ε | (log_expr)

qualifier ::=

ε | type::

4.1.3 Referenced non-terminal symbols

ident	Section 3.2
log_expr	Section 4.9

4.1.4 Semantics

A constraint is always accompanied by an expression and a declaration of variables. The constraint is used to constrain the set of values that can be assigned to variables introduced by this declaration and used in the accompanying expression. The argument of the operator ConstraintYes, log_expr, is a logical expression imposing restrictions on the set to which this operator is applied. If no restrictions are intended, we can use operator ConstraintNo instead of ConstraintYes(true). Note that in the expression accompanying the constraint we can use entities with limited domains, e.g. auxiliary functions (cf. Section 4.3).

A qualifier is used to identify a module in which the qualified entity is defined. Qualified entities include access-programs, input variable events, auxiliary functions, and the empty trace. If the operator QualNo is used, then the entity is defined in the module being specified. Otherwise, the entity is defined in the module identified by argument type of the operator QualYes, i.e., in the module in which the domestic type, c.f. Section 5.2, is equal to argument type of the operator QualYes.

4.2 Variable declarations

4.2.1 Abstract syntax

simple_var_intro_list ::=

[[simple_var_intro]]*

simple_var_intro ::=

SimpleVarIntro(ident)

name_var_intro ::=

NameVarIntro(ident)

var_declaration_list ::=

[[var_declaration]]$^+$

var_declaration ::=

VarDeclaration(var_untyped_declaration_list, trace_set)

var_untyped_declaration_list ::=

[[var_untyped_declaration]]$^+$

var_untyped_declaration ::=

UntypedSimpleVarDeclaration(simple_var_intro)
| UntypedIndexedVarDeclaration(indexed_var_intro)

indexed_var_intro ::=

IndexedVarIntro(ident, trace_expr_list, trace_expr_list)

4.2.2 Presentation syntax

simple_var_intro_list ::=

[[simple_var_intro]]*(,)

simple_var_intro ::=

ident

name_var_intro ::=

ident

var_declaration_list ::=

[[var_declaration]]$^+$(;)

var_declaration ::=

var_untyped_declaration_list : trace_set

var_untyped_declaration_list ::=

[[var_untyped_declaration]]$^+$(,)

var_untyped_declaration ::=

simple_var_intro | indexed_var_intro

indexed_var_intro ::=

    ident[trace_expr_list]..ident[trace_expr_list]

4.2.3 Referenced non-terminal symbols

ident	Section 3.2
trace_expr_list	Section 4.7
trace_set	Section 3.4

4.2.4 Semantics

There are two kinds of variables: trace variables and name variables. The scope of variables is defined in the following sections:

- 4.3.4 (declarations of auxiliary functions),
- 4.5.4 (declarations of input variables and of input variable events),
- 4.7.4.7 (iterations),
- 4.7.4.10 (where expressions),
- 4.9.4.5 (quantified expressions),
- 5.2.4.1 (parameters),
- 5.6.4.1 (legality functions),
- 5.6.4.2.2 and 5.6.4.2.1 (invocation sub-functions),
- 5.7.4 (input variable events),
- 5.8.4 (values of output variables).

4.2.4.1 Trace variables

Trace variables are used to represent traces. Each trace variable is of a certain type. Let x be the type with which the second argument of the VarDeclaration operator, trace_set, is associated; trace variables introduced by the first argument of this operator, var_untyped_declaration_list, are of type x. A trace variable can be assigned either any trace of type x (if trace_set is AllTraces(x)), or a canonical trace of type x (if trace_set is CanTraces(x)). The assigned trace is the *value* of this trace variable.

Trace variables are simple or indexed.

4.2.4.1.1 Simple variables

A *simple variable* is introduced by the argument ident of the SimpleVarIntro operator. In expressions we refer to its value by this identifier.

4.2.4.1.2 Indexed variables

An *indexed variable* is introduced by the argument ident of the IndexedVarIntro operator. The other two arguments of this operator must be lists of the same length. The type of each trace_expr (cf. 4.7.4.1) on these lists must be int.

An indexed_var_intro of the form IndexedVarIntro(v, p, q) is defined iff the value of each trace_expr on the lists p and q is defined and canonical.

In trace expressions we refer to the value of an indexed variable by its identifier and a list of indexing trace expressions (cf. 4.7.4.6).

Example

Declaration *n*: <int>; *a*[1]...*a*[*n*]: <int> introduces an integer *n* and a sequence of integers, *a,* of length *n*.

4.2.4.2 Name variables

Name variables are used to represent names of objects (cf. Section 2.4). Name variables are introduced by the argument ident of the NameVarIntro operator. The sole operation available on names is the equality test.

4.2.5 Comments on the presentation syntax

Both identifiers ident used on the right-hand side of indexed_var_intro are the same and they are equal to the first argument of the IndexedVarIntro operator in the abstract syntax.

4.3 Auxiliary function definitions

4.3.1 Abstract syntax

fct_declaration ::=

FunctionDeclaration(ident, fct_sign, simple_var_intro_list, constraint, trace_expr)

fct_sign ::=

FunctionSignature(trace_set_list, trace_set)

trace_set_list ::=

[[trace_set]]*

4.3.2 Presentation syntax

fct_declaration ::=

ident : fct_sign

ident(simple_var_intro_list) constraint $\stackrel{df}{=}$ trace_expr

fct_sign ::=

trace_set_list → trace_set

trace_set_list ::=

[[trace_set]]*(×)

4.3.3 Referenced non-terminal symbols

constraint	Section 4.1
ident	Section 3.2
simple_var_intro_list	Section 4.2

trace_expr	Section 4.7
trace_set	Section 3.4

4.3.4 Semantics

Each fct_declaration defines an auxiliary function. The meaning of the arguments of the operator FunctionDeclaration is as follows:

1. ident is the identifier of the auxiliary function being introduced.
   - Its scope is the whole specification.
2. fct_sign is the signature of the function.
   - The domain of the function is the Cartesian product of sets, possibly restricted by constraint (cf. p.4 below). Each of these sets is trace_set determined by the first argument of the FunctionSignature operator, i.e., by trace_set_list.
   - The range of the function is the set trace_set being the second argument of the FunctionSignature operator.
3. simple_var_intro_list is the list of formal arguments of the function.
   - It must be of the same length as the first argument of the FunctionSignature operator, i.e., trace_set_list.
   - Each ident inside the simple_var_intro_list introduces a new simple variable whose scope is the forth and fifth argument of the FunctionDeclaration operator, i.e., constraint and trace_expr.
   - The type of the i-th variable on the simple_var_intro_list is the type with which the i-th trace_set in the trace_set_list is associated.
4. constraint (cf. Section 4.1) is used to finally define the function's domain. If the constraint is of the form ConstraintYes(x) then the logical expression x must be defined for any tuple belonging to the Cartesian product defined by trace_set_list.
5. trace_expr is a trace expression defining the value of the function.
   - The type of the trace_expr must be the type of the range of the function.
   - The value of the function is obtained by evaluation of this expression after replacing in it all formal arguments (listed in simple_var_intro_list) by the actual ones. Details are presented in 4.7.4.9. If the logical expression defined by constraint evaluates to true, then the value of the trace_expr must be defined and belong to the range of this function.

4.3.5 Comments on the presentation syntax

Both identifiers ident used on the right-hand side of fct_declaration are the same and they are equal to the first argument of the FunctionDeclaration operator in the abstract syntax.

4.3.6 Built-in auxiliary functions

The following functions are built-in in every specification:

feasible takes one argument which must be a trace. The value of *feasible* is of type bool, and is equal to true iff the value of the argument is a feasible trace (cf. Section 2.1). Otherwise the value of *feasible* is false.

count takes two arguments. The first one must be a trace, while the second must be an identifier of an access-program, or of an input variable event, from the same module as the trace. The value of *count* is of type int. If the second argument denotes an access-program then the value of *count* is equal to the number of invocations of this access-program in the trace. If the second argument denotes an input variable event then the value of *count* is equal to the number of input variable events identified by this input variable event in the trace.

length takes one argument which must be a trace. The value of length is of type int and is equal to the total number of invocations and input variable events in the trace.

reduce takes one argument which must be a feasible trace. The value of *reduce* is the reduced trace, i.e., the canonical trace which is specification equivalent (cf. Section 2.3) to the argument.

Note that the value an application of *feasible* or *reduce* depends on the definitions of the extension function and the output relation for the events appearing in the argument of this application.

4.3.7 Specific auxiliary functions

Note that the predicate "*canonical*" can be treated as an auxiliary function (cf. 5.4.4).

4.4 Access-program declarations

4.4.1 Abstract syntax

program_declaration_list ::=

[[program_declaration]]$^+$

program_declaration ::=

ProgramDeclaration(ident, arg_description_list, result_type)

arg_description_list ::=

[[arg_description]]*

arg_description ::=

ArgDescription(type, arg_mode)

arg_mode ::=

V() | O() | R() | VO() | VR()

result_type ::=

ResultNo() | ResultYes(type) | ResultYesR(type)

4.4.2 Presentation syntax

program_declaration_list ::=

<table><tr><td rowspan="2">program_declaration</td><td>program_declaration</td></tr><tr><td>program_declaration_list</td></tr></table>

program_declaration ::=

ident	arg_description_list	result_type

arg_description_list ::=

ε	arg_description	arg_description_list

arg_description ::=

type : arg_mode

arg_mode ::=

V | O | R | VO | VR

result_type ::=

ε | type | type : R

4.4.3 Referenced non-terminal symbols

ident	Section 3.2
type	Section 3.4

4.4.4 Semantics

Each program_declaration defines an access-program. The meaning of the arguments of the operator Program-Declaration is as follows:

1. ident is the identifier of the access-program being introduced.
   - Its scope is the whole specification.
2. arg_description_list is the description of the arguments of this program.
   - The type and input/output mode of each argument are described by ArgDescription(type, arg_mode).
   - The input/output mode, arg_mode, is characterized by one of the five operators.
     (a) V — a value must be provided in the invocation of this access-program. The argument labeled by V is called an *input argument*. The outcome of this invocation (returned values and state changes) may depend on the value of this argument.
     (b) O — a name of an object must be provided in the invocation of this access-program. The argument labeled by O is called a *deterministic output argument*. The invocation may assign a new value to the object identified by this argument. The new value is obtained deterministically. The previous value of the object identified by this argument cannot be used. The outcome of this invocation may depend on the actual name used.
     (c) R — a name of an object must be provided in the invocation of this access-program. The argument labeled by R is called *a non-deterministic output argument*. The semantics of the R operator is like the one of O with one difference: the new value of the object being identified is obtained non-deterministically. The type of such an argument cannot be domestic.
     (d) VO — a value and a name of an object must be provided in the invocation of this access-program. The argument labeled by VO is called a *deterministic input-output argument*. The semantics of the VO operator is the combination of the semantics of V and O (and hence the previous value of the object can be used to obtain its new value).
     (e) VR — a value and a name of an object must be provided in the invocation of this access-program. The argument labeled by VR is called a *non-deterministic input-output argument*. The semantics of the VR operator is the combination of the semantics of V and R.
3. result_type is the description of the way this program will be used.
   - If the operator ResultNo is used, then the access-program is *procedure-like*.
   - If either the operator ResultYes or the operator ResultYesR is used, then the access-program is *function-like* and

their argument, type, describes the type of values returned by invocations of this program. The returned value is obtained either deterministically (ResultYes) or non-deterministically (ResultYesR). In the latter case, the type of values returned cannot be domestic.

4.4.5 Comments on the presentation syntax

In the productions program_declaration and arg_description_list the tabular notation is used. These productions are used by program_declaration_list to describe rows in the *access-programs table* and related notational conventions are explained in Section 5.3. Below we give only some of them:

- program_declaration_list is a sequence of rows in the access-programs table;
- program_declaration consists of two cells (ident, result_type) and a sequence of cells between them, each being arg_description;

Example

Basic operations on stacks of integers can be declared in the following way:

PUSH	stack:VO	int:V	
POP	stack:VO	int:V	
TOP	stack:V		int

4.5 Input and output variable declarations

4.5.1 Abstract syntax

input_var_declaration_list ::=

[[input_var_declaration]]$^+$

input_var_declaration ::=

InputVarDeclaration(simple_var_intro, type, input_var_condition_list)

input_var_condition_list ::=

[[input_var_condition]]$^+$

input_var_condition ::=

InputVarCondition(condition, ident)

condition::=

AnyChange()

| BecomesTrue(log_expr)

| BecomesFalse(log_expr)

output_var_declaration_list ::=

[[output_var_declaration]]$^+$

output_var_declaration ::=

OutputVarDeclaration(ident, type)

4.5.2 Presentation syntax

input_var_declaration_list ::=

input_var_declaration	input_var_declaration
	input_var_declaration_list

input_var_declaration ::=

simple_var_intro	type	input_var_condition_list

input_var_condition_list ::=

input_var_condition	input_var_condition
	input_var_condition_list

input_var_condition ::=

condition	ident

condition ::=

AnyChange
| @T(log_expr)
| @F(log_expr)

output_var_declaration_list ::=

output_var_declaration	output_var_declaration
	output_var_declaration_list

output_var_declaration ::=

ident	type

4.5.3 Referenced non-terminal symbols

ident	Section 3.2
log_expr	Section 4.9
simple_var_intro	Section 4.2
type	Section 3.4

4.5.4 Semantics

Each input_var_declaration, InputVarDeclaration(simple_var_intro, type, input_var_condition_list), introduces *input variable events* related to an external object observed by the module. Introduced events correspond to different elements of the input_var_condition_list, one event for every input_var_condition. The meaning of the arguments of the operator InputVarDeclaration is as follows:

1. simple_var_intro introduces a simple variable which denotes an external object, and is called an *input variable*. The scope of this variable is the third argument of the operator InputVarDeclaration, input_var_condition_list.
2. type is the type of this input variable. This type cannot be domestic.
3. input_var_condition_list introduces all input variable events concerning this input variable. Each input_var_condition defines a condition of interest for this variable and the identifier of the input variable event. The scope of each identifier is the whole specification. The input_var_condition is interpreted as follows:

- If it is of the form AnyChange(), then the input variable is called an *unconditional input variable*, and the corresponding input variable event informs the module about any change of value of this input variable.
- If it is of the form BecomesTrue(e), then the input variable is called a *conditional input variable*, and the corresponding input variable event informs the module about a change of value of this input variable such that for the new value the expression e becomes true.
- If it is of the form BecomesFalse(e), then the input variable is called a *conditional input variable*, and the corresponding input variable event informs the module about a change of value of this input variable such that for the new value the expression e becomes false.

If the input_var_condition is of the form AnyChange(), then it has to be the only one concerning the given input variable. If there is no input_var_condition of the form AnyChange(), then let for each i=1..n (n is the length of the input_var_condition_list) $p_i = e_i$ if the i-th input_var_condition is equal to BecomesTrue(e_i), and $p_i = \neg e_i$ if it is equal to BecomesFalse(e_i). All the conditions p_i have to be mutually exclusive. This restriction guarantees that there is no pair of events concerning the same input variable that can occur at the same time.

Each output_var_declaration introduces an *output variable*. The ident is its name, the type is its type, and its scope is empty, i.e., it cannot be used inside the same specification.

There is one copy of each input and output variable for each object of the module. The initial values of input variables are determined by the external environment of the module and are not defined in the specification.

Example

An output variable representing measured temperature can be declared in the following way:

temperature	real

The table below introduces two input variable events: FREEZING and WARM, which take place (respectively) when the input variable, temperature, falls below 0.0 or raises above 20.0.

temperature	real	@T(temperature<0.0)	FREEZING
		@T(temperature>20.0)	WARM

4.6 The set of traces

In this section we define the semantic domain of traces.The set of traces of type x, denoted by <<x>>, depends on the declarations of access-programs and input variable events in the specification of type x and is a subset of the language generated by the following abstract grammar.

trace ::=

[[event_description]]*

event_description ::=

InputVarEvent(ident, trace) | ProgramEvent(ident, arg_list) | ProgramEventR(ident, arg_list, trace)

arg_list ::=

[[arg]]*

arg ::=

ArgAst()
| ArgV(trace)
| ArgN(index)
| ArgNV(index, trace)
| ArgNR(index, trace)
| ArgNVR(index, trace, trace)

A trace of type x is a sequence of event descriptions. Each of them (depending on its form) must satisfy the following conditions:

InputVarEvent(e, t)

- e must be the name of an input variable event declared in the specification of type x;
- t must be a trace of the type of the input variable to which event e corresponds;

ProgramEvent(p, $(a_j)_{j=1,2,\ldots,n}$) or ProgramEventR(p, $(a_j)_{j=1,2,\ldots,n}$, r)

- p must be the name of an access-program declared in the specification of type x;
- if the operator is ProgramEvent, then program p must be either procedure-like or function-like with the returned value obtained deterministically;
- if the operator is ProgramEventR, then program p must be function-like with the returned value obtained non-deterministically and r must be a trace of the type of the value returned by p.
- n must be equal to the number of arguments of access-program p;
- for each j =1,2,...,n argument a_j must be of the following form depending on the mode and the type (whether it is x or not) of the j-th argument of access-program p:

Mode	Type x	Type different from x
V()	ArgV(v_j)	ArgV(v_j)
O()	ArgAst() or ArgN(i_j)	ArgN(i_j)
R()	impossible	ArgNR(i_j, o_j)
VO()	ArgAst() or ArgNV(i_j, v_j)	ArgNV(i_j, v_j)
VR()	impossible	ArgNVR(i_j, v_j, o_j)

- for each j=1,2,...,n

  - if v_j exists, then it must be a trace of the type of j-th argument of program p.

  - if o_j exists, then it must be a trace of the type of j-th argument of program p.

- either at least one a_j must be ActAst(), or p is function-like and returns values of type x (this way the subject of the trace is indicated).
- if for certain k, l equality $i_k = i_l$ holds, then

  - types of k-th and l-th argument must be the same, and

  - if both v_k and v_l exist, then equality $v_k = v_l$ must hold, and

  - if both o_k and o_l exist, then equality $o_k = o_l$ must hold

  (if the same object is passed by different arguments, it must be of the same type and its input and output values passed by these arguments must be consistent).

4.6.1 Referenced non-terminal symbols

ident	Section 3.2
index	Section 3.2

4.7 Trace expressions

4.7.1 Abstract syntax

trace_expr ::=
    TraceEmpty(qualifier)
  | TraceConcatenation(trace_expr, trace_expr)
  | TraceInvocation(invocation)
  | TraceVarEvent(event_constructor)
  | TraceVar(var_name)
  | TraceIteration(trace_expr, simple_var_intro, trace_expr, trace_expr)
  | TraceBracketing(trace_expr)
  | TraceApplication(qualifier, ident, trace_expr_list)
  | TraceWhere(trace_expr, var_declaration_list, constraint, log_expr)
  | TraceTable(trace_entry_list)
  | TraceLogExpr(log_expr)

trace_entry_list ::=
    [[trace_entry]]$^+$

trace_entry ::=
    TraceEntry(log_expr, trace_expr)

var_name ::=

VarName(ident) | VarNameIndexed(ident, trace_expr_list)

trace_expr_list ::=

[[trace_expr]]$^+$

event_constructor ::=

Event(qualifier, ident, trace_expr)

4.7.2 Presentation syntax

trace_expr ::=

qualifier _
| trace_expr . trace_expr
| invocation
| event_constructor
| var_name
| [trace_expr]$_{\text{ident = trace\_expr}}^{\text{trace\_expr}}$
| (trace_expr)
| qualifier ident(trace_expr_list)
| trace_expr where var_declaration_list constraint [log_expr]
|

Condition	Value
trace_entry_list	

| log_expr

trace_entry_list ::=

trace_entry	trace_entry
	trace_entry_list

trace_entry ::=

log_expr	trace_expr

var_name ::=

ident | ident[trace_expr_list]

trace_expr_list ::=

[[trace_expr]]$^{+}$(,)

event_constructor ::=

qualifier ident(*, trace_expr)

4.7.3 Referenced non-terminal symbols

constraint	Section 4.1
ident	Section 3.2
invocation	Section 4.8
log_expr	Section 4.9
qualifier	Section 4.1
simple_var_intro	Section 4.2
var_declaration_list	Section 4.2

4.7.4 Semantics

4.7.4.1 Type and value of trace expressions

A trace expression is one of the following expressions (cf. the right-hand sides of the production of trace_expr): the empty trace, a concatenation, an invocation, an input variable event constructor, a variable, an iteration, a bracketing, an auxiliary function application, a “where” expression, a trace table or a logical expression.

Each trace expression is of a certain type. This type is also called the *type of the trace expression*. For a given assignment of variables each trace expression is either *defined* or *undefined*. A defined trace expression can be evaluated to a trace of the same type as this expression (cf. Section 4.6); we also say that this trace is the value of the trace expression. Note that this value does not have to be a canonical trace. An undefined trace expression has no value.

4.7.4.2 Empty trace

An expression described by TraceEmpty(qualifier) allows constructing empty traces. It is always defined and evaluates to the empty trace (i.e. the empty sequence of event_description) of the type identified by the qualifier (cf. Section 4.1).

4.7.4.3 Concatenation

An expression described by TraceConcatenation(trace_expr, trace_expr), called a *concatenation*, allows composing longer traces from shorter ones. Both arguments must be of the same type.

The type of the expression is the type of the arguments. The expression is defined iff both arguments are defined. Let T_1 be the value of the first argument of TraceConcatenation and T_2 be the value of the second one. The expression evaluates to a sequence of event_description being the concatenation of T_1 and T_2.

4.7.4.4 Invocation

An expression described by TraceInvocation(invocation), called an *invocation*, allows either:

- constructing one-element traces, consisting of single invocations of access-programs with possible returned values, or
- application of the output relation or the extension function.

All details about invocations are presented in Section 4.8.

4.7.4.5 Input variable event constructor

An expression described by TraceVarEvent(event_constructor) allows constructing one-element traces consisting of input variable events.

Input variable event constructor has a form Event(qualifier, ident, trace_expr). The first two arguments identify an input variable event. It has to be defined by InputVarCondition(condition, ident) introduced as one of elements of input_var_condition_list being the third argument of InputVarDeclaration(simple_var_intro, type, input_var_condition_list) in a module identified by qualifier, c.f. Section 4.5. The third argument, trace_expr, corresponds to the value of the input variable related to this event and itroduced by simple_var_intro. This trace_expr must be of the same type as the variable.

The type of an input variable event constructor is the type specified by the module where this event is declared.

The input event constructor is defined if the trace_expr is defined and the operator of the corresponding condition is:

- AnyChange, or
- BecomesTrue and the value of the trace_expr substituted into the log_expr argument of the operator makes this expression true, or
- BecomesFalse and the value of the trace_expr substituted into the log_expr argument of the operator makes this expression false.

The value of the input variable event constructor is a single-element trace consisting of InputVarEvent(ident, trace) where ident is the same as the second argument of Event and trace is the value of the trace_expr.

4.7.4.6 Variable

An expression described by TraceVar(var_name), called a *variable*, allows a usage of previously declared variables (cf. Section 4.2) in trace expressions. It is discussed in two steps, depending on the right-hand side of the production describing the argument var_name.

1. var_name is VarName(ident).

The argument ident must be the identifier of a simple variable (cf. Section 4.2) introduced by SimpleVarIntro(ident). The expression described by the operator VarName is always defined, its type is the same as the type of this variable, and evaluates to the value of the variable.

2. var_name is VarNameIndexed(ident, trace_expr_list).

The argument ident must be the identifier of an indexed variable (cf. Section 4.2) introduced by indexed_var_intro, i.e., IndexedVarIntro(ident, trace_expr_list, trace_expr_list). The expressions in the list argument of VarNameIndexed must be of type int and the list argument must be of the same length as the lists in the declaration of the variable.

- The indexed variable described by VarNameIndexed is defined iff
  - the indexed_var_intro is defined,
  - each trace expression in the trace_expr_list of the VarNameIndexed operator is defined, and
  - if the trace expressions in the declaration of the variable evaluate to integers $p_1, p_2, ..., p_n$ (the first list) and to integers $q_1, q_2, ..., q_n$ (the second one), and the trace expressions in the list argument of VarNameIndexed evaluate to integers $k_1, k_2, ..., k_n$, then the logical expression $p_i \le k_i \le q_i$ holds for each i=1,2,...,n.
- The expression described by VarNameIndexed is of the same type as the type of this variable, and evaluates to the value to which $v[k_1, k_2, ..., k_n]$ refers, where v is the identifier of this variable.

4.7.4.7 Iteration

An expression described by TraceIteration(trace_expr, simple_var_intro, trace_expr, trace_expr), called an *iteration*, allows to write a concatenation of a number of trace expressions in a concise way. The last two arguments must be of type int. The argument simple_var_intro introduces a variable whose scope is the first argument, and whose type is int. The type of an iteration is the same as the type of the first argument.

- Let TraceIteration(trace_expr, simple_var_intro, trace_expr, trace_expr) be denoted by *TraceIteration*(e, id, p, q), where e is the first trace_expr, id is the ident introduced in simple_var_intro, and p and q are integers to which the last two expressions trace_expr evaluate.
- The iteration is defined iff:

  - the last two arguments of the operator TraceIteration are defined, their values are canonical, and

  - for any value r such that $p \leq r \leq q$ the value of the expression $e[id \leftarrow r]$ is defined, where $e[id \leftarrow r]$ is the expression e in which all occurrences of trace expressions of the form TraceVar(VarName(id)) are simultaneously replaced with r.
- The value of the iteration is as follows:

  - If $p > q$ holds, then the value of the iteration is equal to the empty trace.

  - If $p \leq q$ holds, then the value of the iteration is equal to the concatenation of the values of the following two expressions:

$$e[id \leftarrow p], \text{ and } TraceIteration(e, id, p+1, q)$$

Example

The expression $[\text{PUSH}(*, i)]_{i=1}^{3}]$ denotes PUSH(*, 1).PUSH(*, 2).PUSH(*, 3).

4.7.4.8 Bracketing

An expression described by TraceBracketing(trace_expr), called a *bracketing*, allows a usage of parentheses within trace expressions as it is known in standard mathematics. Its type is the type of its argument. It is defined iff its argument is defined. Its value is the value of the argument.

4.7.4.9 Application

An expression described by TraceApplication(qualifier, ident, trace_expr_list), called an *application*, allows a usage of auxiliary functions (cf. Section 4.3) within trace expressions. The argument ident must be the identifier of an auxiliary function declared by FunctionDeclaration(ident, fct_sign, simple_var_intro_list, constraint, trace_expr) (cf. 4.3.1) in the module identified by qualifier. The trace_expr_list must be of the same length as the list of formal arguments, simple_var_intro_list, and the type of each trace expression must be the same as the type of the corresponding formal argument.

The type of the application is the type of the range of the auxiliary function.

Let f be the identifier of the auxiliary function under consideration, id_1, id_2,..., id_n be the identifiers introduced in the third argument of FunctionDeclaration operator, simple_var_intro_list, and e be the last argument of the same operator. Let e_1, e_2,..., e_n denote trace expressions forming the trace_expr_list of the TraceApplication operator, and t_i be the value of e_i (if defined) for i=1,2,...,n. The application is defined iff for i=1,2,...,n each e_i is defined, and the tuple (t_1, t_2,..., t_n) is a member of the domain of f. The value of the application is the value of the expression $e[id_1 \leftarrow t_1, id_2 \leftarrow t_2, ..., id_n \leftarrow t_n]$. The traces t_1, t_2,..., t_n are called *actual arguments of the application*.

4.7.4.10 “where” expression

An expression described by TraceWhere(trace_expr, var_declaration_list, constraint, log_expr), called a *where* *expression*, allows formulating specific conditions within trace expressions, constraining values of selected variables.

Let a where expression be denoted by *where*(t, d, c, l), where t stands for trace_expr, d stands for var_declaration_list, c is the argument of the operator ConstraintYes, if constraint has this form, or true if constraint is ConstraintNo(), and l stands for log_expr.

Let us assume the list d in a where expression, *where*(t, d, c, l), contains only the declaration of a single variable.

- The scope of the variable introduced in d is t, c, and l.
- The type of the where expression is the type of t.
- The value of the where expression is defined iff:

  - the value of each trace_expr in d is defined,

  - for each assignment to the variable in d, the value of c is defined,

  - for all the assignments that make c hold, the value of l is defined,

  - among all the assignments to the variable in d, there is exactly one such that l is true, and

  - for this assignment the value of t is defined.
- The value of the where expression is the value of t when the variable declared by d is assigned a value such that c and l are true.

If list d contains the declaration of more than one variable, this expression is an abbreviation of the expression TraceWhere(*new_t*, *new_d*, ConstraintNo(), UniquelyExists(d, c, And(l, EqualTraces(t, *new_t*)))) where *new_d* is a *var_declaration_list* containing only the declaration of a new simple variable, v, being of the same type as t, and *new_t* is a trace_expr built of this variable. In terms of the presentation syntax, this means that the expression "t where d (c) [l]" is an abbreviation of "v where v: <<type_of_t>> [$\exists!$ d (c) [$l \wedge v = t$]]".

Example

The expression "B where B, E: <stack> [$T = B.E \wedge length(E) = 4$]" denotes the value of the stack obtained from T by removing the top 4 elements (cf. Appendix D).

4.7.4.11 Trace table

An expression described by TraceTable(trace_entry_list), called a *trace table*, allows a usage of a tabular notation within trace expressions. Each element of trace_entry_list, described by TraceEntry(log_expr, trace_expr), represents a row in a two-column table.

Let the elements of trace_entry_list be denoted (l_i, e_i) for i=1,2,...,n. Types of all $e_1, e_2, ..., e_n$ must be the same.

- The type of the trace table is the same as the type of expressions e_i.
- A trace table is defined iff:

  - l_1, l_2,..., l_n are defined, and

  - there is exactly one i such that $1 \leq i \leq n$, l_i is true, and

  - for this value of i, the expression e_i is defined.
- The trace table evaluates to the value of expression e_i, where l_i is the unique logical expression which holds.

If the log_expr of a trace_entry starts with the quantifier $\exists!$, the scope of variables bound by this quantifier is extended by trace_expr of this trace_entry. During the evaluation, each variable is assigned the unique value that makes both log_exprs of the log_expr of the trace_entry hold.

4.7.4.12 Logical expression as a trace expression

An expression described by TraceLogExpr(log_expr), called a *trace-logical expression,* allows an interpretation of logical expressions as traces. Logical expressions as such are discussed in Section 4.9. The expression TraceLogExpr(log_expr) is defined iff its argument is defined. The type of the expression is bool (cf. 3.4.5 and Section 6.1), and the value is a trace representing the logical value "true" or "false", depending on the value of the argument log_expr (cf. Section 4.9).

4.8 Invocation constructors and arrow expressions

4.8.1 Abstract syntax

invocation ::=

    Invocation(qualifier, ident, act_arg_list)
  | InvocationR(qualifier, ident, act_arg_list, trace_expr)
  | InvocationArr(qualifier, ident, act_arg_list)

act_arg_list ::=

    [[act_arg]]*

act_arg ::=

    ActAst()
  | ActV(trace_expr)
  | ActN(index)
  | ActNV(index, trace_expr)
  | ActNArr(index)
  | ActNR(index, trace_expr)
  | ActNVArr(index, trace_expr)
  | ActNVR(index, trace_expr, trace_expr)

4.8.2 Presentation syntax

invocation ::=

    qualifier ident(act_arg_list)
  | qualifier ident(act_arg_list)➘ trace_expr
  | qualifier ident(act_arg_list)➘

act_arg_list ::=

    [[act_arg]]*(,)

act_arg ::=

    *
  | trace_expr
  | * index
  | (* index, trace_expr)

| * index ➘
| * index ➘ trace_expr
| (* index, trace_expr) ➘
| (* index, trace_expr) ➘ trace_expr

4.8.3 Referenced non-terminal symbols

ident	Section 3.2
index	Section 3.2
qualifier	Section 4.1
trace_expr	Section 4.7

4.8.4 Semantics

An invocation has a form of either an invocation constructor or an arrow expression. We can distinguish them in the following way:

- if the invocation operator is InvocationArr, or one of the argument operators is ActNArr or ActNVArr then the invocation is an arrow expression,
- otherwise, it is an invocation constructor.

Both forms differ in their evaluation (cf. 4.8.4.3 and 4.8.4.4). The two forms, however, share a number of common features discussed below.

The first two arguments of each of the operators Invocation, InvocationR, or InvocationArr, i.e., qualifier and ident, identify an access-program. The third one is a list of actual arguments of the invocation of this access-program, act_arg_list. This list must be of the same length as arg_description_list of the access-program (cf. Section 4.4).

An invocation, besides its input values and names of objects, describes also its non-deterministic output values (the last argument of operators ActNR, ActNVR, and InvocationR).

If an actual argument contains trace_expr, then this expression must be of the same type as the corresponding argument of the program.

4.8.4.1 Actual argument form

Depending on its input/output mode, arg_mode (cf. Section 4.4), each actual argument in act_arg_list must be of the following form:

arg_mode	act_arg	interpretation
V()	ActV(trace_expr)	value “before”
O()	ActAst()	name of the subject of the trace
	ActN(index)	name of an object not being the subject
	ActNArr(index)	name of an object not being the subject (cf. 4.8.4.4)

arg_mode	act_arg	interpretation
R()	ActNR(index, trace_expr)	name and value “after” of an object not being the subject
VO()	ActAst()	name and value “before”(determined by the context) of the subject
	ActNV(index, trace_expr)	name and value“before” of an object not being the subject
	ActNVArr(index, trace_expr)	name and value“before” of an object not being the subject (cf. 4.8.4.4)
VR()	ActNVR(index, trace_expr, trace_expr)	name, value “before” and value “after” of an object not being the subject

4.8.4.2 Wild-card symbols

Wild-card symbols are mostly used to simplify the syntax of trace expressions and to express the fact that the values associated with different objects can be equal.

An *index* is used to denote a name of an object in situations when the actual name is not important. The behavior of an object after an invocation should not depend on the names of objects passed as arguments. Otherwise, objects in a module may not be homogeneous (cf. Chapter 2). However, the behavior of an object can depend on the equalities of object names, i.e., on the fact that the same object is passed in more than one argument. For this reason, the actual names used need not occur in the trace. We replace them with indices in places where such names would occur (the argument mode O(), or R()). If the indices *i* and *j* are used to name objects of different types, then *i* and *j* must be different. An index must not be used to denote the name of the subject of the trace.

In the case of the subject, a different wild-card symbol is used. The expression ActAst() denotes either the name or the name and value, of this object. This applies to the argument mode O() or, respectively, VO(). Note that in the latter case, the value represents the trace consisting of all preceding invocations. Note also that no wild-card symbol may be used in the case of actual argument with the modes V(), R() or VR().

4.8.4.3 Invocation constructors

An *invocation constructor* is the most straightforward way of constructing one-element traces and has the following form:

Invocation(qualifier, ident, act_arg_list)

or

InvocationR(qualifier, ident, act_arg_list, trace_expr)

The latter must be used, if the program is function-like and returns a value non-deterministically. In this case the last argument, trace_expr, stands for this value.

Either at least one actual argument is of the form ActAst(), or the program is function-like and returns values of the type specified by the module introducing the program (this corresponds to the subject of the trace). If an argument is of the form ActAst(), then the type of this argument is of the type specified by the module introducing the program.

- The type of an invocation constructor is the type specified by the module where the access-program is declared.
- The invocation constructor is defined iff:
  - all trace expressions, trace_expr, in act_arg_list are defined,

- the last argument of the InvocationR operator is defined,

- if the same object is passed by different arguments, they are of the same type, and their input and output values are respectively equal.

- The value of the invocation is a trace consisting of a single event_description obtained from this invocation in the following way:

  - if the invocation constructor is Invocation(qualifier, ident, act_arg_list), then the event_description is ProgramEvent(ident, arg_list),

  - if the invocation constructor is InvocationR(qualifier, ident, act_arg_list, trace_expr), then the event_description is ProgramEventR(ident, arg_list, trace), where trace is the value of the trace_expr,

  - all arguments on the arg_list are obtained from the act_arg_list by evaluation of each trace_expr, and the following replacement of operators:

operator of act_arg	operator of arg
ActAst	ArgAst
ActV	ArgV
ActN	ArgN
ActNV	ArgNV
ActNR	ArgNR
ActNVR	ArgNVR

Example

The following invocation constructor JOIN(*, (*1, *U*)) denotes the invocation of access-program JOIN on two different objects, first of them being the subject.

4.8.4.4 Arrow expressions

Arrow expressions are used in situations where we are interested in the output value of a deterministic argument, or in the value returned by a function-like access-program, i.e., the extension function or the output relation has to be applied. The operators ActNArr and ActNVArr are used to denote the argument of concern, and the operator InvocationArr is used to denote the value returned by a function-like access-program. In both situations there is no subject, and hence none of the argument operators can be ActAst.

If the invocation operator is InvocationArr, the program identified by qualifier and ident must be function-like and no argument operator can be ActNArr or ActNVArr. If a different operator is used then there must be exactly one argument with the operator ActNArr or ActNVArr (if the operator is different from InvocationArr and there is no act_arg with operator ActNArr and ActNVArr, then this invocation is an invocation constructor) and if the operator is InvocationR, then the program must be function-like and return values non-deterministically.

The value of the arrow expression is defined iff:

- each trace_expr inside the invocation is defined and evaluates to a canonical trace,

- all non-deterministic output values are possible with respect to the output relation (cf. Section 5.6),

- if the same object is passed by different arguments, they are of the same type, and their input and output values are respectively equal,

- the value of the legality function (cf. 5.6.4.1) for the invocation where each trace_expr is replaced with its value, is equal to token “%legal%”.

Depending on the operator used the type and the value of an arrow expression is as follows:

- InvocationArr
  - the type of this arrow expression is equal to the type of values returned by this access-program,
  - the value is equal to the value returned by a call of the access-program as described in Section 5.6.
- InvocationR or Invocation
  - the type of this arrow expression is equal to the type of the argument with the operator ActNArr or ActNVArr,
  - the value of this expression is equal to the value returned via this argument by a call of the access-program as described in Section 5.6.

Example

The arrow expression int::MULT(x, y)➤ denotes the result value of an invocation of function-like access-program MULT, multiplying two integers: x and y.

4.9 Logical expressions

4.9.1 Abstract syntax

```
log_expr ::=
        False()
    |   True()
    |   LogEquivalent(log_expr, log_expr)
    |   Implies(log_expr, log_expr)
    |   And(log_expr, log_expr)
    |   Or(log_expr, log_expr)
    |   Not(log_expr)
    |   EqualTraces(trace_expr, trace_expr)
    |   NotEqualTraces(trace_expr, trace_expr)
    |   EqualNames(name_var, name_var)
    |   NotEqualNames(name_var, name_var)
    |   EquivalentTraces(trace_expr, trace_expr)
    |   ForAll(var_declaration_list, constraint, log_expr)
    |   Exists(var_declaration_list, constraint, log_expr)
    |   UniquelyExists(var_declaration_list, constraint, log_expr)
    |   LogTraceExpr(trace_expr)
    |   LogBracketing(log_expr)

name_var ::=
        NameVar(ident)
```

4.9.2 Presentation syntax

log_expr ::=

false
| true
| log_expr ⇔ log_expr
| log_expr ⇒ log_expr
| log_expr ∧ log_expr
| log_expr ∨ log_expr
| ¬ log_expr
| trace_expr = trace_expr
| trace_expr ≠ trace_expr
| name_var = name_var
| name_var ≠ name_var
| trace_expr ≡ trace_expr
| ∀ var_declaration_list constraint [log_expr]
| ∃ var_declaration_list constraint [log_expr]
| ∃! var_declaration_list constraint [log_expr]
| trace_expr
| (log_expr)

name_var ::=

ident

4.9.3 Referenced non-terminal symbols

constraint	Section 4.1
ident	Section 3.2
trace_expr	Section 4.7
var_declaration_list	Section 4.2

4.9.4 Semantics

Each logical expression is either *undefined* or *defined*. In the latter case, it can be evaluated to logical values: false or true. If the value is true, we say that the expression *holds*; if it is false, we say that the expression *does not hold*.

We apply eager evaluation of logical expressions, e.g., if one of the disjuncts is true while the other is undefined, then the value of the disjunction is undefined. We formulate that rule of eager evaluation in the following sections.

4.9.4.1 Simple logical expressions

A logical expression is constructed of one of the following operators: False, True, LogEquivalent, Implies, And, Or, Not, and is called a *simple logical expression*. A simple logical expression is defined iff all its arguments are defined. The evaluation of a defined expression proceeds as in classical logic for corresponding operators:

Expression	Meaning
False()	false
True()	true
LogEquivalent(log_expr, log_expr)	logical equivalence
Implies(log_expr, log_expr)	implication
And(log_expr, log_expr)	conjunction
Or(log_expr, log_expr)	disjunction
Not(log_expr)	negation

4.9.4.2 Name equality

An expression described by EqualNames(name_var, name_var) is called a *name equality*. Both arguments of the EqualNames operator must be name variables of the same type. The value of this expression is true iff the values of its arguments are equal. The expression NotEqualNames(name_var, name_var) is an abbreviation of Not(EqualNames(name_var, name_var)).

4.9.4.3 Trace equality

An expression described by EqualTraces(trace_expr, trace_expr) is called a *trace equality*. The types of both arguments must be the same. The value of this expression is defined iff the values of arguments are defined. The value is true iff the traces being the values of arguments are equal. The expression NotEqualTraces(trace_expr, trace_expr) is an abbreviation of Not(EqualTraces(trace_expr, trace_expr)).

4.9.4.4 Trace equivalence

An expression described by EquivalentTraces(trace_expr, trace_expr) is called a *trace equivalence*. The types of both arguments must be the same. The value of this expression is defined iff the values of arguments are defined. The value is true iff the reduced values of arguments are equal.

4.9.4.5 Quantified expressions

An expression described by ForAll(var_declaration_list, constraint, log_expr), Exists(var_declaration_list, constraint, log_expr), or UniquelyExists(var_declaration_list, constraint, log_expr) is called a *quantified expression*.

Let a quantified expression be denoted *Quantifier*(*d, c, l*), where *Quantifier* is either *ForAll*, *Exists* or *UniquelyExists*, *d* stands for var_declaration_list, *c* is the argument of the operator ConstraintYes, if constraint has that form, or true if constraint is ConstraintNo(), and *l* stands for log_expr.

Let us assume that the list *d* in a quantified expression *Quantifier*(*d*, *c*, *l*) contains only the declaration of a single variable.

- The scope of the variable introduced in *d* is *c* and *l*.
- The value of this logical expression is defined iff:

  - the value of each trace_expr in *d* is defined,

  - for each assignment to the variable in *d*, the value of *c* is defined, and

- for all the assignments that make *c* hold, the value of *l* is defined.

- The value of a quantified expression is defined as follows:
  - if *Quantifier* is *ForAll*, this logical expression is true, iff each assignment satisfying *c* also satisfies *l*;
  - if *Quantifier* is *Exists*, this logical expression is true, iff among the assignments satisfying *c*, there exists at least one that also satisfies *l*;
  - if *Quantifier* is *UniquelyExists*, this logical expression is true, iff among the assignments satisfying *c*, there exists exactly one that also satisfies *l*.

If the list *d* introduces at least two variables, this expression is an abbreviation of the expression *Quantifier*(*d1*, true, *Quantifier*(*r*, *c*, *l*)) where *d1* is the list containing only the declaration of the first variable from *d*, and *r* is the var_declaration_list obtained from *d* by deleting the declaration of the variable moved to *d1*.

Example

The following quantified expression defines the set of canonical traces of a stack of integers:

$$\exists n\text{: <int>; } a[1]\ldots a[n]\text{: <int> } [T = [\text{PUSH}(*, a[i])]_{i=1}^{n}].$$

4.9.4.6 Trace expression as a logical expression

An expression described by LogTraceExpr(trace_expr), called a *logical-trace expression,* allows an interpretation of traces as logical values. The argument, trace_expr, must be of type bool. The expression LogTraceExpr(trace_expr) is defined iff its argument is defined. Its value is true iff the trace_expr evaluates to a trace representing the logical value “true” (cf. 6.1.1).

4.9.4.7 Bracketing

An expression described by LogBracketing(log_expr), called a *bracketing*, allows a usage of parentheses within logical expressions as it is known in standard mathematics. It is defined iff its argument is defined. Its value is the value of the argument.

4.10 Associativity and precedence of operators (presentation syntax)

The operators “where”, “=”, “≠”, and ≡ are right associative, the others are left associative.

The following table describes the precedence of operators. Each box contains operators with the same precedence. An operator has lower precedence than the operators in boxes on its left-hand side.

➘	•	= ≠ ≡	¬	∧	∨	⇒ ⇔	where

Chapter 5 Structure of trace specifications

5.1 Specification

5.1.1 Abstract syntax

specification ::=

Specification(string, informal_introduction, characteristics_section, syntax_section, canonical_section, semantics_section)

| Instance(type, type, act_param_list)

informal_introduction ::=

InformalIntroNo() | InformalIntroYes(text)

act_param_list ::=

[[act_param]]$^+$

act_param ::=

type | trace_expr

5.1.2 Presentation syntax

specification ::=

string **Module Interface Specification**

informal_introduction
characteristics_section
syntax_section
canonical_section
semantics_section

| type = type(act_param_list)

informal_introduction ::=

ε | **Informal Introduction**
text

act_param_list ::=

[[act_param]]$^+$(,)

act_param ::=

type | trace_expr

5.1.3 Referenced non-terminal symbols

canonical_section	Section 5.4
characteristics_section	Section 5.2
semantics_section	Section 5.5
string	Section 3.2
syntax_section	Section 5.3
text	Section 3.2
trace_expr	Section 4.7
type	Section 3.4

5.1.4 Semantics

A trace specification may be defined either from scratch (operator Specification) or as an instance of a parameterized specification (operator Instance). Each trace specification introduces a new type. Identifiers of types introduced in such a way are available at the global scope of a software project.

5.1.4.1 Structure of a specification

If the operator Specification is used, then a trace specification of a module forms a document with a fixed structure. The document has its *title* (which plays an informal role), being described by the argument string of the Specification operator. The second argument of this operator, informal_introduction, is an optional introductory text aiming to help readers to understand the document. Then four sections follow, which correspond to the last four arguments of the Specification operator (they are described in detail in Sections 5.2 – 5.8):

characteristics_section lists the type being specified, foreign types, and specification parameters;

syntax_section lists and characterizes access-programs, input variables, input variable events, and output variables;

canonical_section defines the set of canonical traces. This section may also include definitions of auxiliary functions, used mainly to improve the document readability;

semantics_section defines the changes of objects caused by access-programs invocations and by input variable events, and the values of output variables.

Specifications of the form Instance(type, type, act_param_list) define instances of parameterized specifications and are discussed in 5.2.4.2.

A sample specification can be found in Appendix D.

5.2 Characteristics section

5.2.1 Abstract syntax

characteristics_section ::=

CharacteristicsSection(type, foreign_types, parameters)

foreign_types ::=

ForeignTypesNo() | ForeignTypesYes(type_list) type_list ::=

[[type]]$^+$

parameters ::=

ParametersNo() | ParametersYes(parameter_list, constraint)

parameter_list ::=

[[parameter]]$^+$

parameter ::=

ParameterType(type) | ParameterConsts(simple_var_intro_list, type)

5.2.2 Presentation syntax

characteristics_section ::=

(0) CHARACTERISTICS
• type specified: type
foreign_types
parameters

foreign_types ::=

ε | • foreign types: type_list

type_list ::=

[[type]]$^+$(,)

parameters ::=

ε | • parameters: parameter_list constraint

parameter_list ::=

[[parameter]]$^+$(;)

parameter ::=

type | simple_var_intro_list : type

5.2.3 Referenced non-terminal symbols

constraint	Section 4.1
simple_var_intro_list	Section 4.2
type	Section 3.4

5.2.4 Semantics

The characteristics_section lists the type specified by the specification, the types used in this specification, and the parameters of the specification. The type specified is called the *domestic type*. The types used in the specification are called *foreign* in this specification.

Each identifier of a type from the argument of the ForeignTypesYes operator introduces the following entities from the specification where this type is defined:

- the type itself,
- the empty trace of this type,
- access-programs as constructors of values of this type,
- input variable events as constructors of values of this type,
- predicate "canonical" and auxiliary functions,
- output relation (by means of the *feasible* function and arrow expressions; cf. 4.3.6 and 4.8.4.4),
- extension function (by means of the *reduce* function and arrow expressions; cf. 4.3.6 and 4.8.4.4).

The identifiers of these entities, except the type, have to be qualified (cf. Section 4.1) by the identifier of this type.

A foreign type list can only contain names of types of non-parameterized specifications or names of instances of parameterized specifications.

Dependencies between specifications implied by foreign type lists cannot be circular. For example, if type t_1 appears on the foreign type list in the specification of type t_2, then type t_2 cannot appear on the foreign type list in the specification of type t_1. More formally, it must be possible to define a partial order on the set of all specified types (including instances of parameterized specifications, cf. 5.2.4.2) such that in each specification the types appearing on the foreign type list precede the type specified by the specification.

5.2.4.1 Parameters

Parameterization allows reusing specifications. There are two kinds of specification parameters:

- a *type parameter*, described by the operator ParameterType, introduces a new type, being the argument of this operator,
- *value parameters* are described by the operator ParameterConsts. They introduce a number of simple variables listed in the first argument of this operator, simple_var_intro_list, each of them of the type being the second argument of this operator, type. Values of these variables are constrained by the constraint given as the second argument of the operator ParametersYes. The first argument of the operator ParameterConsts must be a non-empty list.

The scope of each entity introduced by the argument of the ParametersYes operator, parameter_list, is the rest of the specification following this introduction.

Parameterized specifications can only be used to define instances of such specifications (cf. 5.2.4.2).

5.2.4.2 Instances of parameterized specifications

Each specification of the form Instance(type, type, act_param_list) defines an instance of a parameterized specification, i.e., the one in which parameters are of the form ParametersYes(parameter_list, constraint). In that case the following conditions must hold:

- The length of the act_param_list of the Instance operator is equal to the number of entities introduced in the parameter_list of the corresponding ParametersYes operator.
- If the i-th entity introduced by parameter_list is a type then the i-th actual parameter is a type.
- If the i-th entity introduced by parameter_list is a simple variable then
  - the i-th actual parameter is a trace_expr, and
  - this trace_expr is of the type indicated by the second argument of the corresponding ParameterConsts operator.
- The logical expression defined by the second argument of ParametersYes operator, constraint, evaluates to true for the values of trace expressions being actual parameters and corresponding to the simple variables introduced in parameter_list of ParametersYes operator.
- The type being the first argument of Instance operator corresponds to the defined instance of the parameterized specification identified by the type being the second argument of Instance operator. This instance is obtained in a way described below.

  Let assume that a parameterized specification is defined as type t_1 with parameters p_1, ..., p_n. Let an instance of this parameterized specification be defined by *Instance*(t_2, t_1, *act_param_list*) and let *act_param_list* evaluate to actual parameters a_1, ..., a_n. Then this instance introduces a new type t_2 as the instance of type t_1. The specification of t_2 is obtained from t_1 as follows:

  1. the parameters list is removed;
  2. t_2 replaces all occurrences of t_1;
  3. for i=1,2,...,k if a_i is a type (p_i is then a type parameter), a_i replaces all occurrences of p_i and is added to the foreign types list;
  4. for i=1,2,...,k if a_i is a trace expression (p_i is then a value parameter defined by the identifier, say v_i), the value of a_i replaces all occurrences of v_i.

5.3 Syntax section

5.3.1 Abstract syntax

syntax_section ::=

SyntaxSection(access_programs, input_variables, output_variables)

access_programs ::=

AccessProgramsNo() | AccessProgramsYes(program_declaration_list)

input_variables ::=

InputVarsNo() | InputVarsYes(input_var_declaration_list) output_variables ::=

OutputVarsNo() | OutputVarsYes(output_var_declaration_list)

5.3.2 Presentation syntax

syntax_section ::=

(1) SYNTAX
access_programs
input_variables
output_variables

access_programs ::=

ε | ACCESS-PROGRAMS

Program Name	Arg#1	Arg#2	. . .	Result Type
program_declaration_list				

input_variables ::=

ε | INPUT VARIABLES

Variable Name	Type	Condition of interest	Event
input_var_declaration_list			

output_variables ::=

ε | OUTPUT VARIABLES

Variable Name	Type
output_var_declaration_list	

5.3.3 Referenced non-terminal symbols

input_var_declaration_list	Section 4.5
output_var_declaration_list	Section 4.5
program_declaration_list	Section 4.4

5.3.4 Semantics

The syntax_section introduces access-programs, input variables (and input variable events), and output variables. The declarations of access-programs and input variable events determine the set of traces (cf. Section 4.6).

5.3.5 Comments on the presentation syntax

The following conventions apply to the presentation syntax of the access-programs table (cf. Section 4.4):

- program_declaration_list is a sequence of rows in the table.
- program_declaration consists of two cells (ident, result_type) and a sequence of cells between them (arg_description_list) in the table.
- The headers of "Arg#" columns in the table contain subsequent positive integers starting from 1.
- The number of "Arg#" columns in the table is equal to the maximum length of arg_description_list in all program_declarations.
- If in a program_declaration the arg_description_list is shorter than the number of "Arg#" columns, then the outstanding cells are empty.
- result_type is always placed in the Result Type column.
- If all cells in the Result Type column are empty it can be omitted from the table.
- Subsequent cells in the table are aligned to its first row.

The input_var_declaration_list is a sequence of rows in the input variables table. The output_var_declaration_list is a sequence of rows in the output variables table.

5.4 Canonical section

5.4.1 Abstract syntax

```
canonical_section ::=
        CanonicalSection(simple_var_intro, log_expr, empty_equivalence, aux_fct_section)

empty_equivalence ::=
        EmptyEquivalenceNo
    |   EmptyEquivalenceYes(trace_expr)

aux_fct_section ::=
        AuxiliaryFunctionsNo()
    |   AuxiliaryFunctionsYes(fct_declaration_list)

fct_declaration_list ::=
```

[[fct_declaration]]$^+$

5.4.2 Presentation syntax

canonical_section ::=

> **(2) CANONICAL TRACES**
> *canonical*(simple_var_intro) ⇔ log_expr
> empty_equivalence
> aux_fct_section

empty_equivalence ::=

ε | _ ≡ trace_expr

aux_fct_section ::=

ε | AUXILIARY FUNCTIONS
fct_declaration_list

fct_declaration_list ::=

fct_declaration | fct_declaration
fct_declaration_list

5.4.3 Referenced non-terminal symbols

fct_declaration	Section 4.3
ident	Section 3.2
log_expr	Section 4.9
simple_var_intro	Section 4.2
trace_expr	Section 4.7

5.4.4 Semantics

A canonical_section introduces a predicate *canonical* which defines the set of canonical traces. It is treated as an auxiliary function defined as follows:

$$canonical : \text{<<}x\text{>>} \rightarrow \text{bool}$$

$$canonical(T) \overset{\text{df}}{=} e$$

where x is the domestic type and e is log_expr.

Trace T_0 of the domestic type is canonical iff $e[T \leftarrow T_0]$ holds. This predicate like auxiliary functions can be recursive, i.e., "canonical" may occur within the log_expr.

The empty_equivalence of the form EmptyEquivalenceYes(e) specifies a canonical trace, e, equivalent to the empty trace. If the operator EmptyEquivalenceNo is used, then the empty trace must be canonical, i.e., *canonical*(_) must hold.

Auxiliary functions are discussed in Section 4.3.

5.5 Semantics section

5.5.1 Abstract syntax

semantics_section ::=

SemanticsSection(invocation_functions, input_var_event_functions, output_var_values)

invocation_functions ::=

InvocationsFunctionsNo() | InvocationsFunctionsYes(invocation_function_list)

input_var_event_functions ::=

InputVarEventsFunctionsNo() | InputVarEventsFunctionsYes(input_var_event_function_list)

output_var_values ::=

OutputVarValuesNo() | OutputVarValuesYes(output_var_value_list)

5.5.2 Presentation syntax

semantics_section ::=

(3) SEMANTICS
invocation_functions
input_var_event_functions
output_var_values

invocation_functions ::=

ε | ACCESS-PROGRAMS
invocation_function_list

input_var_event_functions ::=

ε | INPUT VARIABLES
input_var_event_function_list

output_var_values ::=

ε | OUTPUT VARIABLES
output_var_value_list

5.5.3 Referenced non-terminal symbols

input_var_event_function_list	Section 5.7
invocation_function_list	Section 5.6
output_var_value_list	Section 5.8

5.5.4 Semantics

In the semantics_section we describe the effects of events affecting the module (cf. Section 2.2). These events are access-program invocations and input variable events. Each event may change the states of objects in the specified module and/or may change the states of objects of foreign types passed as argument of this event. If the specified module defines output variables, the change of the state of a domestic object is followed by the assignment of new values to the output variables.

The semantics_section consists of three sections:

- The invocation_functions section describes the effects of access-program invocations. It defines the extension function (in the case of outputs of the domestic type), and the output relation (in the case of outputs of the foreign types) for each access-program.
- The input_var_event_functions section describes the effects of input variable events. It defines the extension function for each input variable event.
- The output_var_values section describes the value of each output variable as a function of the state of the corresponding domestic object. It defines the output relation for output variables.

5.6 Invocation functions

5.6.1 Abstract syntax

invocation_function_list ::=

  [[invocation_function]]$^+$

invocation_function ::=

  InvocationFunction(legality, invocation_sub_function_list)

legality ::=

  Legality(formal_invocation, token_expr)

token_expr ::=

  Token(token)

  | TokenTable(token_entry_list)

token_entry_list ::=

  [[token_entry]]$^+$

token_entry ::=

  TokenEntry(log_expr, token_expr)

invocation_sub_function_list ::=

[[invocation_sub_function]]$^+$

invocation_sub_function ::=

InvocationSubFunction(formal_invocation, output_expr)

formal_invocation ::=

FmlInvocation(ident, formal_arg_list)
| FmlInvocationArr(ident, formal_arg_list)
| FmlInvocationR(ident, formal_arg_list, simple_var_intro)

formal_arg_list ::=

[[formal_arg]]*

formal_arg ::=

FmlN(name_var_intro)
| FmlV(simple_var_intro)
| FmlNArr(name_var_intro)
| FmlNR(name_var_intro, simple_var_intro)
| FmlNV(name_var_intro, simple_var_intro)
| FmlNVArr(name_var_intro, simple_var_intro)
| FmlNVR(name_var_intro, simple_var_intro, simple_var_intro)

output_expr ::=

OutputValueConstraint(log_expr)
| OutputValue(trace_expr)

5.6.2 Presentation syntax

invocation_function_list ::=

invocation_function	invocation_function invocation_function_list

invocation_function ::=

legality
invocation_sub_function_list

legality ::=

Legality(formal_invocation) = token_expr token_expr ::=

<table><tr><td>token</td><td><table><tr><td>Condition</td><td>Value</td></tr><tr><td colspan="2">token_entry_list</td></tr></table></td></tr></table>

token_entry_list ::=

<table><tr><td>token_entry</td><td>token_entry
token_entry_list</td></tr></table>

token_entry ::=

log_expr	token_expr

invocation_sub_function_list ::=

<table><tr><td>invocation_sub_function</td><td>invocation_sub_function
invocation_sub_function_list</td></tr></table>

invocation_sub_function ::=

    formal_invocation output_expr

formal_invocation ::=

    ident(formal_arg_list)
  | ident(formal_arg_list) ➘
  | ident(formal_arg_list) ➘ simple_var_intro

formal_arg_list ::=

    [[formal_arg]]*(,)

formal_arg ::=

    name_var_intro
  | simple_var_intro
  | name_var_intro ➘
  | name_var_intro ➘ simple_var_intro
  | (name_var_intro, simple_var_intro)
  | (name_var_intro, simple_var_intro) ➘
  | (name_var_intro, simple_var_intro) ➘ simple_var_intro

output_expr ::=

    '|' log_expr
  | = trace_expr

5.6.3 Referenced non-terminal symbols

ident	Section 3.2
log_expr	Section 4.9
name_var_intro	Section 4.2
simple_var_intro	Section 4.2
string	Section 3.2
token	Section 3.2
trace_expr	Section 4.7

5.6.4 Semantics

Each invocation_function describes one access-program; its identifier is the first argument of the operator FmlInvocation in the formal_invocation of legality.

5.6.4.1 Legality function

Intuitively, legal invocations are those for which the module is expected to be useful. A user is supposed to avoid illegal invocations; they will not occur if the module is used correctly. An example of an illegal invocation is the call to TOP on the empty stack — the returned value is useless. We distinguish two kinds of illegal invocations: fatal and erroneous. They are characterized below.

The legality of invocations is defined by the *Legality* function. The range of the *Legality* function is a set of *status tokens* (token). This function partitions invocations into three groups:

- *legal* invocations (the returned token is "%legal%"). Such invocations are correct, they always terminate and return the specified values;
- *fatal* invocations (the returned token is "%fatal%"). If such an invocation occurs, anything can happen, e.g., the invocation does not have to terminate, the computer system may crash; if the invocation terminates, then the returned values are arbitrary;
- *erroneous* invocations (the returned token is different from "%legal%" and "%fatal%"). In this case, the status token corresponds to a warning and we assume that no object is changed by such an invocation.

The value of the *Legality* function depends on the actual arguments of the invocation (not on the output values). The legality must contain formal_arg_list of the same length as the corresponding arg_description_list from the access-program table. Each formal_arg on this list must correspond to the input/output mode of the argument as follows:

arg_mode	formal_arg	Interpretation
V()	FmlV(simple_var_intro)	A value
O(), R()	FmlN(name_var_intro)	A name of an object
VO(), VR()	FmlNV(name_var_intro, simple_var_intro)	A name of an object and its value before the invocation

The type of each simple variable introduced by a formal_arg is the same as the type of the corresponding argument of the access-program. The type of objects denoted by each name variable introduced by a formal_arg is the same as the type of the corresponding argument of the access-program. The variables introduced in the formal_arg_list are formal arguments of *Legality*. Their scope is the token_expr on the right-hand side of the legality.

A token_expr of the form Token(token) is always defined. If a token_expr is of the form TokenTable(token_entry_list) then its definedness conditions and evaluation are similar to these for trace_table (cf. 4.7.4.11). The value of the *Legality* function is the value of the second argument of Legality operator, trace_expr.

Note that the legality function in TAM corresponds to program preconditions in other formal specification methods.

5.6.4.2 Invocation sub-functions

If an invocation is legal, we describe the output values of the arguments and the returned value (if the access-program is function-like). We do it by means of a collection of functions and relations. Each invocation_function describes one access-program. For each output argument including the value returned by the program, there is invocation_sub_function. If an output argument is non-deterministic or the program's result type is labeled by R, then the corresponding invocation_sub_function describes a relation.

The formal_arg_list in each formal_invocation must be of the same length as the corresponding arg_description_list from the access-program table. If an invocation_sub_function describes an output argument then this argument is called a *distinguished argument*, and its form depends on the input/output mode as follows:

arg_mode	formal_arg	Interpretation
O()	FmlNArr(name_var_intro)	A name of an object
R()	FmlNR(name_var_intro, simple_var_intro)	A name of an object and its value after the invocation
VO()	FmlNVArr(name_var_intro, simple_var_intro)	A name of an object and its value before the invocation
VR()	FmlNVR(name_var_intro, simple_var_intro, simple_var_intro)	A name of an object, its value before the invocation, and its value after the invocation

If the same object is passed via several arguments, then all invocation sub-functions corresponding to these arguments must define the same value.

The operator of formal_invocation is defined by the following table:

invocation sub-function describes an output argument, and			invocation sub-function describes a value returned by the program, and	
program is not function-like, or is function-like and deterministic	program is function-like and non-deterministic, and		this value is deterministic	this value is not deterministic
	the distinguished argument is domestic	the distinguished argument is foreign		
FmlInvocation	FmlInvocationR	FmlInvocation	FmlInvocationArr	FmlInvocationR

If the operator is FmlInvocationR, then its third argument, simple_var_intro, represents the non-deterministically returned value.

The form of each non-distinguished argument and the semantics of the invocation sub-function depends on the fact whether the type of the distinguished argument is domestic or foreign, i.e., the invocation sub-function defines an extension function or an output relation.

5.6.4.2.1 The output relation

The form of each non-distinguished argument depends on its input/output mode as follows:

arg_mode	formal_arg	Interpretation
V()	FmlV(simple_var_intro)	A value
O(), R()	FmlN(name_var_intro)	A name of an object
VO(), VR()	FmlNV(name_var_intro, simple_var_intro)	A name of an object and its value before the invocation

The type of each simple variable introduced by a formal_arg is the same as the type of the corresponding argument of the access-program. The type of objects denoted by each name variable introduced by a formal_arg is the same as the type of the corresponding argument of the access-program.

The variables introduced in the formal_invocation are formal arguments of the invocation sub-function. The scope of these variables is the output_expr.

The output_expr of the invocation sub-function defining an output of a foreign type depends on the fact whether the output being described is deterministic.

If the output being described is deterministic, then the invocation_sub_function describes a function, and output_expr is of the form OutputValue(trace_expr). The value of this function is equal to the value of the argument of the operator OutputValue, trace_expr, with the actual arguments substituted for the formal ones. This value must be a canonical trace of the type of the corresponding distinguished argument or the type of the returned value.

If the output being described is non-deterministic, then the invocation_sub_function describes a relation and output_expr is of the form OutputValueConstraint(log_expr). The output value is represented by a simple variable, v, being either the second argument of the operator FmlNR, the third argument of the operator FmlNVR, or the third argument of the operator FmlInvocationR. An output value, T, is possible if the logical expression log_expr holds when we substitute T for v.

5.6.4.2.2 The extension function

The form of each non-distinguished argument depends on its input/output mode as follows:

arg_mode	formal_arg	Interpretation
V()	FmlV(simple_var_intro)	A value
O()	FmlN(name_var_intro)	A name of an object
R()	FmlNR(name_var_intro, simple_var_intro)	A name of an object and its value after the invocation

arg_mode	formal_arg	Interpretation
VO()	FmlNV(name_var_intro, simple_var_intro)	A name of an object and its value before the invocation
VR()	FmlNVR(name_var_intro, simple_var_intro, simple_var_intro)	A name of an object, its value before the invocation, and its value after the invocation

The type of each simple variable introduced by a formal_arg is the same as the type of the corresponding argument of the access-program. The type of objects denoted by each name variable introduced by a formal_arg is the same as the type of the corresponding argument of the access-program.

The variables introduced in the formal_invocation are formal arguments of the invocation sub-function. The scope of these variables is the output_expr.

The output_expr of the invocation sub-function defining an output of the domestic type must be of the form OutputValue(trace_expr) — all domestic outputs are deterministic. The value of this invocation sub-function is equal to the value of the trace_expr with the actual arguments substituted for the formal ones. This value must be a canonical trace of the domestic type.

Note that domestic output values may depend on non-deterministic foreign output values.

5.6.5 Comments on the presentation syntax

Each invocation_function describes one access-program. Therefore, the first ident in each formal_invocation must be the identifier of the described program.

5.7 Input variable event functions

5.7.1 Abstract syntax

input_var_event_function_list ::=

    [[input_var_event_function]]$^+$

input_var_event_function ::=

    InputVarEventFunction(formal_input_var_event, trace_expr)

formal_input_var_event ::=

    FmlInputVarEvent(ident, simple_var_intro, simple_var_intro)

5.7.2 Presentation syntax

input_var_event_function_list ::=

input_var_event_function | input_var_event_function
input_var_event_function_list

input_var_event_function ::=

formal_input_var_event = trace_expr

formal_input_var_event ::=

ident(simple_var_intro , simple_var_intro)

5.7.3 Referenced non-terminal symbols

ident	Section 3.2
simple_var_intro	Section 4.2
trace_expr	Section 4.7

5.7.4 Semantics

Each InputVarEventFunction(formal_input_var_event, trace_expr) describes the effect of the input variable event on the value of the domestic object. This effect does not depend on the identity of the object, thus the name of the object is not delivered among the arguments of InputVarEventFunction. The arguments of operator FmlInputVarEvent of formal_input_var_event are:

1. ident which identifies the input variable event which must be defined by InputVarCondition(condition, ident) introduced as one of elements of input_var_condition_list being the third argument of InputVarDeclaration(simple_var_intro, type, input_var_condition_list) in a module identified by qualifier, c.f. Section 4.5.
2. simple_var_intro given as the second argument of FmlInputVarEvent introduces a simple variable representing a value of the domestic object before the event occurred. This variable is of the domestic type and its scope is restricted to the trace_expr argument of the InputVarEventFunction operator.
3. simple_var_intro given as the third argument of FmlInputVarEvent introduces a simple variable representing a new value of the input variable related to this input variable event. The type of this simple variable is the type of the input variable as defined in the second argument of InputVarDeclaration(simple_var_intro, type, input_var_condition_list) and its scope is restricted to the trace_expr argument of the InputVarEventFunction operator. Note that for the new value of the conditional input variable defined by the BecomesTrue(log_expr) operator, the log_expr must hold, and for the new value of the conditional input variable defined by the BecomesFalse(log_expr) operator, the log_expr must not hold.

The value of the domestic object after the input variable event occurred is equal to the value of the trace_expr with the formal arguments substituted with actual arguments. The trace_expr must be defined and its value must be a canonical trace of the domestic type.

5.8 Definitions of values of output variables

5.8.1 Abstract syntax

output_var_value_list ::=

[[output_var_value]]$^{+}$

output_var_value ::=

OutputVarValue(formal_output_var, trace_expr)

formal_output_var ::=

FmlOutputVar(ident, simple_var_intro)

5.8.2 Presentation syntax

output_var_value_list ::=

output_var_value	output_var_value output_var_value_list

output_var_value ::=

formal_output_var = trace_expr

formal_output_var ::=

ident(simple_var_intro) ➴

5.8.3 Referenced non-terminal symbols

ident	Section 3.2
simple_var_intro	Section 4.2
trace_expr	Section 4.7

5.8.4 Semantics

Each OutputVarValue(formal_output_var_value, trace_expr) describes the value of an output variable as a function of the state of the corresponding domestic object. This value does not depend on the identity of the object, thus the name of the object is not delivered among the arguments of the OutputVarValue operator.

There is exactly one output_var_value equation for every output variable defined in the specification. The first two arguments of the FmlOutputVar operator are:

1. ident which identifies the output variable which has been introduced as the first argument of the OutputVarDeclaration operator in output_var_declaration.
2. simple_var_intro which introduces a simple variable representing a current value of the domestic object. This variable is of the domestic type and its scope is restricted to the trace_expr argument of the OutputVarValue operator.

The type of trace_expr must be the same as the type defined by type in OutputVarDeclaration(ident, type) of output_var_declaration. The value of trace_expr must be defined and canonical.

5.9 Definedness of specifications

A correct specification must satisfy the following conditions:

- The specification of any type introduced by foreign_types is correct.
- Dependencies between specifications implied by foreign types list cannot be circular (c.f. 5.2.4).
- The predicate canonical is total, i.e., the value of the first argument of the operator CanonicalSection, log_expr, is defined.
- If the empty_equivalence is of the form EmptyEquivalenceYes(*e*), then *e* is a canonical trace, and if the operator EmptyEquivalenceNo is used, then the empty trace is canonical.
- The auxiliary functions are defined, i.e., the value of the forth argument of the operator FunctionDeclaration, constraint, is defined, and if it evaluates to true, then the value of the last argument of this operator, trace_expr, is defined and belongs to the range of this function.
- The invocation functions are defined, i.e.,

  - the value of the second argument of the operator Legality, token_expr, is defined,

  - the value of the argument of the operator OutputValueConstraint, log_expr, is defined, and

  - the value of the argument of the operator OutputValue, trace_expr, is defined and canonical.
- The input variable conditions are defined, i.e., the argument of the operators BecomesTrue and BecomesFalse, log_expr, is defined.
- The input variable event functions are defined, i.e., the value of the second argument of the operator InputVarEventFunction, trace_expr, is defined and canonical.
- The value of the reduction function for each canonical trace must be this trace itself.
- Output variable values are defined, i.e., the value of the second argument of the operator OutputVarValue, trace_expr, is defined and canonical.
- Any recursion in the definitions of functions and relations must be terminating.

Chapter 6 Basic types

6.1 Predefined types

Certain types are used very often. They appear on the foreign types list in almost every specification. They are: bool, char, int, real.

We call them *predefined types* and assume that they are foreign in specifications of non-predefined types even if the predefined types are not present on the foreign types list. Auxiliary functions and constants of predefined types need not be qualified. Appendix A and Appendix B contain specifications of bool and int. A specification of real depends very much on the architecture of the computer, therefore for real we informally introduce available operations and the way in which constants of this type are written. Type char is defined as an enumeration type (cf. Section 6.2). As in the case of real, we describe how to use this type in other specifications.

6.1.1 Type bool

The specification of bool can be found in Appendix A. There are two canonical traces of type bool: the empty trace representing false and SUCC(*) representing true.

In a specification where bool is a foreign type, logical expressions and trace expressions that yield traces of bool are interchangeable:

- wherever a log_expr is expected, we can put a trace_expr of type bool (cf. 4.9.4.6);
- wherever a trace_expr of type bool is expected, we can put a log_expr (cf. 4.7.4.12).

Thus bool allows us to define auxiliary predicates: they are auxiliary functions with the range equal to bool.

6.1.2 Type int

The canonical traces of int represent arbitrary integers (thus the set of canonical traces is infinite). Each canonical trace is a sequence of SUCC(*) possibly ended with NEG(*). The represented integer is equal to the number of the invocations of SUCC, or if an invocation of NEG is present, to the negated number of occurrences of SUCC(*).

In specifications where int is foreign, we can use standard notation for integer constants: they are non-empty sequences of digits optionally preceded by "–" or "+". We can also use certain concepts based on int:

- auxiliary functions *count* and *length* (cf. 4.3.6),
- indexed variables (cf. 4.2.4.1.2),
- iterations (cf. 4.7.4.7).

The auxiliary functions from int: "+", (binary) "–", "*", "div", "mod", "<", "≤", ">", "≥" are written in the infix notation (this is an exception to the syntax of auxiliary function applications; user-defined functions cannot be infix). Unary "–" can be applied without parentheses surrounding its argument. The precedence of unary "–" is higher than the precedence of "➘" (cf. Section 4.10). "+", binary "–", "*", "div", "mod" are left associative. Their precedence (also "<", "≤", ">", "≥") is as follows (an operator has lower precedence than the operators in boxes on its left-hand side):

* **mod** **div**	+ –	< ≤ ≥ >

The precedence of "*", "**mod**", "**div**" is lower than the precedence of the dot ("**.**"), while the precedence of "<", "≤", ">", "≥" is equal to the precedence of "=" and "≠" (cf. Section 4.10).

6.1.3 Type real

A specification of real must define the following auxiliary functions: "+", (unary) "–", (binary) "–", "*", "/", "<", "≤", ">", "≥". All these functions should have their standard meaning. All but the unary minus are written in infix notation. The precedence and associativity of them are the same as for the corresponding operators of int ("/" corresponds to "div"). Note that these operators are overloaded which is not allowed for user-defined auxiliary functions.

6.1.4 Type char

Type char is defined as an enumeration type (cf. Section 6.2). A character enclosed in single quotes is a constant of type char. The set of these constants and their ordering is consistent with ASCII.

6.2 Enumeration types

In Appendix C we present a schema to specify enumeration types. Parameter n corresponds to the number of enumeration constants, and canonical traces correspond to these constants. The specification of an enumeration type defines auxiliary functions "<", "≤", ">" and "≥". The infix notation is used in their applications, they have the same precedence as the corresponding functions for int. Remember that this function symbols are overloaded (cf. 6.1.2 and 6.1.3).

The equation:

$$t = [c_1, c_2, \ldots, c_k]$$

can appear in places where the operator Instance is allowed. This equation can be regarded as the introduction of an instance of this schema. The number of constants, k, is the actual value of parameter n of the schema specification, and for i=1,2,...,k, c is an ident (cf. Section 3.2). Constant c_i represents canonical trace $[t::\mathrm{SUCC}(*)]_{j=1}^{i-1}$.

Appendix A Specification of bool

Bool Module Interface Specification

(0) CHARACTERISTICS

- type specified: bool

(1) SYNTAX

ACCESS-PROGRAMS

Program Name	Arg#1	Arg#2	Result Type
AND	bool : V	bool : V	bool
ASSIGN	bool : O	bool : V	
EQUIV	bool : V	bool : V	bool
IMPLIES	bool : V	bool : V	bool
NOT	bool : V		bool
OR	bool : V	bool : V	bool
PRED	bool : VO		
SUCC	bool : VO		
XOR	bool : V	bool : V	bool

(2) CANONICAL TRACES

$canonical(T) \Leftrightarrow T = \_ \vee T = \text{SUCC}(*)$

AUXILIARY FUNCTIONS

$and : \langle\text{bool}\rangle \times \langle\text{bool}\rangle \rightarrow \langle\text{bool}\rangle$

$and\ (x, y) \overset{\text{df}}{=}$

Condition	Value
$x = \_ \vee y = \_$	_
$x \neq \_ \wedge y \neq \_$	SUCC(*)

equiv : <bool> × <bool> → <bool>

equiv $(x, y) \stackrel{\text{df}}{=}$

Condition	Value
$x \neq y$	_
$x = y$	SUCC(*)

not : <bool> → <bool>

not $(x) \stackrel{\text{df}}{=}$ U where U : <bool> $[x \neq U]$

(3) SEMANTICS

ACCESS-PROGRAMS

Legality(AND(T, U)) = %legal%

AND(T, U)➘ = *and*(T, U)

Legality(ASSIGN(n, U)) = %legal%

ASSIGN(n➘, U) = U

Legality(EQUIV(T, U)) = %legal%

EQUIV(T, U)➘ = *equiv*(T, U)

Legality(IMPLIES(T, U)) = %legal%

IMPLIES(T, U)➘ = *not*(*and*(T, *not*(U)))

Legality(NOT(T)) = %legal%

NOT(T)➘ = *not*(T)

Legality(OR(T, U)) = %legal%

OR(T, U)➘ = *not*(*and*(*not*(T), *not*(U)))

Legality(PRED((n, T))) =

Condition	Value
T = _	%fatal%
$T \neq$ _	%legal%

PRED((n, T)➘) = _

Legality(SUCC((*n*, *T*))) =

Condition	Value
T = _	%fatal%
T ≠ _	%legal%

SUCC((*n*, *T*)➘) = SUCC(*)

Legality(XOR(*T*, *U*)) = %legal%

XOR(*T*, *U*)➘ = *not*(*equiv*(*T*, *U*))

Appendix B Specification of int

Integer Module Interface Specification

(0) CHARACTERISTICS

• type specified: int

• foreign types: bool

(1) SYNTAX

ACCESS-PROGRAMS

Program Name	Arg#1	Arg#2	Result Type
ASSIGN	int : O	int : V	
DIV	int : V	int : V	int
EQUAL	int : V	int : V	bool
LESS	int : V	int : V	bool
MINUS	int : V	int : V	int
MOD	int : V	int : V	int
NEG	int : VO		
PLUS	int : V	int : V	int
PRED	int : VO		
SUCC	int : VO		
TIMES	int : V	int : V	int

(2) CANONICAL TRACES

$canonical(T) \Leftrightarrow T = \_ \ \vee T = \text{SUCC(*).NEG(*)} \vee \exists T1 : \text{<<int>>}\ (T = \text{SUCC(*)}.T1)\ [canonical(T1)]$

AUXILIARY FUNCTIONS

$pred : \text{<int>} \rightarrow \text{<int>}$

$pred\ (x) \stackrel{df}{=}$

Condition	Value
$\exists !\ x1 : \text{<int>}\ [\ x = x1.\text{NEG(*)}]$	$\text{SUCC(*)}.x$
$x = \_$	SUCC(*).NEG(*)
$\exists !\ x1 : \text{<int>}\ [\ x = x1.\text{SUCC(*)}]$	$x1$

$succ : <\text{int}> \to <\text{int}>$

$succ\ (x) \stackrel{\text{df}}{=}$

Condition	Value
$\neg\exists!\ x1 : <\text{int}>\ [x = x1.\text{NEG}(*)]$	$x.\text{SUCC}(*)$
$x = \text{SUCC}(*).\text{NEG}(*)$	_
$\exists!\ x1 : <\text{int}>\ [x = x1.\text{SUCC}(*).\text{SUCC}(*).\text{NEG}(*)]$	$x1.\text{SUCC}(*).\text{NEG}(*)$

$+ : <\text{int}> \times <\text{int}> \to <\text{int}>$

$x + y \stackrel{\text{df}}{=}$

Condition	Value
$x = \_$	y
$x = \text{SUCC}(*).\text{NEG}(*)$	$pred(y)$
$\exists!\ x1 : <\text{int}>\ [x = x1.\text{SUCC}(*)]$	$succ(x1 + y)$
$\exists!\ x1 : <\text{int}>\ [x = x1.\text{SUCC}(*).\text{SUCC}(*).\text{NEG}(*)]$	$pred(x1.\text{SUCC}(*).\text{NEG}(*) + y)$

$- : <\text{int}> \to <\text{int}>$

$-x \stackrel{\text{df}}{=}$

Condition	Value
$x = \_$	_
$\exists!\ x1 : <\text{int}>\ [x = x1.\text{SUCC}(*)]$	$x.\text{NEG}(*)$
$\exists!\ x1 : <\text{int}>\ [x = x1.\text{NEG}(*)]$	$x1$

$- : <\text{int}> \times <\text{int}> \to <\text{int}>$

$x - y \stackrel{\text{df}}{=} x + (-y)$

$* : <\text{int}> \times <\text{int}> \to <\text{int}>$

$x * y \stackrel{\text{df}}{=}$

Condition	Value
$x = \_$	_
$\exists!\ x1 : <\text{int}>\ [x = x1.\text{SUCC}(*)]$	$x1 * y + y$
$\exists!\ x1 : <\text{int}>\ [x = x1.\text{NEG}(*)]$	$-(x1 * y)$

$\mathrm{div} : <\mathrm{int}> \times <\mathrm{int}> \to <\mathrm{int}>$

$x \ \mathrm{div} \ y \ (y \neq \_) \overset{\mathrm{df}}{=} z \ \text{ where } \ z, r : <\mathrm{int}> [z * y + r = x \wedge \mathrm{SUCC}(*).\mathrm{NEG}(*) < r \ \wedge r < y]$

$\mathrm{mod} : <\mathrm{int}> \times <\mathrm{int}> \to <\mathrm{int}>$

$x \ \mathrm{mod} \ y \ (y \neq \_) \overset{\mathrm{df}}{=} x - y * (x \ \mathrm{div} \ y)$

$< \ : <\mathrm{int}> \times <\mathrm{int}> \to <\mathrm{bool}>$

$x < y \overset{\mathrm{df}}{=} \exists z : <\mathrm{int}> [x - y = z.\mathrm{NEG}(*)]$

$\leq \ : <\mathrm{int}> \times <\mathrm{int}> \to <\mathrm{bool}>$

$x \leq y \overset{\mathrm{df}}{=} x = y \vee x < y$

$\geq \ : <\mathrm{int}> \times <\mathrm{int}> \to <\mathrm{bool}>$

$x \geq y \overset{\mathrm{df}}{=} \neg \, x < y$

$> \ : <\mathrm{int}> \times <\mathrm{int}> \to <\mathrm{bool}>$

$x > y \overset{\mathrm{df}}{=} \neg \, x \leq y$

(3) SEMANTICS

ACCESS-PROGRAMS

Legality(ASSIGN(n, T)) = %legal%

ASSIGN(n▲, T) = T

Legality(DIV(T, U)) =

Condition	Value
$U = \_$	%fatal%
$U \neq \_$	%legal%

DIV(T, U)▲ = T div U

Legality(EQUAL(T, U)) = %legal%

EQUAL(T, U)▲ = $T = U$

Legality(LESS(T, U)) = %legal%

LESS(T, U)▲ = $T < U$

Legality(MINUS(T, U)) = %legal%

MINUS(T, U)▲ = $T - U$

Legality(MOD(T, U)) =

Condition	Value
$U = \_$	%fatal%
$U \neq \_$	%legal%

MOD(T, U)➘ = T mod U

Legality(NEG((n, U))) = %legal%

NEG((n, U)➘) = $-\ U$

Legality(PLUS(T, U)) = %legal%

PLUS(T, U)➘ = $T + U$

Legality(PRED((n, T))) = %legal%

PRED((n, T)➘) = *pred*(T)

Legality(SUCC((n, T))) = %legal%

SUCC((n, T)➘) = *succ*(T)

Legality(TIMES(T, U)) = %legal%

TIMES(T, U)➘ = $T * U$

Appendix C Schema for enumeration type definitions

Enumeration Module Interface Specification

(0) CHARACTERISTICS

- type specified: enum
- foreign types: bool, int

(1) SYNTAX

ACCESS-PROGRAMS

Program Name	Arg#1	Arg#2	Result Type
ASSIGN	enum : O	enum : V	
ELEM	int : V		enum
EQUAL	enum : V	enum : V	bool
LESS	enum : V	enum : V	bool
ORD	enum : V		int
PRED	enum : VO		
SUCC	enum : VO		

(2) CANONICAL TRACES

$$canonical(T) \Leftrightarrow \exists i : \langle \mathrm{int} \rangle \; (i < n) \; [T = [\mathrm{SUCC}(*)]_{j=1}^{i}]$$

AUXILIARY FUNCTIONS

$$< : \langle \mathrm{enum} \rangle \times \langle \mathrm{enum} \rangle \rightarrow \langle \mathrm{bool} \rangle$$

$$x < y \overset{\mathrm{df}}{=} length(x) < length(y)$$

$$\leq : \langle \mathrm{enum} \rangle \times \langle \mathrm{enum} \rangle \rightarrow \langle \mathrm{bool} \rangle$$

$$x \leq y \overset{\mathrm{df}}{=} x = y \vee x < y$$

$$\geq : \langle \mathrm{enum} \rangle \times \langle \mathrm{enum} \rangle \rightarrow \langle \mathrm{bool} \rangle$$

$$x \geq y \overset{\mathrm{df}}{=} \neg\, x < y$$

$$> : \langle \mathrm{enum} \rangle \times \langle \mathrm{enum} \rangle \rightarrow \langle \mathrm{bool} \rangle$$

$$x > y \overset{\mathrm{df}}{=} \neg\, x \leq y$$

ord : <enum> → <int>

$ord(T) \overset{\text{df}}{=} length(T)$

pred : <enum> → <enum>

$pred\ (T)\ (T \neq \_)\ \overset{\text{df}}{=}\ T1$ where $T1$: <enum> [$T = T1$.SUCC(*)]

succ : <enum> → <enum>

$succ\ (T)\ (length(T) < n - 1)\ \overset{\text{df}}{=}\ T$.SUCC(*)

(3) SEMANTICS

ACCESS-PROGRAMS

Legality(ASSIGN(m, U)) = %legal%

ASSIGN(m➘, U) = U

Legality(ELEM(k)) =

Condition	Value
$0 > k \vee k > n - 1$	%fatal%
$0 \leq k \wedge k \leq n - 1$	%legal%

ELEM(k)➘ = $[\text{SUCC}(*)]_{i=1}^{k}$

Legality(EQUAL(T, U)) = %legal%

EQUAL(T, U)➘ = ($T = U$)

Legality(LESS(T, U)) = %legal%

LESS(T, U)➘ = ($T < U$)

Legality(ORD(T)) = %legal%

ORD(T)➘ = *ord*(T)

Legality(PRED((m, T))) =

Condition	Value
$T = \_$	%fatal%
$T \neq \_$	%legal%

PRED((m, T)➘) = *pred*(T)

Legality(SUCC((*m*, *T*))) =

Condition	Value
$length(T) = n - 1$	%fatal%
$length(T) < n - 1$	%legal%

SUCC((*m*, *T*)▲) = *succ*(*T*)

Appendix D Example Specification

Extended Stack Module Interface Specification

(0) CHARACTERISTICS

- type specified: example

(1) SYNTAX

ACCESS-PROGRAMS

Program Name	Arg#1	Arg#2	Result Type
JOIN	example:VO	example:VO	
MULT	example:VO		
NEG	example:VO		
PLUS	example:VO		
POP	example:VO	int:V	
PUSH	example:VO	int:V	
SHIFT	example:VO	int:V	
TOP	example:V		int

(2) CANONICAL TRACES

$$canonical(T) \Leftrightarrow \exists n\text{: <int>; } a[1] \ldots a[n]\text{: <int> } [T = [\mathrm{PUSH}(*, a[i])]_{i=1}^{n}]$$

(3) SEMANTICS

ACCESS-PROGRAMS

Legality(JOIN((n, T), (m, U))) = %legal%

JOIN((n, T)↘▲, (m, U)) = $T.U$

JOIN((n, T), (m, U)↘▲) =

Condition	Value
$n \neq m$	U
$n = m$	$T.U$

Legality(MULT((*n, T*))) =

Condition	Value
$length(T) < 2$	%too low%
$length(T) \geq 2$	%legal%

MULT((*n*, *T*)➶) = *B*.PUSH(*, int::TIMES(*x*, *y*)➶) where *B*: <example>; *x*, *y*: <int> [*T* = *B*.PUSH(*, *x*).PUSH(*, *y*)][1]

Legality(NEG((*n, T*))) =

Condition	Value
$T = \_$	%empty%
$T \neq \_$	%legal%

NEG((*n*, *T*)➶) = *B*.PUSH(*, int::NEG((*1, *x*)➶)) where *B*: <example>; *x*: <int> [*T* = *B*.PUSH(*, *x*)][2]

Legality(PLUS((*n, T*))) =

Condition	Value
$length(T) < 2$	%too low%
$length(T) \geq 2$	%legal%

PLUS((*n*, *T*)➶) = *B*.PUSH(*, *x* + *y*) where *B*: <example>; *x*, *y*: <int> [*T* = *B*.PUSH(*, *x*).PUSH(*, *y*)]

Legality(POP((*n, T*), *i*)) =

Condition	Value
$i < 0$	%fatal%
$i \geq 0 \wedge length(T) < i$	%too low%
$i \geq 0 \wedge length(T) \geq i$	%legal%

POP((*n*, *T*)➶, *i*) = *B* where *B*, *E*: <example> [$T = B.E \wedge length(E) = i$]

Legality(PUSH((*n, T*), *x*)) = %legal%

PUSH((*n*, *T*)➶, *x*) = *T*.PUSH(*, x)

1. The expression int::TIMES(*x*, *y*)➶ is used for illustration purpose only and could be rewritten as x * y.
2. The expression int::NEG((*1, *x*)➶) is used for illustration purpose only and could be rewritten as -x.

Legality(SHIFT((*n*, *T*), *i*)) =

Condition	Value
$i < 0$	%fatal%
$i \geq 0 \wedge length(T) < 2 * i$	%too low%
$i \geq 0 \wedge length(T) \geq 2 * i$	%legal%

SHIFT((*n*, *T*)➘, *i*) = *B*.*E*2 where *B*, *E1*, *E2*: <example> [$T = B.E1.E2 \wedge length(E1) = i \wedge length(E2) = i$]

Legality(TOP(*T*)) =

Condition	Value
T = _	%empty%
T ≠ _	%legal%

TOP(*T*)➘ = *x* where *B*: <example>; *x*: <int> [*T* = *B*.PUSH(*, *x*)]