CHAPTER 1: INTRODUCTION

1.0 Thesis organisation

This chapter introduces the reader to the older trace assertion method (TAM) and a new TAM. In producing the new TAM (ITAM), this chapter gives a brief introduction to the improvements and the reasons for these improvements.

Chapters 2,3 gives the details of the improvements, and chapter 4 presents the resulting revised specification structure (i.e. *format*), incorporated with these improvements. Chapter 5 discusses the examples found in Appendices A and B.
Chapter 6 presents the conclusion and some suggestions for future work.

In Chapter 2 we present the semantic base for our alternative replacement to using canonical traces, a *canonical representation* that uses sets. It also presents relations used in composing a module specification.
Chapter 3 provides a syntax for describing the canonical representation, and an example using this syntax.

Appendix A presents revised versions of all examples found in [10]. Appendix B presents newer examples. Appendix C gives some notes in using the syntax checking tool when one needs to describe a canonical representation.

1.1 TAM and some basic definitions

The trace assertion method is among the many formal methods recommended for specifying software module requirements. For the reader to understand this method, we

have used terminologies that may be define differently outside of this thesis. We reduce any confusion of some terminologies, by explaining their meaning.

1.1.1 A view of software construction

This work is based on a very simple and pragmatic view of software development. *Software Development* is the design and production of programs that will be used, and often changed, by other people. Programs consist of algorithms and variables. A software developer begins with a previously constructed set of algorithms and the ability to declare new variables of previously defined types. Using these, the software developer constructs new algorithms and new types of variables.

Because software will be used and changed by others, documenting each new program and class of variables is an essential part of the software developer's task, one that is (unfortunately) often neglected or left to others.

In other words, the software developer has three types of products: algorithms; variable types; and documentation. In the remainder of this section, we discuss these in more detail.

1.1.2 What is an algorithm (program)

For our purposes, an *algorithm* is something that constrains the sequence of state changes of a digital computer (a finite state machine), [12]. We refer to a program as *deterministic* if the constraints fully determine the sequence of state changes and *non–deterministic* otherwise, [12].

1.1.3 Variables/Objects and Types/Classes

Although there is no mathematical basis for the distinction, we customarily view a computer program as consisting of two components, a fixed part, called the *control* portion, and a variable part, which we call the *data*. While the data can be viewed as a single, finite

state, machine, we customarily view it as composed of a finite set of smaller machines, called variables. Each variable is characterised by its set of states (values) and the set of possible state transitions (operations). In some programming languages, variables that have been constructed by writing programs, are called *objects*, [12]. All objects have some way to identify them. Two variables that are identical in every way except their identifiers, are said to be of the same *type*, [18]. Variables can be grouped into types or classes; it is most useful to do so on the basis of shared characteristics. Early programming languages provided users with variables of a small number of predetermined types. Later languages provide facilities that allowed users to extend the set of built–in types with additional, user–defined, types. However, even in the early programming languages it was possible for users to define new types of data – but these were then syntactically different from built–in types [9].

1.1.4 Modules and Access Programs

Software is usually constructed by teams consisting of several people. Each programmer writes one or more groups of programs, which we call *modules*, [12]. Even when there is only a single developer, the programs that he/she writes should be grouped into modules. Of the programs included in a module, some may be invoked by programs that are not part of the module. We call these externally accessible programs *access programs*. People developing other modules should be aware only of those access programs. It is considered ”good design” to make sure that those who use a module need not be aware of any aspects of it that are likely to change [9].

A module is referred to as deterministic if all of its programs are deterministic and non–deterministic otherwise.

1.1.5 Modules and Objects

New types of objects are made available by writing programs that implement the operations on those objects. These programs may be grouped into modules as illustrated in

[9] and many subsequent papers. The operations are performed on the objects by invoking the access programs of the module. In [9], each module implemented exactly one object. By adding object identifiers as additional arguments for programs, a module's implementation may be used to create more than one object. If the access program allow their user's to create and/ delete objects, one has effectively introduced a new, user defined, data type. This can be done in any programming language; in [9], the ”old” programming language FORTRAN was used. Many programming language designers have thought that it was helpful to include special features for data type definitions in their languages. The most recent languages to do this are usually called ”object–oriented”.

1.1.6 Documenting Software

The problem of software documentation rarely receives adequate attention from computer scientists, who seem so preoccupied with program creation that they have neglected the problems of program maintenance. Even when thinking about program maintenance, they have paid more attention to tools that try (in vain) to compensate for the programmer's neglect of documentation, than to the problem of what the programmer should have done. Nonetheless, there are few general remarks about software documentation that should be made in this chapter. More detailed discussions of software documentation can be found in [13, 14, 12].

1.1.6.1 Specifications vs. Descriptions

Engineers make a useful distinction between specifications of products and descriptions of those products. This distinction seems to be ignored in Computer Science literature. A *description* is a statement of some of the actual attributes of a product, or a set of products. A *specification* is a statement of properties required of a product, or a set of products, [12].

A description may include attributes that are not required, i.e. incidental properties. For example, a description of a program may include the number of ones in its binary representation. A specification may include attributes that a (faulty) product does not possess. The statement that a product satisfies a given specification, is a description of the product.

Any list of attributes may be interpreted as either a description or a specification. ”A volume of more than 1 cubic meter” may be either an observation about a specific container that has been measured, or a requirement for one that is about to be purchased. If you are given a list of attributes, you must be told whether it is to be interpreted as a description, or as a specification. Sometimes one may use one’s knowledge of the world of guess to decide whether a statement is a description or a specification; for example, when discussing Olympic athletes, the attribute ”steroid–consuming” is unlikely to be a specification. Moreover, a specification may offer a choice of attributes; a description of a specific product must describe its actual attributes.

Since most of the programs we build are deterministic, i.e. their output is determined by their starting states, non–determinism is much less important in connection for descriptions than for specifications. Unless we allow some attributes to be not fully determined, we may find ourselves stating requirements that are not really required.

1.1.6.2 Documenting programs

Individual programs are best documented by describing or specifying the effects that their execution has on their data structure. For terminating programs, we usually want to describe the relation between the initial and final values of those variables. For non–terminating programs we want to describe the sequence of values for those variables identified as inputs or outputs [12].

1.1.6.3 Documenting Modules and Objects

When programs are grouped into information hiding modules, the modules should be described without reference to the hidden data structure. Consequently, methods that are appropriate for individual program descriptions, are not appropriate for black box descriptions of modules, [8]. Black box descriptions of information hiding modules, are the subject of the remainder of this thesis.

1.1.7 Composition

Program developers are given sets of building blocks, primitive programs and primitive data types, from which they construct or compose their own products, larger programs and more convenient data types. A question that is frequently asked about documentation methods, is whether they are ”composable”. This term apparently means that the method is capable of describing how larger units are composed of, or constructed from smaller ones. It is our view that this is an irrelevant question to ask about the documentation method. The composition is done by programmers and described in the programming language.

As is the case with other engineering products, we can provide more than one ”view” of a software product. These ”views” include, but are not limited to: system requirements documents, module interface documents, program function specifications, and the programs themselves. While the various views provide different information, they must be consistent. The content and consistency rules for those documents have been discussed in [13].

1.2 What is a Trace Assertion Method

”Black–box” methods for the specification of module interfaces are used to provide a complete description of the interface to a module without suggesting or revealing the implementation of that module. If programs that use the module are based on such a

specification, they won't have to be changed when the implementation is revised without changing the interface. Designing such abstract interfaces can be broken down into two phases:

• A list of assumptions that are unlikely to change during the life cycle of the product.

• The specification of interface relations whose ability to implement objects of the module is guarantied by those assumptions, [23].

A simple method for black–box module specification, is described in [8]. This method had proven useful in small early trials, but soon proved to have fundamental limitations; it could only be used to describe modules in which the effects of invoking a program were immediately visible. A simple queue could not be described because, unless the queue is empty, the fact that a specific value is inserted is not visible until previously inserted items have been removed.

Trace assertion methods are a way to describe effects whose visibility is delayed. A *trace* is a record of the interactions between the module and the environment, [12]. It describes all data passed to the module and all data returned by the module. Trace Assertion Methods are based on the observation that any information we wish to put in a black–box module specification, can be presented as an assertion about traces. Information that cannot be expressed in that way, does not belong in such a specification. A model for this method is presented in [27].

There are 3 types of assertions defining traces: legality (L); assertions defining the values returned by V–programs (V); assertions defining equivalences among legal traces, [10, 27]. A *legal trace* is a trace for which the module is expected to be useful[10]. A *V–program*, returns values that make up the state of a module. For a V–assertion, if T is a legal trace, X is a syntactically correct call on a V–program , and L(T.X) is true, then V(T.X) describes the value delivered by X when called after an execution of T.

If two traces are defined to be equivalent, the outputs associated with those traces, and all their future output possibilities, must be the same, [10]. In other words, equivalent traces have the same externally visible effect on a module. Formally, let T_1, T_2, S be any traces and X be a V–program call, then T_1 is equivalent to T_2 only if :

$$L(T_1) = L(T_1) \quad \text{and} \quad L(T_1) \rightarrow (V(T_1.S.X) = V(T_2.S.X))$$

The assertions about traces that comprise a specification must answer the following two questions:

• When are two traces equivalent, i.e. given two traces, will the future visible behavior of the module be identical?

• Given a specific trace or history, what values will the module return when one of its programs are invoked? Note that the answer to this question, must be the same for all equivalent traces.

Any method which provides black–box descriptions by making assertions about traces, can be considered, a trace assertion method (TAM), [1].

1.2.1 Detailed explanation of a ”trace”

In giving a detailed explanation of what is a ”trace”, we regard the terminology ”input variables”, as a vector of external state variables that an object observes. We also regard the terminology ”output variables”, as a vector of variables whose values are computed by an object, and can be observed externally, [10].

State changes to an object may be caused only by external invocations of an access program, or changes in the values of the input variables. We refer to these as ”events of interest”. Each invocation of an access program is an event, characterised by the name of the access program and the actual values of possible input arguments.

The complete history of an object is a finite sequence of arbitrary length, $O_0E_1O_1E_2O_2E_3$....., where O_0 is the vector of values of the output variables when the object

is initialised; E_i is the ith event of interest; O_i is the vector of values of the output variables after the ith event of interest. A finite sequence which is a prefix to this history, (i.e. $O_0E_1O_1E_2O_2E_3.....E_n$), is a ”trace”.

1.2.2 What is TAM's relation to algebraic methods

The original move from the approach in [8] to that in [1], was inspired by early work on algebraic specifications [2,3]. Algebraic specifications, another approach to black–box specifications define an algebra by a set of equations. The carrier set of the algebra, implicitly constrained by the set of equations, is considered to be the set of possible values of the ”objects” or ”variables” created by a module. In many cases, TAM specifications and algebraic specifications are similar; it often appears that one is a rewrite of the other in a trivially different syntax.

TAM is state machine based, algebraic specifications are algebra based.

There is however a fundamental difference between the two approaches. In TAM, the objects being discussed are described explicitly; they are traces. In algebraic approaches, the ”carrier set” is constrained implicitly and may not even be constrained enough that the cardinality of the set is known. This has led to a large literature on variations in algebraic methods, where the major source of variation is the decision about which of the carrier sets that satisfies the equations, is actually meant. These concerns are non–issues for TAM. We believe that by making the objects of discourse explicit, the method is closer to the intuition of practising programmers.

At least in theory, every TAM specification can be translated into an algebraic specification (see [30]).

As we will discuss in section 1.3, TAM approaches focus on two issues that have not been central to discussions in the algebraic specification approach: canonical forms, and systematic construction.

There are also many variations of TAM. They differ in more pragmatic issues, including notation and the way that non–determinism is handled. As we will see, TAM researchers tend to be more concerned with organisational issues and notation, rather than the theoretical issues that arise in the algebraic discussions.

A good discussion on some algebraic methods and TAM (before 1989) can be found in [19].

1.2.3 A brief history of the trace assertion method

The first paper on TAM as a “black–box” approach to specifying modules, was done by Bartussek and Parnas, [1].

John Mclean developed a formal theoretical model for TAM software module specifications, and developed a formal deductive system for trace specifications, [20].

Daniel Hoffman developed a systematic way of writing Bartussek and Parnas trace specifications, [4,21]. He based his methodology on the following five heuristics:

(1) Choose a normal form.

(2) Structure the semantics according to normal form prefixes and single element extensions.

(3) Use predicates to decompose complex assertions.

(4) Develop specifications by incrementations.

(5) Write macros to make specifications more readable.

Parnas and Wang, seeking to improve readability, introduced tabular formats, a rigid structure for writing and verifying correctness in specifications. A detailed report on their work can be found in [10].

Wang further suggested improvements to making an even more rigid structure in his thesis, [27].

1.2.3.1 Other related TAM work

Research involving TAM has been not limited to just the Software Engineering Research Group at McMaster University. There is orthogonal work being done at Institute of Informatics, Warsaw University in Poland and the University of Quebec, Hull, Quebec, Canada, [29, 31, 32, 33, 34].

In Wang's work ([27]), he showed that for the purposes of simulating module interfaces, an extension to the traditional term rewriting systems (a *trace rewriting system*) provided an adequate operational semantics for trace specifications. A support tool based on the semantics presented in Wang's work, was further developed, [31].

Other interesting work and discussion of TAM in the Software Engineering Research Group at McMaster University includes, separately, those of Theodore Norvell and Ryshard Janicki.

Norvell suggest ideas, for TAM to use a base of firm mathematical foundations, [28]. His work was built on the ideas of [27, 10, 29].

Other important illustrations of solutions to some problematic modules' (involving the role of non–determinism, normal and exceptional behaviour, value functions and multi–object modules,) have been introduced and discussed by Janicki, [30].

One of the main goals of the group at Warsaw University (Institute of Informatics) is to develop tools to support TAM. Their tools include a trace simulator and editor. Discussion of their approach and that presented by Wang (for simulation), can be found in [38]. More information on their TAM editor can be found in [36, 37]. Their work has recently included a rigorous report on TAM and proof strategies for TAM specifications.

1.2.4 The essential characteristics of trace assertion methods

Although there is considerable variation among the various ”flavors” of TAM, there are two fundamental characteristics that distinguish the method from other approaches. Although Parnas and Bartussek did not use canonical forms explicitly, a canonical representation based on traces plays a central role in TAM, [1]. Further, the recent variations have stressed a systematic and restricted specification style.

1.2.4.1 Use of a canonical representation

Trace assertion methods all provide a way of systematically reducing a trace to an expression that is a unique (hence *canonical*) representative of all traces that are equivalent to that trace. In earlier versions, that canonical representation has been one member of the set of equivalent traces. In most variations, a predicate on the set of possible traces, characterising the set of canonical traces is a key part of the specification. The canonical representation of equivalence classes of traces provides an easy way of determining whether or not two traces are equivalent. Each trace can be reduced to its canonical representation.

The use of canonical representations is a way of assuring that a specification is not biased towards any particular implementation. Efficient implementations usually have many equivalent data structure states and may differ in the number of states. By always using a representation that is canonical, i.e. one that has the minimum number of abstract states, we avoid giving information about the actual representation. In fact, the sets of representations described by all correct specifications written using a trace assertion method, will be isomorphic. One specification will not be closer to a particular implementation than another one. The specification will be ”closer” to an implementation with the minimum number of states, than to other implementations, but there are many such implementations and they are rarely practical enough to be considered.

1.2.4.2 Systematic Descriptions

In the more recent versions of TAM, there has been a stress on imposed (mathematically unnecessary) restrictions to the way that the specifications are organised. Black–box module specifications can be viewed as collections of axioms or equations. Consequently, many researchers have observed that, unlike commands in programs, assertions in a specification can be written in an arbitrary order. However, engineering experience suggests that a rigid organisation has practical advantages. It is easier to systematically detect either incompleteness or inconsistency if there is only one place where information can be found. Rigidity in organisation also facilitates the use of a specification as a reference work. It is important that one need not read or search the whole specification in order to use it.

The key structural restriction in TAM specifications is known as Single Element Extension (SEE). The rules for reducing a trace to its equivalent canonical trace are written, by describing what happens if a canonical trace is extended by a single invocation of an access program. The resulting trace must be equivalent to a canonical trace. Using the SEE method, a deterministic specification is organised as a set of extension functions. There is one such function for each of the access programs. The domain of each function is the cross product of all canonical representations and all possible trace extensions involving the access program. A specification written in this form must be complete and consistent. It is easy to check for completeness; checking for consistency in a deterministic module, requires making sure that each extension function is, in fact, a function. On the other hand, for non–deterministic modules, only operations by access programs which do not contribute in giving the module it's non–deterministic property, can be checked for consistency. For more e details on non–determinism see [30, 28].

This method was often used in the original TAM, but never explicitly stated as a rule or principle. Explicit discussion on the SEE principle first appeared in journal form in [4]. The SEE principle was refined somewhat by using tabular notation to assure coverage of all extension cases in [10].

1.2.4.3 An older TAM format

Below we describe, the general format of the older TAM, used in specifying modules, [10]. This format does not work for really non–deterministic modules because of the Mealy machine basis, (see [30, 28]).

The typical key words for module specification are in standard font. Italics explain what is required at the point of their appearance (see example below).

Conditions	Equivalences
predicate on the trace before access program invocation	*%illegality% or new trace*
⋮	⋮
⋮	⋮

Table 1. A Meta table from TAM's format, section (3)

Conditions	Equivalences
T = _	%empty%
T ≠ _	T1 where T = T1.PUSH(a)

Table 2. Equivalence T.POP of stack example using Table 1.

"T = _" and "T ≠ _" are predicates on a trace, T. "%empty%" is the illegality of the extension to the trace, T. "T1 where T = T1.PUSH(a)" gives the new trace, T1.

In the meta tables below, the symbol :
"$\ldots$" is, continue across.
"$\vdots$" is, continue downwards.
"$\ddots$" is, continue downwards and across.

TYPE IMPLEMENTED: *module type*

(1) SYNTAX

OUTPUT VARIABLES *(optional)*

Variable Name	Type
first variable name	*variable type*
⋮	⋮
⋮	⋮
last variable name	*variable type*

ACCESS PROGRAMS

Program Name	Arg1	…	…	Arg*n*	Value
first access program name	Arg1 *type, if any*	…	…	Arg*n type, if any*	*return value type, if any*
⋮	⋮	⋰	⋰	⋮	⋮
⋮	⋮	⋰	⋰	⋮	⋮
last access program name	Arg1 *type, if any*	…	…	Arg*n type, if any*	*return value type, if any*

(2) CANONICAL TRACES

canonical (T) ↔ (T = *a mathematical expression using access program name/s*)

(3) EQUIVALENCES

Either

T.*program name with arguments* ≡ *a new trace*

or

T.*program name with arguments* ≡

(a table with conditions and corresponding new traces or %legality%

as *shown below)*

Conditions	Equivalences
predicate on the trace before access program invocation	*%legality% or new trace*
⋮	⋮
⋮	⋮

(4) VALUES

OUTPUT VALUES *(optional)*

Either

V[*variable*](T) = *a value*

or

Conditions	Values
predicate on a trace before access program invocation	*%legality% or a value*
⋮	⋮
⋮	⋮

⋮

⋮

RETURN VALUES =

Program Name	Argument No.	Values
first access program name	Value	*variable name or function name or %legality%*
⋮	⋮	⋮
⋮	⋮	⋮
last access program name	Value	*variable name or function name or %legality%*

1.3 Motivation for a new TAM (ITAM)

Although the TAM has been studied extensively and support tools are available [35, 27, 36, 37], its acceptance in industry has been disappointing. Those who have examined it, feel that it has definite advantages over algebraic specifications and other earlier methods, but with a few exceptions, those who have attempted to use it find it clumsy and difficult to use. The expressions seem unnecessarily long and the canonical representations hard to understand. More serious is the need to deduce conclusions from complex mathematical expressions. Programmers, our ultimate audience, prefer information that is stated directly rather than information that is stated implicitly.

It has become clear that although the method is good ”in theory”, and the fundamental characteristics described are good ones, some changes are needed before it is found to be in ”good practice”. The new TAM (ITAM) shares the fundamental characteristics described earlier and is more ”natural” for programmers. Canonical representations are now structured to help programmers understand the composition of any object created by a module.

1.3.1 Canonical Representations

In the older TAM the canonical representations are always traces, i.e. sequences of access program invocations. This had several disadvantages :

• The representations are unnecessarily bulky. In many cases the names of the access programs carries little or no information and are a distraction.

• It is difficult and clumsy to refer to parts of the representation.

• The requirement that the representation be canonical often forces the designer to impose restrictions on the entire canonical representation. Even where some parts of the representation, can be independent of other parts, an order was imposed. The descriptions

of those restrictions is a source of confusion in the specification. (This issue has also been discussed in [30].)

One can represent a sequence as a function whose domain is a set of contiguous natural numbers commencing from 1. This has led us to develop a richer collection of functions (including the function for sequences,) to choose from when describing a canonical representation. This has led us to make the following major adjustments to canonical representations :

• A representation may be structured as a set of named components; any named component may be a set of components, (e.g. Example 2 of Appendix A).

• A set without certain restrictions may be used instead of a sequence in the canonical representations in order to avoid imposing arbitrary ordering. Sets, whose members are in an arbitrary order, can be indexed by a set of : mnemonic (string) names or arbitrarily chosen natural numbers (greater than 0).

• When ordering is necessary (i.e. the need for a sequence), we use a function whose domain is restricted, (see Table 9. in chapter 2). The use of this function is clearly demonstrated by the stack examples (Example 1) in Appendix A.

The set of stacks in Appendix A, Example 2, are independent of each other and there is no need to pick just one continuous ordered representation as its canonical representation. (This can be blamed mainly on the notation provided by some older TAMs.) Instead we pick a set of canonical states to represent all the objects of the module thereby conveying to the reader that the order of the stacks do not matter. On the other hand since there is order within each stack, we use a function whose domain is constrained.

Although we do not show a model for ITAM in this thesis, the model will resemble the one presented in [30].

We describe canonical representations by an expression that describes the characteristic predicate of the set of canonical states[1]. The structure of this descriptive expression and the notation used, is based on common programming languages (e.g. [24]). Such notation is more familiar to programmers than the more traditional mathematical notation we were using.

Preliminary experience with the last three innovations mentioned above suggests that they greatly improve the readability and understanding of module interface specifications. In ITAM, we do use access program names in describing the canonical representation. We regarded this as clutter in the old TAMs. The ability to name components of the representation greatly simplifies making precise reference to parts of the representation. The use of sets is valuable when we wish to specify that there will be acceptable user visible differences in behaviour from one implementation to another, i.e. where we wish to allow a choice to be made by the implementor.

It must be stressed that the use of sets for canonical representations, rather than the old form of traces, does not change the fundamentals of the method. There is nothing we can specify that we could not have specified before. However we believe that the specifications are easier to read and to write. The expressive powerful of the older TAM and ITAM is the same but the latter is more user–friendly.

If the order of objects in a sequence is significant, we can still choose between several representations and the specification writer must make that choice and use the special set for sequences.

If the order of the objects is not significant we use set notation that disregards ordering of objects instead of using a single sequence which leads the reader to think that

1. In an ITAM module specification, the canonical state of an object is any state that satisfies the characteristic predicate for that specific module.

all objects must be ordered. In some later versions of TAM this was done by dividing the sequence into separate sequences, each representing an object with no identifiable order.

The use of sets make specifications of non–deterministic modules easier to read an write.

1.3.2 Canonical Rep. for a bounded ”multi–object” module

As shown in some earlier examples, ”multi–object” module specifications can be written as an infinite set of finite state machines, (e.g. the unbounded examples in [10]). However if the number of objects allowed to be created by a module is known, and the order of creation is arbitrary, we must present the module as a finite set of finite state machines. In the older TAM, it is difficult to write the characteristic predicate of the set of canonical traces for a ”bounded multi–object” module when creation and naming of objects, is done at run–time.

Ensuring the finiteness of multi–object modules has led to the non usage of a subscripted (named) empty trace ”$T_n = _$” in a canonical representation, (given in the multi–object examples of [10]). The disadvantage is, named empty traces represent objects. Realistically there are an infinite amount of names, so by using an infinite amount of names in a ”multi–object” module specification conveys to a the specification reader that there are an infinite amount of objects that can be created on a single system. Thus by using ”$T_n = _$” creates difficulty when one needs to specify a module that will create one finite state machine (i.e. the *primary object*[2]) with a collection of finite state machines.

We are able to remove this disadvantage :–

• by naming only those objects or sets of existing objects created by a module, making it easier to place bounds on the module, thus ensuring one finite state machine.

---

2. A primary object is self–contained. In ITAM it is referred to as ”rep”.

• by using access programs with names as arguments, to create new or delete old objects in order to make specifications more realistic.

We redocumented those not–*fully* bounded module of [10], as *fully* bounded modules. A fully bounded module will create a finite state machine.

1.3.3 Use of Program Function and Other Tables

Earlier versions of TAM introduced the use of tabular notation in a rather *ad hoc* way. The tables used in these documents were different from those studied in our other work [5,11,14,6,7,15]. In more recent work, we have learnt that tables are simply expressions that are useful representations of mathematical relations. In ITAM although we can use any type of tables from our other earlier work, we have chosen to use vector tables [11]. These tables are also known as program function tables [14]. We use them to describe state changes of a module's objects, caused by invocations of that module's programs, (*operation tables*). By using sets, the SEE principle and vector tables, we have made it easier for readers to separately :

• identify objects that have state changes in an object's state representation.

• identify the new state representation of an object that has changed state,

1.3.4 Parameterised Specifications

When languages such as C++, are used in naive ways (which seems to be the usual case), we can end up with hundreds of types of objects that are almost alike. Doing reverse engineering we discover that the TAM specifications for these objects also end up to be almost alike. A simple analogy would be resistors. We would not want to write the descriptive equations for resistors or other circuit components over and over again and, more importantly, nobody would want to read them. Instead, we have written those specifications (e.g. in a textbook), in terms of key parameters such as resistance, power handling capacity, and accuracy, then specify each resistor by giving values to the parameters. The same

technique can be applied to families of more complex circuits. We include parameterisation of specifications in ITAM. (See the parameter ”CAP” in example 1b, Appendix A.)

1.3.5 Using ”Legality” to reduce table size

In the older TAM when an object’s state representation (a trace) was extended by a single access program call under a specific condition, the resulting state of the object would have been either a ”legal” or ”illegal” trace, see [1,10]. Intuitively, legal traces were those for which the module was expected to be useful. Illegal traces contained events that a user of the module was supposed to avoid. Illegal traces would not occur if the module was used correctly. There is no way of preventing an uninformed user from using a modules program in an illegal way especially when that user does not know the present state of the module’s objects.

In ITAM legality is not an issue and is handled differently than before, every access to a program of a module can be now viewed as being ”legal” (i.e. useful and useless extensions are allowable). When all the objects of a module have similar states after the invocation of an access program, this does not mean that they should be handled similarly. In such instances the ”legality” (as replaced by the *extension class*[3] of an access program table) will be different, giving rise to many different cases of legality some useful others useless.

In the ITAM the phrase ’extension class’ appearing in an operation table (shown in chapter 4), is treated as if it were a variable and the value is a *status*. Handling strategies for the status of the extension class of an access program may be taken care of in some other documentation and not in a module specification document.

---

3. The extension class allows two extensions that are equivalent to the same canonical representation to be classified differently. The main purpose is status reporting.

When the extension classes are similar under different conditions, these conditions can be logically “OR”ed together and be represented as one column in a table (as was done in the older TAM).

1.3.6 Using abbreviations to reduce table size

In further reduction of large tables, in ITAM, we have introduced a method of abbreviating conditions found in both auxiliary functions and access programs. Our abbreviations are short pieces of text that are mapped onto longer pieces of text. We evaluate abbreviations before all evaluations of the expressions in which they appear and we do so by simple text substitution without any regard for the semantics of the operations. Abbreviations may themselves be composed of other abbreviations, the latter referred to as an *intermediate* abbreviation.

1.3.7 Using auxiliary relations for further reductions

We have introduced other predefined auxiliary relations to reduce repetitive use of long mathematical expressions in, describing canonical representation. At the same time users of ITAM should be aware of these relations beforehand and will not have to spend any extra time writing their own or learning newer relations that serve the same purpose.

There are two distinct predefined auxiliary functions written to check type and check presence, of objects. These functions are used quite frequently in our examples.

An access program can operate on objects of different types, (see *polymorphism* in [26]). When we specified modules with access programs operating on different types, we found the need for type checking.

When objects are created at run–time, it is essential that the object’s name be unique. Objects cannot be operated on unless they are present. Consequently, checking for the presence of an object is essential.

When using our predefined auxiliary functions, occasionally an ITAM user may find it unnecessary to pass some arguments. In such cases we provide a strategy that allows ITAM users to not provide these arguments, (in chapter 2).

In documenting a module using ITAM, there are only two specific subsections of the ITAM format allotted for defining local auxiliary relations. The scope of using a locally defined auxiliary function will depend on which subsection it was defined within. The two subsections and the scope of using local auxiliary functions are discussed in more detail in Chapter 4. Such small cosmetic rigidity, helps a specification writer to organise the specification so that a reader can understand the document more quickly.

1.3.8 Other format changes to improve reading

ITAM's format consists of four sections, the first being the header, the second the canonical representation section, the third being the syntax section and lastly the operation tables section. Some of these sections are further subdivided. The details of ITAM's format are in Chapter 4.

With respect to return values, we have incorporated the old TAM's Output Variables of the Syntax section and the Values section into ITAM's Operation Tables section. All output values will be returned as an n–vector of values, [16]. We provide a method to refer to an element of the vector by using subscripted access program names as output variables.

In chapter 4 other subtle format changes will be introduced. This will provide a consistent format for readers and writers so that they can locate information more easily.

CHAPTER 2: CANONICAL REPRESENTATIONS

2.1 Semantics for ITAM module specifications

In this part of the document we lay the foundation for module interface specifications.

The meaning of the following terminology should be understood, and most can be found in any good mathematics textbook: relation, binary relation, function, predicate, variable, constant, parameter, argument, ordered pair, string, real, boolean, integer, primitive types.
Tuple, n–tuple, simple n–tuple are explained in [16]. A report on types is found in [18].
The general concept of a set is assumed to be known from any book on set theory, (e.g. [17]).

2.1.1 Terminology and notation

As an aid to the reader, in this and the following chapters, *set descriptors* will be in bold italics.

[4]When we refer to word as a set descriptor, then we are using that word in referring to a specific set.

[5]The set descriptors of primitive types are: ***real***, ***integer***, ***natural***, ***string***, ***boolean***. The primitive types ***real***, ***string***, ***boolean*** are pair–wise disjoint.

For the purpose of ITAM we define value, by saying that the set of values is the smallest set such that :

---

4. e.g. ***integer*** is the collection of all possible integers, thus ***integer*** is a (predefined) set descriptor.
5. ***natural*** $\subset$ ***integer*** and ***integer*** $\subset$ ***real***.

• all elements of the primitive types are values.

• the empty set is a value.

• each set consisting of values is also a value.

value is the set descriptor for the set of all possible values.

In an ITAM module interface specification when we refer to a set as being fixed, we mean there are no operations that are allowed to change the set throughout the specification. The result of changing (i.e. adding, removing or substituting elements of) a set, is a different set.

Every software system composed of modules, should have a method of identifying each module uniquely. The title of a module specification document identifies that particular module specification. For a software system with ITAM module specifications, we identify each module by a unique title. In the older TAM what we refer to as the module's type is what we will call the title in this chapter, (i.e. the string that follows ”type implemented”, of a module in [10]).

2.1.1.1 Abbreviations, new primitive types and the TYPE relation

We will abbreviate the following terminologies and use them as we progress through this thesis.

PT (primitive types) is the set of set descriptors, {***real***, ***integer***, ***natural***, ***string***, ***boolean***}.

PTU is the set, ***real*** ∪ ***string*** ∪ ***boolean***.

A **PE** (element of a primitive type) refers to any one element of **PTU**.

A **UDT** is an abbreviation for user–defined type. There are 2 parts required in defining a **UDT**, a set descriptor and a set description. Both the set descriptor and the set description must be unique in the module specification they appear in. A **UDT** set

description is constructed using the notation in section 2.3.2, Table 11. A user can construct set descriptions that are tuples using this notation. The set descriptor identifies the set given by the description.

name is the set ***string***[6] $\cup$ ***natural***.

We define **contignat**(S) by, $((i \in S) \rightarrow ((i \in$ ***natural***$) \wedge (i>0) \wedge ((i>1) \rightarrow ((i-1) \in S))))$ and ***contignats*** = {S | **contignat**(S)}.

TYPE is a relation that is used in a module's specification, whose domain is the set, ***value***$_M$ and range is a set, ***type***$_M$, where M is the title of the module specification. ***value***$_M$ contains all values used in representing the state of possible objects that can be created by the module, and ***type***$_M$ is all set descriptors of the values used by the module.

Thus for a module specification titled M , $(\forall v, (v \in$ ***value***$_M) \rightarrow ($**TYPE**$(v) \subseteq$ ***type***$_M) \wedge ($**TYPE**$(v) \neq \{\ \}))$.

2.1.2 Representing an object's state in a module's specification

An object's state for a module's specification, titled M, will be a value of a type belonging to ***type***$_M$.

2.1.3 Sets used to represent an object's state in module specifications

We will refer to the 2 elements of an ordered pair as *components*. All structures that represent the externally observable state of an object will be declared in ITAM as a set with constraints. We will constrain the set in the following ways :–

- the type of the elements and, if the elements are ordered pairs, their components.
- the structure of the elements.
- the operations that we can perform on the set.

---

6. Elements of ***string*** are quoted (e.g. "a", "1", "1a", "a1", ...)

We will use three kinds of sets in constructing a state representation for objects: **PT** members, *SP*s and *PSET*s. We define SP and PSET below.

An SP set is expressed either as, a set of Table 4. or Table 5.

Any PSET is expressed as, either of the sets given in Tables 7. to 10. PSETs used in describing canonical representations were developed from the PSET structure constructors appearing in section 2.2.

A summary for the notation to construct both SP and PSET structures can be found in Table 11.

2.1.3.1 SP

■ SP is a user defined set of **PE**s, (refer to tables 2 and 3).

2.1.3.2 PSET

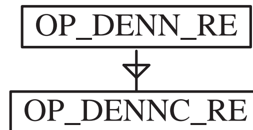
■ PSET is a user defined function, whose domain is a set of **PE**s and range is a set of values.

2.2 The PSET structure–name tree

The PSET structure–name tree, (Fig 1.) was brought about during our trials of the examples of Appendix A and B. The root of this tree is in the middle (i.e. OP) and the branches grow outwards.

The boxed structure–names are used to define other PSETs that we use in describing the canonical representations in the examples of Appendices A and B. Formal definitions of these functions and the way we will write them in a specification, is given in Tables 7. to 10.

An arc form node X to node Y means that Y is a subclass of X. For instance if OP_DENNC_RE is a subclass of OP_DENN_RE, in the tree it will be shown as :



A legend for understanding the cryptic structure names used in the tree is provided on the page following the tree structure.

To read the abbreviations of Fig 1., we assemble the definitions found on the right. For instance, the structure–name OP_DF:ENS_R:E definition will be read as follows : "*ordered pairs of a PSET, with a fixed domain which is, a singleton and a member, of a fixed type; the singleton is a subset of* ***string****, with a range which is, a singleton and a member, of a fixed type*".

2.2.1 Naming structures for sets of ordered pairs

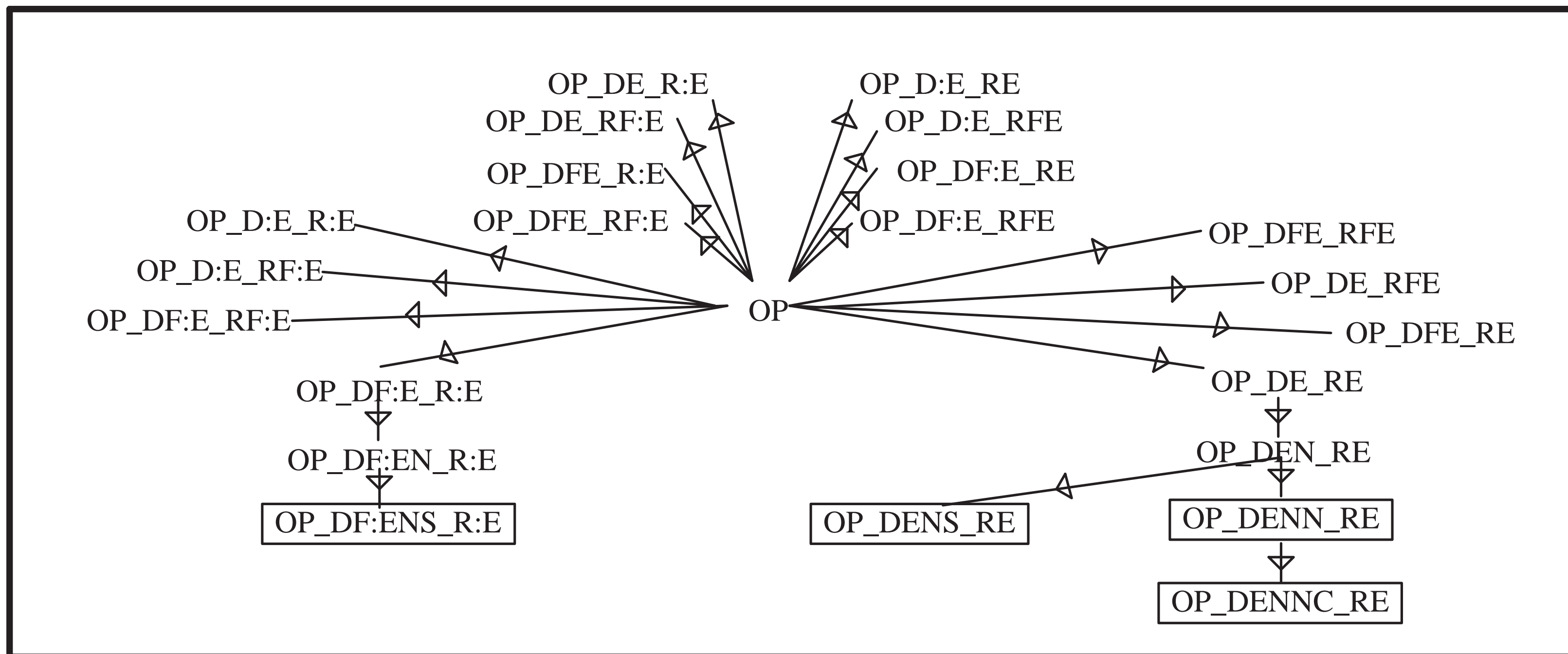


Fig 1. A tree of the PSET's structure names base.

Legend for structure–name constructors:

OP	– ordered pairs of a PSET[7]
_	– , with
DE	– a domain of **PE**s
RE	– a range of elements which is a subset of a defined type[8]
DFE	– a fixed domain of **PE**s
RFE	– a fixed range of elements
D:E	– a domain which is, a singleton and a member, of a defined type; the element of the singleton is a **PE**
R:E	– a range which is, a singleton and a member, of a defined type
DF:E	– a fixed domain which is, a singleton and a member, of a defined type; the element of the singleton is a **PE**
RF:E	– a fixed range which is, a singleton and a member, of a defined type
DEN	– a domain which is a subset of ***name***
DENS	– a domain which is a subset of ***string***
DENN	– a domain which is a subset of ***natural***
DENNC	– a domain which is a member of ***contignats***
DF:ENS	– a fixed domain which is, a singleton and a member, of a defined type; the
singleton	is a subset of ***string***

The tree of Fig 1. allows for growth of other PSET type structures.

---

7. Refer to definition in section 2.1.2.2; only **PE**s are used in the domain. In our examples we only needed ***names***. **PE**s were chosen to accommodate any future changes that may require the other **PTU** elements, in PSET's domain.

8. A defined type is either a **PT** or **UDT** and will appear in the Type Definition section of a module interface specification of an ITAM document. This specification section is explained in chapter 4.

2.3 The syntax for state representations in module specifications

In creating specifications using ITAM, we will use sets (mentioned earlier in the semantics section,) to represent the abstract state of modules. The representative state of a module can be altered only by deletion, addition or substitution.

In our work so far, the only sets that were required to document module specifications without redundancy and with efficiency were sets that were: members of **PT**, SP and PSET.

We found that the only PSET structure–names that were used to explain the functions used in describing canonical representations, are: OP_DF:ENS_R:E, OP_DENS_RE, OP_DENN_RE and OP_DENNC_RE.

Below, 'dt' is used as a set descriptor. It abbreviates the phrase, ***d****efined* ***t****ype*. Syntactically a defined type is denoted by a set descriptor in the canonical representation section (see section 4.2.2, type definition section) of a module specification.

The following are used below : **PT**, **PTU**, ***natural***, ***string***, ***contignats***, and ***type***. Their explanations can be found in section 2.1.1.

2.3.1 SP and PSET

The following table explains some syntax that will be used often in the type definition section for describing canonical representations in ITAM module specifications.

Some fixed syntax	Meaning
type T =	(a) T∈***type***$_M$ (i.e. for a module titled M, T is a set descriptor that is not already used). (b) type – is syntax and is reserved in a module's specification, only to be used in prefixing all set descriptors. (c) = – is syntax which will follow all set descriptors and means 'is defined to be'.
type SNE = {{}}	SNE defines the fixed set {{}}, (refer section 2.4.3) .

Table 3. Common type definition syntax[9]

The syntax for a module titled M, with a set descriptor T, is given below.

We will use the following sets to describe canonical representations.

2.3.1.1 SP syntax

An SP set will be written in either of the 2 ways as shown in the Tables 4. and 5., below. The first one use the structure constructors: parentheses and commas, the other uses SUBP (see Table 11.).

2.3.1.1.1 Explicit SP set

Table 4. shows the typical (mathematical) syntax used in describing a set of elements which is a subset of a **PT** member. We refer to this as an *explicit set*.[10]

Syntax	Meaning
type T = $\{x_1,x_2,....,x_n\}$	$T = \{x_1,x_2,....,x_n\} \wedge (\exists P,(P \in \mathbf{PT}) \wedge (x_1,x_2,....,x_n \in P))$

Table 4. Meaning of an SP set using typical math syntax

9. The "=" notation appearing after a set descriptor must not be confused with the predicate operator =. Here it simply separates the set descriptor from its definition.

SNE will be treated from here on as a reserved string.

10. We use the phrase *explicit set* since all the elements of the set are explicitly listed for the reader.

2.3.1.1.2 SUBP

Table 5. shows the syntax we use to describe a set of elements which are subsets of a defined type, dt. This defined type is a subset of **PTU**.

Syntax	Meaning
type T = SUBP dt	$((dt \in \boldsymbol{type}_M) \wedge (dt \subseteq \mathbf{PTU})) \rightarrow (T=\{S \mid (S \subseteq dt)\})$

Table 5. Meaning of SUBP syntax

2.3.1.2 PSET structure names and syntax

In Fig 1., the boxed structure names were found to be the only PSET structure–names common to all our examples. Later we will define special functions, (those listed in the right column of Table 6.) which use the following structure–names given in the left column. CSET, SSET, NSET and ARRAY, are PSETs. SSET, NSET and ARRAY, are CSETs. The functions SSET NSET and ARRAY are used to describe canonical representations. In our examples, CSET was never used to describe canonical representations but was used in the signatures for predefined auxiliary functions (section 2.4.5.1).

CSET, SSET, NSET and ARRAY can be viewed as relabelling of the structures represented by their corresponding structure names (left column in Table 6.).

REC is a PSET composed from other PSETS. REC has a fixed domain whose elements are the first components (of singletons pairs,) of OP_DF:ENS_R:E structures, and range is the second components from those same structures.

Structure–names from Fig 2.	Special functions
OP_DEN_RE	CSET
OP_DENS_RE	SSET
OP_DENN_RE	NSET
OP_DENNC_RE	ARRAY
OP_DF:ENS_R:E	REC

Table 6. Useful PSET structure–names and sets

The syntax that we use to describe the PSET functions, is not the conventional way. We write them in a fashion that programmers are more familiar with. The structure constructors for PSETS are: SSET; NSET; ARRAY; REC, commas, colon and round brackets, (see Table 11.).

2.3.1.2.1 SSET

Table 7. shows the syntax we use to describe all functions, whose domain are strings and range is a subset of a defined type, dt.

Syntax	Meaning
type T = SSET dt	$(dt \in \boldsymbol{type}_M) \rightarrow (T = \{f{:}S \rightarrow dt \mid S \subseteq \boldsymbol{string}\})$

Table 7. Meaning of SSET syntax

2.3.1.2.2 NSET

Table 8. shows the syntax we use to describe all functions, whose domain is a subset of ***natural*** and range is a subset of a defined type, dt.

Syntax	Meaning
type T = NSET dt	$(dt \in \boldsymbol{type}_M) \rightarrow (T = \{f{:}S \rightarrow dt \mid S \subseteq \boldsymbol{natural}\})$

Table 8. Meaning of NSET syntax

2.3.1.2.2.1 ARRAY

Table 9. shows the syntax we use to describe all functions, that represents a sequence of elements. A set of all elements of the sequence, is a subset of a defined type, dt.

Syntax	Meaning
type T = ARRAY dt	$(dt \in \boldsymbol{type}_M) \rightarrow (T = \{f{:}S \rightarrow dt \mid S \in \boldsymbol{contignats})\})$

Table 9. Meaning of ARRAY syntax

2.3.1.2.3 REC

Syntax	Meaning
type T = REC $(x_1,:dt_1),(x_2,:dt_2),.....,(x_n,:dt_n)$	$((dt_i \in \textbf{\textit{type}}_M) \wedge (\forall q,r,(1 \leq q \leq n) \wedge (1 \leq r \leq n) \wedge ((q \neq r) \rightarrow (x_q \neq x_r)) \wedge (x_q \in \textbf{\textit{string}})))\rightarrow(T = \{(x_i,y_i) \mid y_i \in dt_i\})$

Table 10. Meaning of REC syntax

Table 10. shows the syntax we use to describe a function whose domain is a subset of ***name***. The range is a set of elements where each element is a member of a defined type given in the description (e.g. a dt_i). For further explanation refer to the explanation given for REC in section 2.3.1.2, above.

2.3.2 Summary of notation for UDT descriptions

Notation	Explanation
( )	Used for ordering elements
{ }	Used for sets
,	Used to separate elements
SNE	Refer to Table 3.
SUBP	Refer to Table 5.
∪	Refer to Table 13.
∩	Refer to Table 13.
−	Refer to Table 13.
×	Refer to Table 13.
\+	Refer to Table 18.
\−	Refer to Table 18.
SSET	Refer to Table 7.
NSET	Refer to Table 8.
ARRAY	Refer to Table 9.
:	Used to prefix REC elements that are types
REC	Refer to Table 10.

Table 11. UDT description notation

2.4 A new rule, old relations, connectives, new notation and relations for ITAM

In this section the earlier tables to follow, will have notation that is already familiar to the reader whereas the later tables have, new notation.

The order of the relations to follow in this section will be: the common set relations, appearing first (including the relations on *string* and *real*); new notation and relations common to all sets; predefined PSET relations.

New Rule: We have added a new rule when we use relations (especially the predefined auxiliary functions) in the new access programs (*operation tables*), to operate on an object. As a default, when the condition/s of the relation is/are not satisfied and the unsatisfied condition/s is/are not considered elsewhere in the operation table, then the object's state remains unchanged, (discussed in Chapter 5, example 4).

2.4.1 Common set relations, set operations, and predicate connectives

Assume S and Q are set descriptors and x is a value

Relation Notation	Predicate Operators' Examples	Explanations
$=$	$Q=S$	equality
$\neq$	$Q\neq S$	non–equality
$\in$	$x\in S$	set membership
$\notin$	$x\notin S$	set non–membership
$\subset$	$S\subset Q$	left is proper subset of right
$\subseteq$	$S\subseteq Q$	left is a subset of right

Table 12. Common allowable predicates on set, S

Relation Notation	Set Operations' Examples	Explanations
∪	S∪Q	set union of S and Q
∩	S∩Q	set intersection of S and Q
–	S–Q	set difference between S and Q
×	S × Q	Cartesian product of S and Q
CARD	CARD(S)	number of primary elements of a set, S

Table 13. Common allowable set relations

Assume p,q are predicates and x is a variable

Notation	Examples	Explanations
¬	¬ p	not
→	p→q	implication
↔	p↔q	implication both ways
∨	p ∨ q	or
∧	p ∧ q	and
∀	∀x,p	universal quantification
∃	∃x,p	existential quantification

Table 14. Common connectives and notational conveniences for predicates

2.4.2 Operations on *string* and *real*

Assume x and y are elements of *string*

Relation Notation	Predicate Operator Examples	Explanations
=	x=y	equality
≠	Q ≠ S	non–equality

Table 15. Allowable predicates on strings

Assume x and y are elements of ***real***

Relation Notation	Integer Operations Examples	Explanations
*	x*y	the product of x and y
/	x/y	the maximum number of times we can subtract y from x with a non–negative result
mod	x mod y	the result x–((x/y)*y)
+	x+y	the sum of x and y
–	x–y	the result of subtracting y from x

Table 16. Allowable real operations

Relation Notation	Predicate Operator Examples	Explanations
=	x=y	equality
≠	Q ≠ S	non–equality
<	x<y	x less than y
≤	x ≤ y	x less than or equal to y
>	x>y	x greater than y
≥	x ≥ y	x greater than or equal to y

Table 17. Allowable predicates on reals

2.4.3 New common set notation and relations

Notation	Definitions
$R(n)_x$	if R is a relation, n is in its domain, an m–tuple in its range, and $x \le m$ then $R(n)_x$ is the xth element in the m–tuple of an object o, such that the ordered pair $(n,o) \in R$
S[n]	if S is a PSET, S[n] is an object o, such that the ordered pair $(n,o) \in S$
{ }	the empty set; for empty set S, CARD(S)=0
{{ }}	Set of sets with one element, an empty set
DOM(X)	the domain of a set X
RAN(X)	the range of a set X
S1 \+ s	S1 ∪ {s}, S is a set descriptor and s is an element to be added to S
S1 \– s	S1 – {s}, S is a set descriptor and s is an element to be removed from S

Table 18. New common set notation[11]

Relation	Relation Examples	Definitions
Tchk	Tchk(t,v)	true for $(v,t) \in$ **TYPE**
Tchks	Tchks(S)	true for a set S, for all y, $y \in S$ implies Tchk(t,y)
Cvt	Cvt(t,v1,v2)	true for (Tchk(t,o1) ∧ Tchk(t,o2))
Ls	Ls(S,X,n)	true for sets S, X and an integer, n, CARD(X)=CARD(S)–n and $X \subset S$

Table 19. New common set predicates[12]

11. An example to further clarify, if R(n) = (1,(b,3,((d,5,6r,7f),a5))8) then $R(n)_1$ is 1; $R(n)_{2,1}$ is b; $R(n)_{2,3,4}$ is 7f; $R(n)_3$ is 8; etc...

12. It should be noted that TCHK's first argument is always a set descriptor whose definition is a fixed set.

2.4.4 New PSET predicates

Notation	Operations Examples	Definitions
N_fnd	N_fnd(S,n)	true for n∈DOM(S)
NO_fnd	NO_fnd(S,n,o)	true for (n,o)∈(S)
O_fnd	O_fnd(S,o)	true for ∃n, (n,o)∈S

Table 20. New predicates for a PSET, S

2.4.5 Naming PSET's predefined non–predicate auxiliary functions

The examples of Appendices A and B have shown that only the following 3 basic operations are needed for changing PSETs: delete, add and substitution.

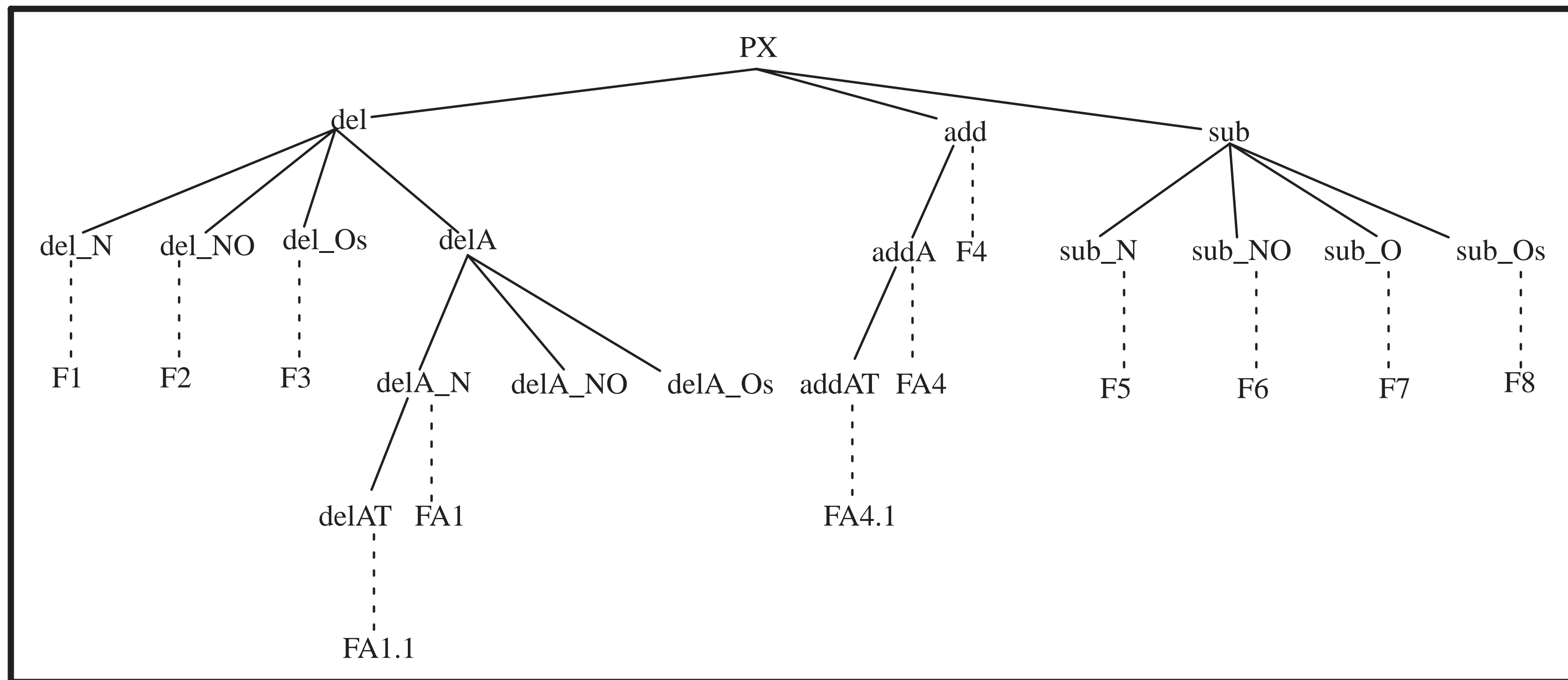


Fig 2. A tree of predefined auxiliary PSET function names.

In Fig 2. predefined auxiliary function table identifiers are connected by dotted lines to their corresponding function definitions names. Table identifiers are located in the upper left corner of the tables, see section 2.4.5.1 . Assembling the function name constructors are done in the same fashion as for Fig 1.

Legend for the PSET function name constructors:

PX	– predefined PSET auxiliary function which are not predicates
del	– delete a pair
add	– add a pair
sub	– substitute
delA	– delete a pair from an ARRAY
delAT	– delete the pair at the top of an ARRAY
addA	– add an object's value into an ARRAY at a given existing index.
addAT	– add an object's value at the top of an ARRAY
_	– , with
N	– a given name
O	– a given object's value
Os	– objects having a common given value
NO	– a given name and corresponding named–object's value

2.4.5.1 New PSET functions that are not predicates

This section consist of both total and partial functions. The signature that is given for any partial function may specify a superset of the domain. We have presented the definition of the partial functions in a tabular format. The right column header specifies a predicate that characterises the domain of the function when the domain is smaller than the superset.

del_N: **CSET** $\times$ ***name*** $\rightarrow$ **CSET**

F1	N_fnd(S,n)
del_N(S,n) =	S \– (n,S[n])

del_NO: **CSET** $\times$ ***name*** $\times$ ***value*** $\rightarrow$ **CSET**

F2	NO_fnd(S,n,o)
del_NO(S,n,o) =	S \– (n,o)

del_Os: **CSET** $\times$ ***value*** $\rightarrow$ **CSET**

F3	O_fnd(S,o)
del_Os(S,o) =	S – {(n,q) | (n,q)∈S ∧ o=q}

add: **CSET** $\times$ ***type*** $\times$ ***name*** $\times$ ***type*** $\times$ ***value*** $\rightarrow$ **CSET**

F4	¬ N_fnd(S,n) ∧ Tchk(t1,n) ∧ Tchk(t2,o)
add(S,t1,n,t2,o) =	S \+ (n,o)

sub_N: **CSET** $\times$ ***type*** $\times$ ***name*** $\times$ ***name*** $\rightarrow$ **CSET**

F5	N_fnd(n1) ∧ ¬ N_fnd(n2) ∧ Cvt(t,n1,n2)
sub_N(S,t,n1,n2) =	S \– (n1,S[n1]) \+ (n2,S[n1])

sub_NO: **PSET** × ***name*** × ***value*** × ***type*** × ***value*** → **PSET**

F6	NO_fnd(S,n,o1) ∧ Tchk(t,o2)
sub_NO(S,n,o1,t,o2) =	S \– (n,o1) \+ (n,o2)

sub_O: **PSET** × ***name*** × ***type*** × ***value*** → **PSET**

F7	N_fnd(S,n) ∧ Tchk(t,o)
sub_O(S,n,t,o) =	S \– (n,S[n]) \+ (n,o)

sub_Os: **PSET** × ***value*** × ***type*** × ***value*** → **PSET**

F8	O_fnd(S,o1) ∧ Tchk(t,o2)
sub_Os(S,o1,t,o2) =	S – {(n,q) | (n,q)∈S ∧ o1=q} ∪ {(n,q) | (n,S[n])∈S ∧ o2=q}

2.4.5.1.1 Predefined auxiliary functions for ARRAY operations

Decr: **ARRAY** × ***integer*** → **NSET**

Decr(S,n) $\stackrel{df}{=}$ {(i,S[i+1]) | (1 ≤ n ≤ i<CARD(S))}

Incr: **ARRAY** × ***integer*** → **NSET**

Incr(S,n) $\stackrel{df}{=}$ {(i+1,S[i]) | (1 ≤ n ≤ i ≤ CARD(S))}

Daft: **ARRAY** × ***integer*** → **NSET**

Daft(S,n) $\stackrel{df}{=}$ {(i,S[i]) | (1 ≤ n ≤ i ≤ CARD(S))}

Dbef: **ARRAY** × ***integer*** → **ARRAY**

$Dbef(S,n) \overset{df}{=} S - Daft(S,n)$

Indx: **NSET** × ***natural*** → ***integer***

	i ≤ CARD(S)
Indx(S,i) =	j | (j ∈ Dom(S)) ∧ (CARD({(k,e) | (((k,e) ∈ S) ∧ (k ≤ j))}) = i)

Reseq: **NSET** → **ARRAY**

$Reseq(S) \overset{df}{=} \{(i,o) \mid o = S[Indx(S,i)] \wedge i \leq CARD(S)\}$

2.4.5.1.2 Operations on ARRAYs

delA_N: **ARRAY** × ***integer*** → **ARRAY**

FA1	N_fnd(S,n)
delA_N(S,n) =	Dbef(S,n) ∪ Decr(S,n)

delAT[13]: **ARRAY** → **ARRAY**

FA1.1	CARD(S) > 0
delAT(S) =	Dbef(S,Card(S))

delA_NO: **ARRAY** × ***integer*** × ***value*** → **ARRAY**

FA2	NO_fnd(S,n,o)
delA_NO(S,n,o) =	delA_N(S,n)

13. delAT(S) is equivalent to delA(S,CARD(S)). In the definition of delAT, S – Daft(S,CARD(S)) is the same as S – (CARD(S),S[CARD(S)]).

delA_Os: **ARRAY** × ***value***→ **ARRAY**

FA3	O_fnd(S,o)
delA_Os(S,o) =	Reseq(del_Os(S,o))

addA: **ARRAY** × ***integer*** × ***type*** × ***value***→ **ARRAY**

FA4	¬ N_fnd(S,n) ∧ Tchk(t,n)
addA(S,n,t,o) =	Dbef(S,n) \+ (n,o) ∪ Incr(S,n)

addAT: **ARRAY** × ***type*** × ***value*** → **ARRAY**

FA4.1	¬ N_fnd(S,n) ∧ Tchk(t,o)
addAT(S,t,o) =	S \+ (CARD(S)+1,o)

2.4.5.2 Reducing the number of arguments for "del", "add" and "sub"

For the new PSET functions that are not predicates, we will adopt a simple convention that was used by some earlier computer system calls (e.g. DEC VAX/VMS). To avoid redundant writing, ITAM specification writers can leave out some predefined auxiliary arguments. For example if a Tchk(b,c) appeared in a table above an addAT(a,b,c) function, then the latter can be written as addAT(a,,c). The perception is that the "Tchk" predicate in the conditions of addAT, is ignored. See further discussions of example 1c in section 5.2.1.

2.4.6 Other predefined functions

Mini(S): **NSET** → ***integer***

Mini(S) $\stackrel{df}{=}$ j | {(i,o) | (i,o)∈S ∧ j≤i) }

Maxi(S): **NSET** $\rightarrow$ ***integer***

Maxi(S) $\stackrel{df}{=}$ j | {(i,o) | (i,o)$\in$S $\wedge$ j$\geq$i) }

Minv(S): **PSET** $\rightarrow$ ***real***

Minv(S) $\stackrel{df}{=}$ v | {(i,o) | (i,o)$\in$S $\wedge$ o$\in$***real*** $\wedge$ v$\leq$o) }

Maxv(S): **PSET** $\rightarrow$ ***real***

Maxv(S) $\stackrel{df}{=}$ v | {(i,o) | (i,o)$\in$S $\wedge$ o$\in$***real*** $\wedge$ v$\geq$o) }

2.4.7 Predefined relations which are not functions

Any_Ls: SET $\times$ integer $\rightarrow$ SET

	CARD(S)>0
ANY_LS(S,n) =	S1 | Ls(S,S1,n)

All_Ls: SET $\times$ integer $\rightarrow$ SET

	CARD(S)>0
ALL_LS(S,n) =	S1 | {X | Ls(S,X,n)} = S1

CHAPTER 3: A SIMPLE GRAMMAR FOR CANONICAL REPs.

3.1 Introduction to the grammar for describing the Canonical Representation

The grammar is written using BNF notation, (see [24]). This grammar is based on set theory. Section 2.4 includes some of the syntax of relations and operations; standard and non–standard notation, used in constructing this grammar.

We use this grammar in describing canonical representations, for a finite state machine called, ”rep”. The grammar presented for ”rep”, will have a constraint which will include its bounds. In constructing the grammar for constraints on ”rep”, we follow ideas presented for predicates, in [16]. The latter paper demonstrates ways of rewriting mathematical expressions without introducing more notation. Such creates redundancy and may complicate reading. However, to reduce writing we did introduced some redundancy into the grammar, as will be seen with some notational conveniences, (e.g. existential quantifier and implication were introduced).

The grammar is based mainly on those sets that we found useful during experimentation, (refer to section 2.3 and Appendices A, B). We foresee no difficulty in modifying our grammar to accommodate any additional sets which may be required in the future. Such additional sets includes the others of Fig 2. not presently used, or any extensions to that figure.

In describing the canonical representation we use set notation of section 2.4. This grammar is designed to guide the writer in describing the canonical representation in small steps. For instance, the grammar allows the writer to present set descriptors in a gradual synthesis which leads to the single set descriptor for declaring “rep”. This makes it easier for readers (e.g. programmers) to understand canonical representations.

3.2 Trace and set syntax for a canonical representation

The following examples show how the trace of a canonical representation can be represented using ITAM’s set structures.

3.2.1 An example using traces

This following example is taken from [10]. It is the canonical representation of the module titled, ”UNBOUNDED PRIORITY INTEGER QUEUE”.

$$\text{canonical (T)} \Longleftrightarrow \left(T = \left[(x_{i-1}).\text{INSERT}(p_i, x_i) \right]_{i=1}^{n} \right) \wedge ((x_0)=(\%\text{empty}\%)) \wedge$$
$$(\forall T1,S1,p,p1,x,x1)((T=T1.\text{INSERT}(p,x).(x).\text{INSERT}(p1,x1).S1) \rightarrow$$
$$(p<p1) \wedge (p=p1) \wedge (x \le x1))$$

3.2.2 The revised example, bounded and using set representation

In this revised version, we define a local auxiliary function ”Numele”. Numele is used to describe the module’s bounds. The original module has no bounds [10].

$$\text{Numele}(x) \overset{df}{=} \sum_{y \in \text{RAN}(x)} \sum_{z \in \text{RAN}(y)} z$$

The following is our description of the canonical representation :–

<u>Type definition</u>
type t1 = NSET integer
type t2 = NSET t1

type t3 = SNE ∪ integer
type S = REC (”oval”, :t3) , (”data”, :t2)

Representation declaration
rep : S | ((CARD(rep[”data”]) ≤ CAPQ) ∧ (Numikey(RAN(rep[”data”])) ≤ CAPIK))

Initial representation
rep = {(”oval”, { }), (”data”,{ })}

3.2.2.1 Revised example using our notation for syntax testing purposes

Type definition
type t1 = NSET integer
type t2 = NSET t1
type t3 = SNE !!U integer
type S = REC (‘oval‘,:t3) , (‘data‘,:t2)
Representation
rep: S | (Numele(rep)<=CAPI)
Initial rep
rep = {(‘oval‘, { }), (‘data‘,{ })}

3.3 Setting precedence

Generally, operator precedence on expressions can eliminate the use of curved brackets (parentheses), [16].

In interpreting the meaning for a **UDT** set description here, the precedence between the following 3 set operators, will be in the assumed order: ∩ (intersection), ∪ (union), – (set difference). We do not use curved brackets when describing a new **UDT**. The distributive law must be applied to create a single set descriptor's definition, from other defined sets, [17]. In case this is too cumbersome or bulky, the writer has the option of predefining some set descriptors and substitute these instead, into the definition for simplicity. By doing this the writer guides the reader to some precedence.

For example, in creating a **UDT** Ty, type Ty =T2 ∩ (T4 – T6) must be written in our grammar as type Ty= T2 ∩ T4 – T2 ∩ T6. The latter part gives the appearance of being bulky for a reader so another way is to predefine T4–T6. Let type Tx = T4 – T6 and then we can write type Ty = T2 ∩ Tx.

We constrain "rep" by using predicates. Our precedence for the predicate connectives is in the following order : ∧ (and), ∨ (or), → (implication).

When we have a set 'S' and an element 'e', using normal set notation we union 2 sets to do this, so here we write S ∪ {e}.

When we want to remove e from S we can write S – {e}. We have added the non–standard notational conveniences \+ and \– to add and remove single elements to a set, in doing this we do not have to enclose these elements in parentheses (to be a set,) before we operate. We have found that this is clear and straight forward at times, in defining **UDT**s.

The set descriptor of the set that is being operated on, to produce the newly defined set, must appear before the \+ and \– operations. For example, if we need to define the set descriptor T1, to be a set of numbers consisting of –1 and all the natural numbers except 1, we write the following:

type T1 = natural[14] \+ –1 \– 1

The operators \+ and \– have equal precedence. We follow the left to right appearance in a sentence when removing or adding an element to a set. The writer has the option of providing his/her own precedence by predefining some set descriptors and using them appropriately.

For the mathematical operations, the precedence is the norm, that is * (multiply), / (quotient), MOD (remainder), + (plus), – (minus). These operations are only for real

14. In an ITAM module specification we write 'natural' for the set descriptor ***natural***.

numbers. It is recommended that any bulkiness resulting from the use of these operations may be substituted with a locally defined auxiliary function.

3.4 Convention

Characters found within ’< >’ are nonterminal symbols.
The strings ’:==’ and ’|’ are the meta–symbols.
The start symbol will be <DerCanRep>

An uppercase nonterminal is mapped to a terminal symbol. Terminal symbols are system dependent. In Appendix C we give the necessary nonterminals that will be required to write a description of a canonical representation and the corresponding terminal symbol for our particular system, (see details in Appendix C). A user can substitute his/her own corresponding terminal symbols to write the description of the canonical representation. Terminal symbols must be recognised by the tool being used for testing a presented grammar. For instance, on the system that we used to test the grammar, the symbol, ∀ was unavailable so instead we used A! .

3.4.1 Grammar for describing the Canonical Representation

<DerCanRep> :== <TypeDef> <CanRep> <NL> <InitRep>

<TypeDef> :== <TD> <NL> <defsornone>

<defsornone> :== <NONE> <NL> | <spectydefs>

<spectydefs> :==<tysentconstr> | <tysentconstr> <spectydefs>

<tysentconstr>:== <TYPE> <setdescreq> <setdescrdef> <NL>

<setdescreq> :== <setdescr> <EQUAL>

<setdescrdef> :== <SETdef> | <SUBPdef> | <SSETdef> | <NSETdef> | <ARRAYdef>
| <RECdef> | <tupdef> | <setdifdef> | <undef> | <indef>

```
                    | <setremadd>[15] | <unSNE>

<SETdef>      :==   <LPARA> <streles> <RPARA> | <LPARA> <realeles> <RPARA>

<SUBPdef>     :==   <SUBP> <subPEdescr>[16]

<SSETdef>     :==   <SSET> <csetdescr>

<NSETdef>     :==   <NSET> <csetdescr>

<ARRAYdef>    :==   <ARRAY> <csetdescr>

<RECdef>      :==   <REC> <RECprs>
<RECprs>      :==   <aRECpr> | <aRECpr> <COMMA> <RECprs>

<tupdef>      :==   <csetdescr> <CROSSPRO> <csetdescr>
                    | <csetdescr> <CROSSPRO> <tupdef>

<setdifdef>   :==   <setdifele> <SETDIFF> <setdifele>
                    | <setdifele> <SETDIFF> <setdifdef>

<setdifele>   :==   <undef> | <indef> | <csetdescr>

<undef>       :==   <unele> <UNION> <unele> | <unele> <UNION> <undef>

<unele>       :==   <indef> | <csetdescr>

<indef>       :==   <csetdescr> <INTERSECT> <csetdescr>
                    | <csetdescr> <INTERSECT>   <indef>

<setremadd>   :==   <csetdescr> <remadd>

<remadd>      :==   <addele> | <remele>

<addele>      :==   <ELEADD> <astrele> | <ELEADD> <areal>
                    | <ELEADD> <astrele> <remadd> | <ELEADD> <areal> <remadd>

<remele>      :==   <ELEMINUS> <astrele> | <ELEMINUS> <areal>
                    | <ELEMINUS> <astrele> <remadd>
                    | <ELEMINUS> <areal> <remadd>
```

15. A notational convenience for adding element/s to a set without using set union or difference.
16. For a SUBP structure (see section 2.3.1.1), <subPEdescr> is a set descriptor for a fixed set of **PE**s.

<unSNE> :== <SNE> <UNION> <csetdescr>

<aRECpr> :== <LBRAC> <astrele> <COMMA> <COLON> <csetdescr> <RBRAC>

<csetdescr>[17] :== <PEdesrc> | <setdescr>

<setdescr> :== <alphanum>

<subPEdescr> :== <alphanum> | <PEdescr>

<PEdescr> :== <REAL> | <INTEGER> | <NATURAL>
| <STRING> | <BOOLEAN>

<astrele> :== <QUOTE> <alphanum> <QUOTE>

<streles> :== <astrele> | <astrele> <COMMA> <streles>

<alphanum> :== <alpha> | <alpha> <an> | <alpha> <ban>
<an> :== <alpha> | <digit> | <alpha> <an> | <digit> <an>
| <alpha> <ban> | <digit> <ban>
<ban> :== <UNDERBAR> <an>

<alpha> :== <UALPHA> | <LALPHA>

<digit> :== <GT0DIGIT> | <ZERO>

<realeles> :== <areal> | <areal> <COMMA> <realeles>

<areal> :== <notint> | <MINUS> <notint> | <anint>

<notint> :== <GT0DIGIT> <repdigit> <PERIOD> <repdigit>
| <GT0DIGIT> <PERIOD> <repdigit>
| <ZERO> <PERIOD> <repdigit>

<repdigit> :== <digit> | <digit> <repdigit>

<anint> :== <posnum> | <MINUS> <posnum> | <ZERO>

<posnum> :== <GT0DIGIT> | <GT0DIGIT> <repdigit>

---

17. All set descriptors appearing on right hand side of a type definition must be defined (i.e. appear on the left hand side only once).

<CanRep>	:==	<CR> <NL> <CanRepdef>
<CanRepdef>	:==	<REP> <csetdescr>[18] <SUCHTHAT> <pred>
<pred>	:==	<primpred>
		| <LBRAC> <FORALL> <vars> <pred> <RBRAC>
		| <LBRAC> <pred> <RBRAC>
		| <pred> <AND> <pred>
		| <pred> <OR> <pred>
		| <pred> <implies> <pred>
		| <NOT> <pred>
<implies>	:==	<RIMPLIES> | <RLIMPLIES>
<primpred>	:==	<prefixpred> | <infixpred>
<prefixpred>[19]	:==	<func1>
<func1>[20]	:==	<alphanum> <LBRAC> <args> <RBRAC>
<args>	:==	<expr> | <expr> <COMMA> <args>
<expr>		| <constant> | <var> | <parameter> | <PEdescr>
		| <objref> | <func1> | <func2> | <func3>
		| <RBRAC> <expr> <LBRAC>
<infixpred>	:==	<setexpr> | <mathexpr> | <strexpr>
<setexpr>	:==	<lftsetarg> <setboolop> <rgtsetarg>
<lftsetarg>[21]	:==	<constant> | <var> | <parameter> | <objref>
		| <func1> | <func2> | <func3> | <func4>
<rgtsetarg>[22]	:==	<var> | <PEdescr> | <objref> | <func2> | <func4>
<setboolop>	:==	<SETNEQUAL> | <SETEQUAL> | <MEM>
		| <NONMEM> | <SUB> | <PROPSUB>

18. This must be a set descriptor that is a **PT** or one defined in the type definition section.
19. Written as a predicate.
20. A predicate or a non–predicate function.
21. A set's element or reference to a set's element.
22. A set or reference to a set.

```
<mathexpr>     :==  <realarg> <realboolop> <realarg>
                    | <realarg> <realboolop> <mathexpr>
<realarg>23    :==  <areal> | <var> | <parameter> | <objref>
                    | <func1> | <func3> | <func4>
<realboolop>   :==  <NEQUAL> | <EQUAL> | <GE> | <GT> | <LT> | <LE>

<strexpr>      :==  <strarg> <strboolop> <strarg>
<strarg>24     :==  <astrele> | <var> | <objref> | <func1>
<strboolop>    :==  <STRNEQUAL> | <STREQUAL>

<constant>     :==  <astrele> | <areal>
<var>          :==  <alphanum>
<vars>         :==  <alphanum> <COMMA> | <alphanum> <COMMA> <vars>

<parameter>    :==  <Ualphanum>
<Ualphanum>    :==  <UALPHA> | <UALPHA> <Uan> | <UALPHA> <bUan>
<Uan>          :==  <UALPHA> | <digit> | <UALPHA> <Uan> | <digit Uan>
                    | <UALPHA> <bUan> | <digit> <bUan>
<bUan>         :==  <UNDERBAR> <Uan>

<objref>       :==  <REPREF> | <REPREF> <objindx>
<objindx>      :==  <LSQBRAC> <index> <RSQBRAC>
                    | <LSQBRAC> <index> <RSQBRAC> <objindx>
<index>        :==  <astrele> | <posnum> | <var> | <parameter> | <objref>
                    | <func1> | <func4>

<func2>        :==  <DOM> <LBRAC> <objref> <RBRAC>
                    | <RAN> <LBRAC> <objref> <RBRAC>

<func3>25      :==  <CARD> <LBRAC> <func3arg> <RBRAC>
<func3arg>     :==  <var> | <func2> | <objref>

<func4>        :==  <total> | <LBRAC> <total> <RBRAC>
<total>        :==  <mathval> <rops> <mathval> | <mathval> <rops> <total>
<rops>         :==  <TIMES> | <DIV> | <MOD> | <PLUS> | <MINUS>
<mathval>      :==  <areal> | <var> | <parameter> | <objref> | <func1> | <func3>
```

---

23. A real or reference to a real.
24. A string or a reference to a string.
25. A real.

<InitRep> :== <IR> <NL> <IRdescr>

<IRdescr> :== <REPREF> <EQUAL> <IRdescr1_2>
| <REPREF> <SUCHTHAT> <pred>

IRdescr1_2 :== <IRdescr1> | <IRdescr2>

<IRdescr1> :== <LPARA> <IRpairs> <RPARA>

<IRdescr2> :== <LPARA> <RPARA>

<IRpairs> :== <apair> | <apair> <COMMA> <IRpairs>
<apair> :== <LBRAC> <name> <COMMA> <obj> <RBRAC>
<name> :== <posnum> | <astrele>
<obj> :== <IRdescr1_2> | <constant>

3.4.2 Grammar tool's terminal symbols

Uppercase Non-terminals	Explanations	Our terminal symbols	Used in:
RPARA	right parenthesis	}	1,3
LPARA	left parenthesis	{	1,3
SUBP	refer to sect 2.3.1.1	SUBP	1
REC	refer to sect. 2.3.1.2	REC	1
SSET	refer to sect. 2.3.1.2	SSET	1
NSET	refer to sect. 2.3.1.2	NSET	1
ARRAY	refer to sect. 2.3.1.2	ARRAY	1
UALPHA	uppercase letters	[A–Z]	1,2,3
LALPHA	lowercase letters	[a–z]	1,2,3
GT0DIGIT	numbers excluding zero	[1–9]	1,2,3
ZERO	zero	0	1,2,3
UNDERBAR	underbar	_	1,2,3
TD	type definition title	Type definition	1

NL	new line character, invisible	\n, keyboard’s carriage return	1,2,3
TYPE	start of a type definition	type	1
NONE	no type definitions	None	1
REAL	set descriptor, real	real	1,2,3
INTEGER	set descriptor, integer	integer	1,2,3
NATURAL	set descriptor,	natural	1,2,3
STRING	set descriptor, string	string	1,2,3
BOOLEAN	set descriptor, boolean	boolean	1,2,3
SNE	refer to sect. 2.3.1	SNE or sne	1
LSQBRAC	left square bracket	[	2,3
RSQBRAC	right square bracket	]	2,3
LBRAC	left curve bracket	(	1,2,3
RBRAC	right curve bracket	)	1,2,3
COMMA	comma	,	1,2,3
COLON	colon	:	1
QUOTE	double quote	‘	1,2,3
PERIOD	period	.	1,2,3
CROSSPRO	cross product	#	1
PLUS	plus	+	2,3
MINUS	minus	_	2,3
MOD	mod	MOD	2,3
TIMES	multiply	*	2,3
DIV	divide	/	2,3
INTERSECT	set intersection	!!N	1
UNION	set union	!!U	1
SETDIFF	set difference	!!–	1
ELEMINUS	remove an element	\\–, keyboard’s \–	1

ELEADD	add an element	\\+, keyboard’s \+	1
CR	canonical representation title	Representation	2
SUCHTHAT	such that	|	2,3
REP	rep is defined to be	rep:	2
REPREF	the object rep	rep	2,3
GT	greater than	>	2,3
GE	greater than or equal	>=	2,3
LT	less than	<	2,3
LE	less than or equal	<=	2,3
NEQUAL	not equal	!=	2,3
EQUAL	equal or assign	=	1,2,3
SETEQUAL	set equality	SET=	2,3
SETNEQUAL	set no–equality	!SET=	2,3
STREQUAL	string equality	STR=	2,3
STRNEQUAL	string non–equality	!STR=	2,3
NOT	not	!	2,3
PROPSUB	proper subset	PSUB	2,3
SUB	subset	SUB	2,3
AND	and	/\\, keyboard’s / and \	2,3
OR	or	\\/, keyboard’s \ and /	2,3
NONMEM	non–membership	!MEM	2,3
MEM	membership	MEM	2,3
CARD	cardinality	CARD	2,3
DOM	set domain	DOM	2,3
RAN	set range	RAN	2,3
EXIST	there exist	E!	2,3
FORALL	for all	A!	2,3
RIMPLIES	right implies	—>	2,3

RLIMPLIES	implies both ways	<—>	2,3
IR	initial representation	Initial rep	3

Table 21. Terminal Symbols[26]

3.5 Grammar testing tool for the canonical representation description section

In describing canonical representations, subtle accommodations must be made when using the testing tool, refer to Appendix C.

26. The numbers in the Used in: column represents:–
    1– The Type definition section
    2– The Representation declaration section
    3– The Initial representation section

CHAPTER 4: A FORMAT FOR ITAM DOCUMENTS

4.1 The four sections of the ITAM module specification

There are 4 main sections to the ITAM module specification document :–

(a) The Header section

(b) The Canonical Representation section

(c) The Syntax section

(d) The Operation Table section.

The subsections of these sections are explained below. Not all subsections are compulsory and only those subsections that are needed must appear in a document.

4.2 The general format

The ideas for the following format grew mostly out of that presented in, [10].

4.2.1 Notable format changes from older TAMs

- The Canonical Representation does not use access program names.
- Syntax section appears after Canonical representation section.
- All auxiliary function tables and access program tables are presented as *program function tables*, [25].
- Table Ids may be found in the upper left corner of auxiliary functions and/or operation tables.
- The return values are now included in their appropriate operation table.

- The return values can be a tuple of values and each value is properly referenced by subscripting, see 2.4.3, Table 18. and examples of Appendices A and B.
- Comments can be made within the following notation, (as done in the programming language PASCAL):
/* *comments* */

In Chapter 5, a discussion of examples of Appendices A and B will add clarity since they follow the format presented in this chapter, for specifying a module interface.

4.2.2 Skeleton of format

The italicised phrases in the format's outline, explains the necessary entries at the point of their respective occurrence. Phrases/words in standard font will appear exactly as they are, in a specification document. Such phrase/words will be refer to as a *reserved* phrase/word.

The format's skeleton is as follow :–

(0) HEADER SECTION

Type Implemented
module's type or title (*module parameters*)

Module parameter(*optional*)
a module parameter : *type*, ..., *a module parameter* : *type*

External type(*optional*)
an external module type (*parameter*, ... , *parameter*) , ... , *an external module type* (*parameter*, ..., *parameter*)

(1) CANONICAL REPRESENTATION SECTION

Auxiliary Functions(*optional*)
Signatures and definitions of auxiliary functions to be used below

Description of Canonical Representation
Type definition
type definitions of non–predefined set descriptors

Representation declaration

rep : *type of canonical representation and constraints*

Initial representation

rep = *initial representation of* rep

or rep | *constraints on* rep

(2) SYNTAX SECTION

Access Programs

ACCESS PROGRAMS

Program Name	Arg1	∴	∴	Arg*n*	Value
first access program name	Arg1 *type, if any*	∴	∴	Arg*n type, if any*	*return value type, if any*
⋮	⋮	⋰	⋰	⋮	⋮
⋮	⋮	⋰	⋰	⋮	⋮
last access program name	Arg1 *type, if any*	∴	∴	Arg*n type, if any*	*return value type, if any*

(3) OPERATION TABLES SECTION

Abbreviations

Abbreviations	Expressions	Table Id.
first abbreviation	*abbreviation's expression*	*function or operation table identifier*
⋮	⋮	⋮
⋮	⋮	⋮
last abbreviation	*abbreviation's expression*	*function or operation table identifier*

Auxiliary Functions

Signatures and definitions of auxiliary functions to be used below

Operation Tables

access program name (*an implicit argument* [rep] *or* [rep,] *other non–implicit arguments*):

Operation table id.	*Condition 1*	...	*Condition n*
Either object' =	*new state*		*new state*
or (subscripted, argumented) access program name =	*return value*	...	*return value*
or (subscripted, argumented) access program name |	*predicate*		*predicate*
⋮	⋮	⋰	⋮
⋮	⋮	⋰	⋮
extension class =	%*status*%	...	%*status*%

4.2.3 Systematic format rules and other notable format changes

The following discussion is additional information to the ITAM format, given above in (section 4.2.1).

4.2.3.1 Header Section

This is explicit in the format above.

- A module's type is its title, (section 2.1.1).
- Module parameter and External type are optional in this section.

4.2.3.2 Canonical Representation Section

This section is divided into 2 subsections, (described by (a) and (b) below):–

(a) An optional section for locally defined Auxiliary Functions.
• These functions can be used from this point onwards, in the document.
• The signature of every auxiliary function will be stated.
• Definitions can be in a linear or in a tabular format.
• An example of the tabular format for defining a function is shown below.

Function table id.	*Condition*	...	*Condition*
<u>*Either*</u> *(subscripted) function name* = ——— <u>*or*</u> *(subscripted) function name* |	*value* ——— *predicate*	...	*value* ——— *predicate*
⋰	⋰	⋰	⋰

Table 22. Tabular format for a function definition

• Only functions in a tabular format can have a Table Id.
• Table Ids are generally required when abbreviations are used in defining the function.
• A subscripted function name is a variable, implying that the function is returning a tuple of values, each value is identified by the subscript, see section 2.4.3, Table 18.
• <u>*true*</u> can be used as a predicate, see [16].
• Either or both of the rows of Table 22. may be found in an auxiliary function table.

(b) A compulsory section for describing the canonical representation.
• This subsection consists of the 3 sections shown above, (section 4.2.2).
• The types *value*, *real*, *integer*, *natural*, *string*, *names*, *boolean*, are predefined, see section 2.1.1.
• The reserved set descriptor, SNE is predefined, see section 2.3.1, Table 3.
• The syntax details is given in Chapters 3, (examples are discussed in 5).

4.2.3.3 Syntax Section

The table may have the 3 headings as shown.

• All arguments will be input.
• All return values will be output.
• No suffices are required after argument types as in the old TAM (e.g. :I :O :V).
• The types *value*, *real*, *integer*, *natural*, *string*, *names*, *boolean*, are predefined, see section 2.1.1.

4.2.3.4 Operation Tables Section

This section may have 3 subsections, (describe by (a), (b) and (c)) :–

(a) An optional subsection for an Abbreviation Table.
• The first 2 columns are compulsory.
• For ease in reading the examples, we preferred to underline and capitalise our abbreviations.
• The list of Table Ids in a given row of an abbreviation table,are there to inform a reader about those tables that use the corresponding abbreviation.
• Any abbreviation that is used in constructing a new abbreviation is refer to as an *intermediate* abbreviation
• Rows without Table Ids are definitely stating intermediate abbreviations.

(b) An optional subsection for locally defined Auxiliary Functions section, see 4.2.3.2(a) above.
• These functions are used only in this section.

(c) Operation tables:
• The first argument of any access program call is the *implicit argument*, rep which will be written as: [rep,] or [rep] , (see Chapter 5, discussing Example 1) .
• An *argumented* access program name is one that requires non–implicit arguments.
• When the return value of an access program is a tuple , the variable referencing each value will be a *subscripted* access program name, see the footnote for Table 18., section 2.4.3.
• A subscripted access program name (with or without arguments) implies that the return value is a tuple of values, each identified by a subscript or a subscript composed of subscripts.
• We treat SNE as a special set descriptor, (Table 3.).
• Similarly for the operation tables, as in the auxiliary function tables, not all rows shown above will be used. All tables will have at least one first row (state of an object) and the last row (extension class).

- *true* can be used as a predicate, see [16].
- The notation in Table 23. , below will be used to specify an object's state.

Notation	Meaning
NC	The Not Change predicate, refer to [14]
'X	State of an object X before an operation, see [14]
X'	State of an object X after an operation, see [14]
NA	(Not Applicable) Given :– specific conditions, C; an object A, the state of A is written as NA when the object A is not defined under conditions C.
CA	(Change Accordingly) Given :– an operation O; specific conditions, C; an object A; an object B; where A is a member of B, the state change of B is written as CA when O causes a state change only to A and no other objects of B under conditions C.

Table 23. Other operation tables *object state* entries

CHAPTER 5: DISCUSSION OF EXAMPLES

5.1 Introduction to examples of Appendices A and B

The older figures of Parnas and Wang are re–presented in Appendix A and are the Examples 1 to 8. The older Figure 4, is now a version of Example 1 since it is based on a stack.

We have also added some newer examples to demonstrate some other issues, (to be discussed below). Newer examples are in Appendix B. The purpose of this chapter is to demonstrate and discuss items that show our improvements. Some of the same improvements occur more than once in the examples of Appendices A and B, but we discuss these improvements only once. Discussion on cosmetic improvements are also included.

All examples demonstrate using ITAM's format, presented in Chapter 4. The fundamentals of describing the Canonical Representation is based on set theory. Any designer using this newer method should follow the BNF grammar presented in Chapter 3.

In discussing the Operation Tables in this chapter, we introduce the following tags for parts of the table using the italicised phrases given in the Operation Table skeleton, section 4.2.2:–

- C – *access program name* (*an implicit argument* [rep] *or*
[rep,] *other non–implicit arguments*)
- H1 – *Operation table id.* , *Condition*

- H2 – *object* '=, *(subscripted, argumented) access program name* = or
  (subscripted, argumented) access program name | or
  (subscripted, argumented) access program name =,
  extension class =
- G – *new state*, *return value*, *predicate*, %*status*%

Predefined auxiliary relations and operations are useful in writing the examples in the Appendices A and B, (see section 2.4). Once these relations and operations are understood it is more efficient for both readers and writers, to use them in ITAM specifications.

In discussing each example, we briefly introduce the main improvements. Then we discuss all improvements that we think we must bring to the readers attention. We discuss the items of a specification from the top downwards. Occasionally this was not adhered to since comparisons among different versions of the same example were compulsory.

We use the terminology *primary object* in this section. A primary object is an object implemented by a module, the object is itself not part of the composition of another object. In an ITAM specification the primary object is, ”rep”.

5.2 Examples of Appendix A

We will present at least one version of all the older figures [10]. All older module specifications which cannot implement a finite state machine, will now have a version which is a finite state machine. Any object implemented by a module is a finite state machine and a finite state machine may be a set of other finite state machines.

5.2.1 Integer stack module examples

This example documents the standard example used in demonstrating a formal method.

In the header section of Example 1a Module parameter and External type are not used. Example 1a does not specify the implementation of a finite state machine.

The main features of this discussion, is to demonstrate the use of ITAM's format; an explanation of how the **TYPE** relation should be understood, (Chapter 2); using auxiliary predefined functions; showing an access program that is polymorphic. We also bring some other cosmetic issues of ITAM, to the readers attention.

5.2.1.1 Demonstrating the use of ARRAY

In the Type definitions we use ARRAY for describing the canonical representation of the module. The Array structure is a PSET used to represent a sequence

5.2.1.2 Argument, "rep"

PUSH([rep,]e) and POP([rep]) demonstrates the difference of the implicit argument mentioned in 4.2.4.3(c). POP does not have any other arguments unlike PUSH.
In trying to relate [rep,] or [rep] to the older TAM, this implicit argument must be viewed as the trace of an object before it is extended by an access program. However the trace is now reorganised and represented as a set that is a function, (PSET).

5.2.1.3 Returned values and Extension class

The return value TOP([rep]) is defined in a predicate, the predicate makes use of the notation _true_ and the extension class gives the status (see section 1.8). An exception handler can take care of this situation in some other document.

The return value is a simple 1–tuple, (section 2.1 and [16]).

The extension class for a condition (*status*) is the value which is a sequence of characters between a pair of "%".

5.2.1.4 Module parameter and External parameter

In 1b and 1c, Module parameter is used but not External type.

The implementation of 1b and 1c will be finite state machines. Using the Module parameter CAP to constrain the canonical representation ensures the finiteness of the object.

5.2.1.5 Considering the TYPE relation

For this example, in the canonical representation description section, S is a user defined set descriptor. ***type***$_{\text{Unbounded_stack}}$ is: {S, ***integer***, ***natural***}.

The set of indices for the ARRAY is a subset of ***natural*** and the set of indexed items are integers therefore ***value***$_{\text{Unbounded_stack}}$ contains values that are all natural numbers and all sets of integer pairs.

5.2.1.6 Polymorphic PUSHCHK

In 1c the argument type as seen in the syntax table, allows the access program PUSHCHK to handle any real. The objects' state is represented by compositions of integers. Determining state changes is dependent on non–integer and integer so the predefined predicate Tchk is used in PUSHCHK. This version of the stack module documents an access program that is polymorphic.

5.2.1.7 Predefined auxiliary functions

This module shows the use of predefined functions CARD, addAT, delN, delAT, Reseq.

5.2.1.7.1 Not providing predefine auxiliary function's arguments

The function addAT is defined with type checking but in this module type checking in addAT is irrelevant so the second argument was ignored, (see section 2.4.5.2).

On looking at the specification of the PUSHCHK operation table, one notices that the second argument ”integer” does not appear in addAT. The convention to leave out arguments was developed for redundant writing (section 2.4.5.2). In this operation table Tchk appear in the operation table header already, so there was no need to provide the ”integer” argument for addAT.

5.2.2 Set of stacks, module examples

This example documents a module with named objects, [10]. We use strings to name the objects.

Both versions of this example are modified to be more practical. For this reason new and modified access programs will be encountered (e.g. delete programs) and bounds provided.

In this example we do not use the notation for an empty trace (i.e. T=_), we explain this further below.

We use a simple n–tuple that is not a simple 1–tuple, to return values.

5.2.2.1 Where to define Auxiliary functions in ITAM's format

In 2b, we used the auxiliary function SUMSTKS instead of using the cumbersome mathematical notation that it defines. SUMSTKS was used in both the canonical representation section and the operation tables section. This is why it is defined in the Canonical Representation section and not in the Operation Tables section, (see section 4.2.2, **Auxiliary Functions** subsections).

5.2.2.2 SSET and cross products

In the Type definitions we demonstrate our use of cross products and SSET. We did not use a sequence here for our definition of the primary object.

In the syntax table ”booltup” is a **UDT** which defines a tuple (using cross product). Notice here that ”booltup” was not used in describing ”rep”, ”booltup” appears in the syntax table, (see section 5.2.8.1 on interchanging sections.)

5.2.2.3 Using the predefined auxiliary function N_fnd

We make use of another predefined auxiliary function N_fnd in most of the operation tables. Using N_fnd has helped us in eliminating the use of “T_n =_” where n is any name (see section 1.3.2). The older module specification in [10], was not a finite state machine, it was an unbounded module. We used both N_fnd and the module’s parameters in maintaining the bounds for each stack and the set of stacks, (CAPSUMSTKS, CAPSET respectively). The module specification with these bounds (Example 2b) implement finite state machines. We may refer to 2b as a finite set of a finite set of objects. The entire set is viewed as a single object, (*primary* object).

N_fnd also helps in making sure that no 2 objects have the same name. This ensures that the set remains a PSET.

5.2.2.4 A return value that is not a simple 1–tuple

STKSTATE in 2a shows one way of presenting return values of simple 2–tuple, (of ”booltup” type). STKSTATE of 2b is the equivalent, presented differently, both examples are shown in Fig 3. and Fig 4. below.

5.2.2.5 Notation: NC, NA, CA

The notation NC can be found in some of the tables.

CA and NA are found in PUSH and POP, their explanations are given in Table 23.

When an object A, is an element of another object B (mentioned in the explanation of CA in Table 23.), we recommend that the information about A's state appear in a row before B's. This creates a familiar consistency in the mind of the reader.

STKSTATE([rep,]n) :

	N_fnd('rep,n)		¬ N_fnd('rep,n)
	CARD('rep[n])=0	CARD('rep[n])>0	
rep'=	NC	NC	NC
STKSTATE([rep,]n)$_1$=	true	true	false
STKSTATE([rep,]n)$_2$=	true	false	false
extension class =	%emptystk%	%notemptystk%	%nofndstk%

Fig 3. Operation Table STKSTATE of Example 2a

STKSTATE([rep,]n) :

	N_fnd('rep,n)		¬ N_fnd('rep,n)
	CARD('rep[n])=0	CARD('rep[n])>0	
rep'=	NC	NC	NC
STKSTATE([rep,]n)=	(true,true)	(true,false)	(false,false)
extension class =	%emptystk%	%notemptystk%	%nofndstk%

Fig 4. Operation Table STKSTATE of Example 2b

5.2.2.6 Local versus predefined auxiliary function

The additional access program, PUTCHK demonstrates a short coming of the design of this operation table and how a locally defined auxiliary function can be clearer. Since addA was used with an argument i, that meant that there is a required redundant check N_fnd('rep[n],i) within addA. This cannot be avoided while keeping column 5. To avoid this redundancy and to save on writing in the design, an auxiliary function local to this

module can be defined as :

LocFun : **ARRAY** × ***integer*** × ***value***

LocFun(S,i,o) $\stackrel{df}{=}$ DBEF(S,i) + (i,o) ∪ INCR(S,i)

and a substitution of LocFun('rep[n],i,e) in place of addA('rep[n],i,,e) can be made.

The choice must be made by the specification writer as to whether an already learnt (predefined function) or a newly written local function that is almost similar, should be used.

5.2.3 Integer queue module examples

This example documents the queue module, [10]. Most of it's improvements are already covered in Example 1.

5.2.4 Bounded integer table list module example

In the older Figure 5, the table list was not bounded. In the design presented as Example 4, we ensure bounds, thus an implementation is a finite state machine. The main feature of this example is the introduction of Abbreviation tables in the Operation Tables Section, (see section 1.9). We briefly discuss the significance of providing both after–states and legality in our operation tables.

In this example it was not necessary to create a separate PARSE function as in the older Figure 5 in [10].

5.2.4.1 Demonstrating the use of REC

In the Type definitions, we demonstrate the use of REC. REC is used in the same sense as the typical records provided by some programming languages, (e.g. PASCAL).

5.2.4.2 Abbreviations

All abbreviations in this example, were used in tables therefore none were intermediate abbreviations, (see section 4.2.3.4 for *intermediate*). An intermediate

abbreviations is mentioned in a later example, sections 5.2.6.2. Our preference to have underlined capitals as abbreviations, must not be regarded as a standard convention. Further explanations on abbreviations, will be discussed below, in 5.2.7.6.

Abbreviations give an operation tables an image of being less cluttered when clutter cannot be otherwise avoided. Meaningful abbreviations are always a good hint for readers to understand their purpose.

5.2.4.3 Transition from a trace of program invocations to a set of objects

Our version of this example is straight forward, CARD(rep[”list”]) – rep[”ptr”]=n, where n is the variable found in the original canonical representation in Figure 5 of [10].

Below is a diagrammatic example of how a conversion from the old trace example to the new, was done. In the older canonical trace, n of GOLEFT was removed and a pointer (i.e. an object,) was put in place in order to reference elements in the sequence of insertions which is now represented by the ARRAY, rep[”list”]. In order to get the exact element before n, rep[”ptr”] must be incremented by 1.

RAN(rep[”list”])	a	b	c	d	e	
DOM(rep[”list”])	1	2	3	4	5	
rep[”ptr”]	0	1	2	3	4	5
n	5	4	3	2	1	0

Fig 5. Converting n in trace to ”ptr” in canonical representation

5.2.4.4 Applying the *New Rule* of section 2.4

In the old specification when G is the length of the insert trace, this means in the new specification that rep[”ptr”]=0, when this occurs the value of parse was ”false”, there are no equivalences on these traces. This is confusing to the reader since the reader may be wondering, “what is the state of the object when parse is false”? The answer to this may be left to an implementor’s choice and is not explicit. In the new documentation when this occurs the object’s state will remain unchanged (*New Rule* in section 2.4) and this condition now have several extension class status, satisfying ”false”.

5.2.4.5 Different status for different conditions

If we look at the new INSERT operation table, under different conditions we see that objects may have the same after–states, (e.g. columns 3 and 5). In these cases the status provided by the extension class reflects the difference. The status has replaced legality which we used in the older TAM, (see section 1.3.5).

5.2.5 Unique integer producer module examples

The older figure 6 featuring a non–deterministic specification, is rewritten as 5a. Since there no bounds on 5a, it will not implement a finite state machine. However the newer versions 5b and 5c, will implement a finite state machine.

This example does not use a delete or add access program. Eventually the system that maintains the history ensuring a production of unique integers, will reach CAP.

5.2.5.1 Demonstrating the use of SUBP

In the type definitions here, we have used SUBP, (see section 2.3.1.1) since unique object states were a requirement in the specification.

5.2.5.2 ITAM table organisation for return values in terms of after–state

When we document non–deterministic modules the independence that exists between rows of operation tables is lost. Fig 6. and Fig 7. (taken from 5b and 5c respectively) are equivalent, the difference is in the syntax. Fig 6. has the return value written in terms of the object's after–state whereas Fig 7. has the object's after–state written in terms of the return value. To be systematic, if an object's after–state is in terms of the return values, then we prefer to let the object's after–state row appear after the return value row. Also, we have adopted the convention of presenting the return values in terms of the after state as in Fig 6. Example 5b is recommended instead of 5c.

GETINT([rep]) :

	true	
	CARD(‘rep) < CAP	CARD(‘rep) = CAP
rep‘ =	‘rep + e | Tchk(integer,e) ∧ e ∉ ‘rep	‘rep
GETINT([rep]) |	GETINT([rep]) = rep‘–‘rep	<u>*true*</u>
extension class =	%successful%	%caprched%

Fig 6. Return value in terms of an object's after–state

GETINT([rep]) :

	true	
	CARD(‘rep) < CAP	CARD(‘rep) = CAP
GETINT([rep]) |	GETINT([rep]) = e | Tchk(integer,e) ∧ e ∉ ‘rep)	<u>*true*</u>
rep‘ =	‘rep + GETINT([rep])	‘rep
extension class =	%successful%	%caprched%

Fig 7. An object's after–state in terms of return value

5.2.6 Bounded priority integer queue module examples

The versions of this example mainly demonstrate, a more complicated module with non–deterministic properties; different methods of placing bounds on the module; reducing H1 size.

5.2.6.1 Syntax when using SNE, +, \+, –, \–

We show how we use SNE in the type definitions, (see section 2.3.1, Table 3.). The set description, SNE ∪ integer, could have been written as, integer \+ {}. To maintain simplicity, our grammar allows only one way, the former.

5.2.6.2 Intermediate abbreviation, function calls in abbreviations

The Abbreviation table of 6b and 6c shows the intermediate abbreviation, HIGHQ’. In 6b we show the use of a locally defined auxiliary function NUMELE within an abbreviation’s expression of ILIM. Predefined auxiliary functions, can appear in an abbreviation’s expression, (e.g. shown by MAXI in ’HIGHQ).

5.2.6.3 Some ways of placing bounds on this module

The original module was unbounded. All 3 versions presented in Example 6 are bounded. Our versions have demonstrated 2 ways in which this can be done.

It is a designer’s choice of how to place bounds on a module so that it implements a finite state machine. This module could have had bounds placed on it in either of several ways as given below. Below we give hints, (a) to (e) of possible ways of placing bounds on the module.

The following parameters are used in the expressions below:

MINQKEY – minimum queue–key
MAXQKEY – maximum queue–key

MAXIKEYPERQ – maximum item–key per queue
MAXSUMIKEY – maximum sum of item–keys
MAXPERIKEY – maximum entries per item–key
MAXSUMI – maximum sum of items for module

(a) The queue key bound, defined as
$(\forall x,\ x \in \mathrm{DOM}(rep[\text{"data"}]) \rightarrow \mathrm{MINQKEY} \le x \le \mathrm{MAXQKEY})$

(b) The maximum number of queues allowed, defined as
$\mathrm{CARD}(rep[\text{"data"}]) \le \mathrm{CAPQ}$

(c) The item key bounds, defined as
$(\forall x,\ x \in \mathrm{RAN}(rep[\text{"data"}]) \wedge (\forall y,\ y \in \mathrm{DOM}(rep[\text{"data"}][x]) \rightarrow \mathrm{MINIKEY} < y < \mathrm{MAXIKEY}))$

(d) Either of the following can be used :–

[i] Maximum number of item keys allowed per queue, defined as
$(\forall x,\ x \in \mathrm{RAN}(rep[\text{"data"}]) \rightarrow (\mathrm{CARD}(x) \le \mathrm{MAXIKEYPERQ})$ or

[ii] maximum number of total item keys allowed for set of queues, defined as

$$\left(\sum_{x \in \mathrm{RAN}(rep[''data''])} \mathrm{Card}(x) \right) \le \mathrm{MAXSUMIKEY}$$

(e) Either of the following can be used :–

[i] Maximum number of items allowed per key, defined as
$(\forall x,\ x \in \mathrm{RAN}(rep[\text{"data"}]) \wedge (\forall y,\ y \in \mathrm{RAN}(rep[\text{"data"}][x]) \rightarrow 1 \le y \le \mathrm{MAXPERIKEY})$ or

[ii] maximum number of items allowed per queue, defined as

$$(\forall x,\ x \in \mathrm{RAN}(rep[\text{"data"}]) \rightarrow 1 \le \left(\sum_{y \in \mathrm{RAN}(rep[''data''][x])} y \right) \le \mathrm{MAXIPERQ})\ \text{or}$$

[iii] maximum number of total items allowed for module, defined as

$$(1 \le (\sum_{y \in \mathrm{RAN}((rep[''data'']))} \sum_{z \in \mathrm{RAN}(y)} z) \le \mathrm{MAXSUMI}\ )$$

Examples 6a,b use a bound on items (i.e. (e) above) whereas 6c use a bound on the summation of the sets of itemkeys (i.e. a combination, (b) and (d)[ii]) .

5.2.6.4 Comments in a specification

Sometimes there may be a dire need to make comments in an ITAM specification. We have made use of a typical notation for commenting, (i.e. using /* and */) as shown after the Representation declaration of ”rep” in Example 6c, (see below).

Representation declaration
rep : S | CARD(rep[”data”]) ≤ CAPQ ∧ Numikey(RAN(rep[”data”])) ≤ CAPIK
/*Maximum number of queues allowed and Maximum number of total item keys allowed for set of queues*/

5.2.6.5 Demonstrating how abbreviations reduce table size

Example 6b,c shows the resulting simplicity of an operation table when abbreviating is done. Further abbreviating could have been done to simplify H1 into a single row. This would have been at the cost of having a larger abbreviation table. If the abbreviation tables of 6b and 6c are extended to include Fig 9. as shown below, the redone H1 (given as Fig 10.) can be used.

<table>
<tr><td rowspan="4">1</td><td rowspan="4">EMPTY</td><td colspan="7">¬ EMPTY</td></tr>
<tr><td colspan="4">¬ ILIM</td><td colspan="3">ILIM</td></tr>
<tr><td colspan="2">QMEM</td><td colspan="2">¬ QMEM</td><td colspan="2">QMEM</td><td rowspan="2">¬ QMEM</td></tr>
<tr><td>IKMEM</td><td>¬ IKMEM</td><td>LCNGQ</td><td>GCNGQ</td><td>IKMEM</td><td>¬ IKMEM</td></tr>
</table>

Fig 8. H1 of INSERT operation table of 6b and 6c

AB1	¬EMPTY ∧ ¬ILIM ∧ QMEM ∧ IKMEM	1
AB2	¬EMPTY ∧ ¬ILIM ∧ QMEM ∧ ¬IKMEM	1
AB3	¬EMPTY ∧ ¬ILIM ∧ ¬QMEM	1
AB4	¬EMPTY ∧ ¬ILIM ∧ ¬QMEM	1
AB5	¬EMPTY ∧ ILIM ∧ QMEM ∧ IKMEM	1
AB6	¬EMPTY ∧ ILIM ∧ QMEM ∧ ¬IKMEM	1
AB7	¬EMPTY ∧ ILIM ∧ ¬QMEM	1

Fig 9. A possible addition to the abbreviation tables of 6b and 6c

1	EMPTY	AB1	AB2	AB3	AB4	AB5	AB6	AB7

Fig 10. A single row H1 for the INSERT operation table

5.2.6.6 Syntax for indexing objects

Our method of identifying an object belonging to a set of objects which may themselves be sets, is a simple one, and obvious to programmers, (e.g. rep[”data”][p][v] in the INSERT operation table). Readers can easily understand this syntax than the actual mathematical expression it stands for.

5.2.6.7 Demonstrating the use of +, \+, –, \– in access programs

In this example the access program INSERT and REMOVE, demonstrate the use of the notation ’+’, ’\+’, ’\–’ and ’–’.

5.2.6.8 Non–deterministic property of this module

It is the REMOVE operation that gives this module it’s non–deterministic quality. The actual predicate shown in the column 4 below demonstrates this.

rep[”oval”]‘ |	NC	<u>*true*</u>	rep[”oval”]‘ ∈ DOM(rep[”data”][MAXI(DOM(rep[”data”]‘))]‘)	NC

Fig 11. The ”predicate of non–determinism”

5.2.7 Bounded multiple binary integer tree module examples

In the 2 versions of this example, we have now added a new access program DELTREE and we have replaced SETNIL and SETTREE with CREATETREE.

DELTREE was added to make a more practical design since both versions of this module is bounded.

Any tree or a tree’s branches can be operated on. Both versions of this example describes a set of trees. The objects created by this module use the access programs of another module, PATH (Example 8), [9,23]. Some operations on objects, depend on information obtained using objects created by the PATH module. This example shows how extension classes, assist in reducing some operation table’s width.

5.2.7.1 External Type

This module assumes the existence of Example 8’s objects. In the Header Section we have included the External type PATH with PATH’s 2 parameters. These parameters are obtained via the tree module.

5.2.7.2 An explicit set

We demonstrate the use of an explicit set (Table 4.), in defining the set descriptor, ”lr”.

type lr = {”l”,”r”}

5.2.7.3 A recurring syntax for dynamic structures

In this example we demonstrate our use of ”recurring” defined types as shown below.

type t1 = SNE $\cup$ tree

type tree = REC (”anint”, :integer), (”left”, :t1), (”right”, :t1)

The set descriptor ”tree” occurs in t1’s definition, and ”t1” occurs in tree’s definition. For those readers familiar with the programming language, Pascal, the equivalent semantics is written as:

type t1 = ↑ tree;

type tree = record anint : integer; left : t1; right : t1 end;

5.2.7.4 Tcard, Int, Gtr functions

All trees are bounded and the set of trees that will be implemented by this module, is also bounded thus ensuring a finite state machine.

We introduced a new auxiliary function Tcard to be used as a term in a predicate which checks on the bounds of this module. Tcard is used in some access programs of this specification. Tcard evaluates the cardinality of a tree.

We also introduced some other auxiliary functions that were not present in the older Figure 8, so as to assist in simplifying our design, (i.e. Int and Gtr).

5.2.7.5 Demonstrating how status reduce table size

We successfully reduced the width of operation table, CREATETREE of 8a, by combining all

extension class = %willrchct%

and produce CREATETREE of 8b. We produced one column with the H1 entry being WARNCT,, (see section 1.8).

We might have tried to simplify H1 of the operation table to be one dimensional (as suggested in 5.2.6 above). However the abbreviation table would be longer and the expressions would have been more intricate for readers to understand. Therefore we decided to leave the table this way.

WARNCT is an abbreviation, mostly made up of intermediate abbreviations and is a good example of how intricate an abbreviation's expression can be.

5.2.7.6 Abbreviations, cost of reducing operation tables and output tuples

By comparing both versions, it is obvious how much larger the Abbreviation tables will get when we try to simplify the operation tables. 8a's Abbreviation table is much smaller that 8b's.

We were able to reduce the specification size further when we combined the auxiliary functions and present the return value of a function as a 5–tuple, as shown in 8b.

The GETVAL abbreviation of Fig 12. demonstrates how tuples can be abbreviated, the underlined portion of the expression is itself an abbreviation but it also denotes an element of a tuple.

VP	VP(p,‘rep[r],d,b)	f2
GETVAL	$\underline{VP}_3$["anint"]	4

Fig 12. An abbreviated tuple's element

5.2.8 Bounded path holder module example

The access programs of Example 7 will use access programs of this module. The main features of this module are : the addition of a delete program; combining access programs in order to reduce the size of a specification document.

The newer version of the older Figure 9 (i.e. this example). We have added a DELPATH access program to make the design more practical.

5.2.8.1 Interchanging the Syntax and Canonical Representation sections

In the Syntax Section of a module specification, set descriptors of **UDT**s may also be used in the argument and value columns of the syntax table. Because of this reason, not all set descriptors found in the canonical representation description section, are used solely for describing Canonical Representations.

In this example we show how the **UDT**, ”lr” is used both in describing the canonical representation and as a subset of a type that is used in returning the value of PATHSTAT. As a convenience to readers, we put the SYNTAX SECTION after the CANONICAL REPRESENTATION SECTION in ITAM’s format. PATHSTAT was created as a replacement when we combined the access programs: EMPTYPATH, GOLEFT and GORIGHT.

5.3 Examples of Appendix B

All examples of Appendix B, are bounded.

5.3.1 A nondeterministic single object, room module example

We demonstrate the use of ***string*** elements, to document a simple but practical example. Each string uniquely represents an object, thus there is no naming of objects. Strings make more meaningful names than natural numbers, and increases legibility.

In access program TAKE, we demonstrate the use of the predefined relation ANYLS. The use of this relation, has given the module it’s non–deterministic property.

5.3.1.1 Explicit sets and their effect on parameters

This example shows the usefulness of a module’s parameters which are used as bounds. One should realize that in this module, a practical value for the Module parameter

must be a value less than 6. When the canonical representation is defined by an explicit set, and the value of the module parameter is equal to or larger than the cardinality of the explicit set, we can disregard the parameter's value.

5.3.1.2 Demonstrating set union syntax

This example demonstrates the syntax when we use set union. Only set descriptors appear in the syntax of the definition, when we describe the canonical representation. The same applies to set intersection and set difference.

5.3.2 A polymorphic, two–rooms module example

The main features of this example is to show how polymorphism and run–time type checking are documented.

All 4 access programs are polymorphic.

The arguments of PUTON and GETOPUTN are specified for only strings that are 'ct' or not 'ct' types. It is assumed that otherwise (i.e. not a string), must be taken care of elsewhere, most likely by a compiler.

However the access programs PUT and SGETOPUTN has taken it to the extreme. Both access programs allow any type of arguments (as specified in the syntax table). Because of this the interpretation of their requirements means that all type checking of their arguments must be done at run–time.

5.3.3 A non–deterministic polymorphic, two–rooms module example

This Example demonstrates that it is possible to combine many features, previously discussed in this chapter, without fear of clashes in the documentation process. This includes non–determinism and polymorphism. We also show how we handle an irrelevant row in an operation table.

5.3.3.1 Eliminating rows of an operation table

In the access program PUTO an object is composed of 2 other objects. An invocation of PUTO changes only one of the latter 2 objects so we only document the changed object and the object it belongs to.

In documenting Example 11, we eliminated the second row after H1 of Fig 13. below since there were no changes to the state of the object, rep[”new”]. Eliminating rows, reduces the size of an operation table.

PUTO([rep,]e) :

	e∉‘rep[”old”]		e∈‘rep[”old”]
	CARD(‘rep[”old”]) < CAPO	CARD(‘rep[”old”]) = CAPO	
rep[”old”]‘=	‘rep[”old”] \+ e	NC	NC
rep[”new”]‘=	NC	NC	NC
rep‘=	CA	NC	NC
extension class =	%successful%	%oldfull%	%dupobj%

Fig 13. Another representation of PUTO of Example 11

5.3.4 A bounded polymorphic multi–object, non–empty room module example

The example documents a module that has an initial state which is not “{}”.

The purpose of this example is to demonstrate the development of a module specification whose primary object’s initial state, is not given by the empty set.

CHAPTER 6: CONCLUSION AND FUTURE WORK

6.1 Conclusion

In our work we accomplished the following goals :–

(a) Allow specification method writers to represent the state of an object, using sets.

(b) View a collection of objects implemented by one module, as a single object composed of objects.

(c) Presents objects' state representation semantics and a syntax that is more in tune with programmers' perceptions. The syntax includes the recursive kind which defines dynamic data type, (e.g. linked lists in PASCAL).

(d) Provide a tool that checks the syntax describing the canonical representation of an objects state. (See Appendix C and Chapter 4 for details.)

(e) Provide commonly used predefined auxiliary functions to simplify mathematical notation.

(f) Provide predefined auxiliary functions to be used when it is necessary to ”check type” and ”check availability” of an object, before performing operations on an object.

(g) Provide a method of abbreviating, long and duplicate, mathematical expressions.

(h) Provide a better format that can be checked for completeness and consistency of access programs (*operation tables*) descriptions.

(i) Show that it is possible for ITAM to document polymorphic and non–deterministic module specifications.

(j) Provide a common, compact and systematic document format for ITAM's module interface specifications.

The examples presented in Appendices A and B are evident.

6.2 Future work

This work may be used to document some software modules. In retrospect we have shown that some examples when compared to those presented in [10], have turned out to be shorter in text volume while others have turned out to be the opposite. This may be due to our more practical considerations. For example: we wanted every implemented module's specification to be a finite state machine; occasionally when a user needed to access an object of a module, object availability checks were done. Such considerations increased the volume of text. Less details, produce smaller volume of text.

The proof of our success with these improvements, should not be assessed on volume of text but how acceptable our method is to a developer. One suggestion would be to take a survey.

This work is limited to the examples we investigated. Therefore we recommend more extensive use of our method, to document software modules in other areas of work (e.g. other engineering disciplines). We need to know what further improvements are necessary. This can only be achieved by putting ITAM to a practical test (e.g. using ITAM to document the TTS module [25]).

6.2.1 Possible extensions to this work in the context of reverse engineering

Many systems contain somewhat similar kinds of software work assignments, where one work assignment may be just like another but have a few extra programs. When reverse engineering is an issue, we will need to investigate and if necessary, provide solutions (an even newer ITAM) that yields systems with minimal redundancy in such cases.

Furthermore, we need to investigate and adjust ITAM to accomodate the inadequacies of some (badly) written programs. We need to consider the case of how we should treat global variables affected by programs of work assignments, where these work assignments produced systems which do not follow a proper ”uses” structure.

6.2.2 Automatically checking a module specification

It would be a worthwhile endeavour to guide a specification writer towards writing understandable module specifications. We would recommend that the module interface specification format presented in this thesis, be further investigated and further simplified for programmers.

A fully automated process can be developed to guide specification writers. Current existing TTS tools may be useful here, [25].

6.2.2.1 Format checking tools

For ITAM specifications we would like to suggest the need for basic format checking tools for the following areas :–

(a) the header section

(b) for auxiliary functions

(c) the syntax section

(d) the operation tables

(e) abbreviation tables

Some extra syntactic rules not covered in this work will be required. After this ground work is laid, then smarter additions can be made to the format checkers to make them more informative to specification writers.

6.2.2.2 Completeness and consistency checks

Automated completeness and consistency checks for the operation tables are always a welcome thought to designers. In the case where current tools may need the header of operation tables to be specified as single dimensional, further work may be needed.

6.2.3 Canonical Representation sets, Abbreviations, Aliases and Macros

In describing canonical representations, our examples only used the SP sets and PSETs that we provided. By using ITAM more extensively, we will be able to find out what new sets (if any) must be added to what we currently provide.

We have made use of abbreviations and macros (auxiliary functions) in some examples of Appendices A and B, (refer to section 5.2.6 and shown by Fig 10. for abbreviations). So far we did not use aliases[27]. However there may be a need as more examples with complicated mathematical expressions are encountered. This needs further investigation.

6.2.4 A trace simulator for deterministic and non–deterministic ITAM modules

Presently, there are trace simulators for deterministic modules based on the older TAMs, [27, 35, 38]. There is a need to construct one for ITAM, which may handle some non–determinism.

Both Norvell [28] and Janicki [30] have suggested the importance of a fundamental change in the domain of extension functions. They have demonstrated the importance of

---

27. Aliases are always names for a variable of a set of variables. They cannot be used for expressions that do not denote variables. Aliases are evaluated before each individual expression evaluation, not during expression evaluation as with functions, or before all evaluations as with abbreviations. Evaluating the alias identifies the variables that will be involved when the expression is evaluated. It is possible that every evaluation of an expression that contain aliases will use different variables.

Example: Suppose that P is an array that always contains a permutation of the numbers from 1 to 10. Suppose that FIVE is an alias for P[i] such that P[i] = 5. and SIX is an alias for P[i] such that P[i] = 6. FIVE plus SIX is always 11, no matter what permutation is stored in the array P.

considering the ”removed values” from an object’s state representation, in order to determine the present state of a module for some cases of non–deterministic behaviour.

In ITAM, we represent the state of an object by a set of values. New states of an object is given by a set operated on by a program invocation and possible input arguments. With ITAM’s current notation for describing a canonical representation (a set), it will be easy to introduce another new element which is an object into the primary object, ”rep”. Here we will refer to this object as, O. The purpose of O is to maintain the history of the values that caused a ”reduced” state change. For ITAM we say an object’s state change is reduced if the elements representing the before–state is more than that of the after–state.

Considering the drunken stack example in [30], we state that all before–states of the object were once an after–state (the initial state is debatable but not significant for this discussion). We let O maintain the history of only the values of the last POP invocation. We then use O to determine the exact after–state from a set of after states. This after–state becomes the new before–state, to be extended by another program invocation. Before a proper simulator can be constructed, further discussion of this topic is required.

Future work on ITAM should include a simulation on ITAM’s specification of a non–deterministic module. However there is difficulty in building a simulator to accomodate non–deterministic modules. Some non–deterministic modules can have a very large set of possible choices to test. Simulating a finite but very large set of possibilities is not an easy task. The effect can be somewhat reduced by simultating duplicate or equivalent states just once. A mechanism would have to be developed to identify such states. Another mechanism can also be developed to determine what possibilities are ”never used” then ignore the simulations on these.

APPENDIX A: OLDER EXAMPLES

Example 1a: Unbounded integer stack module. (Figure 1)

(0) HEADER SECTION

Type Implemented
Unbounded_stack

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition
type S = ARRAY integer

Representation declaration
rep : S

Initial representation
rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Value
PUSH	integer	
POP		
TOP		integer

3) OPERATION TABLES SECTION

Operation Tables

PUSH([rep,]e) :

	true
rep‘=	addAT(‘rep,,e)
extension class =	%successful%

POP([rep)] :

	CARD(‘rep) = 0	CARD(‘rep) > 0
rep‘ =	‘rep	delAT(‘rep)
extension class =	%empty%	%successful%

TOP([rep]) :

	CARD(‘rep) = 0	CARD(‘rep) > 0
rep‘ =	‘rep	‘rep
TOP([rep]) |	*true*	TOP([rep])=‘rep[CARD(‘rep)]
extension class =	%empty%	%successful%

Example 1b: Overflow integer stack module. (Figure 4)

(0) HEADER SECTION

Type Implemented

Overflow_stack(CAP)

Module parameter

CAP : Integer

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition

type S = ARRAY integer

Representation declaration

rep : S | (CARD(rep) $\leq$ CAP)

Initial representation

rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Value
PUSH	integer	
POP		
TOP		integer

3) OPERATION TABLES SECTION

Operation Tables

PUSH([rep,]e) :

	CARD(‘rep) < CAP	CARD(‘rep) = CAP
rep‘=	addAT(‘rep,,e)	addAT(Reseq(del_N(‘rep,1)),,e)
extension class =	%successful%	%full%

POP([rep]) :

	CARD(‘rep) = 0	CARD(‘rep) > 0
rep‘=	‘rep	delAT(‘rep)
extension class =	%empty%	%successful%

TOP([rep]) :

	CARD(‘rep) = 0	CARD(‘rep) > 0
rep‘ =	‘rep	‘rep
TOP([rep]) |	*true*	TOP([rep])=‘rep[CARD(‘rep)] mod 255
extension class =	%empty%	%successful%

Example 1c: Overflow integer stack module with push check.

(0) HEADER SECTION

Type Implemented

Overflow_stack_pushchk(CAP)

Module parameter

CAP : Integer

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition

type S = ARRAY :integer

Representation declaration

rep : S | (CARD(rep) $\leq$ CAP)

Initial representation

rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Value
PUSHCHK	real	
POP		
TOP		integer

3) OPERATION TABLES SECTION

Operation Tables

PUSHCHK([rep,]e) :

	CARD(‘rep) < CAP		CARD(‘rep) = CAP
	Tchk(integer,e)	¬ Tchk(integer,e)	
rep‘=	addAT(‘rep,,e)	‘rep	addAT(Reseq(del_N(‘rep,1)),,e)
extension class =	%successful%	%badtype%	%full%

POP([rep]) :

	CARD(‘rep) = 0	CARD(‘rep) > 0
rep‘=	‘rep	delAT(‘rep)
extension class =	%empty%	%successful%

TOP([rep]) :

	CARD(‘rep) = 0	CARD(‘rep) > 0
rep‘ =	‘rep	‘rep
TOP([rep]) |	*true*	TOP([rep])=‘rep[CARD(‘rep)] mod 255
extension class =	%empty%	%successful%

Example 2a: An unbounded set of unbounded stacks. (Figure 2)

(0) HEADER SECTION

Type Implemented
Unbounded_stackset

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition
type booltup = boolean $\times$ boolean
type Stk = ARRAY integer
type Grp = SSET Stk

Representation declaration
rep : Grp

Initial representation
rep = { }

(2) SYNTAX SECTION

Access programs

Program Name	Arg1	Arg2	Value
CREATESTK	string		
STKSTATE	string		booltup
DELASTK	string		
DELALLSTKS			
PUSH	string	integer	
POP	string		
TOP	string		integer
PUSHCHK	string	real	

(3) OPERATION TABLES SECTION

Operation Tables

CREATESTK([rep,]n) :

	N_fnd(‘rep,n)	¬ N_fnd(‘rep,n)
rep‘=	sub_O(‘rep,n,,{})	add(‘rep,,n,,{})
extension class =	%reinitstk%	%addstk%

STKSTATE([rep,]n) :

	N_fnd(‘rep,n)		¬ N_fnd(‘rep,n)
	CARD(‘rep[n])=0	CARD(‘rep[n])>0	
rep‘=	NC	NC	NC
STKSTATE([rep,]n)$_1$=	true	true	false
STKSTATE([rep,]n)$_2$=	true	false	false
extension class =	%emptystk%	%notemptystk%	%nofndstk%

DELASTK([rep,]n) :

	N_fnd(‘rep,n)	¬ N_fnd(‘rep,n)
rep‘=	del_N(‘rep,n)	‘rep
extension class =	%successful%	%nofndstk%

DELALLSTKS[rep] :

	true
rep‘=	{}
extension class =	%successful%

PUSH([rep,]n,e) :

	N_fnd(‘rep,n)	¬ N_fnd(‘rep,n)
rep[n]‘=	addAT(‘rep[n],,e)	NA
rep‘ =	CA or sub_O(‘rep,n,Stk,rep[n]‘)	‘rep
extension class =	%successful%	%nofndstk%

POP([rep,]n) :

	N_fnd(‘rep,n)		¬ N_fnd(‘rep,n)
	CARD(‘rep[n]) = 0	CARD(‘rep[n]) > 0	
rep[n]‘=	{ }	delAT(‘rep[n])	NA
rep‘ =	‘rep	CA	‘rep
extension class =	%emptystk%	%successful%	%nofndstk%

TOP([rep,]n) :

	N_fnd(‘rep,n)		¬ N_fnd(‘rep,n)
	CARD(‘rep[n]) = 0	CARD(‘rep[n]) > 0	
rep[n]‘=	{ }	‘rep[n]	NA
rep‘ =	‘rep	‘rep	‘rep
TOP([rep,]n) |	*true*	TOP([rep,]n) = ‘rep[n][CARD(‘rep[n])]	*true*
extension class =	%emptystk%	%successful%	%nofndstk%

PUSHCHK([rep,]n,e) :

	Tchk(string,n)			¬ Tchk(string,n)
	N_fnd(‘rep,n)		¬ N_fnd(‘rep,n)	
	Tchk(integer,e)	¬ Tchk(integer,e)		
rep[n]‘=	addAT(‘rep[n],,e)	‘rep[n]	NA	NA
rep‘ =	CA	‘rep	‘rep	‘rep
extension class =	%successful%	%badstkeletyp%	%nofndstk%	%badstktyp%

Example 2b: A bounded set of bounded stacks.

(0) HEADER SECTION

Type Implemented

Bounded_stackgrp(CAPSET, CAPSUMSTKS)

Module parameter

CAPSET : integer

CAPSUMSTKS : integer

(1) CANONICAL REPRESENTATION SECTION

Auxilliary Functions

SUMSTKS: Grp → integer

$$\mathrm{SUMSTKS}(x) \stackrel{df}{=} \sum_{y \in x} \mathrm{CARD}(y)$$

Description of Canonical Representation

Type definition

type booltup = boolean × boolean

type Stk = ARRAY integer

type Grp = SSET Stk

Representation declaration

rep : Grp | ((CARD(rep) $\leq$ CAPSET) $\wedge$ (SUMSTKS(rep) $\leq$ CAPSUMSTKS))

Initial representation

rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Arg2	Arg3	Value
CREATESTK	string			
STKSTATE	string			booltup
DELASTK	string			
DELALLSTKS				
PUSH	string	integer		
POP	string			
TOP	string			integer
PUSHCHK	string	real		
PUTCHK	string	integer	real	

(3) OPERATION TABLES SECTION

Operation Tables

CREATESTK([rep,]n) :

	N_fnd(‘rep,n)	¬ N_fnd(‘rep,n)	
		CARD(‘rep) < CAPSET	CARD(‘rep) = CAPSET
rep‘=	sub_O(‘rep,n,,{})	add(‘rep,,n,,{})	‘rep
extension class =	%reinitstk%	%addstk%	%fullset%

STKSTATE([rep,]n) :

	N_fnd(‘rep,n)		¬ N_fnd(‘rep,n)
	CARD(‘rep[n])=0	CARD(‘rep[n])>0	
rep‘=	NC	NC	NC
STKSTATE([rep,]n)=	(true,true)	(true,false)	(false,false)
extension class =	%emptystk%	%notemptystk%	%nofndstk%

DELASTK([rep,]n) :

	N_fnd(‘rep,n)	¬ N_fnd(‘rep,n)
rep‘=	del_N(‘rep,n)	‘rep
extension class =	%successful%	%nofndstk%

DELALLSTKS[rep] :

	true
rep‘=	{ }
extension class =	%successful%

PUSH([rep,]n,e) :

	N_fnd(‘rep,n)		¬ N_fnd(‘rep,n)
	SUMSTKS(‘rep)=CAPSUMSTKS	SUMSTKS(‘rep)<CAPSUMSTKS	
rep[n]‘=	‘rep[n]	addAT(‘rep[n],,e)	NA
rep‘ =	‘rep	CA	‘rep
extension class =	%stkscaprched%	%successful%	%nofndstk%

POP([rep,]n) :

	N_fnd(‘rep,n)		¬ N_fnd(‘rep,n)
	CARD(‘rep[n]) = 0	CARD(‘rep[n]) > 0	
rep[n]‘=	{}	delAT(‘rep[n])	NA
rep‘ =	‘rep	CA	‘rep
extension class =	%emptystk%	%successful%	%nofndstk%

TOP([rep,]n) :

	N_fnd(‘rep,n)		¬ N_fnd(‘rep,n)
	CARD(‘rep[n]) = 0	CARD(‘rep[n]) > 0	
rep[n]‘=	{}	‘rep[n]	NA
rep‘ =	‘rep	‘rep	‘rep
TOP([rep,]n) |	*true*	TOP([rep,]n) = ‘rep[n][CARD(‘rep[n])]	*true*
extension class =	%emptystk%	%successful%	%nofndstk%

PUSHCHK([rep,]n,e) :

	Tchk(string,n)				¬ Tchk(string,n)
	N_fnd(‘rep,n)			¬ N_fnd(‘rep,n)	
	Tchk(integer,e)		¬ Tchk(integer,e)		
	SUMSTKS(‘rep)< CAPSUMSTKS	SUMSTKS(‘rep)= CAPSUMSTKS			
rep[n]‘=	addAT(‘rep[n],,e)	‘rep[n]	‘rep[n]	NA	NA
rep‘ =	CA	‘rep	‘rep	‘rep	‘rep
extension class =	%successful%	%stkscaprched%	%badstkeletyp%	%nofndstk%	%badstktyp%

PUTCHK([rep,]n,i,e) :

	Tchk(string,n)					¬ Tchk(string,n)
	N_fnd(‘rep,n)				¬ N_fnd(‘rep,n)	
	N_fnd(‘rep[n],i)			¬ N_fnd (‘rep[n],i)		
	Tchk(integer,e)		¬ Tchk (integer,e)			
	SUMSTKS(‘rep)< CAPSUMSTKS	SUMSTKS(‘rep)= CAPSUMSTKS				
rep[n]‘=	addA (‘rep[n],i,,e)	‘rep[n]	‘rep[n]	‘rep[n]	NA	NA
rep‘ =	CA	‘rep	‘rep	‘rep	‘rep	‘rep
extension class =	%successful%	%stkscaprched%	%badstkeletyp%	%indxoutrang%	%nofndstk%	%badstktyp%

Example 3a: Unbounded integer queue module. (Figure 3)

(0) HEADER SECTION

Type Implemented

Unbounded_queue

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition

type Q = ARRAY integer

Representation declaration

rep : Q

Initial representation

rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Value
INSERT	integer	
REMOVE		
FRONT		integer

(3) OPERATION TABLES SECTION

Operation Tables

INSERT([rep,]e) :

	true
rep‘=	addAT(‘rep,,e)
extension class =	%successful%

REMOVE([rep]) :

	CARD(‘rep) = 0	CARD(‘rep) > 0
rep‘ =	‘rep	del_N(‘rep,1)
extension class =	%empty%	%successful%

FRONT([rep]) :

	CARD(‘rep) = 0	CARD(‘rep) > 0
rep‘ =	‘rep	‘rep
FRONT([rep]) |	*true*	FRONT([rep]) = ‘rep[1]
extension class =	%empty%	%successful%

Example 3b: Overflow integer queue module.

(0) HEADER SECTION

Type Implemented
Overflow_queue(CAP)

Module parameter
CAP : Integer

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition
type Q = ARRAY integer

Representation declaration
rep : Q | (CARD(rep) $\leq$ CAP)

Initial representation
rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Value
INSERT	integer	
REMOVE		
FRONT		integer

(3) OPERATION TABLES SECTION

Operation Tables

INSERT([rep,]e) :

	CARD(‘rep) < CAP	CARD(‘rep) = CAP
rep‘=	addAT(‘rep,,e)	addAT(Reseq(del_N(‘rep,1)),,e)
extension class =	%successful%	%full%

REMOVE([rep]) :

	CARD(‘rep) = 0	CARD(‘rep) > 0
rep‘=	‘rep	del_N(‘rep,1)
extension class =	%empty%	%successful%

FRONT([rep]) :

	CARD(‘rep) = 0	CARD(‘rep) > 0
rep‘ =	‘rep	‘rep
FRONT([rep]) |	*true*	FRONT([rep])=‘rep[1]
extension class =	%empty%	%successful%

Example 3c: Overflow integer queue module with insert check.

(0) HEADER SECTION

Type Implemented

Overflow_queue_inserTchk(CAP)

Module parameter

CAP : Integer

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition

type Q = ARRAY integer

Representation declaration

rep : Q | (CARD(rep) $\leq$ CAP)

Initial representation

rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Value
INSERTchk	integer	
REMOVE		
FRONT		integer

(3) OPERATION TABLES SECTION

Operation Tables

INSERTchk([rep,]e) :

	CARD(‘rep) < CAP		CARD(‘rep) = CAP
	Tchk(integer,e)	¬ Tchk(integer,e)	
rep‘=	addAT(‘rep,,e)	‘rep	addAT(Reseq(del_N(‘rep,1)),,e)
extension class =	%successful%	%badtype%	%full%

REMOVE([rep]) :

	CARD(‘rep) = 0	CARD(‘rep) > 0
rep‘	‘rep	del_N(‘rep,1)
extension class =	%empty%	%successful%

FRONT[rep] :

	CARD(‘rep) = 0	CARD(‘rep) > 0
rep‘ =	‘rep	‘rep
FRONT([rep]) |	*true*	FRONT([rep]) = ‘rep[1]
extension class =	%empty%	%successful%

Example 4: Bounded integer table list module. (Figure 5)

(0) HEADER SECTION

Type Implemented

T/L(CAPL)

Module paramete

CAPL : Integer

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition

type t1 = ARRAY integer

type S = REC (”list”, :t1) , (”ptr”, :integer)

Representation declaration

rep : S | (0 ≤ rep[”ptr”] ≤ CARD(rep[”list”]) ≤ CAPL)

Initial representation

rep = {(”list”,{ }),(”ptr”,0)}

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Value
INSERT	integer	
ALTER	integer	
DELETE		
EXLEFT		boolean
EXRIGHT		boolean
OUT		boolean
GOLEFT		
GORIGHT		
CURRENT		integer

3) OPERATION TABLES SECTION

Abbreviations

Abbreviations	Expressions	Table Id.
CURPTR	‘rep[”ptr”]+1	1,2,3,9
EMPTY	CARD(‘rep[”list”]) = 0	1
LLIM	CARD(‘rep[”list”]) ≥ CAPL	1

Operation Tables

INSERT([rep],e) :

1	EMPTY	¬ EMPTY		
		¬ LLIM		LLIM
		rep[”ptr”]=0	rep[”ptr”]>0	
rep[”list”]‘=	{e}	NC	addA(‘rep[”list”],CURPTR,,e)	NC
rep[”ptr”]‘=	1	NC	NC	NC
rep‘ =	CA	NC	CA	NC
extension class =	%emptyIns%	%badptrins%	%ptrins%	%full%

ALTER([rep],e) :

2	rep["ptr"]=0	rep["ptr"] $\neq$ 0
rep["list"]‘=	NC	Sub_O(‘rep["list"],CURPTR,,e)
rep["ptr"]‘=	NC	NC
rep‘ =	NC	CA
extension class =	%nocurrent%	%altered%

DELETE([rep]) :

3	rep["ptr"]=0	rep["ptr"] $\neq$ 0
rep["list"]‘=	NC	delA_N(‘rep["list"],CURPTR)
rep["ptr"]‘=	NC	NC
rep‘ =	NC	CA
extension class =	%nocurrent%	%deleted%

EXLEFT([rep]) :

4	rep[”ptr”] ≤ 1	rep[”ptr”]>1
rep[”list”]‘=	NC	NC
rep[”ptr”]‘=	NC	NC
rep‘ =	NC	NC
EXLEFT([rep]) =	false	true
extension class =	%noleftele%	%leftele%

EXRIGHT([rep]) :

5	rep[”ptr”]=CARD(rep[”list”])	rep[”ptr”]<CARD(rep[”list”])
rep[”list”]‘=	NC	NC
rep[”ptr”]‘=	NC	NC
rep‘ =	NC	NC
EXRIGHT([rep]) =	false	true
extension class =	%norightele%	%rightele%

OUT([rep]) :

6	rep[”ptr”]=0	rep[”ptr”] $\neq$ 0
rep[”list”]‘=	NC	NC
rep[”ptr”]‘=	NC	NC
rep‘ =	NC	NC
OUT([rep]) =	true	false
extension class =	%out%	%notout%

GOLEFT([rep]) :

7	rep[”ptr”]=0	rep[”ptr”] $\neq$ 0
rep[”list”]‘=	NC	NC
rep[”ptr”]‘=	NC	‘rep[”ptr”]–1
rep‘ =	NC	CA
extension class =	%noleft%	%goleft%

GORIGHT([rep]) :

8	rep[”ptr”]=CARD(rep[”list”])	rep[”ptr”]<CARD(rep[”list”])
rep[”list”]‘=	NC	NC
rep[”ptr”]‘=	NC	‘rep[”ptr”]+1
rep‘ =	NC	CA
extension class =	%noright%	%goright%

CURRENT([rep]) :

9	rep[”ptr”]=0	rep[”ptr”] $\neq$ 0
rep[”list”]‘=	NC	NC
rep[”ptr”]‘=	NC	NC
rep‘ =	NC	NC
CURRENT([rep]) |	*true*	CURRENT([rep]) = ‘rep[”list”][CURPTR]
extension class =	%undefined%	%currentele%

Example 5a: Unbounded unique integer producer module. (Figure 6)

(0) HEADER SECTION

Type Implemented

Unbounded_UIP

Module parameter

None

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition

type P = SUBP integer

Representation declaration

rep : P

Initial representation

rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Value
GETINT	integer

(3) OPERATION TABLES SECTION

Operation Tables

GETINT([rep]) :

	true
rep‘ =	‘rep \+ e | Tchk(integer,e) $\wedge$ e$\notin$‘rep
GETINT([rep])=	rep‘–‘rep
extension class =	%successful%

Example 5b: Bounded unique integer producer module.

(0) HEADER SECTION

Type Implemented
Bounded_UIP(CAP)

Module parameter
CAP : Integer

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition
type P = SUBP integer

Representation declaration
rep : P | (CARD(rep) $\leq$ CAP)

Initial representation
rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Value
GETINT	integer

(3) OPERATION TABLES SECTION

Operation Tables

GETINT([rep]) :

	true	
	CARD(‘rep) < CAP	CARD(‘rep) = CAP
rep‘ =	‘rep \+ e | Tchk(integer,e) ∧ e ∉ ‘rep	‘rep
GETINT([rep]) |	GETINT([rep]) = rep‘–‘rep	*true*
extension class =	%successful%	%caprched%

Example 5c: Bounded unique integer producer module.

(0) HEADER SECTION

Type Implemented
Bounded_UIP(CAP)

Module parameter
CAP : Integer

(2) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition
type P = SUBP integer

Representation declaration
rep : P | (CARD(rep) $\leq$ CAP)

Initial representation
rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Value
GETINT	integer

(3) OPERATION TABLES SECTION

Operation Tables

GETINT([rep]) :

	true	
	CARD(‘rep) < CAP	CARD(‘rep) = CAP
GETINT([rep]) |	GETINT([rep]) = e | Tchk(integer,e) ∧ e ∉ ‘rep)	*true*
rep‘ =	‘rep \+ GETINT([rep])	‘rep
extension class =	%successful%	%caprched%

Example 6a : Bounded priority integer queue module. (Figure 7)

(0) HEADER SECTION

Type Implemented

pqueue(CAPI)

Module parameter

CAPI : Integer

(1) CANONICAL REPRESENTATION SECTION

Auxiliary Functions

Numele: t2 $\rightarrow$ integer

$$\text{Numele}(x) \stackrel{df}{=} \sum_{y \in RAN(x)} \sum_{z \in RAN(y)} z$$

Description of Canonical Representation

Type definition

type t1 = NSET integer

type t2 = NSET t1

type t3 = SNE $\cup$ integer

type S = REC (”oval”, :t3) , (”data”, :t2)

Representation declaration

rep : S | (Numele(rep) $\leq$ CAPI)

Initial representation

rep = {(”oval”, { }), (”data”,{ })}

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Arg2	Value
INSERT	integer	integer	
REMOVE	integer		
FRONT			integer

3) OPERATION TABLES SECTION

Operation Tables

INSERT([rep,] p, v) ≡

	‘rep[”data”] = { }	‘rep[”data”] ≠ { }						
		Numele(‘rep[”data”]) < CAP				Numele(‘rep[”data”]) = CAP		
		p∈DOM(‘rep[”data”])		p∉DOM(‘rep[”data”])		p∈DOM(‘rep[”data”])		p∉DOM(‘rep[”data”])
		v∈DOM(‘rep[”data”][p])	v∉DOM(‘rep[”data”][p])	p<Maxi(DOM(‘rep[”data”]))	p>Maxi(DOM(‘rep[”data”]))	v∈DOM(‘rep[”data”][p])	v∉DOM(‘rep[”data”][p])	
rep[”data”][p][v]‘=	NA	’rep[”data”][p][v] + 1	NA	NA	NA	NC	NA	NA
rep[”data”][p]‘=	NA	CA	‘rep[”data”][p] \+ (v,1)	NA	NA	NC	NC	NA
rep[”data”]‘ =	{(p,{(v,1)})}	CA	CA	‘rep[”data”] \+ (p,{(v,1)})		NC	NC	NC
rep[”oval”]‘ =	v	NC	NC	NC	v	NC	NC	NC
rep‘ =	CA	CA	CA	CA	CA	NC	NC	NC
extension class=	%1stIns%	%InsDup%	%InsIK%	%InsQIK%	%Ins-QIKV%	%FullI%	%FullInIK%	%FullInQ%

REMOVE([rep,]) :

<table>
<tr><td rowspan="4"></td><td rowspan="4">‘rep[”data”]={}</td><td colspan="4">‘rep[”data”] ≠ { }</td></tr>
<tr><td colspan="3">rep[”data”][Maxi(DOM(‘rep[”data”]))][‘rep[”oval”]]=1</td><td rowspan="3">‘rep[”data”][Maxi (DOM(‘rep[”data”]))] [‘rep[”oval”]]>1</td></tr>
<tr><td colspan="2">CARD(‘rep[”data”][Maxi(DOM (‘rep[”data”]))]) = 1</td><td rowspan="2">CARD(‘rep[”data”][Maxi (DOM(‘rep[”data”]))]) > 1</td></tr>
<tr><td>CARD(‘rep[”data”])=1</td><td>CARD(‘rep[”data”])>1</td></tr>
<tr><td>rep[”data”] [Maxi(DOM (‘rep[”data”]))] [‘rep[”oval”]]‘ =</td><td>NA</td><td>NA</td><td>NA</td><td>NA</td><td>‘rep[”data”][Maxi (DOM(‘rep[”data”]))] [”oval”] –1</td></tr>
<tr><td>rep[”data”] [Maxi(DOM (‘rep[”data]))]‘=</td><td>NA</td><td>NA</td><td>NA</td><td>‘rep[”data”][Maxi (DOM(‘rep[”data”]))] \– (”oval”,1)</td><td>CA</td></tr>
<tr><td>rep[”data”]‘ =</td><td>NC</td><td>{}</td><td>‘rep[”data”] \– (Maxi(DOM(‘rep [”data”])),{(‘rep [”oval”],1)})</td><td>CA</td><td>CA</td></tr>
<tr><td>rep[”oval”]‘ |</td><td>NC</td><td><u>true</u></td><td colspan="2">rep[”oval”]‘∈DOM (rep[”data”][Maxi(DOM(rep[”data”]‘))]‘)</td><td>NC</td></tr>
<tr><td>rep‘=</td><td>‘rep</td><td>CA</td><td>CA</td><td>CA</td><td>CA</td></tr>
<tr><td>extension class =</td><td>%Empty%</td><td>%ReEmpty%</td><td>%Changepri%</td><td>%Remkey%</td><td>%Remdup%</td></tr>
</table>

FRONT([rep,]a) :

	‘rep[”data”]={}	‘rep[”data”] ≠ {}
rep‘ =	NC	NC
FRONT([rep,]a) =	‘rep[”oval”]	‘rep[”oval”]
extension class =	%Empty%	%Front%

Example 6b : Bounded priority integer queue module.

(0) HEADER SECTION

Type Implemented

pqueue(CAPI)

Module parameter

CAPI : Integer

(1) CANONICAL REPRESENTATION SECTION

Auxiliary Functions

Numele: t2 $\rightarrow$ integer

$$\text{Numele}(x) \stackrel{df}{=} \sum_{y \in RAN(x)} \sum_{z \in RAN(y)} z$$

Description of Canonical Representation

Type definition

type t1 = NSET integer

type t2 = NSET t1

type t3 = SNE ∪ integer

type S = REC (”oval”, :t3) , (”data”, :t2)

Representation declaration

rep : S | (Numele(rep) $\leq$ CAPI)

Initial representation

rep = {("oval", { }), ("data",{ })}

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Arg2	Value
INSERT	integer	integer	
REMOVE	integer		
FRONT			integer

3) OPERATION TABLES SECTION

Abbreviations

Abbreviations	Expressions	Table No.
EMPTY	(‘rep[”data”] = { })	1,2,3
ILIM	(Numele(‘rep[”data”]) ≥ CAP)	1
QMEM	(p∈DOM(‘rep[”data”]))	1
IKMEM	(v∈DOM(‘rep[”data”][p]))	1
GCNGQ	p>‘HIGHQ	1
LCNGQ	p<‘HIGHQ	1
REPHQ‘	rep[”data”][HIGHQ‘]‘	2,3
‘HIGHQ	Maxi(DOM(‘rep[”data”]))	2
HIGHQ‘	Maxi(DOM(rep[”data”]‘))	
‘REPHQ	‘rep[”data”][‘HIGHQ]	2
‘OVAL	‘rep[”oval”]	2

OPERATION TABLES SECTION

Operation Tables

INSERT([rep,] p, v) ≡

1	EMPTY	¬ EMPTY						
		¬ ILIM				ILIM		
		QMEM		¬ QMEM		QMEM		¬ QMEM
		IKMEM	¬ IKMEM	LCNGQ	GCNGQ	IKMEM	¬ IKMEM	
rep[”data”] [p][v]‘=	NA	’rep[”data”] [p][v] +1	NA	NA	NA	NC	NA	NA
rep[”data”][p]‘=	NA	CA	‘rep[”data”][p] \+ (v,1)	NA	NA	NC	NC	NA
rep[”data”]‘ =	{(p,{(v,1)})}	CA	CA	‘rep[”data”] \+ (p,{(v,1)})		NC	NC	NC
rep[”oval”]‘ =	v	NC	NC	NC	v	NC	NC	NC
rep‘ =	CA	CA	CA	CA	CA	NC	NC	NC
extension class=	%1stIns%	%InsDup%	%InsIK%	%InsQIK%	%Ins-QIKV%	%FullI%	%FullInIK%	%FullInQ%

REMOVE([rep,]) :

2	EMPTY	¬ EMPTY			
		rep[”data”][‘HIGHQ][‘OVAL]=1			‘REPHQ[‘OVAL]>1
		CARD(‘REPHQ) = 1		CARD(‘REPHQ)> 1	
		CARD(‘rep[”data”])=1	CARD(‘rep[”data”])>1		
rep[”data”][‘HIGHQ] [‘OVAL]‘ =	NA	NA	NA	NA	‘REPHQ[‘OVAL] –1
rep[”data”][‘HIGHQ]‘=	NA	NA	NA	‘REPHQ \– (‘OVAL,1)	CA
rep[”data”]‘ =	NC	{ }	‘rep[”data”] \– (‘HIGHQ,{(‘OVAL,1)})	CA	CA
rep[”oval”]‘ |	NC	*true*	rep[”oval”]‘ ∈ DOM(REPHQ‘)		NC
rep‘=	NC	CA	CA	CA	CA
extension class =	%Empty%	%ReEmpty%	%Changepri%	%Remkey%	%Remdup%

FRONT([rep,]a) :

3	EMPTY	¬ EMPTY
rep' =	NC	NC
FRONT([rep,]a) =	'OVAL	'OVAL
extension class =	%Empty%	%Front%

Example 6c : Bounded priority integer queue module.

(0) HEADER SECTION

Type Implemented

pqueue(CAPQ, CAPIK)

Module parameter

CAPQ, CAPIK : Integer

(1) CANONICAL REPRESENTATION SECTION

Auxiliary Functions

Numikey : t2 $\rightarrow$ integer

$$\text{Numikey}(x) \overset{df}{=} \sum_{y \in \text{Ran}(x)} \text{Card}(y)$$

Description of Canonical Representation

Type definition

type t1 = NSET integer

type t2 = NSET t1

type t3 = SNE ∪ integer

type S = REC (”oval”, :t3) , (“data”, :t2)

Representation declaration

rep : S | ((CARD(rep["data"]) $\leq$ CAPQ) $\wedge$ (Numikey(RAN(rep["data"])) $\leq$ CAPIK))

/*Maximum number of queues allowed and Maximum number of total item keys allowed for set of queues*/

Initial representation

rep = {("oval", { }), ("data",{ })}

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Arg2	Value
INSERT	integer	integer	
REMOVE	integer		
FRONT			integer

3) OPERATION TABLES SECTION

Abbreviations

Abbreviations	Expressions	Table No.
EMPTY	(‘rep[”data”] = { })	1,2,3
QLIM	(CARD(‘rep[”data”]) ≥ CAPQ)	1
IKLIM	(Numikey(‘rep[”data”]) ≥ CAPIK)	1
QMEM	(p∈DOM(‘rep[”data”]))	1
IKMEM	(v∈DOM(‘rep[”data”][p]))	1
GCNGQ	p>‘HIGHQ	1
LCNGQ	p<‘HIGHQ	1
‘OVAL	‘rep[”oval”]	2,3
‘HIGHQ	Maxi(DOM(‘rep[”data”]))	2
HIGHQ‘	Maxi(DOM(rep[”data”]‘))	
‘REPHQ	‘rep[”data”][‘HIGHQ]	2
REPHQ‘	rep[”data”][HIGHQ‘]‘	2

Operation Tables

INSERT([rep,] p, v) ≡

1	EMPTY	¬EMPTY							
		QMEM			¬QMEM				
		IKMEM	¬IKMEM		QLIM		¬QLIM		
			¬IKLIM	IKLIM	¬IKLIM	IKLIM	IKLIM	¬IKLIM	
								GCNGQ	LCNGQ
rep[”data”] [p][v]‘=	NA	’rep[”data”] [p][v] +1	NA	NA	NA	NA	NA	NA	NA
rep[”data”] [p]‘=	NA	CA	‘rep[”data”] [p] \+ (v,1)	NC	NA	NA	NC	NA	NA
rep[”data”]‘ =	{(p,{(v,1)})}	CA	CA	NC	NC	NC	NC	‘rep[”data”] \+ (p,{(v,1)})	
rep[”oval”]‘ =	v	NC	NC	NC	NC	NC	NC	v	NC
rep‘ =	CA	CA	CA	NC	NC	NC	NC	CA	CA
extension class=	%1stIns%	%InsDup%	%InsIK%	%Full-IK%	%Full-QnIK%	%Full-QIK%	%Full-IKnQ%	%OInsQIK%	%InsQIK%

REMOVE([rep,]) :

2	EMPTY	¬ EMPTY			
		rep[”data”][‘HIGHQ][‘OVAL]=1			‘REPHQ[‘OVAL]>1
		CARD(‘REPHQ) = 1		CARD(‘REPHQ)> 1	
		CARD(‘rep[”data”])=1	CARD(‘rep[”data”])>1		
rep[”data”][‘HIGHQ] [‘OVAL]‘ =	NA	NA	NA	NA	‘REPHQ[‘OVAL] –1
rep[”data”][‘HIGHQ]‘=	NA	NA	NA	‘REPHQ \– (‘OVAL,1)	CA
rep[”data”]‘ =	NC	{ }	‘rep[”data”] \– (‘HIGHQ,{(‘OVAL,1)})	CA	CA
rep[”oval”]‘ |	NC	*true*	rep[”oval”]‘ ∈ DOM(REPHQ‘)		NC
rep‘=	NC	CA	CA	CA	CA
extension class =	%Empty%	%ReEmpty%	%Changepri%	%Remkey%	%Remdup%

FRONT([rep,]a) :

3	EMPTY	¬ EMPTY
rep‘ =	NC	NC
FRONT([rep,]a) =	‘OVAL	‘OVAL
extension class =	%Empty%	%Front%

Example 7a: Bounded multiple binary integer tree module. (Figure 8)

(0) HEADER SECTION

Type Implemented

Tree (CAPN, CAPT, C1, C2)

Module parameter

CAPN, CAPT : Integer

External module

PATH(C1, C2)

(1) CANONICAL REPRESENTATION SECTION

Auxiliary Functions

Tcard : tree → integer

Tcard(t) $\stackrel{df}{=}$

f1	t["left"]={}	t["left"] ≠ {}
t["right"] = {}	1	1 + Tcard(t["left"])
t["right"] ≠ {}	1 + Tcard(t["right"])	1 + Tcard(t["left"]) + Tcard(t["right"])

Description of Canonical Representation

Type definition

type lr = {"l","r"}

type t1 = SNE ∪ tree

type tree = REC ("anint", :integer), ("left", :t1), ("right", :t1)

type S = SSET t1

Representation declaration

rep : S | ((CARD(rep) $\leq$ CAPN) $\wedge$ (CAPT $\geq$ 1) $\wedge$ ($\forall$t, (t$\in$DOM(rep)) $\rightarrow$ (Tcard(rep[t]) $\leq$ CAPT)))

Initial representation

rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Arg2	Arg3	Arg4	Arg5	Value
CREATETREE	string	integer	string	string		
ALTERTREE	string	string	string	tree	string	
ALTERNODE	string	string	integer			
GETNODE	string	string				integer
DELTREE	string					

(3) OPERATION TABLES SECTION

Abbreviations

Abbreviations	Expressions	Table Id.
LVP	Vp(SUBPATH(p),t[”left”])	f2
RVP	Vp(SUBPATH(p),t[”right”])	f2
LUPDBX	Upd(SUBPATH(p),t[”left”],”left”,t,x)	f3
RUPDBX	Upd(SUBPATH(p),t[”right”],”right”,t,x)	f3

LINT	Int(SUBPATH(p),t[”left”])	f4
RINT	Int(SUBPATH(p),t[”right”])	f4
LCNG	Cng(SUBPATH(p),t[”left”],i)	f5
RCNG	Cng(SUBPATH(p),t[”right”],i)	f5
LGVL	Gvl(SUBPATH(p),t[”left”])	f6
RGVL	Gvl(SUBPATH(p),t[”right”])	f6
LGTR	Gtr(SUBPATH(p),t[”left”])	f7
RGTR	GTtr(SUBPATH(p),t[”right”])	f7
CT	(Tcard(‘rep[u]) + Tcard(‘rep[v]) +1 < CAPT)	1
CU	(Tcard(‘rep[u]) +1 < CAPT)	1
CV	(Tcard(‘rep[v]) +1 < CAPT)	1
CN	(CARD(‘rep) < CAPN)	1
RUV	N_fnd(‘rep,r) ∧ N_fnd(‘rep,u) ∧ N_fnd(‘rep,v)	1
R	N_fnd(‘rep,r) ∧ ¬ N_fnd(‘rep,u) ∧ ¬ N_fnd(‘rep,v)	1
RU	N_fnd(‘rep,r) ∧ N_fnd(‘rep,u) ∧ ¬ N_fnd(‘rep,v)	1
RV	N_fnd(‘rep,r) ∧ ¬ N_fnd(‘rep,u) ∧ N_fnd(‘rep,v)	1
UV	¬ N_fnd(‘rep,r) ∧ N_fnd(‘rep,u) ∧ N_fnd(‘rep,v)	1
U	¬ N_fnd(‘rep,r) ∧ N_fnd(‘rep,u) ∧ ¬ N_fnd(‘rep,v)	1

V	¬ N_fnd(‘rep,r) ∧ ¬ N_fnd(‘rep,u) ∧ N_fnd(‘rep,v)	1
NRUV	¬ N_fnd(‘rep,r) ∧ ¬ N_fnd(‘rep,u) ∧ ¬ N_fnd(‘rep,v)	1
N	¬ N_fnd(‘rep,r) ∨ ¬ N_fnd(‘rep,u) ∨ ¬ N_fnd(‘rep,v)	1
SUBT	sub_O(‘rep,r,,{(”anint”,i),(”left”,‘rep[u]),(”right”,‘rep[v])})	1
SUBU	sub_O(‘rep,r,,{(”anint”,i),(”left”,‘rep[u]),(”right”,{})})	1
SUBV	sub_O(‘rep,r,,{(”anint”,i),(”left”,{}),(”right”,‘rep[v])})	1
SUBR	sub_O(‘rep,r,,{(”anint”,i),(”left”,{}),(”right”,{})})	1
ADDU	add(‘rep,,r,,{(”anint”,i),(”left”,‘rep[u]),(”right”,{})})	1
ADDV	add(‘rep,,r,,{(”anint”,i),(”left”,{}),(”right”,‘rep[v])})	1
ADDT	add(‘rep,,r,,{(”anint”,i),(”left”,‘rep[u]),(”right”,‘rep[v])})	1
ADDR	add(‘rep,,r,,{(”anint”,i),(”left”,{}),(”right”,{})})	1
VALIDPATH	Vp(p,‘rep[r])	2,3,4

UPDATEOK	(Tcard(‘rep[r]) – Tcard(Gtr(p,t)) + Tcard(‘rep[x]) $\leq$ CAPT)	2
REPLANT	sub_O(‘rep,r,,‘rep[x])	2
UPDATE	Upd(p,‘rep[r],d,b,‘rep[x])	2
VALIDINTNODE	Int(p,‘rep[r])	3,4
CHANGE	Cng(p,‘rep[r],i)	3
GETVAL	Gvl(p,‘rep[r])	4

Auxiliary Functions

Vp : string $\times$ tree $\rightarrow$ boolean

Vp(p,t) $\stackrel{df}{=}$

<table>
<tr><td rowspan="3">f2</td><td rowspan="3">PATHSTAT([rep,]p)=”e”</td><td colspan="3">Tchk(lr,PATHSTAT([rep,]p))</td></tr>
<tr><td colspan="2">t $\neq$ { }</td><td rowspan="2">t={ }</td></tr>
<tr><td>PATHSTAT([rep,]p)=”l”</td><td>PATHSTAT([rep,]p)=”r”</td></tr>
<tr><td>Vp(p,t)=</td><td>true</td><td>LVP</td><td>RVP</td><td>false</td></tr>
</table>

Upd : string × tree × string × tree × tree → tree

Upd(p,t,d,b,x) $\stackrel{df}{=}$

f3	Vp(p,t)		
	PATHSTAT([rep,]p)="e"	Tchk(lr,PATHSTAT([rep,]p))	
		PATHSTAT([rep,]p)="l"	PATHSTAT([rep,]p)="r"
Upd(p,t,x,d,b)=	Sub_O(b,d,,x)	LUPDBX	RUPDBX

Int : string × tree → boolean

Int(p,t) $\stackrel{df}{=}$

f4	Vp(p,t)			
	PATHSTAT([rep,]p)="e"		Tchk(lr,PATHSTAT([rep,]p))	
	$t \neq \{\}$	t={}	PATHSTAT([rep,]p)='l'	PATHSTAT([rep,]p)='r'
Int(p,t)=	true	false	LINT	RINT

Cng : string × tree × integer → tree

Cng(p,t,i) $\stackrel{df}{=}$

<table>
<tr><td rowspan="3">f5</td><td colspan="3">Vp(p,t)</td></tr>
<tr><td rowspan="2">PATHSTAT([rep,]p)=”e”</td><td colspan="2">Tchk(lr,PATHSTAT([rep,]p))</td></tr>
<tr><td>PATHSTAT([rep,]p)=”l”</td><td>PATHSTAT([rep,]p)=”r”</td></tr>
<tr><td>Cng(p,t)=</td><td>Sub_O(t,”anint”,,i)</td><td><u>LCNG</u></td><td><u>RCNG</u></td></tr>
</table>

Gvl : string × tree → integer

Gvl(p,t) $\stackrel{df}{=}$

<table>
<tr><td rowspan="3">f6</td><td colspan="3">Vp(p,t)</td></tr>
<tr><td rowspan="2">PATHSTAT([rep,]p)=”e”</td><td colspan="2">Tchk(lr,PATHSTAT([rep,]p))</td></tr>
<tr><td>PATHSTAT([rep,]p)=”l”</td><td>PATHSTAT([rep,]p)=”r”</td></tr>
<tr><td>Gvl(p,t)=</td><td>t[”anint”]</td><td><u>LGVL</u></td><td><u>RGVL</u></td></tr>
</table>

Gtr : string $\times$ tree $\rightarrow$ tree

Gtr(p,t) $\stackrel{df}{=}$

<table>
<tr><td rowspan="3">f7</td><td colspan="3">Vp(p,t)</td></tr>
<tr><td rowspan="2">PATHSTAT([rep,]p)=”e”</td><td colspan="2">Tchk(lr,PATHSTAT([rep,]p))</td></tr>
<tr><td>PATHSTAT([rep,]p)=”r”</td><td>PATHSTAT([rep,]p)=”l”</td></tr>
<tr><td>Gtr(p,t)=</td><td>t</td><td><u>LGTR</u></td><td><u>RGTR</u></td></tr>
</table>

Operation Tables

CREATETREE([rep,]r,i,u,v) :

<table>
<tr><td rowspan="5">1</td><td colspan="16">¬ (r=u ∧ r=v)</td><td rowspan="5">(r=u ∧ r =v)</td></tr>
<tr><td colspan="2">RUV</td><td colspan="13">r ≠ u ∧ r ≠ v</td><td>(r=u ∧ r ≠ v) ∨ (r ≠ u ∧ r=v)</td></tr>
<tr><td rowspan="3">CT</td><td rowspan="3">¬ CT</td><td colspan="2" rowspan="2">RU</td><td colspan="2" rowspan="2">RV</td><td rowspan="3">R</td><td colspan="7">CN</td><td rowspan="3">¬ CN</td><td rowspan="3">N</td></tr>
<tr><td colspan="2">UV</td><td colspan="2">U</td><td colspan="2">V</td><td rowspan="2">N</td></tr>
<tr><td>CU</td><td>¬ CU</td><td>CV</td><td>¬ CV</td><td>CT</td><td>¬ CT</td><td>CU</td><td>¬ CU</td><td>CV</td><td>¬ CV</td></tr>
<tr><td>rep‘=</td><td>SUBT</td><td>NC</td><td>SUBU</td><td>NC</td><td>SUBV</td><td>NC</td><td>SUBR</td><td>ADDT</td><td>NC</td><td>ADDU</td><td>NC</td><td>ADDV</td><td>NC</td><td>ADDR</td><td>NC</td><td>NC</td><td>NC</td></tr>
<tr><td>exten-sion class=</td><td>%ok%</td><td>%willrc hct%</td><td>%okno v%</td><td>%willrc hct%</td><td>%okno u%</td><td>%willrc hct%</td><td>%okno uv%</td><td>%crtr%</td><td>%willrc hct%</td><td>%crtrn ov%</td><td>%willrc hct%</td><td>%crtrn ou%</td><td>%willrc hct%</td><td>%crtrn ouv%</td><td>%willrc hcn%</td><td>%mis-tree%</td><td>%same-name%</td></tr>
</table>

ALTERTREE([rep,]r,p,d,b,x) :

<table>
<tr><td rowspan="5">2</td><td colspan="5">r ≠ x</td><td rowspan="5">r=x</td></tr>
<tr><td colspan="4">N_fnd(rep,r) ∧ N_fnd(rep,x) ∧ PATHSTAT([rep,]p) ≠ ”i”</td><td rowspan="4">¬ (N_fnd(rep,r) ∧
N_fnd(rep,x)) ∨
PATHSTAT([rep,]p)=”i”</td></tr>
<tr><td colspan="3">VALIDPATH</td><td rowspan="3">¬ VALIDPATH</td></tr>
<tr><td colspan="2">PATHSTAT([rep,]p) ≠ ”e”</td><td rowspan="2">PATHSTAT
([rep,]p)=”e”</td></tr>
<tr><td>UPDATEOK</td><td>¬ UPDATEOK</td></tr>
<tr><td>rep[r]‘=</td><td>UPDATE</td><td>NC</td><td>REPLANT</td><td>NC</td><td>NC</td><td>NC</td></tr>
<tr><td>rep‘=</td><td>CA</td><td>NC</td><td>CA</td><td>NC</td><td>NC</td><td>NC</td></tr>
<tr><td>extension class=</td><td>%altertree%</td><td>%willrchcapt%</td><td>%replant%</td><td>%invaliddir%</td><td>%invpath/tree%</td><td>%samename%</td></tr>
</table>

ALTERNODE([rep,]r,p,i) :

<table>
<tr><td rowspan="3">3</td><td colspan="3">N_fnd(rep,r) ∧ PATHSTAT([rep,]p) ≠ ”i”</td><td rowspan="3">¬ N_fnd(rep,r) ∨
PATHSTAT([rep,]p)=”i”</td></tr>
<tr><td rowspan="2">VALIDINTNODE</td><td colspan="2">¬ VALIDINTNODE</td></tr>
<tr><td>VALIDPATH</td><td>¬ VALIDPATH</td></tr>
<tr><td>rep[r]‘=</td><td>CHANGE</td><td>NC</td><td>NC</td><td>NC</td></tr>
<tr><td>rep‘=</td><td>CA</td><td>NC</td><td>true</td><td>NC</td></tr>
<tr><td>extension class=</td><td>%alterintnode%</td><td>%notintnode%</td><td>%invaliddir%</td><td>%invpath/tree%</td></tr>
</table>

GETNODE([rep,]r,p) :

4	N_fnd(rep,r) $\wedge$ PATHSTAT([rep,]p) $\neq$ "i"			$\neg$ N_fnd(rep,r) $\vee$ PATHSTAT([rep,]p)="i"
	VALIDINTNODE	$\neg$ VALIDINTNODE		
		VALIDPATH	$\neg$ VALIDPATH	
rep'=	NC	NC	NC	NC
GOLEFT([rep,]r,p) |	GOLEFT([rep,]r,p)=GETVAL	*true*	*true*	*true*
extension class=	%getval%	%notintnode%	%invaliddir%	%invpath/tree%

DELTREE([rep,]r) :

5	N_fnd('rep,r)	$\neg$ N_fnd('rep,r)
rep'=	del_N('rep,r)	NC
extension class =	%deltree%	%invalidtree%

Example 7b: Bounded multiple binary integer tree module.

(0) HEADER SECTION

<u>Type Implemented</u>
Tree (CAPN, CAPT, C1,C2)

<u>Module parameter</u>
CAPN, CAPT : Integer

<u>External type</u>
PATH(C1,C2)

(1) CANONICAL REPRESENTATION SECTION

Auxiliary Functions

Tcard : tree → integer

Tcard(t) $\stackrel{df}{=}$

f1	t[”left”]={}	t[”left”] ≠ {}
t[”right”] = {}	1	1 + Tcard(t[”left”])
t[”right”] ≠ {}	1 + Tcard(t[”right”])	1 + Tcard(t[”left”]) + Tcard(t[”right”])

Description of Canonical Representation

Type definition

type lr = {"l","r"}

type t1 = SNE $\cup$ tree

type tree = REC ("anint", :integer), ("left", :t1), ("right", :t1)

type S = SSET t1

Representation declaration

rep : S | ((CARD(rep) $\leq$ CAPN) $\wedge$ (CAPT $\geq$ 1) $\wedge$ ($\forall$t, (t$\in$DOM(rep)) $\rightarrow$ (Tcard(rep[t]) $\leq$ CAPT)))

Initial representation

rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Arg2	Arg3	Arg4	Arg5	Value
CREATETREE	string	integer	string	string		
ALTERTREE	string	string	string	tree	string	
ALTERNODE	string	string	integer			
GETNODE	string	string				integer
DELTREE	string					

(3) OPERATION TABLES SECTION

Abbreviations

Abbreviations	Expressions	Table Id.
VP	Vp(p,‘rep[r],d,b)	f2
LVPDBX	Vp(SUBPATH(p),t[”left”],”left”,t)	f2
RVPDBX	Vp(SUBPATH(p),t[”right”],”right”,t)	f2
CT	(Tcard(‘rep[u]) + Tcard(‘rep[v]) +1 < CAPT)	1

CU	(Tcard(‘rep[u]) +1 < CAPT)	1
CV	(Tcard(‘rep[v]) +1 < CAPT)	1
CN	(CARD(‘rep) < CAPN)	1
RUV	N_fnd(‘rep,r) ∧ N_fnd(‘rep,u) ∧ N_fnd(‘rep,v)	1
R	N_fnd(‘rep,r) ∧ ¬ N_fnd(‘rep,u) ∧ ¬ N_fnd(‘rep,v)	1
RU	N_fnd(‘rep,r) ∧ N_fnd(‘rep,u) ∧ ¬ N_fnd(‘rep,v)	1
RV	N_fnd(‘rep,r) ∧ ¬ N_fnd(‘rep,u) ∧ N_fnd(‘rep,v)	1
UV	¬ N_fnd(‘rep,r) ∧ N_fnd(‘rep,u) ∧ N_fnd(‘rep,v)	1
U	¬ N_fnd(‘rep,r) ∧ N_fnd(‘rep,u) ∧ ¬ N_fnd(‘rep,v)	1
V	¬ N_fnd(‘rep,r) ∧ ¬ N_fnd(‘rep,u) ∧ N_fnd(‘rep,v)	1
NRUV	¬ N_fnd(‘rep,r) ∧ ¬ N_fnd(‘rep,u) ∧ ¬ N_fnd(‘rep,v)	1
N	¬ N_fnd(‘rep,r) ∨ ¬ N_fnd(‘rep,u) ∨ ¬ N_fnd(‘rep,v)	1
SUBT	sub_O(‘rep,r,{(”anint”,i),(”left”,‘rep[u]),(”right”,‘rep[v])})	1
SUBU	sub_O(‘rep,r,,{(”anint”,i),(”left”,‘rep[u]),(”right”,{ })})	1
SUBV	sub_O(‘rep,r,,{(”anint”,i),(”left”,{ }),(”right”,‘rep[v])})	1
SUBR	sub_O(‘rep,r,,{(”anint”,i),(”left”,{ }),(”right”,{ })})	1
ADDU	add(‘rep,r,,{(”anint”,i),(”left”,‘rep[u]),(”right”,{ })})	1
ADDV	add(‘rep,,r,,{(”anint”,i),(”left”,{ }),(”right”,‘rep[v])})	1

ADDT	add(‘rep,,r,,{(”anint”,i),(”left”,‘rep[u]),(”right”,‘rep[v])})	1
ADDR	add(‘rep,,r,,{(”anint”,i),(”left”,{}),(”right”,{})})	1
VALIDPATH	(Vp_1=true)	2,3,4
REPLANT	sub_O(‘rep,r,,‘rep[x])	2
UPDATEOK	(Tcard(‘rep[r]) – Tcard(Vp_3) + Tcard(‘rep[x]) $\leq$ CAPT)	2
UPDATE	sub_O(Vp_5,Vp_4,,‘rep[x])	2
VALIDINTNODE	(Vp_2=true)	3,4
CHANGE	sub_O(Vp_3,”anint”,,i)	3
GETVAL	Vp_3[”anint”]	4
WARNCT	$\neg$ (r=u $\wedge$ r=v) $\wedge$ ((RUV $\wedge$ $\neg$ CT) $\vee$ ((r $\neq$ u $\wedge$ r $\neq$ v) $\wedge$ ((RU $\wedge$ $\neg$ CU) $\vee$ (RV $\wedge$ $\neg$ CV) $\vee$ (CN- $\wedge$ ((UV $\wedge$ $\neg$ CT) $\vee$ (U $\wedge$ $\neg$ CU) $\vee$ (V $\wedge$ $\neg$ CV))))))	1

Auxiliary Functions

$Vp : string \times tree \times string \times tree \rightarrow boolean \times boolean \times tree \times string \times tree$

$Vp(p,t,d,b) \overset{df}{=}$

f2	PATHSTAT ([rep,]p)="e" ∧ t≠{}	PATHSTAT ([rep,]p)="e" ∧ t={}	Tchk(lr,PATHSTAT([rep,]p)) ∧ t≠{}		Tchk(lr,PATH-STAT([rep,]p)) ∧ t={}
			PATHSTAT ([rep,]p)="l"	PATHSTAT ([rep,]p)="r"	
$\underline{Vp}_1$=	true	true	$\underline{LVPDBX}_1$	$\underline{RVPDBX}_1$	false
$\underline{Vp}_2$=	true	false	$\underline{LVPDBX}_2$	$\underline{RVPDBX}_2$	false
$\underline{Vp}_3$ |	$\underline{Vp}_3$ = t	$\underline{Vp}_3$ = t	$\underline{Vp}_3$ = $\underline{LVPDBX}_3$	$\underline{Vp}_3$ = $\underline{RVPDBX}_3$	*true*
$\underline{Vp}_4$ |	$\underline{Vp}_4$ = d	$\underline{Vp}_4$ = d	$\underline{Vp}_4$ = $\underline{LVPDBX}_4$	$\underline{Vp}_4$ = $\underline{RVPDBX}_4$	*true*
$\underline{Vp}_5$ |	$\underline{Vp}_5$ = b	$\underline{Vp}_5$ = b	$\underline{Vp}_5$ = $\underline{LVPDBX}_5$	$\underline{Vp}_5$ = $\underline{RVPDBX}_5$	*true*

Operation Tables

CREATETREE([rep,]r,i,u,v) :

<table>
<tr><td rowspan="4">1</td><td colspan="11">$\neg (r=u \wedge r=v)$</td><td rowspan="4">$(r=u \wedge r=v)$</td></tr>
<tr><td rowspan="3">RUV $\wedge$ CT</td><td colspan="8">$r \neq u \wedge r \neq v$</td><td rowspan="3">WARNCT</td><td rowspan="3">$(r=u \wedge r \neq v) \vee (r \neq u \wedge r=v) \wedge$ N</td></tr>
<tr><td rowspan="2">RU $\wedge$ CU</td><td rowspan="2">RV $\wedge$ CV</td><td rowspan="2">R</td><td colspan="4">CN</td><td rowspan="2">$\neg$ CN</td></tr>
<tr><td>UV $\wedge$ CT</td><td>U $\wedge$ CU</td><td>V $\wedge$ CV</td><td>N</td></tr>
<tr><td>rep‘=</td><td>SUBT</td><td>SUBU</td><td>SUBV</td><td>SUBR</td><td>ADDT</td><td>ADDU</td><td>ADDV</td><td>ADDR</td><td>NC</td><td>NC</td><td>NC</td><td>NC</td></tr>
<tr><td>extension class=</td><td>%ok%</td><td>%oknov%</td><td>%oknou%</td><td>%oknouv %</td><td>%crtr%</td><td>%crtrnov%</td><td>%crtrnou%</td><td>%crtrnouv %</td><td>%willrchcn %</td><td>%willrchct %</td><td>%mistree%</td><td>%same-name%</td></tr>
</table>

ALTERTREE([rep,]r,p,d,b,x) :

<table>
<tr><td rowspan="5">2</td><td colspan="5">r ≠ x</td><td rowspan="5">r=x</td></tr>
<tr><td colspan="4">N_fnd(rep,r) ∧ N_fnd(rep,x) ∧ PATHSTAT([rep,]p) ≠ ”i”</td><td rowspan="4">¬ (N_fnd(rep,r) ∧ N_fnd(rep,x)) ∨ PATH-STAT([rep,]p)=”i”</td></tr>
<tr><td colspan="3">VALIDPATH</td><td rowspan="3">¬ VALIDPATH</td></tr>
<tr><td colspan="2">PATHSTAT([rep,]p) ≠ ”e”</td><td rowspan="2">PATHSTAT ([rep,]p)=”e”</td></tr>
<tr><td>UPDATEOK</td><td>¬ UPDATEOK</td></tr>
<tr><td>rep[r]‘=</td><td>UPDATE</td><td>NC</td><td>REPLANT</td><td>NC</td><td>NC</td><td>NC</td></tr>
<tr><td>rep‘=</td><td>CA</td><td>NC</td><td>CA</td><td>NC</td><td>NC</td><td>NC</td></tr>
<tr><td>extension class=</td><td>%altertree%</td><td>%willrchcapt%</td><td>%replant%</td><td>%invalidpath%</td><td>%invpath/tree%</td><td>%samename%</td></tr>
</table>

ALTERTNODE([rep,]r,p,i) :

<table>
<tr><td rowspan="3">3</td><td colspan="3">N_fnd(rep,r) ∧ PATHSTAT([rep,]p) ≠ ”i”</td><td rowspan="3">¬ N_fnd(rep,r) ∨ PATHSTAT([rep,]p)=”i”</td></tr>
<tr><td rowspan="2">VALIDINTNODE</td><td colspan="2">¬ VALIDINTNODE</td></tr>
<tr><td>VALIDPATH</td><td>¬ VALIDPATH</td></tr>
<tr><td>rep[r]‘=</td><td>CHANGE</td><td>NC</td><td>NC</td><td>NC</td></tr>
<tr><td>rep‘=</td><td>CA</td><td>NC</td><td>*true*</td><td>NC</td></tr>
<tr><td>extension class=</td><td>%alterintnode%</td><td>%notintnode%</td><td>%invalidpath%</td><td>%invpath/tree%</td></tr>
</table>

GETNODE([rep,]r,p) :

4	N_fnd(rep,r) ∧ PATHSTAT([rep,]p) ≠ ”i”			¬ N_fnd(rep,r) ∨ PATH-STAT([rep,]p)=”i”
	VALIDINTNODE	¬ VALIDINTNODE		
		VALIDPATH	¬ VALIDPATH	
rep‘=	NC	NC	NC	NC
GOLEFT([rep,]r,p) |	GOLEFT([rep,]r,p)=GETVAL	*true*	*true*	*true*
extension class=	%getval%	%notintnode%	%invalidpath%	%invpath/tree%

DELTREE([rep,]r) :

5	N_fnd(‘rep,r)	¬ N_fnd(‘rep,r)
rep‘=	del(‘rep,r)	NC
extension class =	%deltree%	%invalidtree%

Example 8: Bounded path holder module. (Figure 9)

(0) HEADER SECTION

Type Implemented:
Path(CAPN, CAPP)

Module parameter:
CAPN, CAPP : Integer

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition
type lr = {"l", "r"}
type ei = {"e","i"}
type retcode = lr ∪ ei
type t1 = ARRAY lr
type S = SSET t1

Representation declaration
rep : S | ((CARD(rep) ≤ CAPN) ∧ (∀t,(t∈rep) → (CARD(t) ≤ CAPP)))

Initial representation

rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Value
MAKEPATH	string	
DELPATH	string	
PATHSTAT	string	retcode
ADDPATH	string	
SUBPATH	string	

(3) OPERATION TABLES SECTION

Operation Tables

MAKEPATH([rep,]p) :

	N_fnd(‘rep,p)	¬ N_fnd(‘rep,p)	
		CARD(‘rep) < CAPN	CARD(‘rep)=CAPN
rep‘=	sub_O(‘rep,p,,{ })	add(‘rep,,p,,{ })	NC
extension class =	%reinitpath%	%newpath%	%fullset%

DELPATH([rep,]p) :

	N_fnd(‘rep,p)	¬ N_fnd(‘rep,p)
rep‘=	del_N(‘rep,p)	NC
extension class =	%delpath%	%invalidpath%

PATHSTAT([rep,]p) :

	N_fnd(‘rep,p)			¬ N_fnd(‘rep,p)
	CARD(‘rep[p])=0	CARD(‘rep[p])>0		
		‘rep[p][1]=”l”	‘rep[p][1]=”r”	
rep‘=	NC	NC	NC	NC
PATHSTAT([rep,]p)=	”e”	”l”	”r”	”i”
extension class =	%emptypath%	%left%	%right%	%invalidpath%

ADDPATH([rep,]p,c) :

	N_fnd(‘rep,p) ∧ Tchk(lr,c)		¬ N_fnd(‘rep,p) ∧ Tchk(lr,c)	N_fnd(‘rep,p) ∧ ¬ Tchk(lr,c)	¬ N_fnd(‘rep,p) ∧ ¬ Tchk(lr,c)
	CARD(‘rep[p])<CAPP	CARD(‘rep[p])=CAPP			
rep[p]‘=	add_AT(‘rep[p],c)	NC	NA	NC	NA
rep‘=	CA	NC	NC	NC	NC
extension class =	%addir%	%fullpath%	%invalidpath%	%wrongdir%	%inv&wron%

SUBPATH([rep,]p) :

	N_fnd(‘rep,p)		¬ N_fnd(‘rep,p)
	CARD(‘rep[p])>0	CARD(‘rep[p])=0	
rep[p]‘=	delA_N(‘rep[p],1)	NC	NA
rep‘=	CA	NC	NC
extension class =	%subpath%	%emptypath%	%invalidpath%

<u>APPENDIX B:</u> ADDITIONAL EXAMPLES

Example 9: A nondeterministic single object, room module.

(0) HEADER SECTION

<u>Type Implemented</u>
Room(CAP)

<u>Module parameter</u>
CAP : Integer

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

<u>Type definition</u>
type chairs = {"ch1", "ch2", "ch3", "ch4"}
type tables = {"tb2", "tb4"}
type ct = chairs ∪ tables

<u>Representation declaration</u>
rep : ct | (CARD(rep) $\leq$ CAP)

Initial representation

rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Value
PUT	ct	
TAKE		ct

(3) OPERATION TABLES SECTION

Operation Tables

PUT([rep,]e)

	e∉‘rep		e∈‘rep
	CARD(‘rep) < CAP	CARD(‘rep) = CAP	
rep‘=	‘rep \+ e	NC	NC
extension class =	%successful%	%full%	%dupobj%

TAKE([rep]) :

	CARD(‘rep) = 0	CARD(‘rep) > 0
rep‘ =	NC	Any_Ls(‘rep,1)
TAKE([rep]) |	*true*	TAKE([rep]) = ‘rep – rep‘
extension class =	%empty%	%successful%

Example 10: A polymorphic, two–rooms module.

(0) HEADER SECTION

Type Implemented
Room(CAPO,CAPN)

Module parameter
CAPO, CAPN : Integer

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition
type chairs = {"ch1", "ch2", "ch3", "ch4"}
type tables = {"tb2", "tb4"}
type ct = chairs ∪ tables
type S = REC ("old", :ct) , ("new", :string)
type objon1 = {"old","new"}
type objon2 = objon1 \+ "other"
type tup1 = objon1 × boolean × boolean
type tup2 = objon2 × boolean × boolean

Representation declaration

rep : S | ((CARD(rep["old"]) $\leq$ CAPO) $\wedge$ (CARD(rep["new"]) $\leq$ CAPN))

Initial representation

rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Value
PUT	value	
PUTON	string	
GETOPUTN	string	tup1
SGETOPUTN	value	tup2

(3) OPERATION TABLES SECTION

Operation Tables

PUT([rep,]e) :

	Tchk(string,e)					¬ Tchk (string,e)
	Tchk(ct,e)			¬ Tchk(ct,e)		
	e∉ ‘rep[”old”]		e∈ ‘rep[”old”]	CARD (‘rep[”new”]) <CAPN	CARD (‘rep[”new”]) =CAPN	
	CARD(‘rep [”old”]) <CAPO	CARD(‘rep [”old”]) =CAPO				
rep[”old”]‘=	‘rep[”old”] \+ e	NC	NC	NC	NC	NC
rep[”new”]‘=	NC	NC	NC	rep[”new”] \+ e	NC	NC
rep‘=	CA	NC	NC	CA	NC	NC
extension class =	%successold%	%oldfull%	%dupobj%	%successnew%	%newfull%	%badtype%

PUTON([rep,]e) :

	Tchk(ct,e)			¬ Tchk(ct,e)	
	e ∉ ‘rep[”old”]		e ∈ ‘rep[”old”]	CARD(‘rep[”new”]) < CAPN	CARD(‘rep[”new”]) = CAPN
	CARD(‘rep[”old”]) < CAPO	CARD(‘rep[”old”]) = CAPO			
rep[”old”]‘=	‘rep[”old”] \+ e	NC	NC	NC	NC
rep[”new”]‘=	NC	NC	NC	rep[”new”] \+ e	NC
rep‘=	CA	NC	NC	CA	NC
extension class =	%successold%	%oldfull%	%dupobj%	%successnew%	%newfull%

GETOPUTN([rep,]e) :

	Tchk(ct,e)			¬ Tchk(ct,e)		
	CARD(‘rep[”old”]) > 0		CARD(‘rep[”old”]) = 0	e ∉ ‘rep[”new”]		e ∈ ‘rep[”new”]
	e ∈ ‘rep[”old”]	e ∉ ‘rep[”old”]		CARD(‘rep[”new”]) < CAPN	CARD(‘rep[”new”]) = CAPN	
rep[”old”]‘=	‘rep[”old”] \– e	NC	NC	NC	NC	NC
rep[”new”]‘=	NC	NC	NC	’rep[”new”] \+ e	NC	NC
rep‘=	CA	NC	NC	CA	NC	NC
GETOPUTN ([rep],e) =	(”old”,true,true)	(”old”,true, false)	(”old”,false, false)	(”new”,true, true)	(”new”,true, false)	(”new”,false, false)
extension class =	%successold%	%notfound%	%oldempty%	%successnew%	%newfull%	%dupobj%

SGETOPUTN([rep,]e) :

	Tchk(string,e)						¬ Tchk (string,e)
	Tchk(ct,e)			¬ Tchk(ct,e)			
	CARD(‘rep[”old”]) > 0		CARD(‘rep [”old”]) = 0	e ∉ ‘rep[”new”]		e ∈ ‘ rep[”new”]	
	e ∈ ‘rep[”old”]	e ∉ ‘rep[”old”]		CARD(‘rep [”new”])< CAPN	CARD(‘rep [”new”])= CAPN		
rep[”old”]‘=	‘rep[”old”] \– e	NC	NC	NC	NC	NC	NC
rep[”new”]‘=	NC	NC	NC	’rep[”new”] \+ e	NC	NC	NC
rep‘=	CA	NC	NC	CA	NC	NC	NC
SGETOPUTN ([rep],e)$_1$ =	”old”	”old”	”old”	”new”	”new”	”new”	”other”
SGETOPUTN ([rep],e)$_2$ |	SGETOPUTN ([rep],e)$_2$=true	SGETOPUTN ([rep],e)$_2$=false	SGETOPUTN ([rep],e)$_2$=false	SGETOPUTN ([rep],e)$_2$=true	SGETOPUTN ([rep],e)$_2$=false	SGETOPUTN ([rep],e)$_2$=false	*true*
SGETOPUTN ([rep],e)$_3$ |	SGETOPUTN ([rep],e)$_3$=true	SGETOPUTN ([rep],e)$_3$=true	SGETOPUTN ([rep],e)$_3$=false	SGETOPUTN ([rep],e)$_3$=true	SGETOPUTN ([rep],e)$_3$=true	SGETOPUTN ([rep],e)$_3$=false	*true*
extension class =	%successold%	%notfound%	%oldempty%	%successnew%	%newfull%	%dupobj%	%badtyp%

Example 11: A non–deterministic polymorphic, two–rooms module.

(0) HEADER SECTION

Type Implemented
Room(CAPO,CAPN)

Module parameter
CAPO, CAPN : Integer

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition
type chairs = {”ch1”, “ch2”, “ch3”, “ch4”}
type tables = {”tb2”, “tb4”}
type ct = chairs ∪ tables
type S = REC (”old”, :ct) , (“new”, :string)
type objon1 = {”old”,”new”}
type objon2 = {”old”,”new”,”other”}
type booltup = boolean $\times$ boolean
type tup1 = boolean $\times$ objon1 $\times$ string
type tup2 = boolean $\times$ objon2 $\times$ string

Representation declaration

rep : S | ((CARD(rep["old"]) $\leq$ CAPO) $\wedge$ (CARD(rep["new"]) $\leq$ CAPN))

Initial representation

rep = { }

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1	Value
PUT	value	
PUTO	ct	
PUTON	string	
TAKEO		ct
TAKEN		string
GETO	ct	booltup
GETOANYN	string	tup1
SGETOANYN	value	tup2

(3) OPERATION TABLES SECTION

Operation Tables

PUT([rep,]e) :

	Tchk(string,e)					¬ Tchk (string,e)
	Tchk(ct,e)			¬ Tchk(ct,e)		
	e∉ ‘rep[”old”]		e∈ ‘rep[”old”]	CARD(‘rep [”new”])<CAPN	CARD(‘rep [”new”])=CAPN	
	CARD(‘rep [”old”]) <CAPO	CARD(‘rep [”old”]) =CAPO				
rep[”old”]‘=	‘rep[”old”] \+ e	NC	NC	NC	NC	NC
rep[”new”]‘=	NC	NC	NC	rep[”new”] \+ e	NC	NC
rep‘=	CA	NC	NC	CA	NC	NC
extension class =	%successold%	%oldfull%	%dupobj%	%successnew%	%newfull%	%badtype%

PUTO([rep,]e) :

	e∉‘rep[”old”]		e∈‘rep[”old”]
	CARD(‘rep[”old”]) < CAPO	CARD(‘rep[”old”]) = CAPO	
rep[”old”]‘=	‘rep[”old”] \+ e	NC	NC
rep‘=	CA	NC	NC
extension class =	%successful%	%oldfull%	%dupobj%

PUTON([rep,]e) :

	Tchk(ct,e)			¬ Tchk(ct,e)	
	e∉‘rep[”old”]		e∈‘rep[”old”]	CARD(‘rep[”new”]) < CAPN	CARD(‘rep[”new”]) = CAPN
	CARD(‘rep[”old”]) < CAPO	CARD(‘rep[”old”]) = CAPO			
rep[”old”]‘=	‘rep[”old”] \+ e	NC	NC	NC	NC
rep[”new”]‘=	NC	NC	NC	rep[”new”] \+ e	NC
rep‘=	CA	NC	NC	CA	NC
extension class =	%successful%	%oldfull%	%dupobj%	%successnew%	%newfull%

TAKEO([rep]) :

	CARD(‘rep[”old”]) = 0	CARD(‘rep[”old”]) > 0
rep[”old”]‘ =	NC	Any_Ls(’rep[”old”],1)
rep‘ =	NC	CA
TAKEO([rep]) |	*<u>true</u>*	TAKEO([rep])=‘rep[”old”] – rep[”old”]‘
extension class =	%empty%	%successful%

TAKEN([rep]) :

	CARD(‘rep[”new”]) = 0	CARD(‘rep[”new”]) > 0
rep[”new”]‘ =	NC	Any_Ls(’rep[”new”],1)
rep‘ =	NC	sub_O(‘rep,new,,rep[new]‘)
TAKEN([rep]) |	*<u>true</u>*	TAKEN([rep])=‘rep[”new”] – rep[”new”]‘
extension class =	%empty%	%successful%

GETO([rep,]e) :

	CARD(‘rep[”old”]) > 0		CARD(‘rep[”old”]) = 0
	e∈‘rep[”old”]	e∉‘rep[”old”]	
rep[”old”]‘=	‘rep[”old”] \– e	NC	NC
rep‘=	CA	NC	NC
GETO([rep,]e)$_1$=	true	false	true
GETO([rep,]e)$_2$|	GETO([rep,]e)$_2$=true	GETO([rep,]e)$_2$=false	*true*
extension class=	%successful%	%notfound%	%empty%

GETOANYN([rep,]e) :

	Tchk(ct,e)			¬ Tchk(ct,e)	
	CARD(‘rep[”old”]) > 0		CARD (‘rep[”old”])= 0	CARD(‘rep[”new”]) > 0	CARD (‘rep[”new”]) = 0
	e∈‘rep[”old”]	e∉‘rep[”old”]			
rep[”old”]‘=	‘rep[”old”] \– e	NC	NC	NC	NC
rep[”new”]‘=	NC	NC	NC	Any_Ls(’rep[”new”],1)	NC
rep‘=	CA	NC	NC	sub_O(‘rep,new,, rep[new]‘)	NC
GETOANYN ([rep],e)$_1$=	”old”	”old”	”old”	”new”	”new”
GETOANYN ([rep],e)$_2$=	true	true	false	true	false
GETON([rep],e)$_3\vert$	*true*	*true*	*true*	GETOANYN([rep],e)$_3$ =‘rep[”new”]–rep[”new”]‘	*true*
extension class =	%successold%	%notfound%	%oldempty%	%successnew%	%newempty%

SGETOANYN([rep,]e) :

	Tchk(string,e)					¬ Tchk (string,e)
	Tchk(ct,e)			¬ Tchk(ct,e)		
	CARD(‘rep[”old”]) > 0		CARD(‘rep [”old”]) = 0	CARD(‘rep [”new”]) > 0	CARD(‘rep [”new”]) = 0	
	e ∈ ‘rep[”old”]	e ∉ ‘rep[”old”]				
rep[”old”]‘=	‘rep[”old”] \– e	NC	NC	NC	NC	NC
rep[”new”]‘=	NC	NC	NC	Any_Ls (’rep[”new”],1)	NC	NC
rep‘=	CA	NC	NC	sub_O(‘rep, new,,rep[new]‘)	NC	NC
SGETOANYN ([rep],e)$_1$ =	”old”	”old”	”old”	”new”	”new”	”other”
SGETOANYN ([rep],e)$_2$ =	SGETOANYN ([rep],e)$_2$ =true	SGETOANYN ([rep],e)$_2$ =true	SGETOANYN ([rep],e)$_2$ =false	SGETOANYN ([rep],e)$_2$ =true	SGETOANYN ([rep],e)$_2$ =false	***true***
SGETOANYN ([rep],e)$_3$ |	***true***	***true***	***true***	SGETOANYN ([rep],e)$_3$ =rep[”new”]‘	***true***	***true***
extension class =	%successold%	%notfound%	%oldempty%	%successnew%	%newempty%	%badtyp%

Example 12: A bounded polymorphic multi–object, non–empty room module.

(0) HEADER SECTION

Type Implemented
Room(CAP)

Module parameter
CAP : Integer

(1) CANONICAL REPRESENTATION SECTION

Description of Canonical Representation

Type definition
type seat = {"rnd", "sqr", "tri", "obl"}
type chrstr = REC ("bk", :back), ("lg", :leg), ("st", :seat)
type cs = seat ∪ chrstr
type back = {"rnd", "sqr", "cor"}
type leg = {"3legs", "4legs", "6legs"}
type tabshp = {"rtab", "stab", "otab"}
type stlstr = REC ("lg", :leg), ("st", :seat)
type blstcs = cs ∪ back ∪ leg ∪ tabshp ∪ stlstr

type stpart = ARRAY seat
type stool = SSET stlstr
type chair = ARRAY chrstr
type table = SSET tabshp
type room = REC (”obj1”, :chair), (”obj2”, :table), (”obj3”, :stool), (“obj4”, :stpart)

Representation declaration

rep: room | ((CARD(rep[”obj1”]) + CARD(rep[”obj2”] + CARD(rep[”obj3”] + CARD(rep[”obj4”])) ≤ CAP)

Initial representation

rep | ((CARD(rep[”obj1”])=0) ∧ (CARD(rep[”obj2”])>0) ∧ (CARD(rep[”obj3”])>0) ∧ (CARD(rep[”obj4”])=0) ∧

((CARD(rep[”obj2”]) + CARD(rep[”obj3”]))< CAP))

(2) SYNTAX SECTION

Access Programs

Program Name	Arg1
SEL5PUT	blstcs
SEL2PUT	cs

(3) OPERATION TABLES SECTION

Abbreviations

Abbreviations	Expansions	Table Id.
SUMOBJ1	CARD(rep[”obj1”])	1,2
SUMOBJ2	CARD(rep[”obj2”])	
SUMOBJ3	CARD(rep[”obj3”])	
SUMOBJ4	CARD(rep[”obj4“])	1,2
SUMOBJS	SUMOBJ1 + SUMOBJ2 + SUMOBJ3 + SUMOBJ4	1,2

Operation Tables

SEL2PUT([rep,]e) :

1	SUMOBJS<CAP				SUMOBJS =CAP
	Tchk(chrstr,e)		Tchk(seat,e)		
	N_fnd (‘rep[”obj1”],e)	¬ N_fnd (‘rep[”obj1”],e)	¬ N_fnd (‘rep[”obj4”],e)	¬ N_fnd (‘rep[”obj4”],e)	
rep[”obj1”]‘=	NC	add(‘rep[”obj1”],, (SUMOBJ1+1),,e)	NC	NC	NC
rep[”obj4”]‘=	NC	NC	add(‘rep[”obj4”],, (SUMOBJ4+1),,e)	NC	NC
rep’=	NC	CA	CA	NC	NC
extension class=	%dupchrstr%	%putchair%	%putseat%	%dupseat%	%full%

SEL5PUT([rep,]e) :

2	SUMOBJS< CAP			SUMOBJS= CAP
	Tchk[chrstr,e]	Tchk[seat,e]	¬ (Tchk[chrstr,e] ∨ Tchk[seat,e])	
rep[obj1]‘=	‘rep[”obj1”] \+ ((SUMOBJ1 + 1),e)	NC	NC	NC
rep[obj4]‘=	NC	‘rep[”obj4”] \+ ((SUMOBJ4 + 1),e)	NC	NC
rep’=	CA	CA	NC	NC
extension class =	%putchair%	%putseat%	%blsttype%	%full%

APPENDIX C: USING THE CANONICAL REP. GRAMMAR TOOL

Explaining the general grammar

The grammar for describing ITAM canonical representations is given in chapter 4. The grammar is presented in 2 parts. The first part (section 3.4.1) is the general BNF, and includes all nonterminals which will map to terminal symbols used in writing a canonical representation description. In the first part of the grammar we use uppercase lettering to distinguish nonterminals that will be mapped to terminal symbols. In the first part of the grammar we do not map any uppercase nonterminals to terminal symbols. We give the second part of the grammar as these mappings, (in the table of section 3.4.2).

There are reasons for separating the uppercase nonterminal mappings. Firstly, a person can choose which tool will be used to test the grammar. Secondly, depending on the chosen tool and the system that the tests will be done on, it may be necessary to map uppercase nonterminals to a different (from the table in section 3.4.2) set of terminal symbols. We used LEXX and YACC, and on the system that we used to test the grammar, the symbol, ∀ was unavailable so instead we used A!. Table 24. gives those nonterminals that we had to remap because of the unavailability of some terminal symbols found in 3.4.2. (An example of a canonical representation description is given at the end of this Appendix, with set union written as !!U.)

Any user of a canonical representation syntax checking tool should at least be familiar with all nonterminal–to–terminal symbol mappings, for his/her system. These are the terminal symbols the user will need to write a description of the canonical representation on his/her system. Terminal symbols must be recognised by the tool being used for testing a presented grammar.

The grammar checking tool for our system

One should understand the grammatical rules given in section 3.4 before writing an ITAM canonical representation description. The tool is just a guide that helps you to do this, it stops at the first part of a canonical representation description that does not confirm to the rules laid out in section 3.4.1 and 3.4.2 (3.4.2 with any remappings). It is designed to test a file with the portion of an ITAM document describing the canonical representation of a module. The <filename> must be given, (see below for the commands to build and test). When a description that confirms to the grammatical rules, is tested, the tool gives an informative successful message, (i.e. currently, YES!!).

The grammar does not allow leading zeros except when a real is less than 1. For example, a writer can quickly recognize that the set with {002, 2, 3} should be written as {2,3}.

LEXX and YACC is right recursive. The LEXX file was composed of terminal symbols presented in section 3.4.2 and the YACC file from the grammar presented in 3.4.1.

When constructing this grammer using LEXX and YACC, one must make sure that most function names are not written in all uppercase. The only exceptions to this ”all uppercase– rule”, are for CARD, DOM and RAN, (Table 18.)

Another precaution to be taken is to ensure that infix predicates following the quantifiers, should be in parentheses.

In presenting the examples of Appendix A and B some regular symbols were available otherwise we had to create our own. We use the following ”one–to–one” translation when we test the text describing canonical representations.

Uppercase Nonterminals	Our terminal symbols	Our examples’ terminals
QUOTE	‘	”
CROSSPRO	#	$\times$
INTERSECT	!!N	$\cap$
UNION	!!U	$\cup$
SETDIFF	!!–	$-$
CR	Representation	Representation declaration
GE	>=	$\geq$
LE	<=	$\leq$
NEQUAL	!=	$\neq$
SETEQUAL	SET=	$=$
SETNEQUAL	!SET=	$\neq$
STREQUAL	STR=	$=$
STRNEQUAL	!STR=	$\neq$
NOT	!	$\neg$
PROPSUB	PSUB	$\subset$
SUB	SUB	$\subseteq$
AND	/\\, keyboard’s / and \	$\wedge$
OR	\\/, keyboard’s \ and /	$\vee$
NONMEM	!MEM	$\notin$
MEM	MEM	$\in$
EXIST	E!	$\exists$
FORALL	A!	$\forall$
RIMPLIES	—>	$\rightarrow$
RLIMPLIES	<—>	$\leftrightarrow$
IR	Initial rep	Initial representation

Table 24. Translation of familiar symbols for text testing

An Example:

Type definition
type S_1 = {1,2}
type S2 = {‘a‘,‘b‘}
type S3 = SX !!U S_T
type S = SUBP S4
Representation
rep: S | CARD(rep)< CAP
Initial rep
rep | (CARD(rep)=1 /\ (rep = 1 \/ rep STR=‘a‘))

Commands for building and testing our grammar checking tool at the DEC OSF unix prompt using LEXX and YACC:

```
flex fun.l
yacc –d fun.y
gcc lex.yy.c y.tab.c cat
cat <filename> | a.out
```

REFERENCES

[1] Bartussek, W., Parnas, D.L., “Using Assertions About Traces to write Astract Specifications for Softeware Modules”, UNC Report No. TR77–012, December 1977, 26pgs.
– Also in Lecture Notes in Computer Science (65), *Information Sytems Methodology, Proceedings ICS*, Venice, 1978, Springer Verlag, pp211–236.
– Also in *Software Specification Techniques* edited by N. Gehani & A.D. McGettrick, AT&T Bell Telephone Laboratories, 1985, pp 111–130 (QA 76.6 S6437).

[2] Guttag J.V., “The Specification and Application of Programming of Abstract Data Types”, PhD Thesis, University of Toronto, 1975.

[3] Guttag, J.V., Horning, J.J., “The Algebraic Specification of Abstract Data Types”, *Acta Informatica 10*, pp 27–52, 1978.

[4] Hoffman, D.M., “The Trace Specification of Communications Protocols”, *IEEE Transactions on Computers*, Vol. C–34, No. 12, December 1985, pp. 1102–1113.

[5] Heninger, K., Kallander, J., Parnas, D.L., Shore, J., “Software Requirements for the A–7E Aircraft”, NRL Report 3876, November 1978, 523 pgs.

[6] Janicki, R., “Towards a Formal Semantics of Parnas Tables”, Proceedings of the *17th International Conference on Software Engineering*, Seattle WA, April 23–30 1995, pp231–240.

[7] Janicki, R., Parnas, D.L., Zucker, J., “Tabular Representations in Relational Documants”, CRL Report 313, McMaster University, CRL TRIO, November 1995. To appear in “*Relational Methods in Computer Science*”, Ed. C. Brink and G. Schmidt, Springer Verlag.

[8] Parnas, D.L., “A Technique for Software Module Specification with Examples”, *Communications of the ACM*, 15,5, May 1972, pp 330–336.
– Also in *Software Specification Techniques* edited by N. Gehani & A.D. McGettrick, AT&T Bell Telephone Laboratories, 1985, pp 75–88 (QA 76.7 S6437).
– Translated into Russian – book “*Danniye v yazikach programmirovania*” Moscow, Mir (Publishing House), 1984, pp. 9–24.

[9] Parnas, D.L., “On the Criteria to be Used in Decomposing Systems into Modules”, *Communications of the ACM*, 15, 12, December 1972, pp. 1053–1058.
– Translated into Japanese – *BIT*, vol. 14, no. 3, 1982, pp. 54–60.
– Republished in *Great Papers in Computer Science*, edited by Phillip Laplante, West Publishing Co, Minneapolis/St. Paul 1996, pp. 433–441.

[10] Parnas D.L., Wang, Y., “The Trace Assertion Method of Module Interface Specification”, Technical Report 89–261, Queen’s, CIS, Telecommunications Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, October 1989, 39 pp. Available from CRL, McMaster University, Hamilton Ontario.

[11] Parnas D.L., Tabular Representations of Relations”, CRL Report 260, McMaster University, CRL, TRIO, October 1992, 17 Pgs.

[12]Parnas, D.L., “Precise Description and Specification of Software”, in “*Mathematics of Dependable System II*”, edited by V. Stavridou, Clarendon Press, 1997, pp. 1–14

[13] Parnas, D.L., Madey, J., “Functional Documentation for Computer Systems Engineering”, Published in *Science of Computer Programming* (Elsevier) vol.25, No. 1, October 1995, pp. 41–61.

[14] Parnas, D.L., Madey, J., Iglewski, M., “Precise Documentation of Well–structured Programs”, *IEEE Transactions on Software Engineering*, Vol. 20, No. 12, December 1994, pp 948–976.

[15] Zucker, J.I, “Transformations of Normal an Inverted Function Tables”, CRL Report 291, McMaster University, CRL, TRIO, August 1994, 26 pgs.

[16] Parnas, D.L., “Predicate Logic for Software Engineering”, *IEEE Transactions on Software Engineering*, Vol. 19, No. 9, September 1993, pp 856–862.

[17] Halmos, Paul R., “Naive Set Theory”, Springer–Verlag.

[18] Parnas, D.L., Shore, J. E., Weiss, D., “Abstract types defined as classes of variables”, *Proceedings of the Second International Conference on Data: Abstraction, Definition, and Structure*, Salt Lake City, March 1976, pp. 22–24. Also in: NRL Memorandum Report 7998, United States Naval Research Laboratory, Washington DC, April 1976, pp1–10.

[19] Wang, Y., “Formal and Abstract Software Module Specifications – A Survey”, CRL Report 238, McMaster University, CRL, TRIO, November 1991, 90 pgs.
– Previously published as a Technical report 91–307, TRIO Queen’s University.

[20] McLean, J., “A Formal Foundation for the Abstraction of Software”, *Journal of the ACM,* Vol. 31, No. 3, pp. 660–627, July 1984.

[21] Hoffman, D.M., Snodgrass, R., “Trace Specifications: Methodology and Models”, *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, September 1988, pp. 1243–1252.

[22] Parnas, D.L., Smith, D., Pearce, T.W., “Making Formal Software Documentation More Practical”, *Technical Report 88–236*, Department of Computing and Information Science, Queen’s University, June 1988.

[23] Parnas, D.L., “Software Engineering Principles”, “*INFOR, Canadian Journal of Operation Research and Information Processing*”, Vol. 22, No. 4, November 1984, pp. 303–316

[24] Wirth, N., “Algorithms and Data Structures=Programs”, Prentice Hall Series, pgs. 366

[25] Abraham, R., “Evaluating Generalized Tabular Expressions in Software Documentation”, *M.Eng. Thesis*, Department of Electrical and Computer Engineering, McMaster University, Hamilton, ONT. Canada, February 1997, 83 pgs.
– Also in: *CRL Report No. 346*, McMaster University, CRL, TRIO, Hamilton, ON, Canada, February 1997.

[24] Wirth, N., “Compiler Construction”, Addison–Wesley, pgs. 176.

[25] McMaster University SERG, “Table Tool System Developer’s Guide”, CRL Report 339, McMaster University, NSERC and CRL, TRIO, January 1997, 86 pgs.

[26] Schach, R., “CLASSICAL and OBJECT–ORIENTED SOFTWARE ENGINEERING”, Third Edition, 603 pgs.

[27] Wang, Y., “Specifying And Simulating The externally Observable Behaviour Of Modules”, CRL Report 292, McMaster University, Faculty of Engineering, CRL, TRIO, August 1994, 130 pgs.

[28] Norvell, Theodore S., “On Trace Specifications”, CRL Report 305, McMaster University, Faculty of Engineering, CRL, July 1995, 45 pgs.

[29] Iglewski, M, Madey, J., Stencel, K., “On Fundamentals of the Trace Assertion Method”, RR 94/09–6, Department D’Informatique, Universite du Quebec a Hull, Hull, Quebec, September 1994.

[30] Janicki, R., “Foundations of the Trace Assertion Method of Module Interface Specification”, CRL Report 348, McMaster University, Department of Computer Science and Systems, March 1997, 46 Pgs.

[31] Iglewski, M, Kubica, M., Madey, J., “Trace Specification of Non–deterministic Multi–object Modules”, TR 95/05(205), Institute of Informatics, Warsaw University, Warsaw, Poland, 1995.

[32] Iglewski, M, Mincer–Daszkiewicz, J., Stencel, K., “Some experiences with Specification of Non–deterministic Modules”, RR 94/09–7, Department D’Informatique, Universite du Quebec a Hull, Hull, Quebec, September 1994.

[33] Prowell, S. J., “Sequence–Based Software Specification”, Ph.D. Thesis, University of Tennessee, Knoxvile, Tennessee, U.S.A., 1996.

[34] Van Schowen, J., “On the road to practical module interface specification”, a lecture presented at MacMaster workshop on tools for Tabular Notations, McMaster University, Hamilton, Ontario, Canada, 1996

[35] Chunming, Li., “Documentation Based Software Module Reliability Estimation Tool”, CRL Report 337, McMaster University, Faculty of Engineering, CRL, TRIO, December 1996, 122 pgs.

[36] Desruisseaux, B., Iglewski, M, Kubica, M., Roy, P., “Working notes on Requirements specification for the Trace Specification Editor”, RR94/09–5, Universite du Quebec a Hull, Hull, Quebec, September 1994.

[37] Iglewski, M, Kubica, M., Madey, J., “Editor for the Trace Assertion Method”, *Proceedings of the 10th International Conference of CAD/CAM, Robotics and Factories of the Future: CARs & FOF ’94*, Zaremba, M., OCRI, Ottawa, Ontario, Canada, 1994, pp 876–881.

[38] Stencel, K., “Refined Simulation Techniques for the Trace Assertion Method”, CRL Report 314, McMaster University, Faculty of Engineering, CRL, TRIO, December 1995, 11 pgs.

[40] Iglewski, M., Kubica, M., Madey, J., Mincer–Daszkiewicz, j., Stencel K., “TAM’97: the Trace Assertion Method of Module Interface Specification. Reference Manual”, TR 97/01 (238), Institute of Informatics, Warsaw University, Warsaw 1997.