Documentation Paradigms

(A Progress Report)

Michal Iglewski[1], Jan Madey[2], David Lorge Parnas, Philip C. Kelly

Telecommunications Research Institute of Ontario (TRIO)
CRL, McMaster University, Hamilton, Ontario, Canada L8S 4K1

ABSTRACT

This paper illustrates two of the computer systems documents discussed in [11]. A *module interface specification* treats the module as a black-box, identifying all module's access-programs (i.e. programs that can be invoked from outside the module), and describing their externally visible effects. The interface is documented by the trace assertion method [9]. A *module internal design document*, one for each implementation of the module, describes the module's (internal) data structure, the intended interpretation of that data structure (in terms of the module interface), and the effect of each access-program on the module's data structure. Program effect is described by a mapping from a set of data states before program execution to a set of data states after program execution. This mapping is expressed by LD-relations [7,12].

We illustrate these documents by presenting a few modules which were specified, designed, implemented, and tested within the Table Tool Project [15].

The paper contains also a sample implementation of these modules based on the module internal design document. The accompanying paper [1] describes the way this implementation was tested against the module interface specification.

1 Introduction

We are engaged in a long-term project to develop improved techniques for computer systems documentation. Computer systems should be documented with the same professionalism that we find in other engineering areas. Reviewers and maintainers should not have to deal with the ad hoc collection of vague sentences that is currently used to document software controlled systems. Just as mathematical techniques are used to document the properties of electrical components, bridge structures, and piping materials, mathematics is the key to precise, systematic documentation of computer hardware and software.

Computing research is often divided into (1) "theoretical" research, which examines formal models of computers to discover abstract properties of those models, (2) "formal methods" research, which develops deductive systems intended to allow the derivation of programs or proof of specified properties, and (3) development of useful computer systems. Our project attempts to build on all three areas by finding formal objects that can serve as precise descriptions of useful systems and may, eventually, be used to verify desired properties of the artifacts that are described.

Our main effort is a search for practical notations. In the earlier report [11], we have identified the mathematical functions that must be represented in computer system documentation. Our next task is to find useful representations

1 Permanent address: Département d'informatique, Université du Québec à Hull, Hull, Québec, Canada J8X 3X7
2 Permanent address: Institute of Informatics, Warsaw University, Banacha 2, 02-097 Warsaw, Poland

of those functions. These can form the basis of documents that contain the necessary information, are provably consistent, and easily read. Our work must be largely experimental. We experiment with new notational ideas by documenting various programs and asking others to use our "test" documents.

In the course of this research we have learned that tabular representations of mathematical functions [14] are very useful. However, we have also learned that the production of those tabular representations is labour intensive, time consuming, and error prone [10]. A second purpose of our project is to develop tools that make the use of these representations more practical [15].

The present paper grew out of both the experimental effort and the tool-development work. We have been developing precise specifications for the basic building blocks of our table production tools. As we expected, when we applied our "theory", we uncovered some aspects of our methods that were far from ideal. The basic models described in our earlier papers [9,11,12] are essentially unchanged but we have found a need to improve the notation and, in some cases to add some new ideas that extend of the old models. This paper presents recent improvements in our notations by illustrating their application to a few components of a system that we are building. The other purpose is to show how the various documents fit together.

Previous papers from our project have either had a narrow focus (e.g. requirements [16], program functions [12], or module specifications [9]), or have been very abstract [11]. This paper attempts to be both broad and concrete. It is intended to show concrete examples of the ideas in [11] that include module specifications, internal design documents, and code. Thus we are showing two things, (a) what the individual documents are like, and (b) how the documents and system fit together.

In doing this work we have had to face notational issues that did not arise when we worked with a narrow focus. The notations that we had developed for module specifications were not entirely consistent with the notational conventions that we used for program functions and some of those notations were not consistent with the language-dependent coding rules. Our work has required a great many iterations as we "adjusted" the notation for maximum consistency and readability, while remaining consistent with the basic theory reported in [11]. The programs have been tested against the abstract specification [1].

Our work has required a great deal of experimentation and we are not entirely happy with our current conventions. We are publishing this work to (a) give interested readers further insight into our methods, and (b) to obtain feedback and ideas from our colleagues and potential users. We invite comments and suggestions.

The report describes work-in-progress, not a finished product. Revised versions of the introductory papers are also in preparation.

In the next section we explain briefly the concepts of "black-box" and "clear-box" descriptions. In the section 3 we discuss the structure of corresponding documents, module interface specification and module internal design document. We present recent notational changes in both documents and clarify some concepts introduced in [9]. In the section 4 we give an example of documentation of three modules, illustrating the two documents. We also show how these modules could be implemented in C language.

2 "Black-box" and "clear-box" descriptions

Our documentation methods are based on a well established model of software design. As it is known for some 20 years, software should be hierarchically structured [5,6] and consist of a set of information-hiding modules [4], each one of which introduces a new type of variables (often called *objects*). One may group the variables into sets known as *abstract data types*, or, as is more usual, consider the sets of values of these variables to be data types. Each new data type may be used by programs in other modules to create higher-level data types.

In our model, the description of each of these modules consists of three documents. The first one (*a module interface specification*) gives a complete and precise "black-box" description of the behaviour of the variables created by the module. The second one (*a module internal design document*) describes the implementation of the module in terms of a data structure that consists of lower-level data types, i.e. it gives a “clear-box” description. The third document is the code itself. All three documents are mathematical texts and they are correct and consistent with each other only if they meet formal conditions which we will summarize below.

For each object we will have an *abstract* and a *concrete* description of its state. The first one is often also called a *high-level* or an *external* or a *black-box* description. The latter is often also called a *low-level* or an *internal* or a *clear-box* description. For each operation that we can perform on the object, the black-box description will tell us how the abstract state changes. The clear-box description will tell us how the concrete state changes. As part of our internal design documentation we will provide a mapping from the low-level states to the high-level states. The black-box and clear-box documents are consistent if the abstract state specified by the black-box description is the same as that specified in the clear-box documentation.

We provide the black-box descriptions using the “trace assertion method” as described in [9]. A *trace* is a complete history of the visible behaviour of the object. It includes all events affecting the object and all outputs produced by the object. Because the objects are finite state machines, the traces can be partitioned into a finite number of equivalence classes and each equivalence class can be represented by a single *canonical* trace. In the trace method, we describe a set of functions that map from single event extensions of canonical traces to the equivalent canonical traces. This provides a complete black-box description of the abstract values.

To provide the clear-box descriptions we use two kinds of formalisms. We describe the mapping from the concrete states to the abstract states by an *abstraction function*. We describe the behaviour of each program[1] (the effect of its application to a concrete data structure) by a relation and a set. The relation maps from the starting state of programs to the final states of programs. The set is the collection of starting states for which termination is guaranteed. Together this relation and set are called an *LD-relation* [7,12].

The relations and expressions in this report are usually presented in the form of tables whose elements are either conventional expressions or other tables. Each table is equivalent to a boolean expression and is the characteristic predicate of the relation (viewed as a set) [14]. The tables are not intended to be introductory, instead they provide detailed reference material. One uses them to “look things up”, not for conventional reading. Each of our examples also includes very brief introductory descriptions.

3 Interface specification and internal design of modules

In this section we will briefly describe two documents: a *module interface specification* and a *module internal design document*. We will present both the structure of these documents and explain certain relevant issues. The reader should also consult the Appendix A for definitions of the notation used here.

3.1 Module interface specification

This is a black-box description of a module, in which all externally visible features of the module are formally specified.

3.1.1 Document structure

A module interface specification will contain the following sections:

- Characteristics Section
- Syntax Section

1 We use the term *program* to denote a text describing a set of state sequences that may occur on a digital (finite state) machine.

- Canonical Trace Section
- Equivalence Section
- Return Value Section

The Characteristics Section states whether the module specification is parameterized (cf. section 3.1.2) or not, single-object or not (cf. section 3.1.4), deterministic or not[1]. This section also lists the type defined by the module, foreign[2] types used by the module, possible specification parameters (cf. section 3.1.2), and the type of names used by this module (cf. section 3.1.5).

The Syntax Section contains an *access-program table* [3]. The access-program table contains one row for each access-program, one column for each argument, and an additional column for the value returned by the program. Each entry specifies the type of the corresponding argument and characterizes its role (cf. section 3.1.6).

The Canonical Trace Section defines a predicate, "canonical", whose domain is the set of syntactically correct traces. This section may also include definitions of auxiliary functions (including some auxiliary functions which are not used in this specification but in other, project-related, module specifications), possible additional notation for abstract values, and the lexicon of other notational abbreviations.

The Equivalence Section describes the *extension function*[4] for access-programs invocations[5]. This function defines the equivalence relation on traces, denoted "≡". If the empty trace is not canonical, then this section should specify a canonical trace equivalent to the empty trace.

The Return Value Section defines the values returned by access-programs (excluding values returned by output arguments of the domestic type, which are described by the extension function).

3.1.2 Parameterized specifications

Parameterized specifications define a class of module specifications, along with parameters that characterize the members of the class. A parameterized specification may not be used directly. One must first determine the values of the parameters, and then may use the resulting ordinary specification.

A parameter may be either a value of a specified type or a type. Parameters are listed in the Characteristics Section as *specification parameters*. Every parameterized specification has an implicit (first) parameter. This is given as the *type specified* in the parameterized specification. The actual type's name replaces all occurrences of type specified in the resulting ordinary specification.

Example - A parameterized specification called "gen_register".

type specified:	<register_of_names>
specification parameters:	tname, smallest:<integer>, biggest:<integer>

To create an instance of parameterized specification we provide a new name for the type defined by this instance, types for type parameters and values for value parameters.

Example:

gen_register(x_reg, <integer>, 1, 100)

The above given phrase introduces a new type <x_reg> defined as the instance of parameterized specification

1 In a non-deterministic module the Return Value Section describes a relation that is not a function.

2 We use the following terminology: the word *domestic* relates to the module being specified; the word *foreign* relates to any module different from the one being specified.

3 In the general case this section may also contain two other tables: an input table and an event table, both related to input variable events. In this paper we do not deal with input variable events at all.

4 This function is sometimes called a *reduction function*.

5 In the general case the extension function is to be defined also for input variable events.

gen_register. In the specification of this new type:

x_reg	replaces all occurrences of register_of_names,
<integer>	replaces all occurrences of tname,
1	replaces all occurrences of smallest and
100	replaces all occurrences of bigest.

3.1.3 Types

The *value* of an object is represented by the canonical trace equivalent to the actual history of the object. The set of values for a module is called a *type*. The operations allowed on values of a given type are defined by the module's access-programs. In case of standard types (i.e., <integer>) we use conventional notation for values and assume standard operations.

A reference to the existing type can be done by using the name of a module in angle brackets (e.g., <index>, indicating that the type is a set of canonical traces defined by the module called index; cf. section 4.2).

In certain situations we may need to use a subset of a set of values of a previously defined (or standard) type. This can be done either:

- by restricting a set of values from a certain type (e.g., { i: <integer> | i ≥ 0 }), or
- by listing values from a certain type (e.g., { 1, 4, 16, 25 }).

The resulting subtype can be given a name (e.g., <natural> $\overset{\text{df}}{=}$ { i: <integer> | i ≥ 0 }).

3.1.4 Single-object and multi-object modules

If a module is designed to implement a number of objects of type <m>, then we can specify its interface in two different ways:

(1) we can describe the module as implementing a single object of a new type (in terms of a single trace including all operations performed by the module's programs), or

(2) we can describe the module as a manager of a number of independent objects of type <m> (each object being characterized by a separate trace).

The difference between these two views is reflected in the whole specification, and in particular in the access-programs and the canonical traces.

Modules specified according to (1) are called *single-object modules*, while those specified according to (2) are called *multi-object modules*.

3.1.5 Names in multi-object modules

Names of objects are quite distinct from identifiers appearing in a specification (e.g. names of access-programs). *Names of objects* are values of another type, and hence some operations can be performed on them.

There is no special type designated for names of objects, values of any foreign type can be used for this purpose. We often choose integers as names. This does not exclude the usage of integers in the same module for another purpose.

The foreign type chosen for naming domestic objects is indicated in the Characteristics Section. The same set of values can be used by different modules to denote objects of different types. The usage of names in one module can be controlled by another module (cf. an example specification in section 4.3.1).

3.1.6 Access-program table

The purpose of the access-program table is to specify the types of access-programs' arguments, and to describe their input-output characteristics using combinations of the descriptors "N", "V" and "O".

From the module user's point of view, the descriptors are interpreted as follows:

- if an argument is marked "O", then a name must be provided in the invocation. The value associated with this name may be changed. The argument is called an *output argument*. Note that unless the argument is also marked "N", the behavior cannot depend on the actual name used.
- if an argument is marked "V", then a value should be provided. The actual argument can be a name, from which a value will be obtained (however, the value associated with the name can be changed only if the argument was also marked "O"). The argument is called an *input argument*.
- if an argument is marked "N", then a name must be provided and the behavior specified may depend on the actual name used. Unless the argument is also marked "V" the value associated with this name is irrelevant. Unless the argument is also marked "O" the value associated with the name will not be changed.

Note that a value of type <m> can be assigned to an object from the module m only by the use of access-programs from the module m. We assume the existence of an access-program "assignment", usually denoted by ":=", in every module implementation. This access-program is implicitly called by other modules to change the values of objects of type <m>.

3.1.7 Wild-card symbol "*"

The value (canonical trace) of an object is usually independent of names appearing as arguments of access-programs, in the trace of this object - we could view in fact a canonical trace as an equivalence class[1]. Since, however, a canonical trace is a trace (a representative of the equivalence class), we have to use a name in every position where it is required by syntax. There is also often the case that the value of an actual argument is the prefix of the canonical trace in which an invocation with this actual argument appears.

To simplify the notation and improve the readability of specifications in such situations, a wild-card symbol "*" can be used to replace the actual argument. Its meaning depends on the argument descriptors in the access-program table. Let the descriptors of the argument under consideration be:

(1) "O"; if the argument is written as "*", then the wild-card symbol denotes an arbitrary chosen but fixed (for all canonical traces) name of an object of the proper type;

(2) "V"; if the argument is written as "*" and the canonical trace is S1.P(…,*,…).S2, then the wild-card symbol represents the subtrace S1;

(3) "VO"; if the argument is written as "*", then the wild-card symbol represents the pair (n,v) where n and v are defined according to (1) and (2);

(4) "N" or "NO"; the name of the object carries information, hence the wild-card symbol cannot be used;

(5) "NV" or "NVO"; if the argument is written as the pair (n,*) then the wild-card symbol is defined according to (2).

Example: The access-programs INIT and PUT_INT are used to define the canonical traces in the index module (cf. section 4.2.1). Their descriptors are defined by the syntax table as follows:

Program Name	Arg#1	Arg#2	Arg#3	Arg#4	Result Type
INIT	<status>:O	<index>:VO			
PUT_INT	<status>:O	<index>:VO	<integer>:V	<integer>:V	

1 There are some rare cases where the module stores and uses the name as part of the value.

The expression $\mathrm{INIT}(*, *).[\mathrm{PUT\_INT}(*, *, j, a_j)]_{j=1}^{m}$ is used to define the format of canonical traces. It can be interpreted as

$$\mathrm{INIT}(n_s, (n_x,\_)).[\mathrm{PUT\_INT}(n_s, (n_x,\mathrm{prefix}_j), j, a_j)]_{j=1}^{m} \qquad \text{where}$$

- n_s is an arbitrary chosen but fixed name used by the status module,
- n_x is an arbitrary chosen but fixed name used by the index module,
- prefix_j stands for $\mathrm{INIT}(n_s, (n_x,\_)).[\mathrm{PUT\_INT}(n_s, (n_x,\mathrm{prefix}_k), j, a_j)]_{k=1}^{j-1}$

Let’s observe that the choice of names n_s and n_x has no impact on the way that the canonical traces are calculated.

3.1.8 Output values

In a non-deterministic module, the sequence of past events may not fully determine the object state, and hence an object trace includes, besides events affecting the object, outputs produced by the object. For non-deterministic arguments (or function values) the outputs represent the decisions made for non-deterministic choices. For deterministic arguments (or function values) the outputs are redundant information and may be omitted in a trace.

The output values may only be restricted (determined for deterministic modules) by the values of the inputs arguments. These include the object being operated on. For single-object modules, the object created by the module is an implicit argument. Output values are described in terms of the trace of the object and the arguments of the most recent invocation.

Example:

The canonical traces in gen_register module (cf. section 4.3.1) are defined as sequences $[\mathrm{PICK}(*)\ \ x_i]_{i=1}^{n}$, where x_i is the value returned by the i-th invocation of PICK in the trace.

3.1.9 Extension function

The signature of the extension function, ef, for an access-program P [1] in the multi-object module m is

$$ef : (N \times C) \times I \rightarrow C \times S$$

where N is a set of names used to denote objects of type <m>, C is a set of canonical traces defined by m, I is a set of access-program invocations of P (extended by returned values if P is non-deterministic), and S is a set of status tokens including at least the token %legal%. In case of a single-object module, the signature of ef is

$$ef : C \times I \rightarrow C \times S$$

To simplify the definition of the extension function, its domain is partitioned, and the function is described as the union of subfunctions. For every access-program there is one subfunction, so called a legality function, which defines the status token, and there is one extension subfunction for any output argument of the domestic type which defines the new value of this argument.

We assume that if the name of an object whose trace is being extended does not appear in the event description, the extension function will map onto the same canonical trace. The corresponding subfunction need not be written.

We will also use the following notational conventions:

- the extension subfunctions will be written in infix notation with the symbol “≡>” denoting these subfunctions (cf. examples below),
- the output argument being defined will be followed by the symbol “ ”.

1 In the general case the extension function is also defined for input variable events.

Examples: Let’s consider three access-programs from the index module in a simplified form (cf. section 4.2.1).

Program Name	Arg#1	Arg#2	Arg#3	Result Type
PUT_INT	<index>:VO	<integer>:V	<integer>:V	
GET_INT	<index>:V	<integer>:V		<integer>
CATENATE	<index>:NVO	<index>:NVO		

Depending on the number of domestic output arguments, we can use the following abbreviations:

- access-program without domestic output arguments:

  “Legality(GET_INT(T,i)) = %no_init%” is an abbreviation for “∀x ef((x,T), GET_INT(T,i)) = (T, %no_init%)”
- access-program with at least one domestic output argument:

  “Legality(PUT_INT((*,T), n, i)) = %legal%

  PUT_INT((*,T) , n, i) ≡> T.PUT_INT((*,T), n, i)” is an abbreviation for

  “∀x ef((x,T), PUT_INT((x,T),n,i)) = (T.PUT_INT(*,n,i), %legal%)”

3.1.10 The equality operator

The specifier of a module can use the equality operator to compare values of two terms. This does not mean that a user of this module is provided automatically with the equality operator. The latter is only the case when the specifier has created an access-program having this meaning. The specifier is free to choose the name of this program.

3.2 Module internal design document

This is a clear-box description of a module, in which the effect of access-programs’ invocations is expressed in terms of the internal data structure.

3.2.1 Document structure

A module internal design document will contain the following sections:

- Data Structure Section
- Abstraction Function Section
- Program Function Section

The Data Structure Section describes the module’s local data structures and specifies their initial values; those data structures will usually include objects of foreign types.

The Abstraction Function Section defines a mapping from a domain consisting of pairs (object name, data state) to a range containing all canonical traces for objects from this module.

The Program Function Section describes modifications of module data structures resulting from an access-program invocation and the possible return values [1] of this invocation. The program functions [cf. section 3.2.3] will be defined in the form of schemata, one for each access-program.

Auxiliary functions and the lexicon of additional notational conventions are described in the Data Structure Section.

1 In contrast to the trace method, the return values do not form a separate section but are included in the Program Function Section.

3.2.2 Abstraction function

An *abstraction function* af for an internal design of a multi-object module m has the following signature

$$\text{af: <name>} \times \text{local\_data\_structure} \rightarrow \text{<m>}$$

where <name> is a set of values used to denote objects of type <m>, and local_data_structure contains vectors of elements of already-defined foreign data types (data states) that are used to represent objects of type <m>. If m is a single-object module then <name> is omitted from the domain.

3.2.3 Schema of program functions

Program functions were introduced by Mills and others [3]. In the present paper the term "program function" is understood in a broader sense, as a "program LD-relation" (cf. section 2 and [12]). Additionally, we need to deal with more general concepts ("parameterized program functions" and "schemata of parameterized program functions") to be able to describe behaviour of parameterized programs. This section is an informal introduction to these concepts.

(a) What is a program function?

Our work is based on the assumption that a "program" is a text that describes a set of state change sequences in a previously identified data structure. The set of possible executions (as they are called in [12]) must be constrained by the program's description. The only information (beyond the text describing the program) that should be needed to determine the set of executions that could result from invocation of a program, should be the initial data state. For programs in this sense, an LD-relation gives the possible stopping states for each starting state. We call such an LD-relation, a *program function* (in short *pf*).

(b) When is a "program" not a program?

The texts that we call "procedures with arguments" are not programs in the sense used in the above paragraph. In general, until the actual arguments are known (i.e. until we see the invocation) we do not know which actual elements of the data state will be affected by the execution nor do we know what effect the execution will have on the data structure. Consequently, procedures with arguments cannot be described by program functions (because determining the set of possible executions requires knowing the actual arguments used in an invocation).

(c) What are parameterized program functions and schemata of parameterized program functions?

Procedures with arguments can be described by a function that maps from a description of those arguments to a program function. We have found two useful notations for these function-valued-functions (functionals), a simple one, known as a *parameterized program function* (in short *ppf*), and a more complex one, known as a *schema of parameterized program functions* (in short *sppf*).

(d) What is a parameterized program function?

The actual domain of a program function is a set of tuples containing one element for every variable in the machine. In practice, however, each program uses and affects only a very small part of that data structure. Consequently, for practical purposes, we write the program function as if the domain contained only a relatively small number of variables, which we consider relevant, or potentially relevant, to the behaviour of the specific program being described. For an invocation of a procedure without arguments, this consists of a *local data structure* used by that text. For an invocation of a procedure with arguments, the data state used in the program function description includes the local data structure plus additional data elements indicated by the actual arguments. We want to describe the program's effects by an LD-relation on this *extended data structure*.

In many cases, it is possible to describe the effects of a program execution in terms of an extended data structure, which comprises the data structure used in the program text, extended by variables that correspond to the formal arguments to the procedure that we are documenting. In other words, we follow programming language tradition and use the identifiers of the formal arguments as “stand-ins” for the actual arguments. The resulting *parameterized program function* (*ppf*) is described by a table in which the formal argument names may be used as if those arguments were part of the data structure used by the program. The actual program function for an invocation is obtained by substituting the actual argument for the formal argument in the ppf description.

We only use the ppf when the effect of the program is independent of the names of the variables. If the code is such that the identities of the actual arguments can change the behaviour (e.g. if aliasing could cause changes in the effect of the program), this simple substitution scheme does not suffice and we must use a schema of parameterized program functions (see below).

A pf is simply a special case of a ppf that arises when there are no arguments. For parameterized programs we use ppf where possible and sppf elsewhere.

(e) What is a schema of parameterized program functions?

For those cases where the effect of the program execution depends on the identities of the actual arguments we cannot use simple substitution to get the program function from a parameterized program function. Instead, we must produce a table in which the entries, themselves are representations of functions (e.g. tables) and denote the possible ppfs. The arguments to this schema of parameterized program functions should be the names of those variables whose names matter. The function described by the sppf will be evaluated for each invocation of the procedure in the program text and the result of that evaluation will be a table describing the pf or ppf for that invocation.

Below we summarize the domains and ranges of these three classes of functions.

- pf: the domain of a pf consists of tuples representing the local data state. The range is a set of tuples representing the local data state and one element for each formal argument used for output. If the access-program returns values in pascal-function style, then every tuple in the range will contain an additional element, corresponding to the returned value.
- ppf: the domain of a ppf consists of tuples containing, for every argument:
  - a name, if the argument has the single descriptor “N” or “O” [1],
  - a value, if the argument has the single descriptor “V”,
  - a pair (name,value) in other cases.

  The range is a set of program functions.
- sppf: The domain consists of tuples containing a name for every argument marked “N”. The range is a set of parameterized program functions.

Example: Let’s consider three access-programs from the index module (cf. section 4.2.1).

Program Name	Arg#1	Arg#2	Arg#3	Result Type
INIT	<status>:O	<index>:VO		
GET_INT	<status>:O	<index>:V	<integer>:V	<integer>
CATENATE	<status>:O	<index>:NVO	<index>:NVO	

The signatures of the corresponding ppf/sppf are as follows:

ppf_INIT: $[\text{<stname>}_1] \times (\text{<xname>},\text{<index>})_2 \rightarrow (ds \rightarrow ds \times \text{<status>}_1)$

ppf_GET_INT: $[\text{<stname>}_1] \times \text{<index>}_2 \times \text{<integer>}_3 \rightarrow (ds \rightarrow ds \times \text{<status>}_1 \times \text{<integer>}_R)$

sppf_CATENATE: $\text{<xname>}_2 \times \text{<xname>}_3 \rightarrow$

1 For standard types, we will use the type <address> as the type of names.

$$([\text{<stname>}_1] \times (\text{<xname>},\text{<index>})_2 \times (\text{<xname>},\text{<index>})_3 \rightarrow (\text{ds} \rightarrow \text{ds} \times \text{<status>}_1))$$

The type ds represents the data structure of the module; other types used in the signatures have indices indicating the corresponding argument position (the result type has the index R); if the value of an argument is not used by the ppf, its type is written in square brackets[1]; values of type <stname> are used as names in the status module, values of type <xname> are used as names in the index module; the result type of GET_INT, <integer>, appears as the last element in the range of program functions. The behavior of CATENATE depends on the identities of the actual arguments; program functions are obtained by evaluating the schema of parameterized program functions for a given invocation.

3.2.4 Notational conventions

3.2.4.1 Function domain descriptions

In both documents we accompany every definition of a function *f* by an independent description of its domain. The domain is described by indicating a set, called the Universe, and by possibly restricting this set to a subset. The Universe can be restricted in two ways:

(a) by providing the characteristic predicate of the domain; we will denote it by "*f_domain*(…)".

(b) by including restrictions in the table describing the function; the table can contain one or more rows with a predicate on the left-hand side and the empty (shadowed) right-hand side. This is the statement that the values characterized by the left-hand side predicate are not in the domain.

If neither (a) nor (b) is used, then by default the domain is the Universe.

Example: The definitions A) and B) of the auxiliary function *putInt*, given below, are equivalent. The Universe that includes the domain is the set <index> × <integer> × <integer>, *canPut* and *nbElems* are auxiliary functions (cf. section 4.2.1).

A) *putInt*: <index> × <integer> × <integer> → <index>

putInt_domain(x, n, i) ⇔ (x ≠ _) ∧ *canPut*(x,n)

putInt(x, n, i) $\overset{\text{df}}{=}$

Condition	Value
n < (*nbElems*(x) + 1)	α
n = (*nbElems*(x) + 1)	β

B) *putInt*: <index> × <integer> × <integer> → <index>

putInt(x, n, i) $\overset{\text{df}}{=}$

Condition	Value
(x = _) ∨ ¬ *canPut*(x,n)	▨
(x ≠ _) ∧ *canPut*(x,n) ∧ (n < (*nbElems*(x) + 1))	α
(x ≠ _) ∧ *canPut*(x,n) ∧ (n = (*nbElems*(x) + 1))	β

If the convention A) is used, the union of all the row headings must be true if the condition *putInt_domain*(x, n, i) is satisfied.

If the convention B) is used, the union of the row headings for non-shadowed rows must be equivalent to "*putInt_do-*

1 It may happen, e.g., in case of an argument whose only specifier is "O". We keep this argument in the ppf signature to obtain consistency with a program declaration in a traditional programming language.

main”.

3.2.4.2 “Before” and “after” notation

The following notational short-cut will be used in expressions (tables) describing program functions. The mathematical variables corresponding to values of local objects before the invocation will be denoted by the same textual name as their programming counterpart, primed on their left (to be read: “before”). The mathematical variables corresponding to values of these objects after the invocation will be denoted by the same name, primed on their right (to be read: “after”). For example, if x is an object, then “ ‘x ” is read “x before” and represents the value of this object before the invocation of the program.

3.2.4.3 Table formats

The tables defining program functions can contain two kinds of entries:

- expressions; the leftmost column contains a variable followed by the equality symbol “=”; the value is defined by an expression of the corresponding type.
- predicates; the leftmost column contains a variable, followed by the “such that” symbol, “|”; the value is characterized by a boolean expression (cf. Example in section 3.2.4.4).

If a table contains an entry of the form “v’ | true”, then this is the statement that the value of v’ could be any value of the type specified earlier in the document.

3.2.4.4 The *NC* and *NCP* notations

If $v_1,\ldots,v_k$ are variables whose values should not change, then we can shorten notation in the specification (both outside and inside a table) by using the standard predicate *NC* (“not changed”) defined as follows:

$$NC(v_1, \ldots, v_m) \overset{df}{=} (v_1' = {}'v_1) \wedge \ldots \wedge (v_m' = {}'v_m)$$

If a variable v is a pointer, then the notation *NCP*(v) means that both v and the whole data structure “reachable” via v are not changed (this is to be understood recursively). The section 4.2.2 contains an example of formal definition of the predicate *NCP*.

The program functions are often described by a conjunction of the predicate *NC* and the predicate represented by the table.

Example: The program function pf_PICK (cf. section 4.3.2) is defined as follows (the module data structure is composed of two variables, table and fl; “ ” represents the value returned by the program PICK):

pf_PICK $\overset{df}{=}$

		‘fl < 0	‘fl ≥ 0
table’	|	*NC*(table)	(table’[‘fl].used = TRUE) $\wedge\ \forall$ j (0 ≤ j ≤ (size–1)) [(j ≠ ‘fl) ⇒ *NC*(table[j])]
fl’	=	‘fl	‘table[‘fl].next
	=	0	‘fl + smallest

3.2.4.5 Built-in types vs. data structures

To simplify the notation, it is possible to mix the usage of abstract values (traces) with data state expressions if such a usage is not ambiguous.

Example: Consider the function pf_PICK from the internal design document for the module gen_register, given above. The value returned by PICK is of the type <integer>. The expression “fl+smallest” describing this value is given in terms of module data structure, and will yield a value of C type “int”. It should be interpreted as af(fl+smallest) where af is the abstraction function from the internal design document for integer module and the C type “int” is used to implement objects of type <integer>. Additionally, one of the values of the C type “int” (usually, 0) should be interpreted as the empty trace of <integer> type.

3.2.4.6 Miscellaneous

See the Notation Guide (Appendix A) for explanation of additional symbols (e.g. “ ”).

4 Example of documentation

4.1 Introduction

For the illustration of the documents discussed in the previous sections, we will present sample modules which were specified, designed, implemented, and tested within the Table Tool Project (in short: TTP) [15]. We have selected two multi-object modules (“index” and “status”) and one parameterized single-object module (“gen_register”).

A section devoted to each module will contain:

(1) Module Interface Specification, as discussed in section 3.1, in which we have additionally a brief informal description of the module.

(2) Module Internal Design Document, as discussed in section 3.2. This document is to be written in terms of the programming language chosen for the implementation. In case of TTP, this language is C [2].

(3) Example implementation in C. Certain implementation conventions which apply to all modules in TTP are presented separately, in section 4.1.2.

The choice of the modules from TTP was motivated as follows:

- The index module defines a seemingly simple data type - a one-dimensional vectors of integers. Since, however, we want to allow full dynamism (a vector can expand or shrink), a precise description of the semantics of all operations is not trivial.
- The gen_register module is an example of a parameterized specification. As we have mentioned before, each multi-object module specification (e.g. index) must state the type of values which will be used as names. These names can be generated using an instance of the parameterized single-object module specification gen_register. The gen_register specification has three parameters: an (ordered) type whose values will be used as names, and two values of the type defined by the first parameter, representing the range of possible names. Each instance of this class of modules can be viewed as a register of names used by a multi-object module.
- The status module defines the set of status tokens used by all modules in TTP. It is a very simple module, however, it illustrates how an error reporting mechanism could be implemented (cf. section 4.1.1).

Footnotes in this document are explanations that would not be needed in actual documents.

4.1.1 Error communication

In the trace assertion method one can express the fact that the user of a module should avoid certain input events (invocations of access-programs). In such cases the extension function yields a status token different than %legal%.

In the module internal design document we can take this information into account and specify appropriate behaviour of the access-programs in case of non-legal status token, i.e. design a special error communication mechanisms.

The example presented in this paper illustrates one of many possible solutions for the error handling problem. It is as follows:

- all access-programs from the index module have an additional output argument of type <status>. It is used to indicate whether or not an invocation is “legal” and to classify the illegal ones. Its value corresponds directly to the status token defined by the extension function.
- the instances of the gen_register specification and the status module can be viewed as system modules. To avoid loops, they have been designed in such a way that every invocation of these modules’ access-programs is legal.

4.1.2 Implementation conventions

For each module named m we will have two files with C code:

- m.h - containing the shared declarations of the module,
- m.c - containing the code of access-programs.

The file global.h contains the declarations shared by all modules.

For each access-program we will have one C function. As the name of an object determines the elements of the module data structure, we do not need to pass this structure as an argument to the C function. Similarly, in case of a single-object module the object name (which we have chosen to be the module name) can be omitted.

Henceforth, in all C functions corresponding to access-programs:

- for types treated as built-in programming language types (integer, boolean) we will pass the values of corresponding objects if the argument is not supposed to be changed, and their address otherwise,
- for other types we will pass the names of the corresponding objects.

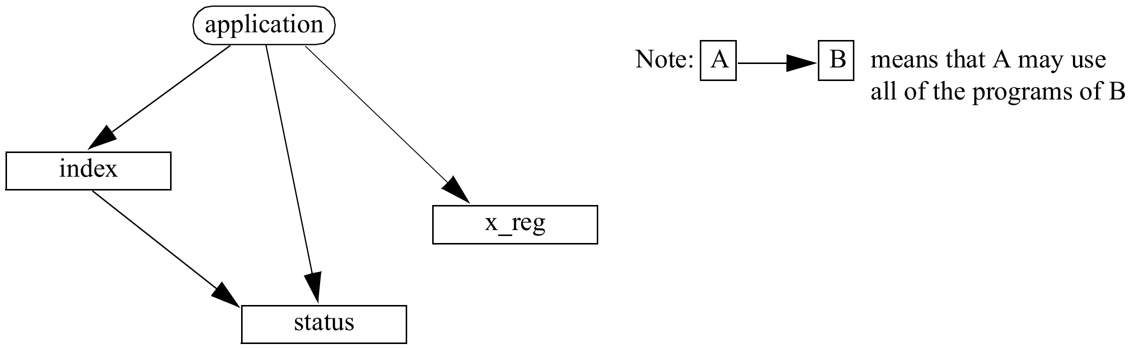
The C name of the access-program P from the module m will be written as m_P.

4.1.3 Depedencies between modules

Three modules discussed in this paper can be used by an external program “application”. The figure below illustrates the dependencies between them. The module “x_reg” is an instance of the parameterized module gen_register and is defined as follows:

gen_register(x_reg, <integer>,1,100)

An example of application is the program “index_test” used to test the index module [1].



4.2 The index module

4.2.1 index Module Interface Specification

Informal description

The index module is a multi-object module that manages objects of type <index>. The set of non-empty canonical index traces is isomorphic to the set of one dimensional vectors of integers, of maximum length 50. A non-empty index can be shortened by removing one element from it; an index with length below 50 can be extended by appending an element to it.

In the following informal descriptions of the access-programs[1], s is a name denoting an object of type <status>, x and y are names denoting objects of type <index>, T and U are values of type <index>, i and n are values of type <integer>. The first argument in each access-program returns a value of <status> type. It is used to indicate whether or not an invocation is legal and to classify the illegal ones. Its exact meaning is described in the Return Value Section.

INIT(s, x) associates an index of length zero with x.

DESTROY(s, x) clears the history of the object x by associating the empty trace with x.

PUT_INT(s, x, n, i) sets the nth element of the object x to i if n is between 1 and the length of x, or adds a new element with the value i if n equals the length of x plus one.

GET_INT(s, T, n) returns the nth element of T.

IS_EQUAL(s, T, U) returns true if T has the same value (canonical trace) as U.

COPY(s, U, x) associates the value of U with x.

NB_ELEMS(s, T) returns the number of elements in T.

DELETE(s, x, n) shortens the object x by deleting its nth element.

CATENATE(s, x, y) associates with x an index with all the elements of x followed by all the elements of y.

(0) CHARACTERISTICS

- type specified: <index>
- features: multi-object, deterministic
- foreign types: <boolean>, <integer>, <status>
- type of names: { i: <integer> | $1 \le i \le 100$ }

1 Note that these informal descriptions are not, and are not intended to be, complete or very precise. They are introductory.

(1) SYNTAX

ACCESS-PROGRAMS

Program Name	Arg#1	Arg#2	Arg#3	Arg#4	Result Type
CATENATE	<status>:O	<index>:NVO	<index>:NVO		
COPY	<status>:O	<index>:V	<index>:VO		
DELETE	<status>:O	<index>:VO	<integer>:V		
DESTROY	<status>:O	<index>:VO			
GET_INT	<status>:O	<index>:V	<integer>:V		<integer>
INIT	<status>:O	<index>:VO			
IS_EQUAL	<status>:O	<index>:V	<index>:V		<boolean>
NB_ELEMS	<status>:O	<index>:V			<integer>
PUT_INT	<status>:O	<index>:VO	<integer>:V	<integer>:V	

(2) CANONICAL TRACES

canonical(T) $\Leftrightarrow$ (T = _) $\vee$ (T = INIT(*, *).$[\text{PUT\_INT}(*, *, j, a_j)]_{j=1}^{m}$) $\wedge$ $(0 \le m \le \text{xmax})$

LEXICON

xmax $\overset{\text{df}}{=}$ 50 /* maximum length of an index */

AUXILIARY FUNCTIONS

canGet: <index> × <integer> → <boolean>

canGet(x, n) $\overset{\text{df}}{=}$ (x ≠ _) $\wedge$ $(1 \le n)$ $\wedge$ (n ≤ *nbElems*(x))

canPut: <index> × <integer> → <boolean>

canPut(x, n) $\overset{\text{df}}{=}$ (x ≠ _) $\wedge$ $(1 \le n)$ $\wedge$ (n ≤ (*nbElems*(x) + 1))

getInt: <index> × <integer> → <integer>

getInt_domain(x, n) $\Leftrightarrow$ *canGet*(x, n)

getInt(x, n) $\overset{\text{df}}{=}$ i where x = T1.PUT_INT(*, *, n, i).T2

nbElems: <index> → <integer>

nbElems(x) $\overset{\text{df}}{=}$ *count*(x, PUT_INT)

putInt: <index> × <integer> × <integer> → <index>

putInt_domain(x, n, i) $\Leftrightarrow$ *canPut*(x, n) *putInt*(x, n, i) $\overset{\text{df}}{=}$

Condition	Value
n < (*nbElems*(x) + 1)	T1.PUT_INT(*, *, n, i).T2 where x = T1.PUT_INT(*, *, n, j).T2
n = (*nbElems*(x) + 1)	x.PUT_INT(*, *, n, i)

(3) EQUIVALENCES

Legality(CATENATE(*, (x,T), (y,U))) =

Condition		Value
(T = _) ∨ (U = _)		%no_init%
(T ≠ _) ∧ (U ≠ _) ∧ ((*nbElems*(T) + *nbElems*(U))	> xmax)	%out_of_space%
	≤ xmax)	%legal%

CATENATE(*, (x,T) , (y,U)) ≡> T.[PUT_INT(*, *, j+*nbElems*(T), a_j)]$_{j=1}^{m}$

where U = INIT(*, *).[PUT_INT(*, *, j, a_j)]$_{j=1}^{m}$

CATENATE(*, (x,T), (y,U)) ≡>

Condition	Value
(x ≠ y)	U
(x = y)	T.[PUT_INT(*, *, j+*nbElems*(T), a_j)]$_{j=1}^{m}$ where U = INIT(*, *).[PUT_INT(*, *, j, a_j)]$_{j=1}^{m}$

Legality(COPY(*, U, (*,T))) =

Condition	Value
(T = _) ∨ (U = _)	%no_init%
(T ≠ _) ∧ (U ≠ _)	%legal%

COPY(*, U, (*,T)) ≡> U

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

Condition		Value
T = _		%no_init%
(T ≠ _) ∧	¬*canGet*(T, n)	%bad_access%
	canGet(T, n)	%legal%

DELETE(*, (*,T) , n) ≡> T1.[PUT_INT(*, *, j–1, a_j)]$_{j=n+1}^{m}$

where T = T1.[PUT_INT(*, *, j, a_j)]$_{j=n}^{m}$

Legality(DESTROY(*, (*,T))) =

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

DESTROY(*, (*,T)) ≡> _

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

Condition		Value
T = _		%no_init%
(T ≠ _) ∧	¬*canGet*(T, n)	%bad_access%
	canGet(T, n)	%legal%

Legality(INIT(*, (*,T))) =

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

INIT(*, (*,T)) ≡> INIT(*, *)

Legality(IS_EQUAL(*, T1, T2)) = %legal%

Legality(NB_ELEMS(*, T)) =

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

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

Condition		Value
T = _		%no_init%
(T ≠ _) ∧	¬*canPut*(T, n)	%bad_access%
	canPut(T, n) ∧ (n > xmax)	%out_of_space%
	canPut(T, n) ∧ (n ≤ xmax)	%legal%

PUT_INT(*, (*,T), n, i) ≡> *putInt*(T, n, i)

(4) RETURN VALUES[1]

Invocation	Value		
CATENATE(* , (x,T), (y,U))	$(T = \_) \vee (U = \_)$		init
	$(T \neq \_) \wedge (U \neq \_) \wedge ((\mathit{nbElems}(T) + \mathit{nbElems}(U))$	$> \mathrm{xmax})$	space
		$\leq \mathrm{xmax})$	legal
COPY(* , U, (*,T))	$(U = \_) \vee (T = \_)$		init
	$(U \neq \_) \wedge (T \neq \_)$		legal
DELETE(* , (*,T), n)	$T = \_$		init
	$(T \neq \_) \wedge \neg \mathit{canGet}(T, n)$		access
	$(T \neq \_) \wedge \mathit{canGet}(T, n)$		legal
DESTROY(* , (*,T))	$T = \_$		init
	$T \neq \_$		legal
GET_INT(*, T, n)	$\mathit{getInt}(T, n)$		
GET_INT(* , T, n)	$T = \_$		init
	$(T \neq \_) \wedge \neg \mathit{canGet}(T, n)$		access
	$(T \neq \_) \wedge \mathit{canGet}(T, n)$		legal
INIT(* , (*,T))	$T \neq \_$		nonempty
	$T = \_$		legal
IS_EQUAL(*, T1, T2)	$T1 = T2$		
IS_EQUAL(* , T1, T2)	legal		
NB_ELEMS(*, T)	$\mathit{nbElems}(T)$		
NB_ELEMS(* , T)	$T = \_$		init
	$T \neq \_$		legal
PUT_INT(* , (*,T), n, i)	$T = \_$		init
	$(T \neq \_) \wedge \neg \mathit{canPut}(T, n)$		access
	$(T \neq \_) \wedge \mathit{canPut}(T, n) \wedge (n$	$> \mathrm{xmax})$	space
		$\leq \mathrm{xmax})$	legal

1 In the following table we give the values returned by each access-program. The types of those values are given in the Syntax Section. The meaning of the values can be found in the specification for that type.

4.2.2 index Internal Design Document

Informal description

The internal data structure of index module consists of two variables, t and h. The variable t is an array of pointers of indexObj type. Pointers correspond to single objects of index type. The empty trace is represented by the null pointer value. If the pointer value t[x] is different from the empty trace then t[x]->els is a pointer to a variable size array and t[x]->len contains the number of its elements.

The variable h of the type "heap" represents the whole dynamic memory which can be allocated by C allocation routines. In fact, all variables of indexObj type are a part of h. The variable h could be viewed as a set of "allocated" disjoint blocks of addresses, possibly with assigned values. We assume that the following three operators, *mallocated*, $->_h$, $[]_h$, have been defined on values of type heap:

- '*mallocated*(h,p,i)' allows to check if a block of length i is allocated at address p,
- 'structure$->_h$field' corresponds to the C operator of dereferencing (we write a subscript $_h$ to show that the operator is to be evaluated for a given contents of the variable h),
- 'ptr$[i]_h$' corresponds to the C operator of subscripting.

The invariant "well formed data structure" (*wfds*) constrains the behaviour of the module internal data structure. The predicate *wfds* need not be stated explicitly in the descriptions of the abstraction function and program functions; they "assume" that the constraint will hold in all states encountered. The predicate *wfds* makes use of the predicate *NCP* discussed in section 3.2.4.4. We show how *NCP* can be described in a formal way, for a dynamic data structure defined in C programming language.

(1) DATA STRUCTURE

CONSTANT DEFINITIONS

Constant Name	Definition
EMPTY	(indexObj)0 /* null pointer (empty trace) */
SIZE	100 /* max. number of index objects */
XMAX	50 /* max. length of an index */

TYPE DEFINITIONS

Type Name	Definition
indexArr	typedef int (*indexArr)[];
indexStr	typedef struct { int len; /* $0 \leq$ len $\leq$ XMAX */ indexArr els; /* (len > 0) $\Rightarrow$ "els is a valid pointer" */ } indexStr;
indexObj	typedef indexStr * indexObj;
vector	typedef indexObj vector[SIZE];
heap	"a set of disjoint blocks of addresses, possibly with assigned values"

VARIABLE DEFINITIONS AND INITIALIZATIONS

Type Name/Definition	Variable Name	Initial Condition
vector	t	$\forall i\ (0 \le i \le (\mathrm{SIZE}-1))\ [t[i] = \mathrm{EMPTY}]$
heap	h	$\forall i,p\ [\ \neg \mathit{mallocated}(h,p,i)\ ]$

DATA STATE CONSTRAINTS

wfds(t,h)

AUXILIARY FUNCTIONS

areEqEls: vector × heap × int × int × vector × heap × int × int × int → <boolean>

$$\mathit{areEqEls\_domain}(t1, h1, x, a, t2, h2, y, b, k) \Leftrightarrow (0 \le x < \mathrm{SIZE}) \wedge (0 \le y < \mathrm{SIZE}) \wedge (a \ge 0) \wedge (b \ge 0) \wedge (k \ge 0) \wedge (a+k \le \mathrm{XMAX}) \wedge (b + k \le \mathrm{XMAX}) \wedge (t1[x] \ne \mathrm{EMPTY}) \wedge (t2[y] \ne \mathrm{EMPTY}) \wedge \wedge ((k > 0) \Rightarrow ((\mathit{len}(t1,h1,x) > 0) \wedge (\mathit{len}(t2,h2,y) > 0)))$$

$$\mathit{areEqEls}(t1, h1, x, a, t2, h2, y, b, k) \overset{\mathrm{df}}{=} \forall j\ (0 \le j \le (k-1))\ [\ \mathit{els}(t1,h1,x)[a+j]_{h1} = \mathit{els}(t2,h2,y)[b+j]_{h2}\ ]$$

canGet: vector × heap × <integer> × <integer> → <boolean>

$\mathit{canGet\_domain}(t, h, x, n) \Leftrightarrow t[x] \ne \mathrm{EMPTY}$

$\mathit{canGet}(t, h, x, n) \overset{\mathrm{df}}{=} (1 \le n \le \mathit{len}(t,h,x))$

canPut: vector × heap × <integer> × <integer> → <boolean>

$\mathit{canPut\_domain}(t, h, x, n) \Leftrightarrow t[x] \ne \mathrm{EMPTY}$

$\mathit{canPut}(t, h, x, n) \overset{\mathrm{df}}{=} (1 \le n \le (\mathit{len}(t,h,x)+1))$

els: vector × heap × int → indexArr

$\mathit{els\_domain}(t, h, x) \Leftrightarrow t[x] \ne \mathrm{EMPTY}$

$\mathit{els}(t, h, x) \overset{\mathrm{df}}{=} t[x]\text{->}_h\mathrm{els}$

isEqual: vector × heap × int × int → <boolean>

$\mathit{isEqual}(t, h, y, z) \overset{\mathrm{df}}{=}$

Condition	Value
$(t[y] = \mathrm{EMPTY}) \wedge (t[z] = \mathrm{EMPTY})$	true
$(t[y] = \mathrm{EMPTY}) \wedge (t[z] \ne \mathrm{EMPTY}) \vee$ $(t[y] \ne \mathrm{EMPTY}) \wedge (t[z] = \mathrm{EMPTY})$	false
$(t[y] \ne \mathrm{EMPTY}) \wedge (t[z] \ne \mathrm{EMPTY})$	$(\mathit{len}(t,h,y) = \mathit{len}(t,h,z)) \wedge$ $\forall j\ (1 \le j \le \mathit{len}(t,h,z))\ [\ \mathit{els}(t,h,y)[j-1]_h = \mathit{els}(t,h,z)[j-1]_h\ ]$

len: vector $\times$ heap $\times$ int $\rightarrow$ int

len_domain(t, h, x) $\Leftrightarrow$ t[x] $\neq$ EMPTY

len(t, h, x) $\overset{df}{=}$ t[x]-$>_h$len

$mallocated$: heap $\times$ void* $\times$ int $\rightarrow$ <boolean>

$mallocated$(h, p, i) $\overset{df}{=}$ “h is such that a block of length i is allocated starting at address p”

nc_ptr: vector $\times$ heap $\times$ vector $\times$ heap $\times$ int $\rightarrow$ <boolean>

nc_ptr(‘t, ‘h, t’, h’, i) $\overset{df}{=}$ (‘t[i] = t’[i]) $\wedge$

((‘t[i] $\neq$ EMPTY) $\Rightarrow$ ((len(‘t,‘h,i) = len(‘t,h’,i)) $\wedge$

((len(‘t,‘h,i) > 0) $\Rightarrow$ (els(‘t,‘h,i) = els(‘t,h’,i))) $\wedge$

$\forall$j(0 $\leq$ j < len(‘t,‘h,i)) [els(‘t,‘h,i)[j]$_{‘h}$ = els(‘t,h’,i)[j]$_{h’}$])

$wfds$: vector $\times$ heap $\rightarrow$ <boolean>

$wfds$(t,h) $\overset{df}{=}$ $\forall$i (0$\leq$i<SIZE) [((t[i] = EMPTY) $\vee$ $mallocated$(h, t[i], sizeof(indexStr))) $\wedge$

((t[i] $\neq$ EMPTY) $\Rightarrow$ ((0 $\leq$ t[i]-$>_h$len $\leq$ XMAX) $\wedge$

((t[i]-$>_h$len > 0) $\Rightarrow$ $mallocated$(h, t[i]-$>_h$els, t[i]-$>_h$len * sizeof(int))))] $\wedge$

$\forall$i,j ((0$\leq$i,j<SIZE) $\wedge$ (t[i] $\neq$ EMPTY) $\wedge$ t[j] $\neq$ EMPTY)) [

((i $\neq$ j) $\Rightarrow$ (t[i] $\neq$ t[j]) $\wedge$

((t[i]-$>_h$len > 0) $\wedge$ (t[j]-$>_h$len > 0) $\Rightarrow$ (t[i]-$>_h$els $\neq$ t[j]-$>_h$els))) $\wedge$

((t[j]-$>_h$len > 0) $\Rightarrow$ (t[i] $\neq$ t[j]-$>_h$els))]

LEXICON OF TYPES

ds $\overset{df}{=}$ vector $\times$ heap

<xname> $\overset{df}{=}$ {i: <integer> | 1 $\leq$ i $\leq$ SIZE }

<stname> $\overset{df}{=}$ {i: <integer> | 1 $\leq$ i $\leq$ 1000 }

LEXICON OF MACROS

NCP(t[i]) $\overset{df}{=}$ nc_ptr(‘t, ‘h, t’, h’, i)

NCP_all $\overset{df}{=}$ $\forall$j (0 $\leq$ j < SIZE) [NCP(t[j])]

NCP_but_one(x) $\overset{df}{=}$ $\forall$j ((0 $\leq$ j < SIZE) $\wedge$ (j $\neq$ x)) [NCP(t[j])]

(2) ABSTRACTION FUNCTION

af:<xname> $\times$ vector $\times$ heap $\rightarrow$ <index>

af(x, t, h) $\overset{df}{=}$

Condition	Canonical Trace
(t[x] = EMPTY)	_
(t[x] $\neq$ EMPTY)	INIT(*, *).[PUT_INT(*, *, j, els(t,h,x)[j−1]$_h$)]$_{j=1}^{len(t,h,x)}$

(3) PROGRAM FUNCTIONS

Pf Name	Arg#1	Arg#2	Arg#3	Arg#4	Signature of Value
sppf_CATENATE	<xname>	<xname>			$\langle xname\rangle_2 \times \langle xname\rangle_3 \rightarrow$ $([\langle stname\rangle_1] \times$ $(\langle xname\rangle,\langle index\rangle)_2 \times$ $(\langle xname\rangle,\langle index\rangle)_3 \rightarrow$ $(ds \rightarrow ds \times \langle status\rangle_1))$
ppf_COPY	[<stname>]	<index>	(<xname>, <index>)		$ds \rightarrow ds \times \langle status\rangle_1$
ppf_DELETE	[<stname>]	(<xname>,<index>)	<integer>		$ds \rightarrow ds \times \langle status\rangle_1$
ppf_DESTROY	[<stname>]	(<xname>,<index>)			$ds \rightarrow ds \times \langle status\rangle_1$
ppf_GET_INT	[<stname>]	<index>	<integer>		$ds \rightarrow ds \times \langle status\rangle_1 \times \langle integer\rangle_R$
ppf_INIT	[<stname>]	(<xname>,<index>)			$ds \rightarrow ds \times \langle status\rangle_1$
ppf_IS_EQUAL	[<stname>]	<index>	<index>		$ds \rightarrow ds \times \langle status\rangle_1 \times \langle boolean\rangle_R$
ppf_NB_ELEMS	[<stname>]	<index>			$ds \rightarrow ds \times \langle status\rangle_1 \times \langle integer\rangle_R$
ppf_PUT_INT	[<stname>]	(<xname>,<index>)	<integer>	<integer>	$ds \rightarrow ds \times \langle status\rangle_1$

sppf_CATENATE(x, y) $\overset{df}{=}$ [1]

x = y	x ≠ y
ppf1_CATENATE	ppf2_CATENATE

ppf1_CATENATE(s, x, y) $\overset{df}{=}$ *NC*(t) ∧

		(‘t[x] ≠ EMPTY) ∧ (2 * *len*(‘t,‘h,x)	
	(‘t[x] = EMPTY)	> XMAX)	≤ XMAX)
h’ |	NCP_all	NCP_all	NCP_but_one(x) ∧ E1
s =	init	space	legal

with E1 standing for (*len*(t’,h’,x) = 2 * *len*(‘t,‘h,x))
∧ *areEqEls*(t’, h’, x, 0, ‘t, ‘h, x, 0, *len*(‘t,‘h,x))
∧ *areEqEls*(t’, h’, x, *len*(‘t,‘h,x), ‘t, ‘h, x, 0, *len*(‘t,‘h,x))

ppf2_CATENATE(s, x, y) $\overset{df}{=}$ *NC*(t) ∧

		((‘t[x] ≠ EMPTY) ∧ (‘t[y] ≠ EMPTY)) ∧ (*len*(‘t,‘h,x) + *len*(‘t,‘h,y)	
	(‘t[x] = EMPTY) ∨ (‘t[y] = EMPTY)	> XMAX)	≤ XMAX)
h’ |	NCP_all	NCP_all	NCP_but_one(x) ∧ E2
s =	init	space	legal

1 In this particular situation, the table defining sppf_CATENATE could be replaced by the simple expression “ppf2_CATENATE”. We introduce two tables to illustrate how a more complicated schema of program functions could be documented.

with E2 standing for (*len*(t’, x) = *len*(‘t, x) + *len*(‘t, y))
∧ *areEqEls*(t’, h’, x, 0, ‘t, ‘h, x, 0, *len*(‘t, ‘h, x))
∧ *areEqEls*(t’, h’, x, *len*(‘t, ‘h, x), ‘t, ‘h, y, 0, *len*(‘t, ‘h, y))

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

	(‘t[x] = EMPTY) ∨ (‘t[y] = EMPTY)	(‘t[x] ≠ EMPTY) ∧ (‘t[y] ≠ EMPTY)
t’ |	*NC*(t)	∀j ((0 ≤ j < SIZE) ∧ (j ≠ x)) [*NC*(t[j])] ∧ (t’[x] ≠ EMPTY)
h’ |	NCP_all	NCP_but_one(x) ∧ *isEqual*(t’, h’, x, y)
s =	init	legal

ppf_DELETE(s, x, n) $\overset{\text{df}}{=}$ *NC*(t) ∧

	‘t[x] = EMPTY	(‘t[x] ≠ EMPTY) ∧	
		¬*canGet*(‘t, ‘h, x, n)	*canGet*(‘t, ‘h, x, n)
h’ |	NCP_all	NCP_all	NCP_but_one(x) ∧ (*len*(t’, h’, x) = (*len*(‘t, ‘h, x) –1)) ∧ *areEqEls*(t’,h’,x, 0, ‘t,‘h,x, 0, n–1) ∧ *areEqEls*(t’, h’, x, n, ‘t, ‘h, x, n+1, *len*(‘t,‘h,x)–n)
s =	init	access	legal

ppf_DESTROY(s, x) $\overset{\text{df}}{=}$

	‘t[x] = EMPTY	‘t[x] ≠ EMPTY
t’ |	*NC*(t)	∀j ((0 ≤ j < SIZE) ∧ (j ≠ x)) [*NC*(t[j])] ∧ (t’[x] = EMPTY)
h’ |	NCP_all	NCP_but_one(x)
s =	init	legal

ppf_GET_INT(s, x, n) $\overset{\text{df}}{=}$ NCP_all ∧

	‘t[x] = EMPTY	(‘t[x] ≠ EMPTY) ∧	
		¬*canGet*(‘t, ‘h, x, n)	*canGet*(‘t, ‘h, x, n)
s =	init	access	legal
=			*els*(‘t,‘h,x)[n].‘h

ppf_INIT(s, x) $\overset{\text{df}}{=}$

	‘t[x] ≠ EMPTY	‘t[x] = EMPTY
t’ |	*NC*(t)	∀j ((0 ≤ j < SIZE) ∧ (j ≠ x)) [*NC*(t[j])] ∧ (t’[x] ≠ EMPTY)
h’ |	NCP_all	NCP_but_one(x) ∧ (*len*(t’, h’, x) = 0)
s =	nonempty	legal

ppf_IS_EQUAL(s, x, y) $\overset{df}{=}$ NCP_all ∧

	true
s =	legal
=	*isEqual*(‘t, ‘h, x, y)

ppf_NB_ELEMS(s, x) $\overset{df}{=}$ NCP_all ∧

	‘t[x] = EMPTY	‘t[x] ≠ EMPTY
s =	init	legal
=	▨	*len*(‘t,‘h,x)

ppf_PUT_INT(s, x, n, k) $\overset{df}{=}$ *NC*(t) ∧

	‘t[x] = EMPTY	(‘t[x] ≠ EMPTY) ∧		
		¬*canPut*(‘t,‘h,x,n)	*canPut*(‘t,‘h,x,n) ∧	
			(n > XMAX)	(n ≤ XMAX)
h’ |	NCP_all	NCP_all	NCP_all	NCP_but_one(x) ∧ [n < (*len*(‘t, ‘h, x) + 1) : *len*(t’,h’,x) = *len*(‘t,‘h,x) ; n = (*len*(‘t,‘h,x) + 1) : *len*(t’,h’,x) = *len*(‘t,‘h,x) + 1] ∧ $(els(t’,h’,x)[n-1]_{h’} = k)$ ∧ ∀j (1 ≤ j ≤ *len*(t’,h’,x)) $[(j \neq n) \Rightarrow (els(‘t,‘h,x)[j-1]_{‘h} = els(t’,h’,x)[j-1]_{h’})]$
s =	init	access	space	legal

4.2.3 C code of the index module

/* **index.h** file */

```
#include “global.h”
#include “status.h”              /* includes st_reg.h where st_reg is specified by gen_register(st_reg, <integer>, 1, 1000) */
#define xname int
#define XMAX 50
extern void     index_module_init(void);
extern void     index_INIT          (stname, xname);
extern void     index_DESTROY   (stname, xname);
extern void     index_PUT_INT     (stname, xname, int, int);
extern int      index_GET_INT     (stname, xname, int);
extern bool     index_IS_EQUAL  (stname, xname, xname);
extern void     index_COPY         (stname, xname, xname);
extern int      index_NB_ELEMS(stname, xname);
extern void     index_DELETE      (stname, xname, int);
extern void     index_CATENATE (stname, xname, xname);
```

/* **index.c** file */

```
#include “index.h”
#include “st_reg.h”
```

```
#define SIZE 100

typedef int* indexArr;
typedef struct {
    int len;
    indexArr els;
} indexStr;
typedef indexStr* indexObj;

typedef indexObj ds[SIZE];
static ds t;

#define EMPTY (indexObj)0
#define canGet(x,i) (1<=(i) && (i) <= (x)->len)
#define canPut(x,i) (1<=(i) && (i) <= (x)->len+1)

/******** local functions ********/

static void set_value(int n, indexObj v)
{  t[n] = v; }

static indexObj get_value(int n)
{  return t[n]; }

static indexStr createStr(int const n)
{  indexStr str;

    str.len = n;
    if (n>0) str.els = (indexArr)malloc(n*sizeof(int));
    return str;
}

static void updateStr(indexObj const tx, indexStr const str)
{  if (tx->len > 0) free(tx->els);
    *tx = str;
}

static void moveEls(indexArr const x, indexArr const y, int const n)
{  /* may call only with valid pointer values */
    if (n>0) memmove(x,y,n*sizeof(int));
}

/******** module initialization ********/
void index_module_init(void)
{  int i;
    for (i=0;i<SIZE;i++)
        set_value(i,EMPTY);
}
```

```
/******** access-programs ********/

void index_INIT(stname s, xname x)
{  indexObj tx;

   status_LEGAL(s);
   tx = get_value(x);
   if (tx != EMPTY)
      status_NONEMPTY_TRACE(s);
   else {
      tx = (indexObj)malloc(sizeof(indexStr));
      tx->len = 0;
      set_value(x, tx);
   }
}

void index_DESTROY(stname s, xname x)
{  indexObj tx;

   status_LEGAL(s);
   tx = get_value(x);
   if (tx == EMPTY)
      status_NO_INIT(s);
   else {
      if (tx->len > 0) free(tx->els);
      free(tx);
      set_value(x, EMPTY);
   }
}

void index_PUT_INT(stname s, xname x, int n, int i)
{  indexObj tx;

   status_LEGAL(s);
   tx = get_value(x);
   if (tx == EMPTY) status_NO_INIT(s);
   else if (!canPut(tx,n)) status_BAD_ACCESS(s);
   else if (n > XMAX) status_OUT_OF_SPACE(s);
   else if (n-1 < tx->len) tx->els[n-1] = i;
   else {
      indexStr const str = createStr(n);
      moveEls(str.els, tx->els, n-1);
      str.els[n-1] = i;
      updateStr(tx,str);
   }
}

int index_GET_INT(stname s, xname x, int n)
{  indexObj tx;
```

```
    status_LEGAL(s);
    tx = get_value(x);
    if (tx == EMPTY) {
        status_NO_INIT(s);
        return UNDEFINED;
    }
    else if (!canGet(tx,n)) {
        status_BAD_ACCESS(s);
        return UNDEFINED;
    }
    else return tx->els[n-1];
}
```

```
bool **index_IS_EQUAL**(stname s, xname x, xname y)
{  int i;
    indexObj tx, ty;

    status_LEGAL(s);
    tx = get_value(x);
    ty = get_value(y);
    if (tx == ty) return TRUE;
    else if (tx == EMPTY || ty == EMPTY) return FALSE;
    else if (tx->len != ty->len) return FALSE;
    else
        for (i=0; i < tx->len; i++)
            if (tx->els[i] != ty->els[i]) return FALSE;
    return TRUE;
}

void **index_COPY**(stname s, xname y, xname x)
{  indexObj ty, tx;

    status_LEGAL(s);
    ty = get_value(y);
    tx = get_value(x);
    if (ty == EMPTY || tx == EMPTY)
        status_NO_INIT(s);
    else {
        indexStr const str = createStr(ty->len);
        moveEls(str.els, ty->els, ty->len);
        updateStr(tx, str);
    }
}

int **index_NB_ELEMS**(stname s, xname x)
{  indexObj tx;

    status_LEGAL(s);
    tx = get_value(x);
    if (tx == EMPTY) {
        status_NO_INIT(s);
        return UNDEFINED;
    }
    else return tx->len;
}

void index_DELETE(stname s, xname x, int n)
{  indexObj tx;
    status_LEGAL(s);
    tx = get_value(x);
    if (tx == EMPTY) status_NO_INIT(s);
    else if (!canGet(tx,n)) status_BAD_ACCESS(s);
    else {
        indexStr const str = createStr(tx->len-1);
        moveEls(str.els, tx->els, n-1);
        moveEls(&str.els[n-1], &tx->els[n], tx->len-n);
        updateStr(tx, str);
    }
}

void index_CATENATE(stname s, xname x, xname y)
{  indexObj tx, ty;
    status_LEGAL(s);
    tx = get_value(x);
    ty = get_value(y);
    if (tx == EMPTY || ty == EMPTY) status_NO_INIT(s);
    else if (tx->len + ty->len > XMAX) status_OUT_OF_SPACE(s);
    else {
        indexStr const str = createStr(tx->len+ty->len);
        moveEls(str.els, tx->els, tx->len);
        moveEls(&str.els[tx->len], ty->els, ty->len);
        updateStr(tx, str);
    }
}
```

4.3 The gen_register(tname, smallest: tname, biggest: tname) module

4.3.1 gen_register(tname, smallest: tname, biggest: tname) Module Interface Specification

Informal description

This parameterized module provides mechanisms for controling the usage of names in a multi-object module. It has three parameters denoting the subrange smallest..biggest of the type tname whose values will be used as names. The type tname has to be well-ordered.[1]

PICK returns a value (a name) which was not yet “used”; this value becomes a “used” one. If there are no more “unused” names, then the value equivalent to the empty trace in the type tname is returned.[1]

DISPOSE allows a “used” name to become an “unused” one.

Example of application (written in C):

```
    /* gen_register(x_reg, <integer>, 1, 100)       the module x_reg is used to manage names in the index module */
    /* gen_register(st_reg, <integer>, 1, 1000)     the module st_reg is used to manage names in the status module */
    int X1, X2, S;
    X1 = x_reg_PICK;
    S = st_reg_PICK;
    …
    X2 = x_reg_PICK;
    if (X2 == 0) …                                  /* no more unused names; 0 represents the empty trace */
```

(0) CHARACTERISTICS

• type specified: <register_of_names>

• features: parameterized, single-object, non-deterministic

• specification parameters: tname, smallest: <integer>, biggest: <integer>

• foreign types: <integer>, <status>, tname

(1) SYNTAX

ACCESS-PROGRAMS

Program Name	Arg#1	Result Type
PICK		tname:R
DISPOSE	tname:V	

(2) CANONICAL TRACES

$$\begin{aligned} \text{canonical}(T) \Leftrightarrow\ & (T = [\text{PICK}(*)\ \ x_i]_{i=1}^{n}\ ) \\ & \wedge \forall T1,T2,T3,x,y\ [(T = T1.\text{PICK}(*)\ \ x.T2.\text{PICK}(*)\ \ y.T3) \Rightarrow (x < y)]^{2} \\ & \wedge \forall i\ (1 \le i \le n)\ [x_i \neq \_\ ] \\ & \wedge (0 \le n \le (\text{biggest}-\text{smallest}+1)) \end{aligned}$$

(3) EQUIVALENCES

Legality(PICK(T)) = %legal%

1 We assume the existence of a well-ordering operator (access-program) “<” in the module tname.

1 Because instances of the gen_register specification can be treated as system modules, we have adopted such a solution to avoid problems with “illegal” invocations.

2 We assume here that the operator “<” is defined on the type tname.

PICK(T) a ≡>

Condition	Value
$a \neq \_$	T1.PICK(*) a.T2 where $(T = T1.T2) \wedge$ canonical(T1.PICK(*) a.T2)
$a = \_$	T

Legality(DISPOSE(T, a)) = %legal%

DISPOSE(T , a) ≡>

Condition	Value
$\exists$T1,T2 [T = T1.PICK(*) a.T2]	T1.T2
$\neg\exists$T1,T2 [T = T1.PICK(*) a.T2]	T

(4) RETURN VALUES

PICK(T) a |

Condition		Value
count(register_of_names, PICK) < (biggest−smallest+1) =	false	$a = \_$
	true	$(a \neq _) \wedge (\neg\exists$T1,T2)[T = T1.PICK(*) a.T2]

4.3.2 gen_register(<integer>, smallest, biggest) Module Internal Design Document

Informal description

This document describes a parameterized module design written on the base of parameterized single-object module interface specification: gen_register(tname, smallest, biggest), where

- the parameter tname (which denotes the type of values used as names) has been replaced by the type <integer>,
- the specification parameters smallest and biggest of <integer> type (which determine the range of names that could be used) are left as parameters whose values we will define later.

This document should be treated as a parameterized internal design document. To obtain a complete module design we have to replace smallest and biggest by integer values.[1]

In the document we mix the usage of abstract values of type <integer> with values of type int defined in the C language (cf. section 3.2.4.5). As the module interface specification “distinguishes” the empty trace when the set of canonical traces is defined (it requires that used names be different from the empty trace), we define “0” to be the denotation of the empty trace in the integer module.

1 It would be difficult to have a fully parameterized design document using available features of the implementation language C. Our design depends on the properties of the first parameter (it is used to subscript a table).

(1) DATA STRUCTURE

CONSTANT DEFINITIONS

Constant Name	Definition
FALSE	0
TRUE	1
size	biggest–smallest+1

TYPE DEFINITIONS

Type Name	Definition
bool	typedef int;

item	typedef struct {
	  bool used;
	  int next;
	} item;

items	typedef item items[size];

VARIABLE DEFINITIONS AND INITIALIZATIONS

Type Name/Definition	Variable Name	Initial Condition
items	table	$\forall$ i $(0 \le i \le (size-1))$ [table[i].used = FALSE] $\wedge$ $\forall$ i $(0 \le i \le (size-2))$ [(table[i].next = (i+1)) $\wedge$ (table[size–1].next = –1)]
int	fl /* freelist */	fl = 0

DATA STATE CONSTRAINTS

wfds(table, fl)

AUXILIARY FUNCTIONS

nb_names: items $\times$ int $\times$ int $\rightarrow$ int

nb_names_domain(t, i, max) $\Leftrightarrow$ $(0 \le i \le max) \wedge (max < size)$

nb_names(t, i, max) $\overset{df}{=}$

Condition	Value
(i < max) $\wedge$ (t[i].used = TRUE)	1 + *nb_names*(t, i+1, max)
(i < max) $\wedge$ (t[i].used = FALSE)	*nb_names*(t, i+1, max)
(i = max) $\wedge$ (t[i].used = TRUE)	1
(i = max) $\wedge$ (t[i].used = FALSE)	0

fl_length: items × P(int) × int → int

fl_length(t, set, i) $\overset{df}{=}$

Condition	Value
(i = –1) ∧ (set ≠ {})	–(size+1)
(i = –1) ∧ (set = {})	0
(i ≠ –1) ∧ (i ∉ set)	–(size+1)
(i ≠ –1) ∧ (i ∈ set)	1 + *fl_length*(t, set–{i}, t[i].next)

wfds: items × int → <boolean>

wfds(t, fl) $\overset{df}{=}$ (–1 ≤ fl ≤ (size–1)) ∧ ((fl ≥ 0) ⇒ (*fl_length*(t, {x | ¬t[x].used}, fl) ≥ 0))

LEXICON

ds $\overset{df}{=}$ items × int

(2) ABSTRACTION FUNCTION

af: ds → gen_register(<integer>, smallest, biggest)

af((table, fl)) $\overset{df}{=}$

Condition	Canonical Trace
true	$[\text{PICK}(*)\ \ x_i]_{i=1}^{n}$, where (n = *nb_names*(table, 0, size–1)) ∧ ∀i (1 ≤ i ≤ n) [((i > 1) ⇒ ($x_{i-1} < x_i$)) ∧ (table[x_i–smallest].used = TRUE)]

(3) PROGRAM FUNCTIONS

Pf Name	Arg#1	Signature of Value
pf_PICK		ds → ds × <integer>_R
ppf_DISPOSE	<integer>_1	ds → ds

pf_PICK[1] $\overset{df}{=}$

		‘fl < 0	‘fl ≥ 0
table’	|	*NC*(table)	(table’[‘fl].used = TRUE) ∧ ∀ j (0 ≤ j ≤ (size–1)) [(j ≠ ‘fl) ⇒ *NC*(table[j])]
fl’	=	‘fl	‘table[‘fl].next
	=	0	‘fl + smallest

1 As the access-program PICK has no arguments, we describe PICK in the form of program function.

ppf_DISPOSE(n) $\overset{df}{=}$

	‘table[n–smallest].used = FALSE	‘table[n–smallest].used = TRUE
table’ |	NC(table)	(table’[n–smallest].used = FALSE) $\wedge$ (table’[n–smallest].next = ‘fl) $\wedge$ $\forall$ j (0 $\le$ j $\le$ (size–1)) [(j $\ne$ (n–smallest)) $\Rightarrow$ NC(table[j])]
fl’ =	‘fl	n – smallest

4.3.3 C code of the gen_register module

/* **x_reg.h** file */

```
/* x_reg = gen_register(int, smallest, biggest) */
#define xname int
#define smallest 1
#define biggest 100

extern void      x_reg_module_init(void);
extern xname     x_reg_PICK(void);
extern void      x_reg_DISPOSE(xname);
```

/* **x_reg.c** file */

```
/* x_reg = gen_register(int, smallest, biggest) */
#include “global.h”
#include “x_reg.h”
#define SIZE (biggest-smallest+1)

typedef struct {
   bool used;
   int next;
} item;
typedef item items[SIZE];

static items table;
static int fl;

/******** module initialization ********/
void x_reg_module_init(void)
{  int i;

   for (i=0;i<=SIZE-1;i++)
      table[i].used = FALSE;
   for (i=0;i<=SIZE-2;i++)
      table[i].next = i+1;
   table[SIZE-1].next = -1;
   fl = 0;
}
```

```
/******** access-programs ********/
int x_reg_PICK(void)
{  int old_fl;

   if (fl<0) return 0;
   else {
      old_fl = fl;
      table[old_fl].used = TRUE;
      fl = table[old_fl].next;
      return old_fl+smallest;
   }
}

void x_reg_DISPOSE(int n)
{
   if (table[n-smallest].used) {
      table[n-smallest].used = FALSE;
      table[n-smallest].next = fl;
      fl = n-smallest; }
}
```

4.4 The status module

4.4.1 status Module Interface Specification

Informal description

The status module is a multi-object module. Its values correspond to status tokens used in all module interface specifications in TTP and are used to denote the status of objects of other modules (within TTP). Each access-program, but IS_EQUAL, is used to deliver a value, on the request of other modules.

Example of application (written in C):

```
/* gen_register(x_reg, <integer>, 1, 100)     the module x_reg is used to manage names in the index module */
/* gen_register(st_reg, <integer>, 1, 1000)   the module st_reg is used to manage names in the status module */
int X, S, Legal;
X = x_reg_PICK;
S = st_reg_PICK;
Legal = st_reg_PICK;
status_LEGAL(Legal);
index_INIT(S, X); … index_GET_INT(S, X, 1);
if (! status_IS_EQUAL(S1, Legal) …
```

(0) CHARACTERISTICS

• type specified: <status>

• features: multi-object, deterministic

• foreign types: <boolean>

• type of names: { i: <integer> | $1 \leq i \leq 1000$ }

(1) SYNTAX

ACCESS-PROGRAMS

Program Name	Arg#1	Arg#2	Result Type
BAD_ACCESS	<status>:O		
IS_EQUAL	<status>:V	<status>:V	<boolean>
LEGAL	<status>:O		
NONEMPTY_TRACE	<status>:O		
NO_INIT	<status>:O		
OUT_OF_SPACE	<status>:O		

(2) CANONICAL TRACES

canonical(T) ⇔ (T = _)

∨ (T = BAD_ACCESS(*))

∨ (T = NO_INIT(*))

∨ (T = NONEMPTY_TRACE(*))

∨ (T = OUT_OF_SPACE(*))

EQUIVALENT NOTATION FOR ABSTRACT VALUES

Canonical trace	Equivalent notation
_	legal
BAD_ACCESS(*)	access
NO_INIT(*)	init
NONEMPTY_TRACE(*)	nonempty
OUT_OF_SPACE(*)	space

(3) EQUIVALENCES

Legality(BAD_ACCESS(*)) = %legal%

BAD_ACCESS(*) ≡> access

Legality(IS_EQUAL(T1, T2)) = %legal%

Legality(LEGAL(*)) = %legal%

LEGAL(*) ≡> legal

Legality(NO_INIT(*)) = %legal%

NO_INIT(*) ≡> init

Legality(NONEMPTY_TRACE(*)) = %legal%

NONEMPTY_TRACE(*) ≡> nonempty

Legality(OUT_OF_SPACE(*)) = %legal%

OUT_OF_SPACE(*) ≡> space

(4) RETURN VALUES

IS_EQUAL(T1, T2)   = (T1 = T2)

4.4.2 status Module Internal Design Document

Informal description

This document describes a design of the multi-object status module.

(1) DATA STRUCTURE

CONSTANT DEFINITIONS

Constant Name	Definition
SIZE	1000

TYPE DEFINITIONS

Type Name	Definition
status	enum {Legal, Access, Init, Nonempty, Space }
ds	status ds[SIZE]

VARIABLE DEFINITIONS AND INITIALIZATIONS

Type Name/Definition	Variable Name	Initial Condition
ds	table	∀ i (0 ≤ i ≤ (SIZE–1)) [table[i] = Legal]

LEXICON

<stname> $\overset{df}{=}$ {i: <integer> \| 1 ≤ i ≤ SIZE }

(2) ABSTRACTION FUNCTION

af: <stname> × ds → <status>

af(i,t) $\overset{df}{=}$

Condition	Canonical Trace
t[i–1] = Access	access
t[i–1] = Legal	legal
t[i–1] = Init	init
t[i–1] = Nonempty	nonempty
t[i–1] = Space	space

(3) PROGRAM FUNCTIONS

Pf Name	Arg#1	Arg#2	Value
ppf_BAD_ACCESS	<stname>		ds → ds × <status>
ppf_IS_EQUAL	<stname>	<stname>	ds → ds × <boolean>
ppf_LEGAL	<stname>		ds → ds × <status>
ppf_NONEMPTY_TRACE	<stname>		ds → ds × <status>
ppf_NO_INIT	<stname>		ds → ds × <status>
ppf_OUT_OF_SPACE	<stname>		ds → ds × <status>

ppf_BAD_ACCESS(i) $\overset{df}{=}$ ∀j:int (0 ≤ j ≤ (SIZE–1)) [(j ≠ (i–1)) ⇒ *NC*(table[j])] ∧

	true
table'[i–1]=	Access
i =	access

ppf_IS_EQUAL(i, j) $\overset{df}{=}$ *NC*(table) ∧

	true
=	'table[i–1] = 'table[j–1]

ppf_LEGAL(i) $\overset{df}{=}$ ∀j:int (0 ≤ j ≤ (SIZE–1)) [(j ≠ (i–1)) ⇒ *NC*(table[j])] ∧

	true
table'[i–1]=	Legal
i =	legal

ppf_NONEMPTY_TRACE(i) $\overset{df}{=}$ ∀j:int (0 ≤ j ≤ (SIZE–1)) [(j ≠ (i–1)) ⇒ *NC*(table[j])] ∧

	true
table'[i–1]=	Nonempty
i =	nonempty

ppf_NO_INIT(i) $\overset{df}{=}$ $\forall$j:int (0 $\le$ j $\le$ (SIZE−1)) [(j $\ne$ (i−1)) $\Rightarrow$ *NC*(table[j])] $\wedge$

	true
table'[i−1] =	Init
i =	init

ppf_OUT_OF_SPACE(i) $\overset{df}{=}$ $\forall$j:int (0 $\le$ j $\le$ (SIZE−1)) [(j $\ne$ (i−1)) $\Rightarrow$ *NC*(table[j])] $\wedge$

	true
table'[i−1] =	Space
i =	space

4.4.3 C code of the status module

```
/* status.h file */
#include "global.h"
#define stname int
extern void     status_module_init();
extern void     status_BAD_ACCESS(stname);
extern void     status_LEGAL(stname);
extern void     status_NONEMPTY_TRACE(stname);
extern void     status_OUT_OF_SPACE(stname);
extern void     status_NO_INIT(stname);
extern bool     status_IS_EQUAL(stname,stname);

/* status.c file */
#include "status.h"
#define SIZE 1000
typedef enum {
   Legal,
   Init,
   Access,
   Nonempty,
   Space
} status;
typedef status ds[SIZE];
static ds t;
/******** module initialization ********/
void status_module_init()
{  int i;

   for (i=0;i<SIZE;i++)
      t[i] = Legal;
}
void status_BAD_ACCESS(stname x)
```

```
{  t[x-1] = Access; }

 void status_LEGAL(stname x)
{  t[x-1] = Legal; }

 void status_NONEMPTY_TRACE(stname x)
{  t[x-1] = Nonempty; }

 void status_OUT_OF_SPACE(stname x)
{  t[x-1] = Space; }

 void status_NO_INIT(stname x)
{  t[x-1] = Init; }

bool status_IS_EQUAL(stname x, stname y)
{  return t[x-1] == t[y-1]; }
```

5 Conclusions

As it was mentioned in the introduction, we began this work expecting difficulties. There was no experience in the application of all our methods in a single software project - previous papers either have had a rather narrow focus (e.g. [10]), or have been quite abstract (e.g. [11]). We knew that some aspects of the methods need to be revised and modified. We felt the lack of software tools supporting preparation of documentation, in general, and the tabular notation, in particular. This resulted in a long and painful, iterative process. In fact we are still not happy with the present state of affairs. It is obvious that a new description of the trace assertion method is badly needed. The most recent report on it [9] is practically obsolete and we had to record many changes and extensions to it in the present paper. On the other hand it was necessary to gain the needed experience before updating the report on the method. This paper is not uniform in depth and combines a non-trivial example of applications of ideas presented in [11], with an update of the formal documentation methods and the notational conventions. Moreover, the update is also not complete - we have not provided a complete description, merely a record of the most important modifications and proposals with respect to [9].

During this work we had to address certain aspects of our methods which were earlier either not needed or not recognized and hence not treated in previous papers at all. For example, the notion of parameterized specifications (both, on the module interface level and on the internal design level) was “in our heads” but is introduced for the very first time here. Also a concept of multi-object modules, though already present in [9], had to be significantly clarified. Another important issue was related to the error treatment. In the trace method, status tokens are used to provide information about the legality of a given operation and to classify possible incorrect invocations of access-programs. We need an equivalent tool at the lower level of specification (internal design). Our current solution of that problem (“status” module) is far from ideal and had been changed many times.

There are also certain aspects of the methods that have not been discussed in this paper at all and require further studies. They include the usage of input variables, non-determinism in case of output domestic arguments, and different formats of parameterization of module interface specifications.

Acknowledgements

We are grateful to J. Bojanowski, B. Desrosiers, M. Kubica, A. Malton, A.P. Ravn, K. Stencel, A.J. van Schouwen, and Yabo Wang for helpful comments on earlier versions of this paper.

This work was supported by the Province of Ontario through TRIO, the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Canadian Institute of Telecommunications Research (CITR).

References

1. Iglewski, M., Madey, J., Parnas, D.L., “Using Formal Documentation for Black-Box Testing”, *Technical Report,* McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada; *in preparation*
2. Kernighan, B.W., Ritchie, D.M., “The C Programming Language”, Prentice-Hall, 1978.
3. Mills, H.D., “The New Math of Computer Programming”, *Communications of the ACM*, Vol. 18, No. 1, January 1975, pp. 43-48.
4. Parnas, D.L., “On the Criteria to be Used in Decomposing Systems into Modules”, *Communications of the ACM*, Vol. 15, No. 12, December 1972, pp. 1053-1058.
5. Parnas, D.L., “On a ’Buzzword’: Hierarchical Structure”, *Proceedings of the IFIP Congress ’74*, North Holland, 1974, pp. 336-339.
6. Parnas, D.L., “Designing Software for Ease of Extension and Contraction”, *Proceedings of the Third International Conference on Software Engineering*, May 1978, pp. 264-277. Also in: *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 2, March 1979, pp. 128-138.
7. Parnas, D.L., “A Generalized Control Structure and Its Formal Definition”, *Communications of the ACM*, Vol. 26, No. 8, August 1983, pp. 572-581.
8. Parnas, D.L., Wadge, W.W., “Less Restrictive Constructs for Structured Programs”, *Technical Report 86-186*, Queen’s University, C&IS, Kingston, Ontario, Canada, October 1986, 16 pp.
9. Parnas, D.L., Wang, Y., “The Trace Assertion Method of Module Interface Specification”, *Technical Report 89-261*, Queen’s University, C&IS, Telecommunications Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, October 1989, 39 pp. (*Revised version in preparation)*
10. Parnas, D.L., Asmis, G.J.K., Madey J., “Assessment of Safety-Critical Software in Nuclear Power Plants”, *Nuclear Safety*, Vol. 32, No. 2, 1991, pp. 189-198.
11. Parnas, D.L., Madey, J., “Functional Documentation for Computer Systems Engineering. (Version 2)”, *CRL Repor*t *237*, McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada; September 1991, 14 pp.
12. Parnas, D.L., Madey, J., “Documentation of Real-Time Requirements”, in: *Real-Time Systems. Abstraction, Languages and Design Methodologies*, Krishna M. Kavi (ed.), IEEE Computer Society Press, 1992, pp.48-56.
13. Parnas, D.L., Madey, J., Iglewski, M., “Formal Documentation of Well-Structured Programs”, *CRL Repor*t *259*, McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada; September 1992, 37 pp.
14. Parnas, D.L., “Tabular Representation of Relations”, *CRL Report 260*, McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada; October 1992, 17 pp.
15. Parnas, D.L. et al., “Table-Tool Project Design Issues”, *Technical Report,* McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada; *in preparation*
16. van Schouwen, A.J., Parnas, D.L., Madey, J., “Documentation of Requirements for Computer Systems”, Presented at RE ‘93 IEEE International Symposium on Requirements Engineering, San Diego, CA, 4-6 January, 1993.

Appendix A

Notation Guide

In this appendix we have put together basic notational conventions used in module interface specifications and internal design documents. This includes both the standard mathematical notions, and the method dependent ones employed in these two documents (though not all of them are used in the present paper).

A.1 Symbols

A.1.1 Logic

The logic used is a many-sorted predicate logic, i.e. it has:

- Variables $x_1,\ldots$, each having an associated type (or sort) $T_1,\ldots$.
- Constant symbols $c_1,\ldots$, each having an associated type.
- Function symbols $f_1,\ldots$, each with an associated arity *n* and a signature giving a type for each of the *n* arguments and a result type.

Variable, constant and function symbols are used to form syntactical and type correct terms $t_1, \ldots$.

Predicate symbols $p_1,\ldots$ each with an associated arity *n* and a signature giving a type for each of the *n* arguments. Predicate symbols are used to form syntactical and type correct atomic formulas. A formula P, Q, … is either an atomic formula or one of the formulas described in the following table:

formal symbolism	meaning
true, false	logical constants
$\neg P$	negation
$P \wedge Q$	conjunction
$P \vee Q$	disjunction
$P \Rightarrow Q$	implication
$P \Leftrightarrow Q$	equivalence
$\forall x_1{:}T_1,\ldots,x_n{:}T_n\ [P]$	universal quantification: “for all x_1 from T_1, …, and x_n from T_n, P holds”
$\exists x_1{:}T_1,\ldots,x_n{:}T_n\ [P]$	existential quantification: “there exist x_1 in T_1, …, and x_n in T_n such that P holds”
$\forall x_1{:}T_1,\ldots,x_n{:}T_n\ (P)\ [Q]$	$\forall x_1{:}T_1,\ldots,x_n{:}T_n\ [P \Rightarrow Q]$
$\exists x_1{:}T_1,\ldots,x_n{:}T_n\ (P)\ [Q]$	$\exists x_1{:}T_1,\ldots,x_n{:}T_n\ [P \wedge Q]$
$\exists! x{:}T[P]$	unique existence: $\exists x{:}T\ [P(x) \wedge \forall y{:}T\ (P(y))\ [\ x = y]]$
$x = t(x_1,\ldots,x_m)$ where $P(x_1,\ldots,x_m)$	unique value for x determined by t over variables $x_1,\ldots,x_m$, satisfying P: $\exists! y\ [(x = y) \wedge \forall x_1,\ldots,x_m\ [P(x_1,\ldots,x_m) \Rightarrow y = t(x_1,\ldots,x_m)]]$
$t_1 = t_2$	term equality (if t1 and t2 are defined)
$t_1 \neq t_2$	$\neg(t_1 = t_2)$ (if t1 and t2 are defined)

$T_1,\ldots,T_n$ can be omitted, if known by the context, e.g. $\forall i\ (i > 1)[P(i)]$.

A.1.2 Sets

Let S, T, S_1, …, S_n be sets; e_1, …, e_n terms; and P be a predicate.

formal symbolism	meaning
$x \in S$	x is a member of S
$x \notin S$	$\neg(x \in S)$
$S \subseteq T$	set inclusion
$S \subset T$	strict set inclusion
$S \cap T$	set intersection
$S \cup T$	set union
$S - T$	set difference
$\| S \|$	cardinality of S
$\{x:T \mid P\}$	the set containing exactly those x from T for which P holds (T may be omitted, if known by the context)
$P(S)$	power set of S (the set containing all subsets of S)
$(e_1, e_2, \ldots, e_n)$	n-tuple
$S_1 \times S_2 \times \ldots \times S_n$	Cartesian product: the set of all n-tuples such that the k-th component is an element of S_k

A.1.3 Traces

Let x,y be names of objects (cf. section 3.1.5); P an access-program; T1, T2 traces; and S be a portion of a trace [1].

formal symbolism	meaning
_ (underscore)	the empty trace
.	separator between elements in written representation of traces
$[S]_{i=m}^{n}$	portion Z of a trace, obtained as follows: Let S_α denote the string S in which all occurrences of i have been replaced by α; Z = _ if $m > n$, or $Z = S_m.S_{m+1}. \ldots .S_n$ if $m \leq n$
$\equiv>$	symbol used as the infix denotation of the extension function ef (cf. section 3.1.9)
$(x,T1) \equiv (y,T2)$	trace equivalence (defined by the extension function)
$T1 \equiv T2$	$\forall x,y\ [(x,T1) \equiv (y,T2)]$
*	wild-card symbol used in canonical traces (cf. section 3.1.7)

1 If a trace is understood as a string of symbols, then by a portion of a trace we mean any substring of this string.

A.1.4 Other symbols

formal symbolism	meaning
lhs $\overset{df}{=}$ rhs	lhs is defined to be rhs
f: U $\rightarrow$ R	signature of a function f with the domain drawn from the Universe U and the range R (cf. section 3.2.4.1)
<m>	type (set of canonical traces) defined by the module m, e.g. <index>; parameterized types are written without angle brackets
m :: p	access-program p from the module m
m :: *f*	auxiliary function *f* (written in italics) from the module m
P(…)	a value returned by the invocation P(…); P must be a deterministic access-program
P(…, a , …)	a value returned by P in the output argument a; P must be a deterministic access-program
P(…) v	v is a value returned by the invocation P(…) (cf. section 3.1.8); program P may be non-deterministic
P(…, a b, …)	b is a value returned by P in the output argument a (cf. section 3.1.8); program P may be non-deterministic
	used in program functions to indicate that the value being defined is the value returned by the program
v	used in program functions to indicate that the value v being defined is the value returned by the program
a	used in program functions to indicate that the value being defined is the output value of deterministic argument a
a v	used in program functions to indicate that the value v being defined is the output value of argument a
m	if m is the name of a single-object module then m (passed as an implicit argument) may be also used in the return value table to denote the module value
%s%	status token s, e.g. %legal%
/* … */	comment

A.2 Functions

A.2.1 Functions defined for each module

function	meaning
canonical(T)	true, iff T is a canonical trace
ef((x,T), e)	extension function; for a given name x, and a canonical trace T extended by an event e, it returns the equivalent canonical trace and a status token (cf. section 3.1.9)
af(x, ds)	abstraction function; returns the canonical trace corresponding to the name x and the data state ds (cf. section 3.2.2)
ppf_P(…)	parameterized program function related to the access-program P (cf. section 3.2.3)
sppf_P(…)	schema of parameterized program functions related to the access-program P (cf. section 3.2.3)

A.2.2 Standard functions

function	meaning
length(T)	number of events in trace T
count(T, P)	number of times an invocation of P occurs as an event in trace T
NC(v_1,…,v_n)	“not changed” predicate, cf. section 3.2.4.4
NCP(v_1,…,v_n)	“not changed pointer” predicate, cf. section 3.2.4.4

A.3 Access-program argument descriptors

Each access-program argument is characterized in the access-program table of a module interface specification by one or more descriptors. The meaning of these descriptors is explained in the table below. The symbols n and v used in the table denote, respectively, a name (cf. section 3.1.5) and a value associated with this name. The further interpretation of the descriptors is given in section 3.1.6.

descriptor	actual arg.	informal meaning
O	n	output argument; the access-program calculates a value which is returned via n
V	v	input argument; one of the access-program outputs is a function of v
VO	(n, v)	input-output argument; the access-program calculates a value which is returned via n; one of the access-program outputs is a function of v
N	n	input argument; one of the access-program outputs is a function of n
NO	n	input-output argument; the access-program calculates a value which is returned via n; one of the access-program outputs is a function of n
NV	(n, v)	input argument; one of the access-program outputs is a function of n, and one of the access-program outputs is a function of v
NVO	(n, v)	input-output argument; the access-program calculates a value which is returned via n; one of the access-program outputs is a function of n, and one of the access-program outputs is a function of v

The descriptor ‘R’ used after the Result Type in the Syntax Table means that the value returned by the corresponding access-program is described by a relation.