Dealing with Non-Functional Properties
in an Imperative Programming Language

Xavier Franch

Report LSI-95-44-R

UPC
Facultat d'Informàtica
de Barcelona - Biblioteca
26 SET. 1995
Dealing with Non-Functional Properties in an
Imperative Programming Language

Xavier Franch
Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Barcelona
franch@lsi.upc.es

Abstract: An extension of a modular imperative programming language is presented. This extension is aimed to support the automatic selection of implementations depending on their behaviour with respect to time and space efficiency, which is measured using the big-Oh asymptotic notation. The language provides some constructs to state both efficiency behaviour of public types and functions and efficiency requirements over implementations. Efficiency requirements may be stated at different places in programs; so, different implementations may be selected for different objects of the same type. An algorithm and a set of rules to compute the set of implementations that satisfy their requirements are presented. If the selection succeeds, programs may be translated into executable code.
1 Introduction

Abstract data types (ADT for short) provide a two-level stratification of types: the upper level, specification, describes the behaviour of the type to its users; the lower level, implementation, consists of code for representing type values and its functions that must satisfy the specification when executed. There exists a single specification for every ADT; however, there could exist many implementations, each one of them with its own non-functional properties; thus, an implementation becomes interesting when it is the best suited for the set of non-functional requirements in its context of use. In this report, we are interested in a particular kind of non-functional requirements, namely efficiency of implementations, this is to say, how a given implementation consumes computer resources during execution.

To be more precise, we address to the following situation: given an ADT with a full determined specification\(^1\), and given a program using this ADT with some efficiency requirements, we want to select an implementation for the ADT in this program. The usual procedure to solve this question consists on a preliminary informal checking of all the already existing implementations against the requirements; if some implementations satisfy the requirements, we pick up one; otherwise, we should write a new implementation for this particular situation, that will be available for future use when needed, or else we could relax somehow the requirements until an existing implementation is valid for them.

This procedure has many drawbacks. First of all, the design task is not automated at all; the designer is responsible of the process of checking implementations against the requirements. Also, the resulting program is very sensitive to modifications coming from changes on program requirements or from later creation of new implementations of ADTs; in both cases, a new design analysis should be carried out. A third drawback is that design decisions and efficiency characteristics of software do not appear on the software itself but optionally in documentation. Last, usual programming languages do not permit to attach two different implementations to the same ADT in different points in the same program\(^2\).

In this report, a procedure to associate implementations to specifications is presented that solves all the drawbacks mentioned above. It relies on the enrichment of a classical imperative programming language, in this case Modula-2 (although the framework may be easily adapted to similar programming languages), with some new constructs concerning efficiency, to obtain the design-oriented Modula-NF programming language (NF stands for "non-functional"). Programming in Modula-NF does not require to explicitly check implementations against requirements, but to establish efficiency requirements and efficiency properties of software; implementations are automatically selected by means of an algorithm and some rules incorporated to the parser. Changes on requirements demand just changes on those software components establishing them; construction of new implementations does not require any change in existing software. In both cases, software may be actualised re-running the implementation selection algorithm. Furthermore, the same ADT may have more than one implementation attached in different parts of the program if the efficiency requirements demand

\(^1\) The degree of formalism in the specification is not rellevant to the work presented here.

\(^2\) Inheritance may partially solve this problem, but with some restrictions (see section 7).
it; this is true even if two objects of the same type interact provided that an abstraction function has been defined for implementations. To obtain executable code, Modula-NF programs may be translated to other languages via semantic rules once implementation selection has been successful.

The report is organised as follows. In section 2, the Modula-NF programming language constructions related to efficiency are presented, together with various examples illustrating their use. Sections 3 and 4 define the rules to select implementations and the algorithm to check them against requirements. Section 5 focuses on the existence of more than an implementation for a given type in a single program. The translation of a software application from Modula-NF to Modula-2 is addressed in section 6, emphasizing how name clashes are avoided and how the abstraction function is translated. Last, section 7 compares our approach with other existing ones and also presents the conclusions and future work (extension to more powerful languages, use of other measure units and -partial- automation of cost computation from code).

2 The Modula-NF Programming Language

The family of imperative programming languages stemming from Algol is a wide one. Among various possibilities, in this report we have chosen Modula-2 [Wir82] as a basis for our work mainly for three reasons. First of all, it is a well-known language in the academic environments and, in particular, it is the language used for our students to develop software in the first programming courses. Second, it is a modular language oriented to programming with ADT (in spite of various drawbacks, as non-privacy of types in the general case or lack of genericity). Last, it may serve as a first step to more powerful languages, including genericity, inheritance, error management, etc.; in fact, it could be said that the work done for Modula-2 could be a seed for languages as Oberon, Ada, Eiffel and so on.

Concerning the programming language level, the goal of our work is to add several language constructs on Modula-2 aimed to select implementations for ADTs in an automatic manner, given some efficiency declarations, properties and constraints, to define the Modula-NF programming language. As efficiency criteria, we have chosen the big-Oh (short, O) asymptotical notation [Knu76, Bra85], which is a classical one for data structures. O notation describes program efficiency by means of sets of functions which range on some measure units that represent the size of the information processed. Its definition as a set of functions is: given a function \( f: \mathcal{N}^+ \rightarrow \mathcal{N}^+ \), we define \( O(f) \) as:

\[
O(f) = \{ g: \mathcal{N}^+ \rightarrow \mathcal{N}^+ / \exists c_0 \in \mathcal{R}^+, \exists n_0 \in \mathcal{N}^+: \forall n \geq n_0: \text{c}_0 f(n) \geq g(n) \},
\]

being \( \mathcal{N}^+ \) and \( \mathcal{R}^+ \) the domain of positive natural and real numbers, respectively. So, \( O(f) \) catches all those functions which are an asymptotic bound for \( f \). As program efficiency is measured with a set, inclusions, memberships, unions and intersections are well-defined. Also, two other useful operations, sums and products, are defined in the following way:

\[
O(f) + O(g) = O(f + g) = O(\text{max}(f, g)), \text{being max}(f, g)(n) = \text{max}(f(n), g(n))
\]
\[
O(f) \cdot O(g) = O(f \cdot g), \text{being } (f \cdot g)(n) = f(n) \cdot g(n)
\]

From these operations and given some heuristic rules to analyse programs and to infer their efficiency in terms of the \( O \) notation, some magnitude orders arise often, which are represented by the following functions: \( O(1) \), constant; \( O(\log n) \), logarithmic; \( O(n) \), linear; \( O(n \log n) \), almost-linear; \( O(n^2) \), square; \( O(n^3) \), cube; \( O(k^n) \), for a given constant \( k \), exponential (usually,
the concrete value of \( k \) is not relevant). Non-exponential orders are named polynomial orders in general.

Once the \( O \) notation has been presented, let's focus on its use in the Modula-NF programming language. We are interested in measuring which is the space occupied for data structures implementing types, which is the time required to execute a function and which is the auxiliary space required for a function during its execution. In section 7, other candidates are proposed as future work, as well as the goal of allowing designers to define their own criteria.

Before addressing to efficiency, three other differences between Modula-2 and Modula-NF must be pointed out. First, to enhance programming with ADTs, global variables are forbidden. Second, different implementations for the same ADT must be encapsulated in files with different names to ease later implementation selection; furthermore, their name must differ from the corresponding definition file name. The linking between an implementation file and its corresponding definition one is made with an explicit declaration in the header line of implementations:

```
IMPLEMENTATION MODULE implem-id IMPLEMENTS definition-id;
```

The name of the files containing the modules must agree with the name of the modules themselves, with extensions .DEFth and .MODth. Hereafter, we call these files definition files and implementation files, or even definition and implementations as a shorthand, respectively.

Last, every definition \( D \) should introduce a new ADT at most, whose representation must not appear in \( D \), because more than one implementation could exist for it; so, we require types to be declared as private and then their representation should always appear in implementations (as a pointer in the general case, because of Modula-2 restrictions on private types).

The additional language constructs concerning efficiency may be grouped into three different classes: declaration of the efficiency parameters (hereafter, measure units) for measuring an ADT, efficiency results for a given implementation and efficiency requirements inside a given implementation. In all of them, the built-in Modula-NF predicates \textsc{TIME} and \textsc{SPACE} are used for referring to the time and the space wasted by types and functions.

### 2.1 Declaration of Measure Units

Each measure unit of a given ADT should be declared in a single statement in its definition, after the exported symbols of the module, unless it has been already declared in any imported definition. This is to say, measure units propagate in the hierarchy of modules of a program.

The reason to declare measure units in definitions is to inform both the users and the implementers of the type which are the magnitudes that determine the efficiency of the implementation. Typically, they will be the number of elements to be stored in a data structure implementing the ADT. For legibility purposes, the identifiers used in this declaration may collide with identifiers of constants, types or functions of the type; this coincidence will not affect the compilation of the program, as far as efficiency constructs will disappear in a pre-process step (see section 6).

When more than a single measure unit is known in the scope of a module, either by declaration in this module or by declaration in an imported one, relationships between them may be stated in a different clause. As explained in section 4, there must exist relationships enough to fix the value of all measure units in terms of just one of them; this is why properties may be stated not
only on definition modules but also in the main program. Each relationship is a boolean expression involving Modula-2 boolean and relational operators (in a slight abuse of notation, the operation symbols on sets associated to the O notation are translated to relational symbols in Modula-NF), plus sums and products, powers and logarithms, and the new relational operators '<<' and '>>', to state that a measure unit is negligible with respect to another one. The constants in this notation are the measure units themselves, together with 1 to identify O(1) cost and EXP to identify exponential cost. Subsection 4.1 gives a formal specification of these operators.

Hereafter, we give the name of NF-expressions to those Modula-NF expressions mentioned above; when necessary, we can be more explicit and call them boolean, arithmetic or relational NF-expressions.

Last, efficiency of program schemes may be declared in the definition. Program schemes are different ways of using ADTs by means of their operations; so, their cost will usually be an arithmetic expression involving operations of the type. However, the cost of an scheme may sometimes not be determined in the definition; then, it could be left undefined and it must be fixed in each implementation module for the type.

```
<def module> ::= DEFINITION MODULE id ';'
    <import> <export> <declarations>
    <measure> <properties> <schemes>
    END id '.

<measure> ::= MEASURES <lid> ';'
<properties> ::= PROPS <costbool> ';'
<costbool> ::= <costbool> [ '!' <costbool> ]
<costbool> ::= ... boolean NF-expressions with: NOT, AND, OR,
            1, EXP, +, *, pot, log, <, >, <=, >=, <, >, =, <>
<schemes> ::= SCHEMES <ischeme> ';'
<ischeme> ::= <ischeme> [ ';' <ischeme> ]
<ischeme> ::= (TIME | SPACE) '(' <id> ')' [ '=' <arithNFexpr> ]
<arithNFexpr> ::= ... arithmetic NF-expressions with:
            1, EXP, +, *, pot, log, TIME, SPACE
```

For instance, we present the definition for lists of *elems*. There is a new measure unit *n*, the number of elements in the list; also, the measure unit *max* for elements' size is implicitly imported. The property states that the size of elements is negligible with respect to the size of lists. A program scheme to traverse lists is defined, with a time efficiency given in terms of list operations; another scheme is introduced to traverse lists in order, without any information about its cost neither in time nor in space.
DEFINITION MODULE Lists;
    FROM Elems IMPORT ...;
    ...
    MEASURES n; (* n: number of elements *)
    PROPS n >> Elems.max;
    SCHEMES
        TIME(traversal) = TIME(ListReset) + TIME(ListEnd) +
                          n*(TIME(ListGet)+TIME(ListNext));
        SPACE(ord_traversal); TIME(ord_traversal);
    END Lists;

2.2 Efficiency Results for a Given Implementation

Each implementation module for an ADT should state the time and space costs of their exported types and functions in a separate area, previous to the real implementation. In the case of types, just the space is declared; in the case of functions, both the execution time and the auxiliary space (this is, not including neither the space for the parameters nor the space for the result). Symbols with the same efficiency results may be grouped in the same declaration.

<imp module> ::= IMPLEMENTATION MODULE id IMPLEMENTS id ';'
    <import> <eff. results> <declarations> <ops>
    END id ';
<eff. results> ::= EFFICIENCY <eff. list> ';'
<eff. list> ::= <eff. res> [ ';' <eff. list> ]
<eff. res> ::= (SPACE | TIME) ('<'lid>'') '=' <arithNFexpr>

Note that the cost of every scheme introduced in the definition module is implicitly defined from the cost of the ADT operations, unless no arithmetic expression were attached to it. In this case, the time and/or the space of these schemes have to be explicitly given too.

For instance, the cost for an implementation for lists with an unlinked array may look as:

EFFICIENCY SPACE(list) = n;
    TIME(ListCreate) = 1; TIME(ListAdd, ListRemove) = n;
    TIME(ListReset, ListGet, ListNext, ListEnd) = 1;
    TIME(ord_traversal) = n * log(n); SPACE(ord_traversal) = n;

By default, auxiliary space for functions is $O(1)$. 
Note that, up to now, efficiency results are given by constant NF-expressions. However, the cost for a new type or function could depend on the cost of other types and functions declared in any used module (see next subsection). Even more, the cost may be left unspecified for some types and/or functions. These two variations are not recommended when defining elementary ADTs like lists above, but they may arise when building enrichments of types.

2.3 Efficiency Requirements on Types and Functions

Given the language constructs introduced up to now, it is possible to fulfil efficiency requirements expressed with a suitable notation in the appropriate contexts.

Efficiency requirements are stated over definition modules with a boolean NF-expression or else by directly writing the name of the candidate implementations. The main difference between both kinds of binding is their temporal nature: binding by name is static, this is to say, neither changes on the efficiency of the selected implementations nor creation of new implementations for the type affect the result, while binding by expressions behaves just the other way round.

Note that, in both cases, more than one implementation may fit to a given efficiency constraint or, on the other hand, there may not exist any implementation satisfying the requirement. As far as there should exist at the end a single implementation associated to every variable in the program, it is supposed that other efficiency requirements stated in other points of the program will solve this ambiguity; if it is not the case, requirements would have to be reviewed.

Boolean NF-expressions for requirements may include new efficiency operators: OPS and TYPES, which may be applied to the name of a definition module, returning the set of exported operations or types, respectively; and MIN, to select the implementation with lowest time or space cost for a type, function or scheme, provided that an optional condition is satisfied.

```
<eff. reqs.> ::= REQUIRES <lreqs> ;
<lreqs> ::= <req> [ ',' <lreqs> ]
<req> ::= ON id ':' <boolNFexprwithIMPLEWITH>
<boolNFexprwithIMPLEWITH> ::= ...

... boolean NF-expression with IMPLEMENTED WITH, OPS, TYPES and MIN
```

For example, an efficiency requirement over lists may be: "the cost of individual operations must be constant and traversals must no exceed linear time; furthermore, auxiliary space must be constant for all operations; last, ordered traversal should be as fast as possible, provided that it does not exceed linear auxiliary space". The equivalent Modula-NF expression is:
Efficiency requirements can be stated at different points of implementations.

- Module level. The requirements must be satisfied in all the types and functions inside the implementation. They appear following the EFFICIENCY construct.

```plaintext
<imp module> ::= ... <eff. results> <eff. reqs.>
```

- Type representation level. The requirements must be satisfied for all the components inside the declaration.

```plaintext
<type repr> ::= TYPE <eff. reqs.> <type declarations>
```

- Procedure code level. The requirements must be satisfied for all the objects (variables and parameters) and procedure calls appearing inside the procedure.

```plaintext
<proc impl> ::= PROCEDURE id '(' <params> ')'; <eff. reqs.> <body>
```

- Type representation component level. The requirements must be satisfied just for this component inside the type representation.

```plaintext
<array> ::= ARRAY ['...'] OF id <eff. reqs.>
```

- Object level. The requirements must be satisfied just for this object (variable or parameter) in the procedure where the object is introduced.

```plaintext
<object designator> ::= <did> 'id <eff. reqs.>
```

- Procedure call level. The requirements must be satisfied just for this procedure call.

```plaintext
<proc call> ::= <designator> <actual params> <eff. reqs.>
```

In the last three cases, two abbreviations are allowed. First, the definition name may be omitted, because it may be inferred from the unit being restricted. Second, in case of IMPLEMENTED WITH restriction with a single implementation name $M$, the unit $x$ being restricted may appear as $M.x$.

Restrictions in each level are considered to constrain restriction in upper ones, which could cause contradictions or ambiguities. This problem is addressed in section 3.
2.4 Examples

2.4.1 Tables

There is a first example for tables, that is an ADT mapping keys to values. Tables may be accessed individually to add pairs key-value, to delete pairs or to get the value associated to a key; also, ordered traversal are defined by means of four operations. To simplify details, both the key and the value are supposed to be of a given type message; messages do have their own efficiency characteristics, which are expressed with respect to a measure unit \( n \) that models message size. Concerning efficiency of tables, there is a single measure unit and a scheme to model ordered traversals, both in time and space. This measure is compared to the one from Message.

Implementations for tables should give different efficiency results. We present four strategies: implementation by an ordered list, by an AVL binary search tree, by hashing and by a mixed hashing-list structure. In every case, efficiency results are stated. Requirements on messages are identical in all the cases: additional space on operations is required to be constant. Note that, with these requirements and the property stated in the definition modules, the cost of all table operations is fully determined in the four implementation modules.

Next, many ways of enriching tables are presented. First of all, we outline three different main programs. The first one, Dictionary, is designed to support efficient access to a dictionary represented with a table; individual queries must be optimised as far as they will be the most usual operations; additions are also required to be not too slow. On the other hand, the second program, ListOfPeople, is intended to provide alphabetically ordered lists of people in a wide community and thus this is the optimisation criteria together with no linear additions. Note that, in both cases, variable implementations are implicitly selected because of module level restrictions. Last, we give two different ways of combining both structures in the same main program; the first one declares two different variables while the second one creates a new, private type with two different components. Requirements are not stated at module level but at object and type component level, respectively.

As a different enrichment, a new ADT is defined over tables by adding set operations on them. The measure unit is not explicitly given, since it is known from Table. In this case, we give two different implementations. The first one states exactly the efficiency of these operations requiring some particular efficiency constraints on table implementation. The second one gives no hints about table implementations; so, it is not possible to determine the actual efficiency of the new operations, which is left in terms of the efficiency of the table implementation selected for the type.
DEFINITION MODULE Table;
FROM Message IMPORT message...
EXPORT QUALIFIED table, TableCreate, TableAdd, TableDelete, TableGet,
        TableFirst, TableNext, TableEnd;
TYPE table;
PROCEDURE TableCreate (VAR t: table);
PROCEDURE TableAdd (VAR t: table; key, info: message);
PROCEDURE TableDelete (VAR t: table; key: message);
PROCEDURE TableGet (t: table; key: message): message;
PROCEDURE TableFirst (VAR t: table; VAR key, info: message);
PROCEDURE TableNext (VAR t: table; VAR key, info: message);
PROCEDURE TableEnd (t: table): BOOLEAN;
(* Efficiency information area *)
MEASURES n; (* n: number of elements in the table *)
PROPERTIES n >> Message.n; (* Message.n is neglected when compared to Table.n *)
SCHEMES TIME(ord_traversal) = TIME(TableFirst) + n*TIME(TableNext) + TIME(TableEnd);
        SPACE(ord_traversal) = SPACE(TableFirst) + SPACE(TableNext) + SPACE(TableEnd);
END Table.

IMPLEMENT MODULE TableList FOR Table;
FROM Message IMPORT message...
EFFICIENCY
SPACE(table) = n; TIME(TableCreate) = 1;
TIME(TableAdd, TableDelete, TableGet) = n;
TIME(TableFirst, TableNext, TableEnd) = 1;
REQUIRES ON Message:
SPACE(OPS(Message)) = 1;
... implementation using a linked ordered list
END TableList.

IMPLEMENT MODULE TableUnordHash FOR Table;
FROM Message IMPORT message...
EFFICIENCY
SPACE(table, TableFirst) = n;
TIME(TableAdd, TableDelete, TableGet) = 1;
TIME(TableFirst) = n * log(n);
TIME(TableNext, TableEnd) = 1;
TIME(TableCreate) = n
REQUIRES ON Message:
SPACE(OPS(Message)) = 1;
... implementation using a hash table, ordering the elements with links when starting traversals
END TableUnordHash.

IMPLEMENT MODULE TableAVL FOR Table;
FROM Message IMPORT message...
EFFICIENCY
SPACE(table) = n;
TIME(TableCreate) = n;
TIME(TableAdd, TableDelete, TableGet) = log(n);
TIME(TableFirst, TableNext) = log(n);
REQUIRES ON Message:
SPACE(OPS(Message)) = 1;
... implementation using AVL trees
END TableAVL.

IMPLEMENT MODULE TableOrdHash FOR Table;
FROM Message IMPORT message...
EFFICIENCY
SPACE(table) = n;
TIME(TableCreate) = n;
TIME(TableAdd) = n;
TIME(TableDelete, TableGet) = 1;
TIME(TableFirst, TableNext, TableEnd) = 1;
REQUIRES ON Message:
SPACE(OPS(Message)) = 1;
... implementation using a hash table, ordering the elements with (double) links as they are added
END TableOrdHash.
### MODULE Dictionary
- FROM Table IMPORT ...
- REQUIRES ON Table:
  - TIME(TableAdd) < n AND TIME(TableGet) = 1;
- VAR d: table;
- ...
- END Dictionary.

### MODULE ListOfPeople
- FROM Table IMPORT ...
- REQUIRES ON Table: TIME(ord_traversal) <= n AND TIME(TableAdd) < n;
- VAR sl: table;
- ...
- END ListOfPeople.

### MODULE ListOfPeopleWithDictionary1
- FROM Table IMPORT ...
- VAR d: table REQUIRES TIME(TableAdd) < n AND TIME(TableGet) = 1;
- sl: table REQUIRES TIME(ord_traversal) <= n AND TIME(TableAdd) < n;
- ...
- END ListOfPeopleWithDictionary1.

### MODULE ListOfPeopleWithDictionary2
- FROM Table IMPORT ...
- TYPE AllInfo = RECORD
  - d: table REQUIRES TIME(TableAdd) < n AND TIME(TableGet) = 1;
  - sl: table REQUIRES TIME(ord_traversal) <= n AND TIME(TableAdd) < n;
- END;
- VAR a: AllInfo;
- ...
- END ListOfPeopleWithDictionary2.

### DEFINITION MODULE SetTable
- FROM Table IMPORT table;
- EXPORT QUALIFIED TableUnion, TableIntersection, TableDifference;
- PROCEDURE TableUnion (t1, t2: table; VAR t3: table);
- PROCEDURE TableIntersection (t1, t2: table; VAR t3: table);
- PROCEDURE TableDifference (t1, t2: table; VAR t3: table);
- END SetTable.

### IMPLEMENTATION MODULE SetTableImp1
- FROM Table IMPORT ...;
- EFFICIENCY TIME(TableUnion, TableIntersection, TableDifference) = n;
- REQUIRES ON Table:
  - TIME(ord_traversal) = n AND TIME(TableCreate) <= n AND TIME(TableAdd) = 1;
- ...
- END SetTableImp1.

### IMPLEMENTATION MODULE SetTableImp2
- FROM Table IMPORT ...;
- EFFICIENCY TIME(TableUnion, TableIntersection, TableDifference) =
  - TIME(ord_traversal) + TIME(TableCreate) + n*TIME(TableAdd);
- ...
- END SetTableImp2.
2.4.2 Directed graphs

Next we address to the ADT of directed, labelled graphs. There exists a set of nodes and a set of edges, which are viewed as pairs of nodes with labels; to simplify matters, we suppose that nodes are from cardinal type and that labels are integer numbers. Operations on graphs are: creation of a graph with all its nodes and with no edges; addition and deletion of individual edges; querying the label on an edge; and obtaining the list of successors of a given node together with their labels. List of pairs of cardinals and integers are supposed to exist with a definition including a measure unit \( n \) for list size. There are two measure units for graphs, the number of edges and the number of nodes, related with a boolean expression; as an additional property, the (maximum) size of a list is fixed to be the number of nodes. A scheme is defined to visit all the edges in a graph, which is a task to be performed in various classical graph algorithms; its efficiency is left undefined.

Two usual implementations for graphs are presented. Types are declared as pointers because their representation is not public. *AdjacencyMatrix* uses a two-dimension array to store labels; so, the cost of the operations and the type are straightforward. Note that list implementation requirements are global to the module but, in fact, they could have been declared at procedure *GraphSucc* level, because it is the only place where lists appear. This is not true in *AdjacencyLists*, because there are two different lists, the list of successors attached to each node and the list to be built by *GraphSucc*, and their requirements are not the same: adjacency lists are explicitly obliged to be implemented with pointers, while the other lists may be implemented with any module satisfying the same requirements as before. As a general requirement, operations on list are required to have constant extra space, and so this restriction is stated at module level (and this is why the first conjunction on procedure level restrictions of list could be omitted).

As an enrichment, procedures for solving the all-pair shortest path problem are outlined. The procedure is defined in a .DEF file, placing the result in a two-dimension array indexed by nodes (to simplify details, we just focus on the labels of the path, not on its component nodes). To implement it, we use the algorithms of Floyd and (iterated) Dijkstra. Floyd algorithm does not depend neither on the implementation nor on the appearance of graphs (that is, the relationship between \( e \) and \( n \)); so, the only requirement is to obtain all the labels in no more than cube time on the number of nodes, which is a condition satisfied for both implementations (and, probably, for any other representation of graphs we can think about). On the other hand, Dijkstra do depend on both the implementation and the appearance of graphs. Concerning implementation, Dijkstra algorithm is always \( O(n^2) \) in time when the graph is implemented with adjacency matrix; furthermore, auxiliary space is \( O(n) \) to mark treated nodes. When the graph is implemented with adjacency lists, a more sophisticated version exists that spends \( O((e+n) \log n) \) in time and \( O(e + n) \) in auxiliary space (using a heap to organise non-definitive shortest paths, which implementation should probably meet other efficiency criteria not shown here). Clearly, this cost is better than the former one only when the number of edges is less than \( O(n^2) \); so, this version states this relationship between \( e \) and \( n \) as a requirement, and then the resulting repeated Dijkstra algorithm stands \( O(n(e+n) \log n) \). Note that, to be more general, graphs could be required not to be implemented by adjacency lists exclusively, but with any implementation with an all-edge visit time of \( O(e) \).
Finally, three different main programs using the all-pair shortest path procedure are sketched. The first one fixes the implementation with adjacency matrix, which discards Dijkstra implementation with heap to fulfill the requirement on apsp. To break the tie, the third conjunction of the requirement constraint picks up Floyd implementation. On the other hand, the second program requires implementation with adjacency lists; so, two solutions are possible depending on the relationships between \( n \) and \( e \) and the user will have to supply more information to select the correct implementation (for instance, stating as a property that \( n = e \) if the graph is sparse, or \( e = \text{pot}(n, 2) \) if it is dense). Finally, the third main program directly picks up an implementation for the apsp procedure; note that the implementation for graphs may not be inferred from this particular apsp implementation.

The algorithm that computes implementation selection is presented in section 4.

```
DEFINITION MODULE Graph;
    FROM ListCardInt IMPORT list;
    EXPORT QUALIFIED graph, GraphCreate, GraphAdd, GraphDelete, GraphLabel, GraphSucc;
    CONST n = 1000;  (* this declaration of \( n \) is not relevant to the study of efficiency *)
    TYPE graph;
    PROCEDURE GraphCreate(VAR g: graph);
    PROCEDURE GraphAdd(VAR g: graph; u, v: CARDINAL; et: INTEGER);
    PROCEDURE GraphDelete (VAR g: graph; u, v: CARDINAL);
    PROCEDURE GraphLabel (g: graph; u, v: CARDINAL): INTEGER;
    PROCEDURE GraphSucc (VAR g: graph; u: CARDINAL; VAR l: list);
    (* Efficiency information area *)
    MEASURES n, e;  (* n: number of nodes; e: number of edges *)
    PROPS e <= \( \text{pot}(n, 2) \); ListCardInt.n = n;
    SCHEMES TIME(AllEdgesVisit);
END Graph.
```
IMPLEMENTATION MODULE AdjacencyMatrix FOR Graph;
    FROM ...;
    EFFICIENCY
    SPACE(graph) = pot(n, 2); TIME(GraphCreate) = pot(n, 2);
    TIME(GraphAdd, GraphDelete, GraphLabel) = 1;
    TIME(GraphSucc) = n; TIME(AllEdgesVisit) = pot(n, 2);
    REQUIRES ON List:
    SPACE(ListCreate, ListAdd) = 1 AND TIME(ListCreate) <= n AND TIME(ListAdd) = 1;
    (* begin of Modula-2 implementation *)
    TYPE graph = POINTER TO privgraph; privgraph = ARRAY [1..n, 1..n] OF INTEGER;
    ...
END AdjacencyMatrix.

IMPLEMENTATION MODULE AdjacencyLists FOR Graph;
    FROM ...;
    EFFICIENCY
    SPACE(graph) = c+n; TIME(GraphCreate) = n;
    TIME(GraphAdd, GraphDelete, GraphLabel, GraphSucc) = n;
    TIME(AllEdgesVisit) = c+n;
    REQUIRES ON List: SPACE(OPS(List)) = 1;
    (* begin of Modula-2 implementation *)
    TYPE
    REQUIRES On List: IMPLEMENTED WITH ListCardIntByPointers;
      graph = POINTER TO privgraph;
      privgraph = ARRAY [1..n] OF list;
    ...
    PROCEDURE GraphSucc (VAR g: graph; u: CARDINAL; VAR l: list);
    REQUIRES ON List:
    SPACE(ListCreate, ListAdd) = 1 AND TIME(ListCreate) <= n AND TIME(ListAdd) = 1;
    BEGIN
      ListCreate(0); ...
    END GraphSucc;
    ...
END AdjacencyLists.
DEFINITION MODULE AllPairsShortestPath;
   FROM Graph IMPORT ...;
   EXPORT QUALIFIED asp;
   PROCEDURE asp(g: graph; VAR res: ARRAY [1..n, 1..n] OF INTEGER);
END AllPairsShortestPath.

IMPLEMENTATION MODULE Floyd FOR AllPairsShortestPath;
   FROM Graph IMPORT ...;
   EFFICIENCY TIME(asp) = pot(n, 3);
   REQUIRES ON Graph: TIME(AllEdgesVisit) <= pot(n, 3);
   (* begin of Modula-2 implementation *)
   PROCEDURE asp(g: graph; VAR res: ARRAY [1..n, 1..n] OF INTEGER);
   BEGIN
      ...
      (* initialise res with edge labels in no more than O(n^3) time *)
      ...
      (* perform Floyd iteration in exactly O(n^3) time *)
   END asp;
   ...
END Floyd.

IMPLEMENTATION MODULE Dijkstra FOR AllPairsShortestPath;
   FROM Graph IMPORT ...;
   EFFICIENCY TIME(asp) = pot(n, 3); SPACE(asp) = n;
   (* begin of Modula-2 implementation *)
   PROCEDURE asp(g: graph; VAR res: ARRAY [1..n, 1..n] OF INTEGER);
   BEGIN
      ...
   END asp;
   ...
END Dijkstra.

IMPLEMENTATION MODULE DijkstraWithHeap FOR AllPairsShortestPath;
   FROM Graph IMPORT ...;
   EFFICIENCY TIME(asp) = (c+n)*n*log(n); SPACE(asp) = c+n;
   REQUIRES ON Graph: IMPLEMENTED WITH AdjacencyLists AND (c+n)*log(n) <= pot(n, 2);
   (* begin of Modula-2 implementation *)
   PROCEDURE asp(g: graph; VAR res: ARRAY [1..n, 1..n] OF INTEGER);
   BEGIN
      ...
   END asp;
   ...
END DijkstraWithHeap.
MODULE Main1;
  FROM ...;
  REQUIRES ON Graph:
    MIN(TIME(apsp)) AND IMPLEMENTED WITH AdjacencyMatrix AND
    MIN(SPACE(apsp));
...
END Main1.

MODULE Main2;
  FROM ...;
  REQUIRES ON Graph: MIN(TIME(apsp)) AND IMPLEMENTED WITH AdjacencyLists;
...
END Main2.

MODULE Main3;
  FROM ...;
  REQUIRES ON AllPairsShortestPath: IMPLEMENTED WITH Floyd;
...
END Main3.
3 Associating Implementations to Objects and Procedure Calls

As mentioned in the previous section, the statement of efficiency requirements at many different levels could cause conflicts and ambiguities if some rules were not defined to avoid them. In this section, a set of rules is given to associate the appropriate implementations to objects and procedure calls from efficiency requirements stated in a hierarchy of modules; as a result, implementations may really be selected not by name but from their efficiency characteristics.

Before stating the rules, some syntactic domains and functions are introduced to provide a comfortable framework.

3.1 Syntactic Domains and Syntactic Functions

Syntactic elements are classified with respect to efficiency requirement levels. Domains are defined as sets. Functions are declared as higher-order ones in the general case; properties on them are stated, usually to determine their domain and/or rank. Their concrete values in a given hierarchy of modules is computed by means of C code associated to grammar rules in a Yacc description.

3.1.1 Module Level

The basic syntactic domains are those ones for definition and implementation modules. We include on them a pair of modules, MODULA2.DEFth and MODULA2.IMPth, to model the definition and implementation of predefined Modula-2 features.

\[
\text{DefModules}_H = \{ U_d \mid U_d \text{ is a definition in a hierarchy } H \} \cup \{ \text{MODULA2.DEFth} \}
\]

\[
\text{ImplModules}_H = \{ U_i \mid U_i \text{ is an implementation in a hierarchy } H \} \cup \{ \text{MODULA2.IMPth} \}
\]

Relationships between both domains are kept with a pair of syntactic functions mapping definition modules to implementations ones and vice versa. Many trivial properties are stated.

\[
\text{DefToImpl}_H : \text{DefModules}_H \to \exists \text{ImplModules}_H
\]

\[
\text{DefToImpl}_H(U_d) = \{ U_i \} \mid \{ U_i \} \text{ is the set of already existent implementations for } U_d \text{ in } H
\]

\[
\text{ImplToDef}_H : \text{ImplModules}_H \to \exists \text{DefModules}_H
\]

\[
\text{ImplToDef}_H(U_i) = U_d \mid U_i \text{ is an implementation for } U_d \text{ in } H
\]

P1 \text{DefToImpl}_H and \text{ImplToDef}_H are total functions.

P2 An implementation must be referred to a single definition.

\[
\forall U_d, U_d' : U_d, U_d' \in \text{DefModules}_H:\n\]

\[
U_d \neq U_d' \Rightarrow \text{DefToImpl}_H(U_d) \cap \text{DefToImpl}_H(U_d') = \emptyset
\]
P3 Functions DefToImplH and ImplToDefH are each one the inverse of the other.

\[ \forall U_d: U_d \in \text{DefModules}_H: [\forall U_i: U_i \in \text{DefToImpl}_H(U_d): \text{ImplToDef}_H(U_i) = U_d] \land \\
\land \forall U_i: U_i \in \text{ImplModules}_H: [U_i \in \text{DefToImpl}_H(\text{ImplToDef}_H(U_i))] \]

P4 MODULA2.IMPth is the only module implementing MODULA2.DEFth.

\[ \text{DefToImpl}_H(\text{MODULA2.DEFth}) = \{ \text{MODULA2.IMPth} \} \land \\
\land \text{ImplToDef}_H(\text{MODULA2.IMPth}) = \text{MODULA2.DEFth} \]

Finally, restrictions at module level are considered. First of all, a function to record imported modules is introduced. Then, a second function keeps track of which modules are restricted at module level. This function serves as domain of the goal one, that points out the set of implementation modules for which the definition is restricted to. Properties on these functions are referred to their valid domains and ranks, while the problem of determining this set from some given efficiency requirements is addressed in section 4.

ImportedModulesH: ImplModulesH → 2(DefModulesH)

ImportedModulesH(U_i) = \{ U_d \} \setminus \{ U_d \} is the set of definitions imported by U_i

RestrictedModulesH: ImplModulesH → 2(DefModulesH)

RestrictedModulesH(U_i) = \{ U_d \} \setminus \{ U_d \} are restricted at module level in U_i

RestrictionsH: ImplModulesH → DefModulesH → 2(ImplModulesH)

RestrictionsH(U_i)(U_d) = \{ U'_i \} \setminus \{ U'_i \} is the set of implementations satisfying the restrictions stated at module level for U_d in U_i

P5 All restricted modules must be previously imported.

\[ \forall U_i: U_i \in \text{ImplModules}_H: \text{RestrictedModules}_H(U_i) \subseteq \text{ImportedModules}_H(U_i) \]

P6 Restrictions are stated with respect to restricted modules.

\[ \forall U_i: U_i \in \text{ImplModules}_H: \text{dom}(\text{Restrictions}_H(U_i)) = \text{RestrictedModules}_H(U_i) \]

P7 Restrictions over a definition must refer to its valid implementations.

\[ \forall U_i: U_i \in \text{ImplModules}_H: \\
\forall U_d: U_d \in \text{RestrictedModules}_H(U_i): \\
\text{Restrictions}_H(U_i)(U_d) \subseteq \text{DefToImpl}_H(U_d) \]
3.1.2 Type Representation Level

First of all, we introduce a syntactic domain for types, including standard ones from Modula-2. Associated to it, a function mapping from types to definition modules introducing them is defined; note that this function is partial, because auxiliary types in implementations are not bound to any definition.

\[ \text{Types}_H = \{ t \mid t \text{ is a type introduced in any module in } H \} \]

**P8** Modula-2 predefined, non-structured types are implicitly declared in \( H \).

\[ (\text{INTEGER, CARDINAL, REAL, BOOLEAN, CHAR, BITSET}) \subseteq \text{Types}_H \]

\[ \text{TypeDef}_H : \text{Types}_H \rightarrow \text{DefModules}_H \]

\[ \text{TypeDef}_H(t) \equiv \text{definition module that defines } t \]

**P9** Modula-2 predefined, non-structured types are declared in MODULA2.DEFth.

\[ \text{TypeDef}_H(\text{INTEGER}) = \text{TypeDef}_H(\text{BITSET}) = \text{TypeDef}_H(\text{REAL}) = \text{TypeDef}_H(\text{CHAR}) = \]

\[ = \text{TypeDef}_H(\text{BOOLEAN}) = \text{TypeDef}_H(\text{CARDINAL}) = \text{MODULA2.DEFth} \]

Then, types are classified into three (non-disjunct) sets depending on their import/export characteristics. Two other sets are defined later on them (but only in implementations). Many integrity properties are stated to ensure implementation coherence.

**P10** \( \text{TypeDef}_H \) domain is the union of all types introduced in definition modules.

\[ \text{dom}(\text{TypeDef}_H) = \bigcup U_d : U_d \in \text{DefModules}_H : \text{NewTypesDef}_H(U_d) \]

**P11** \( \text{NewTypesDef}_H \) and \( \text{TypeDef}_H \) should be coherent.

\[ \forall U_d : U_d \in \text{DefModules}_H : \forall t : t \in \text{dom}(\text{TypeDef}_H) : t \in \text{NewTypesDef}_H(U_d) \iff \text{TypeDef}_H(t) = U_d \]
P12 A module can only export types either imported or introduced as new.

\[ \forall U_d: U_d \in \text{DefModules}_H: \]
\[ \text{ExportTypesDef}_H(U_d) \subseteq \text{ImportTypesDef}_H(U_d) \cup \text{NewTypesDef}_H(U_d) \land \]
\[ \land \text{ImportTypesDef}_H(U_d) \cap \text{NewTypesDef}_H(U_d) = \emptyset \]

P13 Types imported by a module are the union of those types exported by its imported modules.

\[ \forall U_d: U_d \in \text{DefModules}_H: \]
\[ \text{ImportTypesDef}_H(U_d) = \]
\[ = \bigcup U_d': U_d' \in \text{ImportedModules}_H(U_d): \text{ExportTypesDef}_H(U_d') \]

being \( \text{ImportedModules}_H: \text{DefModules}_H \rightarrow \mathcal{P}(\text{DefModules}_H) \) defined as the already introduced function \( \text{ImportedModules}_H: \text{ImpModules}_H \rightarrow \mathcal{P}(\text{DefModules}_H) \)

P14 All types declared as new in a definition should be exported.

\[ \forall U_d: U_d \in \text{DefModules}_H: \text{NewTypesDef}_H(U_d) \subseteq \text{ExportTypesDef}_H(U_d') \]

P15, P16 As P12, P13 but referring to implementation modules.

P17 All implementations export the same types as their definition.

\[ \forall U_d: U_d \in \text{DefModules}_H: \]
\[ \forall U_i: U_i \in \text{DefToImpl}_H(U_d): \text{ExportTypesImpl}_H(U_i) = \text{ExportTypesDef}_H(U_d) \]

P18 All implementations introduce the same new public types as their definition.

\[ \forall U_d: U_d \in \text{DefModules}_H: \]
\[ \forall U_i: U_i \in \text{DefToImpl}(U_d)_H: \]
\[ \text{NewTypesImpl}_H(U_i) \cap \text{ExportTypesImpl}_H(U_i) = \text{NewTypesDef}_H(U_d) \]

P19 All implementations import the same types as their definition, at least.

\[ \forall U_d: U_d \in \text{DefModules}_H: \]
\[ \forall U_i: U_i \in \text{DefToImpl}(U_d)_H: \]
\[ \text{ImportTypesImpl}_H(U_i) = \text{ImportTypesDef}_H(U_d) \]
P20 A public type may not be defined more than once.

\[ \forall U_d, U'_d: U_d, U'_d \in \text{DefModules}_H: \]
\[ U_d \neq U'_d \Rightarrow \text{NewTypesDef}_H(U_d) \cap \text{NewTypesDef}_H(U'_d) = \emptyset \]

All\text{TypesImpl}_H, PrivateTypesImpl\text{Impl}_H: \text{ImplModules}_H \rightarrow \mathcal{P}(\text{Types}_H)

All/PrivateTypesImpl\text{Impl}_H(U_i) \equiv \{ t \} introduced/introduced as private in \text{U}_i

P21 These sets are computable from the ones above.

\[ \text{AllTypesImpl}_H(U_i) = \text{NewTypesImpl}_H(U_i) \cup \text{ImportTypesImpl}_H(U_i) \]
\[ \text{PrivateTypesImpl}_H(U_i) = \text{NewTypesImpl}_H(U_i) - \text{ExportTypesImpl}_H(U_i) \]

Last, restrictions at type representation level are taken into account. The strategy is the same as in module level: a function to fix the domain and another one to record the set of selected implementations.

RestrictedModulesOnRepr\text{Repr}_H: \text{ImplModules}_H \rightarrow \mathcal{P}(\text{DefModules}_H)

RestrictedModulesOnRepr\text{Repr}_H(U_i) \equiv \{ U_d \} / \{ U_d \} are restricted at type representation level in \text{U}_i

P22 All restricted modules should be previously imported.

\[ \forall U_i: U_i \in \text{ImplModules}_H: \text{RestrictedModulesOnRepr}_H(U_i) \subseteq \text{ImportedModules}_H(U_i) \]

RestrictionsOnRepr\text{Repr}_H: \text{ImplModules}_H \rightarrow \text{DefModules}_H \rightarrow \mathcal{P}(\text{ImplModules}_H)

RestrictionsOnRepr\text{Repr}_H(U_i)(U_d) \equiv \{ U'_i \} / \{ U'_i \} is the set of implementations satisfying the restrictions formulated at type representation level for \text{U}_d in \text{U}_i

P23 Restrictions are formulated with respect to restricted modules.

\[ \forall U_i: U_i \in \text{ImplModules}_H: \]
\[ \text{dom}(\text{RestrictionsOnRepr}_H(U_i)) = \text{RestrictedModulesOnRepr}_H(U_i) \]

P24 Restrictions over a definition must refer to its valid implementations.

\[ \forall U_i: U_i \in \text{ImplModules}_H: \]
\[ \forall U_d: U_d \in \text{RestrictedModulesOnRepr}_H(U_i): \]
\[ \text{RestrictionsOnRepr}_H(U_i)(U_d) \subseteq \text{DefToImp}_H(U_d) \]
3.1.3 Type Component Level

Type component level is more difficult to deal with than the previous ones, because we need a mathematical model to access components inside types. The most straightforward solution is to model type representations with a kind of parse tree (containing only information relevant to the implementation selection problem); then, a classical mathematical model for trees may be used where node positions are accessed by a natural-number string, each natural number referring to a level on the tree and its value being the position with respect to its parent.

More precisely, let $\mathcal{N}_0$ be the domain of positive natural numbers, let $V^*$ be the strings of elements of domain $V$ and let $\text{ExtendedTypes}_H$ be the domain of nodes of the tree, defined as:

$$\text{ExtendedTypes}_H = \text{TypeNames}_H \cup \{\text{RECORD}, \text{ARRAY}, \text{POINTER}\}$$

with RECORD, ARRAY and POINTER as flags for these structures. Suppose also that $\text{TypeDef}_H$ function is extended to $\text{ExtendedTypes}_H$ in its domain, such that $\text{TypeDef}_H(\text{RECORD}) = \text{TypeDef}_H(\text{ARRAY}) = \text{TypeDef}_H(\text{POINTER}) = \text{MODULA2.DEFth}$. Then, we define the model $\text{TypeRepresentation}_H$ as:

$$\text{TypeRepresentation}_H = \{f : \mathcal{N}_0^* \to \text{ExtendedTypes}_H / f \text{ is valid}\}$$

where validity of functions $f : \mathcal{N}_0^* \to \text{ExtendedTypes}_H$ is defined as:

- $f$ is closed under prefix:
  $$\forall s : s \in \mathcal{N}_0^* : \forall k : k \in \mathcal{N}_0 : s \in \text{dom}(f) \Rightarrow s \in \text{dom}(f)$$

- $f$ is compact:
  $$\forall s : s \in \mathcal{N}_0^* : \forall k : k \in \mathcal{N}_0 : s \in \text{dom}(f) \Rightarrow [\forall j : 1 \leq j < k : s \in \text{dom}(f)]$$

- $f$ is correct with respect to Modula-2 type construction rules (a child for array and pointer component type, at least a child for record fields, and no children for user-defined types):
  $$\forall s : s \in \text{dom}(f) :$$
  $$f(s) = \text{ARRAY} \lor f(s) = \text{POINTER} \Rightarrow f(s1) \in \text{dom}(f) \land [\forall k : k > 1 : s \in \text{dom}(f)]$$
  $$\land f(s) = \text{RECORD} \Rightarrow f(s1) \in \text{dom}(f)$$
  $$\land f(s) \neq \text{ARRAY} \land f(s) \neq \text{RECORD} \land f(s) \neq \text{POINTER} \Rightarrow \forall k : k \geq 1 : s \in \text{dom}(f)$$

To manipulate this model, a pair of operations are useful:

- **MakeTypeRepr**
  $$\text{MakeTypeRepr}_H : \text{TypeRepresentation}_H \rightarrow \text{ExtendedTypes}_H \rightarrow \text{TypeRepresentation}_H$$
  $$\text{MakeTypeRepr}_H(f_1, ..., f_n)(i) = f, \text{ defined as:}$$
  $$\text{dom}(f) = 1\#\text{dom}(f_1) \cup ... \cup n\#\text{dom}(f_n) \cup \{\lambda\}, \text{ and}$$
  $$f(\lambda) = t \land \forall k : 1 \leq k \leq n : \forall s : s \in \text{dom}(f) : f(ks) = f_k(s)$$

3 We do not consider neither enumerate nor subrange nor set nor procedure types because they do not have structured component types.
where $\lambda$ is the empty sequence and #: $\mathcal{N}_0 \times \mathcal{A}(\mathcal{N}_0^*) \rightarrow \mathcal{A}(\mathcal{N}_0^*)$ is defined as:

$$n#\{s_1, ..., s_k\} = \{ns_1, ..., ns_k\}$$

- **TypeCompH**: $\text{TypeRepresentation}_H \rightarrow \mathcal{N}_0 \rightarrow \text{TypeRepresentation}_H$
  $$\text{TypeComp}_H(f)(n) = g,$$ defined as:
  $$\text{dom}(g) = \{se \mathcal{N}_0^* \mid \text{ns} \in \text{dom}(f)\},$$ and
  $$\forall s: s \in \text{dom}(g): g(s) = f(\text{ns})$$

Also, an auxiliary function is needed to fix domains to user-defined types in various contexts:

- **UserDefinedTypesH**: $\text{TypeRepresentation}_H \rightarrow \mathcal{A}(\mathcal{N}_0^*)$
  $$\text{UserDefinedTypes}_H(f) = \{s \in \mathcal{N}_0^* \mid s \in \text{dom}(f) \land f(s) \in \text{Types}_H - \{\text{INTEGER, ...}\}\}$$

Last, a way to connect types with their representation inside implementation modules is needed, yielding to a new function.

**TypeToTypeReprH**: $\text{ImplModules}_H \rightarrow \text{Types}_H \rightarrow \text{TypeRepresentation}_H$

$$\text{TypeToTypeRepr}_H(U_i)(t)$$ = representation for $t$ inside $U_i$

**P25** Mapping to type representations is defined on implementation new types.

$$\forall U_i: U_i \in \text{ImplModules}_H: \text{dom}($$\text{TypeToTypeRepr}_H(U_i)) = \text{NewTypesImpl}_H(U_i)$$

Now, restrictions on type components may be stated in the usual way. Note that, given a type component, it only makes sense to restrict the module defining its type and so the function yields not a set but a single definition.

**RestrictedModuleOnTypeCompH**: $\text{ImplModules}_H \rightarrow \text{Types}_H \rightarrow \mathcal{N}_0^* \rightarrow \text{DefModules}_H$

$$\text{RestrictedModuleOnTypeComp}_H(U_i)(t)(s) \equiv U_d / U_d \text{ is restricted at component } s \text{ of type } t \text{ in } U_i$$

**P26** Restricted modules may only appear on newly defined types.

$$\forall U_i: U_i \in \text{ImplModules}_H: \text{dom}($$\text{RestrictedModuleOnTypeComp}_H(U_i)) = \text{NewTypesImpl}_H(U_i)$$

**P27** Only imported user-defined types may be restricted as type components.

$$\forall U_i: U_i \in \text{ImplModules}_H:$$

$$\forall t: t \in \text{NewTypesImpl}_H(U_i):$$

$$s \in \text{dom}($$\text{RestrictedModuleOnTypeComp}_H(U_i)(t)) \Rightarrow$$

$$s \in \text{UserDefinedTypes}_H($$\text{TypeToTypeRepr}_H(U_i)(t)) \land$$

$$\land \text{TypeDef}_H($$\text{TypeToTypeRepr}_H(U_i)(t)(s)) \in \text{ImportedModules}_H(U_i)$$
P28 All restricted type components should refer to its type definition module, which must be an imported one.

\[ \forall U_i \in \text{ImplModules}_H: \]
\[ \forall t : \text{t} \in \text{NewTypesImpl}_H(U_i): \]
\[ \forall s : \text{s} \in \text{UserDefinedTypes}_H(\text{TypeToTypeRepr}_H(U_i)(t)): \]
\[ \text{defined}(\text{RestrictedModuleOnTypeComp}_H(U_i)(t)(s)) \Rightarrow \]
\[ \text{RestrictedModuleOnTypeComp}_H(U_i)(t)(s) = \]
\[ \text{TypeDef}_H(\text{TypeToTypeRepr}_H(U_i)(t)(s)) \]

RestrictionsOnTypeComp\(_H\):

\[ \text{ImplModules}_H \rightarrow \text{Types}_H \rightarrow \mathcal{N}_0^* \rightarrow \text{DefModules}_H \rightarrow \mathcal{P}(\text{ImplModules}_H) \]

RestrictionsOnTypeComp\(_H(U_i)(t)(s)(U_d) \equiv \{ U'_i \} / \{ U'_i \} \) is the set of implementations for the definition \( U_d \) satisfying the restrictions stated for the component \( s \) of type \( t \) in \( U_i \).

P29 Restrictions may only appear on newly defined types.

\[ \forall U_i \in \text{ImplModules}_H: \]
\[ \text{dom}(\text{RestrictionsOnTypeComp}_H(U_i)) = \text{NewTypesImpl}_H(U_i) \]

P30 Restrictions may only appear on restricted type components.

\[ \forall U_i \in \text{ImplModules}_H: \]
\[ \forall t : \text{t} \in \text{NewTypesImpl}_H(U_i): \]
\[ \text{dom}(\text{RestrictionsOnTypeComp}_H(U_i)(t)) = \]
\[ \text{dom}(\text{RestrictedModuleOnTypeComp}_H(U_i)(t)) \]
**P31** All restricted type components should refer to its type definition module.

\[
\forall U_i \in \text{ImplModules}_H:
\forall t \in \text{NewTypesImpl}(U_i)_H:
\forall s \in \text{UserDefinedTypes}_H(\text{TypeToTypeRepr}_H(U_i)(t)):
\text{defined}(\text{RestrictedModuleOnTypeComp}_H(U_i)(t)(s)) \Rightarrow
\text{RestrictionsOnTypeComp}_H(U_i)(t)(s) = \text{TypeDef}_H(\text{TypeToTypeRepr}_H(U_i)(t)(s))
\]

**P32** Restrictions over a definition must refer to its valid implementations.

\[
\forall U_i \in \text{ImplModules}_H:
\forall t \in \text{NewTypesImpl}(U_i)_H:
\forall s \in \text{UserDefinedTypes}_H(\text{TypeToTypeRepr}_H(U_i)(t)) \land
\text{defined}(\text{RestrictedModuleOnTypeComp}_H(U_i)(t)(s)):
\text{RestrictionsOnTypeComp}_H(U_i)(t)(s)(U_d) \subseteq
\subseteq \text{DefToImp}_H(U_i)(t)(s)(U_d),
\text{being } U_d = \text{TypeDef}_H(\text{TypeToTypeRepr}_H(U_i)(t)(s))
\]

### 3.1.4 Procedure Level

Two domains for procedures are defined, one for procedure definition and the other for procedure implementation, which are related by a pair of functions, and its elements are classified into three groups depending on their import/export characteristics. Also, additional functions are given, to relate procedures with other procedures and with modules and to obtain the types of the objects appearing in the code of a procedure inside an implementation; these functions will be useful to associate implementations to procedure calls.

\[
\text{DefProcs}_H = \{ f / f \text{ is a procedure defined in any module in } H \} \\
\text{ImplProcs}_H = \{ f / f \text{ is a procedure implemented in any module in } H \}
\]

**P33** Modula-2 predefined procedure are implicitly declared and implemented in \( H \).

\[
\{ \text{INC, DEC, ...} \} \subseteq \text{DefProcs}_H \land \{ \text{INC, DEC, ...} \} \subseteq \text{ImplProcs}_H
\]

(Italics stand for procedure implementation.)
P48 All implementations import the same procedures as their definition at least.

\[ \forall U_d : U_d \in \text{DefModules}_H : \]
\[ \forall U_i : U_i \in \text{DefToImpl}(U_d)_H : \]
\[ \text{ImportProcsImpl}_H(U_i) \equiv \text{ImportProcsDef}_H(U_d) \]

P49 A public procedure may not be defined more than once.

\[ \forall U_d, U'_d : U_d, U'_d \in \text{DefModules}_H : \]
\[ \text{NewProcsDef}_H(U_d) \cap \text{NewProcsDef}_H(U'_d) = \emptyset \]

P50 NewProcsDef\_H must be coherent with respect to DefProcToDef\_H.

\[ \forall U_d : U_d \in \text{DefModules}_H : \]
\[ \forall f : f \in \text{NewProcsDef}_H(U_d) : \text{DefProcToDef}_H(f) = U_d \]
\[ \land \]
\[ \forall f : f \in \text{dom}(\text{DefProcToDef}_H) : f \in \text{NewProcsDef}_H(\text{DefProcToDef}_H(f)) \]

AllProcsImpl\_H, PrivateProcsImpl\_H: ImplModules\_H \rightarrow P(DefProcs\_H)

P51 These sets are computable from the ones above.

AllProcsImpl\_H(U_i) = NewProcsImpl\_H(U_i) \cup ImportProcsImpl\_H(U_i)

PrivateProcsImpl\_H(U_i) = NewProcsImpl\_H(U_i) - ExportProcsImpl\_H(U_i)

Code\_H: ImplModules\_H \rightarrow DefProcs\_H \rightarrow ImplProcs\_H

Code\_H(U_i)(g) \equiv f \mid f \text{ is the implementation for } g \text{ inside } U_i

P52 Code\_H is only defined on new procedures inside implementations.

\[ \forall U_i : U_i \in \text{ImplModules}_H : \text{dom}(\text{Code}_H(U_i)) = \text{NewProcsImpl}_H(U_i) \]

P53 Code\_H must follow procedure definition and implementation relationship.

\[ \forall U_i : U_i \in \text{ImplModules}_H : \]
\[ \forall f : f \in \text{NewProcsImpl}_H(U_i) : \]
\[ \text{ImplToDef}_H(\text{Code}_H(U_i)(f)) = f \]
**P54** Types used inside a procedure code should be known inside its implementation.

\[ \forall f : \text{ImplProcs}_H \rightarrow \mathcal{P}(\text{Types}_H) \]

\[ \text{UsedTypesInProc}_H(f) \equiv \{ t \} / \{ t \} \text{ is the set of types used inside } f \]

\[ \forall f : \text{ImplProcs}_H; \text{UsedTypesInProc}_H(f) \subseteq \text{AllTypes}_H(\text{ImplProcToImpl}_H(f)) \]

Now, restrictions themselves are similar to type component level, forgetting natural-number sequences.

**P55** Restrictions may be stated just in new procedures.

\[ \forall U_i : U_i \in \text{ImplModules}_H; \text{dom}(\text{RestrictedModulesOnProc}_H(U_i)) = \text{NewProcs}_H(\text{Impl}_H(U_i)) \]

**P56** All restricted modules must be previously imported and used in the procedure.

\[ \forall U_i : U_i \in \text{ImplModules}_H; \]

\[ \forall f : \text{NewProcs}_H(U_i); \]

\[ \text{RestrictedModulesOnProc}_H(U_i)(f) \subseteq \text{ImportedModules}_H(U_i) \]

\[ \land \text{RestrictedModulesOnProc}_H(U_i)(f) \subseteq \text{TypeDef}_H^*(\text{UsedTypesInProc}(f)), \]

being \( \text{TypeDef}_H^* : \mathcal{P}(\text{Types}_H) \rightarrow \mathcal{P}(\text{DefModules}_H) \) the extension of \( \text{TypeDef}_H \)

**P57** Restrictions may be stated just in new procedures.

\[ \forall U_i : U_i \in \text{ImplModules}_H; \text{dom}(\text{RestrictionsOnProc}_H(U_i)) = \text{NewProcs}_H(\text{Impl}_H(U_i)) \]

**P58** Restrictions are formulated with respect to restricted modules.

\[ \forall U_i : U_i \in \text{ImplModules}_H; \]

\[ \forall f : \text{NewProcs}_H(U_i); \]

\[ \text{dom}(\text{RestrictionsOnProc}_H(U_i)(f)) = \text{RestrictedModulesOnProc}_H(U_i)(f) \]
P59 Restrictions over a definition must refer to its valid implementations.

\[ \forall U_l: U_l \in {\text{ImplModules}}_H: \]
\[ \forall f: f \in {\text{NewProcsImpl}}_H(U_l): \]
\[ \forall U_d: U_d \in {\text{RestrictedModulesOnProc}}_H(U_l)(f): \]
\[ {\text{RestrictionsOnProc}}_H(U_l)(f)(U_d) \subseteq {\text{DefToImpl}}_H(U_d) \]

3.1.5 Object Level

In this section, restrictions at object level are treated. First, we define two domains for variables and parameters, together with functions to know their type. Also, a pair of functions are defined to find out the set the objects introduced inside a procedure.

\[ {\text{Vars}}_H = \{ x \mid x \text{ is a variable introduced in any module in } H \} \]
\[ {\text{Params}}_H = \{ x \mid x \text{ is a parameter introduced in any module in } H \} \]

\[ {\text{AllVars}}_H: {\text{ImplProcs}}_H \rightarrow \mathcal{P}({\text{Vars}}_H) \]
\[ {\text{AllParams}}_H: {\text{ImplProcs}}_H \rightarrow \mathcal{P}({\text{Params}}_H) \]
\[ {\text{AllVars}}/{\text{Params}}_H(f) \equiv \{ x \} \setminus \{ x \} \text{ is the set of variables/parameters declared in } f \]

P60 Objects may not be declared in different procedures.

\[ \forall f,g: f,g \in {\text{ImplProcs}}_H: \]
\[ (f \neq g) \Rightarrow {\text{AllVars}}_H(f) \cap {\text{AllVars}}_H(g) = \emptyset \land {\text{AllParams}}_H(f) \cap {\text{AllParams}}_H(g) = \emptyset \]

\[ {\text{VarToCode}}_H: {\text{Vars}}_H \rightarrow {\text{ImplProcs}}_H \]
\[ {\text{ParamToCode}}_H: {\text{Params}}_H \rightarrow {\text{ImplProcs}}_H \]
\[ \text{Var}/\text{ParamToCode}_H(x) \equiv f/f \text{ is the procedure implementation declaring } x \]

P61 \text{Var}/\text{ParamToCode}_H \text{ are the inverse of AllVars/Params}_H \text{ and vice versa}

\[ \forall f: f \in {\text{ImplProcs}}_H: [\forall x: x \in {\text{AllVars}}_H(f): \text{VarToCode}_H(x) = f] \land \]
\[ \land \forall x: x \in {\text{Vars}}_H: x \in {\text{AllVars}}_H(\text{VarToCode}_H(x)) \]
\[ \land \]
\[ \forall f: f \in {\text{ImplProcs}}_H: [\forall x: x \in {\text{AllParams}}_H(f): \text{ParamToCode}_H(x) = f] \land \]
\[ \land \forall x: x \in {\text{Params}}_H: x \in {\text{AllParams}}_H(\text{ParamToCode}_H(x)) \]
P62 Type of objects should be known inside the module.

\[
\forall U_i: U_i \in \text{ImplModules}_H:
\forall f: f \in \text{AllProcImpl}_H(U_i):
\forall x: x \in \text{AllVars}_H(f): \text{VarType}_H(x) \in \text{AllTypesImpl}_H(U_i) \land
\forall x: x \in \text{AllParams}_H(f): \text{ParamType}_H(x) \in \text{AllTypesImpl}_H(U_i)
\]

RestrictedVarsOnProc\(_H\): ImplModules\(_H\) \rightarrow ImplProcs\(_H\) \rightarrow \mathcal{P}(\text{Vars}_H)

RestrictedParamsOnProc\(_H\): ImplModules\(_H\) \rightarrow ImplProcs\(_H\) \rightarrow \mathcal{P}(\text{Params}_H)

RestrictedVars/ParamsOnProc\(_H\)(U\(_i\))(f) = \{x\} / \{x\} are restricted in procedure f in U\(_i\)

P63 Objects may be restricted just in new procedures.

\[
\forall U_i: U_i \in \text{ImplModules}_H:
\text{dom}(\text{RestrictedVarsOnProc}_H(U_i)) = \text{NewProcsImpl}_H(U_i) \land
\land \text{dom}(\text{RestrictedParamsOnProc}_H(U_i)) = \text{NewProcsImpl}_H(U_i)
\]

P64 Only objects of an imported type declared in new procedures may be restricted.

\[
\forall U_i: U_i \in \text{ImplModules}_H:
\forall f: f \in \text{NewProcsImpl}_H(U_i):
\text{RestrictedVarsOnProc}_H(U_i)(f) \subseteq \text{AllVars}_H(f)
\land \forall x: x \in \text{RestrictedVarsOnProc}_H(U_i)(f):
\text{TypeDef}_H(x) \in \text{ImportedModules}_H(U_i)
\land
\text{RestrictedParamsOnProc}_H(U_i)(f) \subseteq \text{AllParams}_H(f)
\land \forall x: x \in \text{RestrictedParamsOnProc}_H(U_i)(f):
\text{TypeDef}_H(x) \in \text{ImportedModules}_H(U_i)
\]

30
RestrictionsOnVar_H: ImplModules_H → ImplProcs_H → Vars_H → ℙ(ImplModules_H)
RestrictionsOnVar/Param_H(U_i)(f)(x) = {U_i'} / {U_i'} is the set of implementations satisfying the restrictions formulated at procedure f for x in U_i

P65 Objects may be restricted just in new procedures.

∀U_i: U_i ∈ ImplModules_H:
    dom(RestrictionsOnVar_H(U_i)) = NewProcsImpl_H(U_i)
    ∧
    dom(RestrictionsOnParam_H(U_i)) = NewProcsImpl_H(U_i)

P66 Restrictions are stated on restricted objects.

∀U_i: U_i ∈ ImplModules_H:
    ∀f: f ∈ NewProcsImpl_H(U_i):
        dom(RestrictionsOnVar_H(U_i)(f)) = RestrictedVarsOnProc_H(U_i)(f)
        ∧
        dom(RestrictionsOnParam_H(U_i)(f)) = RestrictedParamsOnProc_H(U_i)(f)

P67 Restrictions should be defined over the definition module introducing object's type.

∀U_i: U_i ∈ ImplModules_H:
    ∀f: f ∈ NewProcsImpl_H(U_i):
        ∀x: x ∈ RestrictedVarsOnProc_H(U_i)(f):
            RestrictionsOnVar_H(U_i)(f)(x) ⊆
            ≤ DefToImp_H(TypeDef_H(Var_Type_H(x)))
        ∧
        ∀x: x ∈ RestrictedParamsOnProc_H(U_i)(f):
            RestrictionsOnParam_H(U_i)(f)(x) ⊆
            ≤ DefToImp_H(TypeDef_H(Param_Type_H(x)))

3.1.6 Procedure Call Level

Last, procedure call level restrictions are dealt with. The strategy is similar for objects, introducing a domain for procedure calls and relating them to procedures.
\[
\text{ProcCalls}_H = \{ x \mid x \text{ is a procedure call made in any procedure in } H \} \\
\text{ProcCallToProc}_H: \text{ProcCalls}_H \rightarrow \text{DefProcs}_H \\
\text{ProcCallToProc}_H(x) = \{ g \mid g \text{ is the procedure called by } x \} \\
\text{AllProcCalls}_H: \text{ImplProcs}_H \rightarrow \mathcal{P}(\text{ProcCalls}_H) \\
\text{AllProcCalls}_H(f) = \{ x \} / \{ x \} \text{ is the set of procedure calls made in } f
\]

**P68** Procedure calls are related just to one procedure implementation.

\[
\forall f, g: f, g \in \text{ImplProcs}_H: (f \neq g) \Rightarrow \text{AllProcCall}_H(f) \cap \text{AllProcCall}_H(g) = \emptyset
\]

**P69** Procedure calls must correspond to procedures known in the implementation.

\[
\forall f: f \in \text{ImplProcs}_H: \\
\forall x: x \in \text{AllProcCalls}_H(f): \\
\text{ProcCallToProc}_H(x) \in \text{AllProcsImpl}_H(\text{ImplProcToImpl}_H(f))
\]

**P70** ProcCallToCode\(_H\) is the inverse of AllProcCalls\(_H\) and vice versa

\[
\forall f: f \in \text{ImplProcs}_H: [\forall x: x \in \text{AllProcCalls}_H(f): \text{ProcCallToCode}_H(x) = f] \land \\
\land \forall x: x \in \text{ProcCalls}_H: x \in \text{AllProcCalls}_H(\text{ProcCallToCode}_H(x))
\]

**P71** Procedure calls may be restricted just in new procedures.

\[
\forall U_i: U_i \in \text{ImplModules}_H: \\
\text{dom}(\text{RestrictedCallsOnProc}_H(U_i)) = \\
\text{Code}_H^*(U_i)(\text{NewProcsImpl}_H(U_i)) \\
\text{being } \text{Code}_H^* \text{ the extension of } \text{Code}_H \text{ to sets of procedures}
\]
P72 Only imported procedure calls made in new procedures may be restricted.

\[ \forall U_i \in \text{ImplModules}_H: \]
\[ \forall f \in \text{Code}_H^*(U_i)(\text{NewProcsImpl}_H(U_i)): \]
\[ \text{RestrictedCallsOnProc}_H(U_i)(f) \subseteq \text{AllProcCalls}_H(f) \]
\[ \land \forall f \in \text{RestrictedCallsOnProc}_H(U_i)(f): \]
\[ \text{DefProcToDef}_H(\text{ProcCallToProc}_H(x)) \subseteq \text{ImportedModules}_H(U_i) \]

RestrictionsOnCall_H : ImplModules_H → ImplProcs_H → ProcCalls_H → τ(ImplModules_H)
RestrictionsOnCall_H(U_i)(f)(x) = \{ U' \} / \{ U' \} is the set of implementations satisfying the restrictions formulated at procedure f for x in U_i

P73 Procedure calls may be restricted just in new procedures.

\[ \forall U_i \in \text{ImplModules}_H: \]
\[ \text{dom}(\text{RestrictionsOnCall}_H(U_i)) = \text{Code}_H^*(U_i)(\text{NewProcsImpl}_H(U_i)) \]

P74 Restrictions are stated on restricted procedure calls.

\[ \forall U_i \in \text{ImplModules}_H: \]
\[ \forall f \in \text{Code}_H^*(U_i)(\text{NewProcsImpl}_H(U_i)): \]
\[ \text{dom}(\text{RestrictionsOnCall}_H(U_i)(f)) = \text{RestrictedCallsOnProc}_H(U_i)(f) \]

P75 Restrictions should be defined over the definition module introducing the called procedure.

\[ \forall U_i \in \text{ImplModules}_H: \]
\[ \forall f \in \text{Code}_H^*(U_i)(\text{NewProcsImpl}_H(U_i)): \]
\[ \forall x : x \in \text{RestrictedCallsOnProc}_H(U_i)(f): \]
\[ \text{RestrictionsOnCall}_H(U_i)(f)(x) \subseteq \]
\[ \subseteq \text{DefToImp}_H(\text{DefProcToDef}_H(\text{ProcCallToProc}_H(x))) \]
3.2 Implementation Selection Rules

Implementations are attached to every variable, parameter and procedure call appearing in procedures by means of three functions, \( \text{IMP}_{\text{VAR},H} \), \( \text{IMP}_{\text{PARAM},H} \) and \( \text{IMP}_{\text{CALL},H} \). Their result should be a single implementation, but in fact they are allowed to return any non-unitary set of implementations (even an empty one), in which case restrictions should be modified somewhere in the hierarchy of modules.

To define these goal functions, in addition to syntactic domains and functions introduced in the previous section, other functions are needed to keep track of restrictions at all other possible levels. A function corresponding to a level is defined as the function corresponding to its previous level constrained with the restrictions stated in its level; in the upper level, the constrained set for every definition is that of all its implementations.

All of these functions are presented next, giving first their signature and their informal description, next their domain and rank conditions and last their definition.

3.2.1 Module Level

Signature:

\[
\text{IMP}_{\text{MOD},H} : \text{ImplModules}_H \rightarrow \text{DefModules}_H \rightarrow \mathcal{P}(\text{ImplModules}_H)
\]

\[
\text{IMP}_{\text{MOD},H}(U_i)(U_d) = \{U'_i\} / \{U'_i\} \text{ is the set of implementations associated to } U_d \text{ considering up to module level in } U_i
\]

Domain and rank:

\[
\forall U_i : U_i \in \text{ImplModules}_H : \\
\quad \text{dom}(\text{IMP}_{\text{MOD},H}(U_i)) = \text{ImportedModules}_H(U_i) \wedge \\
\quad \forall U_d : U_d \in \text{ImportedModules}_H(U_i) : \text{IMP}_{\text{MOD},H}(U_i)(U_d) \subseteq \text{DefToImp}_H(U_d)
\]

Definition:

\[
\forall U_i : U_i \in \text{ImplModules}_H : \\
\quad \forall U_d : U_d \in \text{ImportedModules}_H(U_i) \wedge U_d \in \text{RestrictedModules}_H(U_i) : \\
\quad \quad \text{IMP}_{\text{MOD},H}(U_i)(U_d) = \text{Restrictions}_H(U_i)(U_d) \wedge \\
\quad \quad \forall U_d : U_d \in \text{ImportedModules}_H(U_i) \wedge U_d \in \text{RestrictedModules}_H(U_i) : \\
\quad \quad \quad \text{IMP}_{\text{MOD},H}(U_i)(U_d) = \text{DefToImp}_H(U_i)(U_d)
\]
3.2.2 Type Representation Level

Signature:

\[ \text{IMP\_TYPES}_H : \text{ImplModules}_H \rightarrow \text{Types}_H \rightarrow \mathcal{T}(\text{ImplModules}_H) \]

\[ \text{IMP\_TYPES}_H(U_i)(t) = \{ U'_i \} / \{ U'_i \} \text{ is the set of implementations satisfying the} \]

\[ \text{restrictions formulated up to type representation level for } t \text{ in } U_i \]

Domain and rank:

\[ \forall U_i : U_i \in \text{ImplModules}_H : \]

\[ \text{dom}(\text{IMP\_TYPES}_H(U_i)) = \text{ImportTypesImpl}_H(U_i) \]

\[ \wedge \]

\[ \forall t : t \in \text{ImportTypesImpl}_H(U_i) : \text{IMP\_TYPES}_H(U_i)(t) \subseteq \text{DefToImp}_H(\text{TypeDef}_H(t)) \]

Definition:

\[ \forall U_i : U_i \in \text{ImplModules}_H : \]

\[ \forall t : t \in \text{ImportTypesImpl}_H(U_i) \wedge \text{TypeDef}_H(t) \in \text{RestrictedModulesOnRepr}_H(U_i) : \]

\[ \text{IMP\_TYPES}_H(U_i)(t) = \text{RestrictionsOnRepr}_H(U_i)(\text{TypeDef}_H(t)) \]

\[ \wedge \]

\[ \forall t : t \in \text{ImportTypesImpl}_H(U_i) \wedge \text{TypeDef}_H(t) \notin \text{RestrictedModulesOnRepr}_H(U_i) : \]

\[ \text{IMP\_TYPES}_H(U_i)(t) = \text{IMP\_MOD}_H(U_i)(\text{TypeDef}_H(t)) \]
3.2.3 Type Component Level

Signature:

\[
\text{IMPTYPECOMPS}_H : \text{ImplModules}_H \rightarrow \text{Types}_H \rightarrow \mathcal{K}_* \rightarrow \mathcal{T}(\text{ImplModules}_H)
\]

\[
\text{IMPTYPECOMPS}_H(U_i)(t)(s) = \{U_i'\} / \{U_i\}
\]

is the set of implementations satisfying the restrictions formulated up to type component level for the component \(s\) of \(t\) in \(U_i\).

Domain and rank:

\[
\forall U_i : U_i \in \text{ImplModules}_H:
\]

\[
\text{dom}(\text{IMPTYPECOMPS}_H(U_i)) = \text{NewTypesImpl}_H(U_i) \land
\]

\[
\forall t : t \in \text{NewTypesImpl}_H(U_i):
\]

\[
\text{dom}(\text{IMPTYPECOMPS}_H(U_i)(t)) = \text{dom}(\text{TypeToTypeRepr}_H(U_i)(t))
\]

\[
\land
\]

\[
\forall s : s \in \text{dom}(\text{TypeToTypeRepr}_H(U_i)(t)):
\]

\[
\text{IMPTYPECOMPS}_H(U_i)(t)(s) \subseteq
\]

\[
\subseteq \text{DefToImpl}_H(\text{TypeDef}_H(\text{TypeToTypeRepr}_H(U_i)(t)(s)))
\]

Definition:

\[
\forall U_i : U_i \in \text{ImplModules}_H:
\]

\[
\forall t : t \in \text{NewTypesImpl}_H(U_i):
\]

\[
\forall s : s \in \text{dom}(\text{TypeToTypeRepr}_H(U_i)(t)) \land
\]

\[
\text{se dom}(\text{RestrictedModuleOnTypeComp}_H(U_i)(t)):
\]

\[
\text{IMPTYPECOMPS}_H(U_i)(t)(s) = \text{RestrictionsOnTypeComp}_H(U_i)(t)(s)(\text{TypeDef}_H(t))
\]

\[
\land
\]

\[
\text{IMPTYPECOMPS}_H(U_i)(t)(s) \subseteq \text{DefToImpl}_H(\text{TypeDef}_H(\text{TypeToTypeRepr}_H(U_i)(t)(s)))
\]

\[
\land
\]

\[
\forall s : s \in \text{dom}(\text{TypeToTypeRepr}_H(U_i)(t)) \land
\]

\[
\text{se dom}(\text{RestrictedModuleOnTypeComp}_H(U_i)(t)) \land
\]

\[
\text{se UserDefinedTypes}_H(\text{TypeToTypeRepr}_H(U_i)(t)):
\]

\[
\text{IMPTYPECOMPS}_H(U_i)(t)(s) = \text{IMPTYPECOMPS}_H(U_i)(t)(s)
\]

\[
\land
\]

\[
\forall s : s \in \text{dom}(\text{TypeToTypeRepr}_H(U_i)(t)) \land
\]

\[
\text{se dom}(\text{RestrictedModuleOnTypeComp}_H(U_i)(t)) \land
\]

\[
\text{se UserDefinedTypes}_H(\text{TypeToTypeRepr}_H(U_i)(t)):
\]

\[
\text{IMPTYPECOMPS}_H(U_i)(t)(s) = \{\text{MODULA2.IMPth}\}
\]
3.2.4 Procedure Level

Signature:

\[ \text{IMP}_{\text{PROC},H}: \text{ImplModules}_H \rightarrow \text{ImplProcs}_H \rightarrow \text{DefModules}_H \rightarrow \mathcal{P}(\text{ImplModules}_H) \]

\[ \text{IMP}_{\text{PROC},H}(U_i)(f)(U_d) = \{ U_i' \} \cap \{ U_i' \} \text{ is the set of implementations satisfying the restrictions formulated up to procedure } f \text{ level for } U_d \text{ in } U_i \]

Domain and rank:

\[ \forall U_i: U_i \in \text{ImplModules}_H: \]
\[ \text{dom}(\text{IMP}_{\text{PROC},H}(U_i)) = \text{NewProcsImpl}_H(U_i) \]
\[ \land \]
\[ \forall f: f \in \text{NewProcsImpl}_H(U_i): \]
\[ \text{dom}(\text{IMP}_{\text{PROC},H}(U_i)(f)) = \text{TypeDef}^*_{H}(\text{UsedTypesInProc}_H((U_i)(f)) \cup \text{CallDef}^*_{H}(\text{AllProcCalls}_H((U_i)(f))) \]
\[ \land \]
\[ \forall U_d: U_d \in \text{TypeDef}^*_{H}(\text{UsedTypesInProc}_H((U_i)(f))) \cup \text{CallDef}^*_{H}(\text{AllProcCalls}_H((U_i)(f))): \]
\[ \text{IMP}_{\text{PROC},H}(U_i)(f)(U_d) \in \text{DefToImpl}_H(U_d) \]

Definition:

\[ \forall U_i: U_i \in \text{ImplModules}_H: \]
\[ \forall f: f \in \text{NewProcsImpl}_H(U_i): \]
\[ \forall U_d: U_d \in \text{TypeDef}^*_{H}(\text{UsedTypesInProc}_H(f)) \cup \text{CallDef}^*_{H}(\text{AllProcCalls}_H(U_i)(f)) \]
\[ \land U_d \in \text{RestrictedModules}_H(U_i): \]
\[ \text{IMP}_{\text{PROC},H}(U_i)(f)(U_d) = \text{IMP}_{\text{MOD},H}(U_i)(U_d) \]
\[ \land \]
\[ \forall U_d: U_d \in \text{TypeDef}^*_{H}(\text{UsedTypesInProc}_H(f)) \cup \text{CallDef}^*_{H}(\text{AllProcCalls}_H(U_i)(f)) \]
\[ \land U_d \in \text{RestrictedModules}_H(U_i): \]
\[ \text{IMP}_{\text{PROC},H}(U_i)(f)(U_d) = \text{RestrictionsOnProc}_H(U_i)(f)(U_d) \cap \text{IMP}_{\text{MOD},H}(U_i)(U_d) \]

being \( \text{CallDef}^*_{H}: \mathcal{P}(\text{ProcCalls}_H) \rightarrow \mathcal{P}(\text{DefModules}_H) \) defined as:

\[ \text{CallDef}^*_{H}(s) = \bigcup x: x \in s: \text{DefProcToDef}_H(\text{ProcCallToProc}^*_{H}(s)) \]
3.2.5 Object Level

Signature:

\[ \text{IMPVAR,H}: \text{ImplModules}_H \rightarrow \text{ImplProcs}_H \rightarrow \text{Vars}_H \rightarrow \mathcal{P}(\text{ImplModules}_H) \]
\[ \text{IMPPARAM,H}: \text{ImplModules}_H \rightarrow \text{ImplProcs}_H \rightarrow \text{Params}_H \rightarrow \mathcal{P}(\text{ImplModules}_H) \]
\[ \text{IMPVARPARAM,H}(U_j)(f)(x) = \{ U'_i \} / \{ U'_i \} \text{ is the set of implementations satisfying} \]
\[ \text{the restrictions formulated up to object level for } x \text{ in } U_i \]

Domain and rank:

\[ \forall U_j: U_j \in \text{ImplModules}_H: \]
\[ \text{dom}(\text{IMPVAR,H}(U_j)) = \text{dom}(\text{IMPPARAM,H}(U_j)) = \text{NewProcsImpl}_H(U_j) \land \]
\[ \forall f: f \in \text{NewProcsImpl}_H(U_j): \]
\[ \text{dom}(\text{IMPVAR,H}(U_j)(f)) = \text{AllVars}_H(f) \land \]
\[ \forall x: x \in \text{AllVars}_H(f): \]
\[ \text{IMPVAR,H}(U_j)(f)(x) \in \text{DefToImp}_H(\text{TypeDef}_H(\text{VarType}_H(x))) \land \]
\[ \text{dom}(\text{IMPPARAM,H}(U_j)(f)) = \text{AllParams}_H(f) \land \]
\[ \forall x: x \in \text{AllParams}_H(f): \]
\[ \text{IMPPARAM,H}(U_j)(f)(x) \in \text{DefToImp}_H(\text{TypeDef}_H(\text{ParamType}_H(x))) \]

Definition:

\[ \forall U_j: U_j \in \text{ImplModules}_H: \]
\[ \forall f: f \in \text{NewProcsImpl}_H(U_j): \]
\[ \forall x: x \in \text{AllVars}_H(f) \land x \in \text{RestrictedVarsOnProc}_H(U_j)(f): \]
\[ \text{IMPVAR,H}(U_j)(f)(x) = \text{RestrictionsOnVar}_H(U_j)(f)(x) \cap \]
\[ \text{IMPPROC,H}(U_j)(f)(\text{TypeDef}_H(\text{VarType}_H(x))) \land \]
\[ \forall x: x \in \text{AllVars}_H(f) \land x \notin \text{RestrictedVarsOnProc}_H(U_j)(f): \]
\[ \text{IMPVAR,H}(U_j)(f)(x) = \text{IMPPROC,H}(U_j)(f)(\text{TypeDef}_H(\text{VarType}_H(x))) \land \]
\[ \forall x: x \in \text{AllParams}_H(f) \land x \in \text{RestrictedParamsOnProc}_H(U_j)(f): \]
\[ \text{IMPPARAM,H}(U_j)(f)(x) = \text{RestrictionsOnParam}_H(U_j)(f)(x) \cap \]
\[ \text{IMPPROC,H}(U_j)(f)(\text{TypeDef}_H(\text{ParamType}_H(x))) \land \]
\[ \forall x: x \in \text{AllParams}_H(f) \land x \notin \text{RestrictedParamsOnProc}_H(U_j)(f): \]
\[ \text{IMPPARAM,H}(U_j)(f)(x) = \text{IMPPROC,H}(U_j)(f)(\text{TypeDef}_H(\text{ParamType}_H(x))) \]
3.2.6 Procedure Call Level

Signature:

\[ \text{IMP}_{\text{CALL},H}(\text{ImplModules}_H \rightarrow \text{ImplProcs}_H \rightarrow \text{ProcCalls}_H \rightarrow \mathcal{D}(\text{ImplModules}_H)) \]

\[ \text{IMP}_{\text{CALL},H}(U_j)(f)(x) \equiv \{ U'_{i} \} / \{ U'_{i} \} \text{ is the set of implementations satisfying the restrictions formulated up to procedure call level for } x \text{ in } U_i \]

Domain and rank:

\[ \forall U_i : U_i \in \text{ImplModules}_H : \]
\[ \text{dom}(\text{IMP}_{\text{CALL},H}(U_i)) = \text{NewProcsImpl}_H(U_i) \]
\[ \wedge \]
\[ \forall f : f \in \text{NewProcsImpl}_H(U_i) : \]
\[ \text{dom}(\text{IMP}_{\text{CALL},H}(U_i)(f)) = \text{AllProcCalls}_H(U_i)(f) \]
\[ \wedge \]
\[ \forall x : x \in \text{AllProcCalls}_H(U_i)(f) : \]
\[ \text{IMP}_{\text{CALL},H}(U_i)(f)(x) \in \text{DefToImp}_H(\text{DefProcToDef}_H(\text{ProcCallToProc}_H(x))) \]

Definition:

\[ \forall U_i : U_i \in \text{ImplModules}_H : \]
\[ \forall f : f \in \text{NewProcsImpl}_H(U_i) : \]
\[ \forall x : x \in \text{AllProcCalls}_H(U_i)(f) \land x \in \text{RestrictedCallsOnProc}_H(U_i)(f) : \]
\[ \text{IMP}_{\text{CALL},H}(U_i)(f)(x) = \]
\[ \text{RestrictionsOnCall}_H(U_i)(f)(x) \cap \]
\[ \text{IMP}_{\text{PROC},H}(U_i)(f)(\text{DefProcToDef}_H(\text{ProcCallToProc}_H(x))) \]
\[ \wedge \]
\[ \forall x : x \in \text{AllProcCalls}_H(U_i)(f) \land x \notin \text{RestrictedCallsOnProc}_H(U_i)(f) : \]
\[ \text{IMP}_{\text{CALL},H}(U_i)(f)(x) = \]
\[ \text{IMP}_{\text{PROC},H}(U_i)(f)(\text{DefProcToDef}_H(\text{ProcCallToProc}_H(x))) \]
4 Checking Implementations against Requirements

In this section, we address the problem of determining which implementations satisfy a set of given requirements. We divide this study in three parts. First, an abstract data type for NF-expressions is given, and so the result of sums, products and comparisons of costs will be well-defined. This data type requires a single measure unit appearing in NF-expressions, in order to avoid undefined results when comparing two different expressions (which could happen if there were two or more unrelated measure units). Hereafter, we call NF$_n$-expressions those NF-expressions involving a single measure unit $n$.

As far as we require a single measure unit in NF$_n$-expressions, subsection 4.2 presents an algorithm to select which measure unit is the best suited to act as $n$, and to translate the other ones into NF$_n$-expressions. Last, an algorithm for checking an implementation against a set of requirements is given. Implementation properties and requirements are translated into NF$_n$-expressions, and then they are compared in the algebra of NF$_n$-expressions.

4.1 An Abstract Data Type for NF$_n$-expressions

The definition of O notation as well as its usual operations has been introduced in section 2. Although it is mathematically clear and comfortable, it is difficult to deal with in our operational framework. So, we are going next to restrict the set of representative values in O notation and then we are going to introduce an ADT modelling the result.

Values (this is to say, functions measuring program efficiency) are given as a product of powers and logarithms over a single measure unit $n$; powers may also appear on logarithms; for practical reasons, we consider log log $n = 1$, as far as this is the case for reasonable values of $n$. Also, a single additional value exists to represent exponential costs; this uniqueness stems from the fact that most of the times the important thing on exponential costs is precisely its exponential nature.

Operations are those ones on O notation introduced in section 2, satisfying the properties inherent to the definition of O adapted to our set of values. About $<<$, we consider the NF$_n$-expression $x$ negligible with respect to $y$ ($x << y$) if $y$ is exponential and $x$ polynomial or, else, being $x$ and $y$ polynomial, $x = n^k \ast (\log n)^r$ and $y = n^i \ast (\log n)^s$, if $x$ is constant or $k < i$.

In order to provide a computable form of the mathematical model underlying NF$_n$-expressions, we state the properties of these operations by means of equations which define an ADT; so, we can view NF$_n$-expressions (in the initial semantics framework for abstract data types [GTW78, EM85]) as an isomorphic class of algebras represented by the quotient-term algebra built from this equational specification. To show its correctness, we should verify that there is an isomorphism between this quotient-term algebra and the intended mathematical model.
values: \{n^k \cdot (\log n)^i / i, k \geq 0\} \cup \{b^n\}

operations:
1, n, log n, EXP: \rightarrow \text{NF}_n\text{-expression} \quad \text{for clarity, we write } b^n \text{ for EXP}

pot: \text{NF}_n\text{-expression nat} \rightarrow \text{NF}_n\text{-expression} \quad \text{for clarity, we write } x^k \text{ for } \text{pot}(x, k)

log: \text{NF}_n\text{-expression} \rightarrow \text{NF}_n\text{-expression}

+, *, \cdot: \text{NF}_n\text{-expression NF}_n\text{-expression} \rightarrow \text{NF}_n\text{-expression}

\langle, \rangle, \langle=, =, \rangle, \langle>, \langle<, \rangle>: \text{NF}_n\text{-expression NF}_n\text{-expression} \rightarrow \text{bool}

equations: \forall x, y, z \in \text{NF}_n\text{-expression}; \forall k, i, r, s \in \text{nat}
\begin{align*}
x^0 &= 1; x^1 = x \\
(\log(x^r))^k &= \text{IF } r = 0 \text{ THEN } 1 \text{ ELSE } (\log(x))^k \text{ ENDIF} \\
\log(\log(x^r)) &= 1 \\
(x^r)^s &= x^{kr}; (b^n)^r = b^n; \log(b^n) = n \\
x \cdot y &= y \cdot x; (x \cdot y) \cdot z = x \cdot (y \cdot z) \\
b^n \cdot x &= b^n \\
x^k \cdot x^r &= x^{kr}; (x \cdot y)^k = x^k \cdot y^k \\
x + y &= \text{IF } x > y \text{ THEN } x \text{ ELSE } y \text{ ENDIF} \\
(x < x) &= \text{false}; (x < y \Rightarrow \text{NOT } y < x) = \text{true}; (x < y \text{ AND } y < z \Rightarrow x < z) = \text{true} \\
\quad \text{(* for the following laws, note that a non-exponential cost may be expressed as } n^k \cdot (\log(n))^r \text{ *)} \\
n^k \cdot (\log(n))^r < n^i \cdot (\log(n))^s = (k < i) \text{ OR } (k = i \text{ AND } r < s) \\
n^k \cdot (\log(n))^r < b^n = \text{true}; b^n < x = \text{false} \\
n^k \cdot (\log(n))^r < b^n = \text{true}; b^n < x = \text{false} \\
(x = y) = \text{NOT}(x < y) \text{ AND } \text{NOT}(y < x); (x <> y) = \text{NOT}(x = y) \\
(x <= y) = \text{NOT}(y < x); (x >= y) = \text{NOT}(x < y); x >= y = y <= x
\end{align*}

4.2 Translating NF-expressions into NF_n-expressions

As far as we require just a measure unit to be able to perform asymptotic arithmetic and logic, an algorithm is needed to translate efficiency requirements and properties from possibly many measure units to just one of them. It is clear that this translation will not be always possible, either because some information is missing or because some inconsistencies arise during translation or during later checking; in these cases, error messages will be given to the user.
The algorithm is split into four steps:

- **Step 1**: information retrieval. The modular organisation of the program is examined to collect all the formula referred to measure unit relationships.
- **Step 2**: basic measure unit selection. The appropriate set of formula is then processed to choose a measure unit $n$ suited to be used as a basis for the rest of them.
- **Step 3**: measure unit computation. All measure units are computed as $\text{NF}_n$-expressions using just the basic measure unit selected in the previous step.
- **Step 4**: consistence checking. Measure unit values viewed as $\text{NF}_n$-expressions are checked against all measure unit relationships not used during steps 2 and 3.

These steps consist of one or more procedures which are put together in subsection 3.2.5. To avoid not relevant details, procedures are written using a kind of "high-level" (mainly for set notation) Modula-2, sometimes sacrificing real code for clarity in explanation; also, the code for auxiliary procedures and ADTs is not included, being their names self-explanatory enough (hopefully!). Concerning NF-expressions, operations are defined to view them as a parse tree: an operation $\text{Root}$ is available to obtain their root; operations $\text{First}$, $\text{Second}$, etc., to obtain children one by one; $\text{Children}$, to obtain children altogether; and operations $\text{Form0}$, $\text{Form1}$, etc., to form a tree from a label stored in the new root and zero, one, etc., trees as children.

### 4.2.1 Information Retrieval

The hierarchy of modules is traversed to collect all measure unit formula, which are grouped into two different sets. The first one contains those NF-expressions stating equalities of the form $x = f(x_1, \ldots, x_k)$ and also comparisons $x << y$ provided that no equalities for $x$ exist\(^4\), which will be used to determine measure unit relationships, while the second one contains the rest of NF-expressions, which will be used during step 4 to check their validity against the results obtained with the first set.

---

\(^4\) $\Theta$-expressions of the form $f(x_1, \ldots, x_k) = x$ and $x >> y$ are automatically reversed when building parse trees.
Name: InformationRetrieval
Input: \( H \), the whole module hierarchy.
Output: \( S_1 \), \( S_2 \), two set of NF-expressions.
Comments: \( S_0 \) stores '<<' NF-expressions, which are processed at the end: they are added at \( S_1 \) only if the measure unit under definition (which is the first child) has not been defined yet.
Algorithm:

\[\begin{align*}
\text{VAR } & M: \text{Module}; \text{SM: set of Module; } F: \text{NF-expression; SF, S0: set of NF-expression;} \\
& S_1, S_2, S_0 := \text{EmptySet}; \text{SM := AddSet(EmptySet, TopModule(H));} \\
& \text{WHILE NOT EmptySet?(SM) DO} \\
& \quad M := \text{SelectOneFrom(SM); SM := DeleteSet(SM, M);} \\
& \quad SF := \text{EfficiencyFormula(M);} \\
& \quad \text{FOR ALL } F \text{ IN SF DO} \\
& \quad \quad \text{IF MUEquality(F) THEN } S_1 := \text{AddSet}(S_1, F) \\
& \quad \quad \text{ELSIF MUNegligible(F) THEN } S_0 := \text{AddSet}(S_0, F) \\
& \quad \quad \text{ELSE } S_2 := \text{AddSet}(S_2, F); \\
& \quad \text{END;} \\
& \quad \text{FOR ALL } F \text{ IN S0 DO} \\
& \quad \quad \text{IF MUNotDefined}(S_1, \text{First}(F)) \text{ THEN } S_1 := \text{AddSet}(S_1, F) \\
& \quad \quad \text{ELSE } S_2 := \text{AddSet}(S_2, F) \\
& \quad \text{END;} \\
& \quad \text{SM := Union(SM, ImportedModules(H, M))} \\
& \text{END;} \\
\end{align*}\]
4.2.2 Basic Measure Unit Selection

Once relevant formula have been grouped together, they are examined to select a measure unit \( n \) that permits to state the other ones as \( \text{NF}_n \)-expressions. To perform this selection and also the later translation step, the set \( S1 \) of formula is used to build a directed graph \( G \) with the following sets \( V \) of nodes and \( E \) of edges:

- Nodes. \( V \) contains all measure units that have been neither declared to be equal (this is to say, identical measure units are grouped into a single node) nor negligible with respect other measure unit. Also, there exist two special nodes for \( O(1) \) and \( O(b^n) \) costs.

- Edges. For every equality of the form \( x = f(x_1, ..., x_k) \) different from \( x = y \), \( k \) edges from \( x_1, ..., x_k \) to \( x \) exist in \( E \), and for every comparison of the form \( x << y \) an edge from \( y \) to \( x \) exist in \( E \).

The meaning of the graph is: nodes represent different magnitudes in a program given their stated relationships, while every edge from a node \( x \) to a node \( y \) means that \( x \)'s value could be necessary in order to fix \( y \)'s one.

The procedure is presented below. Creation of the graph is divided in two steps: first, a node is defined for every measure unit and edges are labelled with ':=' , '>>' or '='. Second, nodes are merged as described above and '=' edges are discarded while other ones loose their label (we are just interested in knowing evaluation precedence). When creation ends, nodes contain as information a set of measure units (those ones declared to be equal). Errors arise for many reasons: two different kind of relationships may be found between two measure units\(^5\), a '<='- or '='-cycle exists in the graph (no basic measure unit may be selected) or two measure units may not be related anyway. To carry out the second step and to detect the first two kind of errors, the transitive closure \( TC \) of the labelled graph is computed applying the following rules in order of appearance to obtain path labels, being \( \bot \) the absence of path and \( TC(u, v) \) the path from \( u \) to \( v \).

\[
\begin{align*}
TC(u, v) &= \bot \quad TC(v, w) = \bot: \quad TC(u, w) = \bot \\
TC(u, v) &= '=' \quad TC(v, w) = '=': \quad TC(u, w) = '=' \\
TC(u, v) &= '>>' \quad TC(v, w) = '>>': \quad TC(u, w) = '>>' \\
TC(u, v) &= '=' \land TC(v, w) = '=': \quad TC(u, w) = '='
\end{align*}
\]

(Application of these rules is denoted by \( \otimes \) in the algorithm.)

When the final graph has been built, the basic measure unit is one from the node that does not have predecessors, once the nodes whose cost is a constant \( \text{NF} \)-expression have been deleted (because they must not affect to this computation).

\(^5\) Some errors of this kind may not be detected in this step. More precisely, it is just checked if two different paths compute two different labels. Errors may also occur with two paths yielding both to '=' label; these errors will be detected later.
Name: GraphCreation

Input: $S$, a set of NF-expressions of the form $x = f(x_1, ..., x_k)$ and $x \ll y$.

Output: $G$, a labelled and directed graph as described above.

Algorithm:

VAR LHS, RHS: NF-expression; F: set of NF-expression;
G := AddNode(AddNode(EmptyGraph, b^n), 1);
FOR ALL F IN S DO
    LHS := First(F); RHS := Second(F);
    G := AddNode(G, LHS); G := AddNodes(G, AllMeasureUnits(RHS));
    IF MUEquality(F) THEN
        IF IsMeasureUnit?(RHS) \lor IsConstant?(RHS) (* IsConstant? = 1 or b^n *)
            THEN G := AddEdge(AddEdge(G, LHS, RHS, '='), RHS, LHS, '=')
        ELSE FOR ALL mu IN AllMeasureUnits(RHS) DO
            G := AddEdge(G, mu, LHS, 'c')
        END;
    ELSE G := AddEdge(G, RHS, LHS, '>>');
END;
Name: TransitiveClosure

Input: $G$, a directed and labelled graph resulting from $GraphCreation$.

Output: $TC$, the closure transitive of the graph $G$.

Errors: two different paths between two nodes yield to two different labels; '$<$' or '$<=$' cycles do exist.

Algorithm (Warshall algorithm variation):

VAR $u$, $v$, $w$: NF-expression;

FOR ALL $(u, v)$ IN Nodes($G$) DO $TC[u, v] := '<$' END;

FOR ALL $(u, v)$ IN Nodes($G$) DO

    IF ExistsEdge($G$, $u$, $v$) THEN $TC[u, v] := EdgeLabel(G, u, v)$

END;

FOR ALL $(w, u, v)$ IN Nodes($G$) DO

    IF $u = v$ THEN

        IF $(TC[u, w] \circ TC[w, v] <> '=')$ AND
            $(TC[u, w] \circ TC[w, v] <> '<$') THEN Error

        ELSE IF $(TC[u, v] <> '<$') AND $(TC[u, v] <> TC[u, w] \circ TC[w, v])

            THEN Error

        ELSE $TC[u, v] := TC[u, w] \circ TC[w, v]$;

END;
Name: GraphSimplification
Input: $G$, the directed graph obtained in GraphCreation.
- $S$, set of formula of the form $x = f(x_1, ..., x_k)$ and $x \ll y$
- $TC$, the result of TransitiveClosure for $G$.
Output: $Gout$, the ultimate directed labelled graph as described above.
Comment: nodes include a set for the magnitude units that they represent. A function $MUToNode$ returns the node a measure unit belongs to. The constant values $1$ and $b^n$ are treated as measure units during this procedure.
Algorithm:

VAR u, v, F: NF-expression; e: Edge; S, Saux: set of NF-expression;
Gout := EmptyGraph; S := Nodes(G);
(* first, nodes are merged appropriately *)
WHILE NOT EmptySet?(S) DO
  u := SelectOneFrom(S);
  Saux := \{v ∈ Nodes(G) / TC[u, v] = '='\} ∪ \{u\};
  Gout := AddNode(Gout, Saux); S := Difference(S, Saux);
END;
(* next, relevant edges are included *)
FOR ALL e IN Edges(G) DO
  IF MUToNode(From(e)) ≠ MUToNode(To(e)) THEN
    Gout := AddEdge(Gout, To(e), From(e))
  END;
(* last, formula are attached to node; $x = y$ equalities are discarded *)
FOR ALL F IN S DO
  IF NOT (MUEquality(F) AND IsMeasureUnit?(Second(F))) THEN
    T := AddFormulaToNode(T, MUToNode(First(F)), F);
  END;
Name: MeasureUnitSelection

Input: $G$, directed graph obtained from GraphSimplification.
Output: $n$, basic measure unit selected.

$S_I$, $S_{exp}$, set of constant measure units.

$G$, with constant nodes deleted.

Errors: two measure units are not related.

Algorithm: VAR $u$, $v$: NF-expression; $SMU$: set of NF-expression;

$G := \text{DeleteConstantMU}(\text{DeleteConstantMU}(G, 1, S_I), b^n, S_{exp})$;

$SMU := \{v \in \text{Nodes}(G) / \| \{u \in \text{Nodes}(G) / (u, v) \in \text{Edges}(G)\} \| = 0\}$;

IF $\| SMU \| > 1$ THEN Error
ELSE (* Let $SMU = \{v\} (*)$

$n := \text{SelectOneFrom}($SetOfMeasureUnits$(v))$;

Name: DeleteConstantMU

Input: $G$, directed graph obtained from GraphSimplification.

$k$, either 1 or $b^n$.

Output: $G$, with nodes whose cost equals $k$ removed.

$S_{out}$, set of deleted measure units.

Algorithm: VAR $u$, $v$: NF-expression; $SMU$: set of NF-expression;

$S_{out} := \text{EmptySet}$;

IF ExistsNode?(G, $k$) THEN

$S := \text{AddSet}(\text{EmptySet}, k)$; (* set of nodes under treatment *)

WHILE NOT EmptySet?(S) DO

$u := \text{SelectOneFrom}(S)$; $S := \text{DeleteSet}(S, u)$;

FOR ALL $v \in \text{Successors}(G, u)$ DO

IF $\| \{w \in \text{Nodes}(G) / (w, v) \in \text{Edges}(G)\} \| = 1$ THEN

$S := \text{AddSet}(S, v)$

END;

$S_{out} := \text{Union}(S_{out}, \text{SetOfMeasureUnits}(u))$;

$G := \text{DeleteNodeAndItsEdges}(G, u)$

END;
4.2.3 Measure Unit Computation

Once a measure unit \( n \) has been selected to work with, the rest of them must be expressed as \( \text{NF}_{\pi} \)-expressions. To perform this task we may use the same graph as the step before and make a topological sort starting from the basic measure unit: when another measure unit (a node) is reached, its associated formula is selected and all of the measure unit appearances different from \( n \) are substituted by the appropriate values; this substitution is always possible, as far as the topological sort strategy assures that the involved units have been previously treated. Measure units known as constant are updated before the topological sort. A previous step arranges formula to be easily accessed from the measure unit they define; formula of the form \( x = y \), being \( y \) a measure unit, are discarded, as far as they do not hold relevant information.

'\( \ll \)' \( \text{NF} \)-expressions require special care. Note that \( x \ll y \) is attached to the node containing \( x \) if and only if no equality definition exists for \( x \). In this case, a value for \( x \) must be conjectured such that the relationship holds; recalling the equational specification for \( \text{NF}_{\pi} \)-expressions, the algorithm chooses \( x = 1 \), because \( x \ll y \) holds then for any value of \( y \); as this is a strong decision that eventually may not be accepted by the designer, a warning must be given for this kind of assumptions; the designer could add an equality for \( x \) and reprocess the program to reject the assumption.
Name: FormulaTranslation

Input: G, directed graph resulting from MeasureUnitSelection.

S1, Sexp, sets of constant measure units selection.

n, basic measure unit.

Output: Tout, a mapping from measure units to single NF_n-expressions.

Algorithm:

VAR E, F, x: NF-expression; v: Node;

(* all measure units equal to n are set to n *)

Tout := CreateMapping;

FOR ALL x IN SetOfMeasureUnits(MUToNode(G, n)) DO
    Tout := AddEntry(Tout, x, n)
END;

G := MarkAsVisited(G, MUToNode(G, n));

(* constant measure units are updated first *)

FOR ALL x IN S1 DO Tout := AddEntry(Tout, x, 1) END;

FOR ALL x IN Sexp DO Tout := AddEntry(Tout, x, EXP) END;

(* iterated selection of candidate nodes *)

WHILE NOT AllNodesVisited(G) DO

    v := SelectOneFrom({x∈ Nodes(G) / x not visited ∧ x predecessors visited});

    G := MarkAsVisited(G, v);

    (* NF_n-expression computation from any formula for any m. unit *)

    (* O-Simplify computes the simplest form of a NF-expression *)

    E := O-Simplify(SubstituteRightHandSide(
        SelectOneFrom(SetOfFormula(v)), Tout)));

    (* all measure units declared as equal should have the same value *)

    FOR ALL x IN SetOfMeasureUnits(v) DO Tout := AddEntry(Tout, x, E) END;

    FOR ALL x IN SetOfMeasureUnits(v) DO
        FOR ALL F IN Get(T, x) DO
            IF SubstituteRightHandSide(F, Tout) ≠ E THEN Error
        END;
    END;

END;
Name: SubstituteRightHandSide
Input: $F$, formula defining a measure unit involving possibly many others measure units.
$T$, a (partial) mapping from measure units to formula.
Output: $E$, right-hand side of $F$ as a NF$_n$-expression.
Algorithm:

IF MUEquality($F$)
    THEN $E :=$ SubstituteOExpr($T$, Second($F$))
ELSIF $E :=$ Form0(1);

Name: SubstituteOExpr
Input: $E$, a general NF-expression involving possibly many measure units.
$T$, a (partial) mapping from measure units to formula.
Output: $E_{out}$, the equivalent of $E$ as a NF$_n$-expression.
Comment: SubstituteOExpr is a recursive procedure that follows tree form of NF-expressions until leaves, which are already NF$_n$-expressions, are reached.
Algorithm:

IF Root($E$) = '*' THEN
    $E_{out} :=$ Form2('*, SubstituteOExpr(First($E$), $T$), SubstituteOExpr(Second($E$), $T$))
ELSIF Root($E$) = 'log' THEN $E_{out} :=$ Form1('log', SubstituteOExpr(First($E$), $T$))
ELSIF Root($E$) = 'pot' THEN $E_{out} :=$ Form1('pot', SubstituteOExpr(First($E$), $T$))
ELSIF Root($E$) = 'exp' THEN $E_{out} :=$ Form1('exp', SubstituteOExpr(First($E$), $T$))
ELSIF ... (* treatment of other operators: TIME, ... *)
ELSE /* Root($E$) is a measure unit */ $E_{out} :=$ Form0(Get($T$, Root($E$)))
END;

4.2.4 Consistency Checking

Last, even if all previous steps have succeeded, a last consistency checking is necessary to test resulting values against comparisons, equalities and inequalities not used during the algorithm. These formula have been collected in a set during the first step; so, the procedure consists of obtaining its elements, substituting the measure units by their values and proving the validity or not of the resulting formula.
Name: ConsistencyCheck
Input: \( T \), a mapping from measure units to \( \text{NF}_n \)-expressions.
\( S \), a set of formula not used in previous steps.
Output: \( b \), a boolean indicating consistency of all the formula in \( S \).
Algorithm:
\[
\text{VAR } F: \text{NF-expression};
\]
\[
b := \text{true};
\]
\[
\text{WHILE } b \text{ AND NOT EmptySet?}(S) \text{ DO }
\]
\[
F := \text{SelectOneFrom}(S); S := \text{DeleteSet}(S, F);
\]
\[
F := \text{Form2(}\text{Root}(F), \text{SubstituteOExpr}(T, \text{First}(F)),
\]
\[
\text{SubstituteOExpr}(T, \text{Second}(F)));
\]
\[
b := \text{Evaluate}(F) (* \text{traversal of the parse tree form of the formula } *)
\]
END;

4.2.5 Translation Algorithm
The final algorithm may be written as the combination of the procedures presented in the previous subsections.

Name: MeasureUnitsTranslation
Input: \( H \), hierarchy of modules
Output: \( Tout \), a mapping from measure units to \( \text{NF}_n \)-expressions.
Errors: any error explained during this section.
Algorithm:
\[
<S1, S2> := \text{InformationRetrieval}(H);
\]
\[
G := \text{GraphCreation}(S1); TC := \text{TransitiveClosure}(G);
\]
\[
\text{Gdef} := \text{GraphSimplification}(G, S1, TC);
\]
\[
<n, Sct, Sexp, Gdef> := \text{MeasureUnitSelection}(\text{Gdef});
\]
\[
\text{Tout} := \text{FormulaTranslation}(\text{Gdef, Sct, Sexp, n});
\]
\[
\text{IF NOT ConsistencyCheck}(\text{Tout, S2}) \text{ THEN Error;}
\]
4.3 Checking Implementations against Efficiency Requirements

Once comparison and arithmetic for NF-expressions have been defined, including the necessary translation step, it remains to use this ability for checking implementations against requirements, both of them expressed as general NF-expressions.

To be more precise, we are interested in defining the mappings:

\[ \text{Restrictions}_{\mathcal{H}}: \text{ImplModules}_{\mathcal{H}} \rightarrow \text{DefModules}_{\mathcal{H}} \rightarrow \mathcal{T}(\text{ImplModules}_{\mathcal{H}}) \]

used in section 3 from given implementation properties and efficiency requirements at the appropriate levels. In order to do this, we introduce an algorithm \textit{SingleModuleChecking} together with some auxiliary procedures. The algorithm checks the set of implementations of a definition against a given requirement; it is supposed to be invoked many times at many levels (once for every restricted definition module at every possible level).

```
Name: SingleModuleChecking
Input: \( M \), a set of implementations corresponding to the definition module under study, \( U \).
\( R \), a boolean NF\(_n\)-expression (efficiency requirement over \( U \)).
\( STIM, SSP \), two mappings from \( U \) implementation modules to mappings from their public types, functions and schemes to NF\(_n\)-expressions (the efficiency properties of the implementations in \( M \)).
\( T \), a mapping from measure units to NF\(_n\)-expressions.
Output: \( M_{out} \), restriction of \( M \) after applying \( R \).
Algorithm: (* analysis of the NF\(_n\)-expression parse tree form *)

IF Root\((R)\) = "ImplementedWith" THEN \( M_{out} := \text{Intersection}(M, \cup \text{Children}(R)) \)
ELSE IF Root\((R)\) = 'NOT' THEN
    \( M_{out} := \text{Difference}(M, \text{SingleModuleChecking}(M, \text{First}(R), \text{STIM}, \text{SSP}, T)) \);
ELSIF Root\((R)\) = 'AND' THEN
    \( M_{out} := \text{Intersection}(\text{SingleModuleChecking}(M, \text{First}(R), \text{STIM}, \text{SSP}, T),\)
    \( \text{SingleModuleChecking}(M, \text{Second}(R), \text{STIM}, \text{SSP}, T)) \);
ELSIF Root\((R)\) = 'OR' THEN
    \( M_{out} := \text{Union}(\text{SingleModuleChecking}(M, \text{First}(R), \text{STIM}, \text{SSP}, T),\)
    \( \text{SingleModuleChecking}(M, \text{Second}(R), \text{STIM}, \text{SSP}, T)) \);
ELSIF Root\((R)\) = 'MIN' THEN
    \( M_{out} := \text{SelectMinimalCost}(M, R, \text{STIM}, \text{SSP}, T) \);
ELSE (* \( R \) must be a boolean expression of the form \( A < \text{rel}op > B *\))
    \( M_{out} := \{ m \in M / \text{SatisfiesRelationalExpression}(R, \text{STIM}, \text{SSP}, m, M, T) \} \);
```
Name: SelectMinimalCost
Input: \( M, R, STIM, SSP, T \), as before.
Output: \( M_{out} \), restriction of \( M \) after applying \( R \).
Algorithm: (* all implementations are checked to compute the minimum value satisfying the boolean expression *)

\[
\text{VAR Min: NF-expression; \quad (* stores the minimal value at every moment *)}
\]
\[
\text{Min := \( b^n \); Mout := EmptySet;}
\]
\[
\text{FOR ALL } m \text{ IN M DO}
\]
\[
\text{IF SubstituteTypesAndFunctions(Second(R), STIM, SSP, m, M, T) = true THEN}
\]
\[
\text{IF SubstituteTypesAndFunctions(First(E), STIM, SSP, m, M, T) < Min THEN}
\]
\[
\text{Min := SubstituteTypesAndFunctions(First(E), STIM, SSP, m, M, T);}
\]
\[
\text{Mout := EmptySet}
\]
\[
\text{ELSIF SubstituteTypesAndFunctions(First(E), STIM, SSP, m, M, T) = Min THEN}
\]
\[
\text{Mout := Union(Mout, m)}
\]
\[
\text{END;}
\]
\[
\text{END;}
\]

Relational expression satisfactibility consists of converting the general NF-expressions into \( NF_n \)-expressions by substituting TIME, SPACE and MIN operators by their values and by translating all measure units into a function on \( n \). Then, expressions can be compared using \( NF_n \)-expressions laws. Note that the substitution procedure assumes that TIME and SPACE operators are applied over single symbol names; also, note that OPS and TYPES do not appear in the algorithm; we suppose that both situations are dealt with in parse tree building time (TIME and SPACE on symbol lists are expanded to as many single TIME and SPACE operators as required; OPS and TYPES are expanded to all involved symbols).

Name: SatisfiesRelationalExpression
Input: \( R \), a relational general \( NF_n \)-expression.
\[
STIM, SSP, \text{ two mappings from implementation modules to mappings from their public types, functions and schemes to } NF_n \text{-expressions (the efficiency properties of the implementations in } M)\).
\]
\[
M, \text{ the set of implementations corresponding to the definition module under study.}
\]
\[
m, \text{ an implementation module to be tested.}
\]
\[
T, \text{ a mapping from measure units to } NF_n \text{-expressions.}
\]
Output: \( b \), a boolean indicating if \( S \) satisfies \( R \).
Algorithm:

\[
\begin{align*}
E_{\text{Left}} := & \text{SubstituteTypesAndFunctions(First(R), STIM, SSP, m, M, T);} \\
E_{\text{Right}} := & \text{SubstituteTypesAndFunctions(Second(R), STIM, SSP, m, M, T);} \\
b := & (\text{Form2(Root(R), E_{\text{Left}}, E_{\text{Right}}) = true})
\end{align*}
\]

END;

Name: SubstituteTypesAndFunctions

Input: \( E \), an arithmetic general NF\(_{n}\)-expression (including MIN, TIME and SPACE).

\( \text{STIM}, \text{SSP} \), two mappings from implementation modules to mappings from their public types, functions and schemes to NF\(_{n}\)-expressions (the efficiency properties of the implementations in \( M \)).

\( M \), the set of implementations corresponding to the definition module under study.

\( m \), an implementation module to be tested.

\( T \), a mapping from measure units to NF\(_{n}\)-expressions.

Output: \( E_{\text{out}} \), an arithmetic NF\(_{n}\)-expression (without MIN, TIME nor SPACE).

Algorithm:

\( (* \text{TIME and SPACE expressions require access to mapping values} *) \)

IF Root(E) = 'TIME' THEN Eout := Form0(Get(Get(STIM, m), First(E)))
ELSIF Root(E) = 'SPACE' THEN Eout := Form0(Get(Get(SSP, m), First(E)))
ELSIF Root(E) = '+' THEN
    Eout := Form2('+', SubstituteTypesAndFunctions(First(E), STIM, SSP, m, M), 
             SubstituteTypesAndFunctions(Second(E), STIM, SSP, m, M))
ELSIF ...
    (* similar for the rest of valid operators *)
ELSE Eout := Form0(Get(T, Root(E)))
    (* a leave containing a single measure unit *)
END;

4.4 An Example

In this section we present an example that shows the evolution of the algorithms presented before. To be illustrative enough in a short space, we just introduce measure units, properties and requirements.

4.4.1 Hierarchy of Modules

A hierarchy of definition modules is shown at the figure below. The top module is the main program (the only implementation module in this hierarchy) which, as it has been said, may declare some final properties about measure units. The relationships stated in the hierarchy
allows later to select a basic measure unit \( n \) and to express the rest of measure units as \( \text{NF}_n \)-expressions.

### 4.4.2 Translation of Expressions

Here we address to the problem of translating all the formula into \( \text{NF}_n \)-expressions. First, formula are collected into two sets traversing the hierarchy, depending on their use or not in this translation.

\[
\text{InformationRetrieval}(H) \rightarrow S_1 = \{ x << a, y << a, b = a, z = \text{pot}(b, 2), t = z \times \log(b), \\
    s = t, f = 1, c = \text{pot}(f, 2), a = c \times r \}
\]

\[
S_2 = \{ z > 1, a < t \}
\]

Set \( S_1 \) is the basis for generating the directed and labelled graph \( G \), with a node for each measure unit and an edge or more for each formula (\( b^n \) node is not shown):
(Note the direction of '→' edges.) Next, G transitive closure is computed resulting the following matrix with no cycles (empty cells stand for '⊥' values):

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f</th>
<th>r</th>
<th>s</th>
<th>t</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>l</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>c</td>
<td>c</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td>c</td>
<td>c</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For instance, the transitive inclusion $a ≡ s$ follows from $a = b$, $b ≡ t$ and $t = s$.

Now, the graph may be simplified by merging equal nodes and discarding equal edges, yielding to $G_{def}$:

```
```

Formula have been added to nodes; note that equalities between measure units are not necessary to state because they may be deduced from the graph structure.

From $G_{def}$, the basic measure unit should be selected once constant nodes have been temporarily deleted, in this case yielding to the graph:
and the sets \( Sct = \{ f, c \} \) and \( Sexp = \emptyset \).

Then, the basic measure unit is clearly \( r \) and the others are expressed as \( O_r \)-expressions:

\[
\text{FormulaTranslation}(\text{Gdef}, \{ f, c \}, \emptyset, r) \rightarrow f = 1; c = 1 \text{ (constant propagation)}
\]

\[
r = r \text{ (first step)}
\]

\[
a = r; b = r \text{ (second step)}
\]

\[
z = \text{pot}(r, 2) \text{ (third step)}
\]

\[
x = 1; y = 1 \text{ (fourth step)}
\]

\[
t = s = \text{pot}(r, 2) \ast \text{log}(r) \text{ (fifth step)}
\]

(Other orderings are possible in the last three steps.)

Last, consistency of set \( S2 \) is checked against these formula:

\[
z > 1 \rightarrow \text{pot}(r, 2) > 1 \rightarrow \text{true}
\]

\[
a < t \rightarrow r < \text{pot}(r, 2) \ast \text{log}(r) \rightarrow \text{true}
\]

### 4.4.3 Requirements Checking

Last, let's see how some requirements may be checked against a given set of implementations. Let's focus on \( M2 \), and let's suppose that there exist four different implementations for it, \( M21 \) to \( M24 \), with efficiency:

\[
\begin{align*}
\text{M21: } \text{TIME}(f1) &= s; \text{TIME}(f2) = s \ast \text{log}(a); \text{TIME}(f3) = a + s; \\
\text{M22: } \text{TIME}(f1) &= a; \text{TIME}(f2) = \text{pot}(s, 2); \text{TIME}(f3) = a + s; \\
\text{M23: } \text{TIME}(f1) &= 1; \text{TIME}(f2) = s \ast \text{log}(a); \text{TIME}(f3) = \text{pot}(s, 2); \\
\text{M24: } \text{TIME}(f1) &= 1; \text{TIME}(f2) = s; \text{TIME}(f3) = a + s;
\end{align*}
\]

for three functions \( f1, f2 \) and \( f3 \) defined inside \( M2 \). Now, let's suppose that \( M1 \) states as requirement the NF-expression at module level:

\[
\text{REQUIREMENTS ON M2:}
\]

\[
\text{TIME}(f1) < \text{pot}(a, 2) \text{ AND MIN(TIME}(f3), \text{TIME}(f2) < \text{pot}(a, 3))
\]
As the requirement is stated at module level, the input value of $M$ is the set $\{M21, M22, M23, M24\}$ of all possible implementations. As the NF-expression is a conjunction, both operands are checked and they result are intersected to obtain the set of modules satisfying both parts.

The first expression results in the set $\{M22, M23, M24\}$:

- **M21**: $\text{TIME}(f1) < \text{pot}(a, 2) \rightarrow s < \text{pot}(a, 2) \rightarrow \text{pot}(r, 2) * \log(r) < \text{pot}(r, 2) = \text{false}

- **M22**: $\text{TIME}(f1) < \text{pot}(a, 2) \rightarrow a < \text{pot}(a, 2) \rightarrow r < \text{pot}(r, 2) = \text{true}

- **M23**: $\text{TIME}(f1) < \text{pot}(a, 2) \rightarrow 1 < \text{pot}(a, 2) \rightarrow 1 < \text{pot}(r, 2) = \text{true}

- **M24**: $\text{TIME}(f1) < \text{pot}(a, 2) \rightarrow 1 < \text{pot}(a, 2) \rightarrow 1 < \text{pot}(r, 2) = \text{true}

(Note the two-step substitution: first, $\text{TIME}(f1)$ by its value in every implementation and, next, the result is translated into an NF$_r$-expression.)

The second NF-expression involves the MIN operator; so, every implementation from $M$ is studied to see if the boolean NF-expression of MIN holds, which is mandatory for it to be considered, and then its cost is checked:

- **M21**: $\text{TIME}(f2) < \text{pot}(a, 3) \rightarrow s * \log(a) < \text{pot}(a, 3) \rightarrow (\text{pot}(r, 2) * \log(r)) * \log(r) \rightarrow \text{pot}(r, 3) = \text{false}

  - $\text{TIME}(f3) = a + s \rightarrow r + \text{pot}(r, 2) * \log(r) = \text{pot}(r, 2) * \log(r)$

- **M22**: $\text{TIME}(f2) < \text{pot}(a, 3) \rightarrow \text{pot}(s, 2) < \text{pot}(a, 3) \rightarrow \text{pot}(\text{pot}(r, 2) * \log(r), 2)

  - $\text{pot}(r, 3) = \text{pot}(\text{pot}(r, 2), 2) * \log(r), 2 < \text{pot}(r, 3) = \text{pot}(r, 4) * \log(r), 2 < \text{pot}(r, 3) = \text{false}

  - $\text{TIME}(f3) = a + s \rightarrow r + \text{pot}(r, 2) * \log(r) = \text{pot}(r, 2) * \log(r)$

- **M23**: $\text{TIME}(f2) < \text{pot}(a, 3) \rightarrow \text{pot}(s, 2) \rightarrow \text{as M21 ... = true}

  - $\text{TIME}(f3) = \text{pot}(s, 2) \rightarrow \text{pot}(\text{pot}(r, 2) * \log(r), 2) = \text{pot}(r, 4) * \log(r), 2$

- **M24**: $\text{TIME}(f2) < \text{pot}(a, 3) \rightarrow s < \text{pot}(a, 3) \rightarrow \text{pot}(r, 2) * \log(r) < \text{pot}(r, 3) = \text{true}

  - $\text{TIME}(f3) = a + s \rightarrow r + \text{pot}(r, 2) * \log(r) = \text{pot}(r, 2) * \log(r)$

For instance, the computation of the boolean NF-expression for M21 follows two substitution steps as before, and then laws on NF-expression calculus are applied:

- $(\text{pot}(r, 2) * \log(r)) * \log(r) < \text{pot}(r, 3) = \{\text{by associativity of product}\}$

- $\text{pot}(r, 2) * (\log(r) * \log(r)) < \text{pot}(r, 3) = \{\text{by } x^k \cdot x^r = x^{k+r} \text{ law, with } k = r = 1\}$

- $\text{pot}(r, 2) * \text{pot}(\log(r), 2) < \text{pot}(r, 3) = \{\text{by fourth law on } <, \text{ with } k = r = 2, i = 3, s = 0\}$

From these results, **M22** is discarded as candidate because the boolean NF-expression does not hold and the resulting set is $\{M21, M24\}$, as **M23** has worse efficiency results than them.

So, the final result for the requirement happens to be $\{M22, M23, M24\} \cap \{M21, M24\} = \{M24\}$. This means that, unless otherwise stated in lower levels (type level, ...), all objects declared inside MI from any type exported by M2 are selected to be implemented with M24, and that all function calls for $f1$, $f2$ and $f3$ invoke the procedures coded in M24. Moreover, note that further restrictions at lower levels only can yield to an empty set of implementations, in which case restrictions should be relaxed somewhere.
5 Interaction of Objects with Different Implementations

Once selection of implementations has been totally defined, a question is left. According to their particular requirements stated at different levels, it could be the case for two objects of the same type to require two different implementations, which is not allowed in Modula-2 programs. In this section, we show how this situation is coped and, to be more precise, how these objects may be used indistinctly in the same context and how they may even interact freely provided that an abstraction function is defined for their types. Also, we will see how the scheme can be extended to procedure calls, and we will address to the problem of how this kind of interaction may interfere with efficiency costs and how the designer may prevent this problem.

5.1 The Abstraction Function and the Representation Function

Given a type \( t \) introduced in a definition \( T \) and implemented in two different .MODth files \( \text{Timp1} \) and \( \text{Timp2} \), given an operation \( \text{PROCEDURE} \ f (a, b; t; ...) \) defined in \( T \) and given two objects \( x \) and \( y \) of type \( t \), selected to be implemented with \( \text{Timp1} \) and \( \text{Timp2} \) respectively, the former question may be summarised as: is the call \( f(x, y, ...) \) executable or not? Obviously, both \( \text{Timp1} \) and \( \text{Timp2} \) are required to have their own implementation for \( f \), built on their particular representation of type \( t \), and they should be not aware of whether other implementations for \( t \) do exist. So, other alternatives to support the execution of these "mixed" expressions are considered:

- To write a huge collection of implementations for \( f \), one for every "interesting" (or even, possible) combination of its parameter implementations.
- To explicitly define conversion functions between any pair of implementations for \( t \).
- To define a function to convert objects from any implementation for \( t \) to a common domain, and vice versa.

The first two methods suffer from a lack of modularity (where are those functions to be put in?) and they are not well suited for program maintenance (construction of new implementations require modification of older ones). So, we choose the third option, and then it is necessary to fix the common domain.

The simplest solution is to define a pair of functions, the abstraction function and the representation function. The first one [Hoa72] maps an object to an equivalent term (i. e., a sequence of operations) at the definition level, while the second one performs the inverse mapping. Given a definition \( T \) introducing an ADT \( t \), both functions should be defined in every module \( A \) implementing \( T \), yielding to two families \((\text{abs}_A, t)\) and \((\text{repr}_A, t)\) of abstraction and representation functions. The important point is that, given an object \( x \) of type \( t \), its implementation \( V \) may be switched to \( W \) by first applying \( \text{abs}_V, t \) over it and then \( \text{repr}_W, t \).

Abstraction functions are written in implementations just as any other function; among other things, auxiliary functions may be defined to be used inside them. A Modula-NF predefined type \( \text{TERM} \) is available to store the result of \( \text{abs}_A, t \) in a parse-tree form; it is a generic type whose operations are those from \( t \). Also, efficiency should be stated in the file header; in this sense, operations on terms are \( O(1) \) in time, and the space of the resulting term is about the
number of calls to operations on terms during abstraction function execution (which, in the
general case, will be a function on the size of the data structure implementing \( t \)).

On the other hand, the representation function is just the execution of the operations contained
in the term into the chosen implementation. This is no more than a preorder traversal of the term
considered as a tree; so, its code does not have to be supplied by the implementer. However, its
efficiency should be stated; it is trivial to compute from the composition of the tree (that is, its
size and which operation symbols are stored in its nodes). As a convention, the name of the
representation function for a type \( t \) is \( t_{Repr} \).

The next figure presents three examples of abstraction functions for tables and graphs as
defined in section 2. The first one is the abstraction function for tables implemented with
ordered lists; it is easily defined in a recursive manner, and this will be the case in a great deal
of ADTs, mainly when implemented with linked structures. The other two examples are
abstraction functions for graphs; in the case of an adjacency matrix, a nested loop is defined to
traverse all the positions of the array (9999 gre value stands for free array position); this loop is
usual when an array has to be explored. In the case of adjacency lists, the implementation of
lists by means of an already defined ADT requires using its operations instead of directly
accessing to the representation; an auxiliary function has been defined to treat every list.

```
IMPLEMENTATION MODULE TableList FOR Table;
    FROM ...;
    EFFICIENCY ...
        SPACE(tableAbs) = n; TIME(tableAbs) = n; TIME(tableRepr) = \text{poly}(n, 2);
    ...
    TYPE table = POINTER TO node;
        node = RECORD
            k, v: message;
            link: table
        END;

    PROCEDURE tableAbs (x: table): TERM;
        VAR t: TERM;
        BEGIN
            IF x = NIL THEN t := TableCreate;
                ELSE t := TableAdd(tableAbs(x^, link), x^, k, x^, v);
            END;
            RETURN t;
        END;
    ...
END TableList.
```
IMPLEMENTATION MODULE AdjacencyMatrix FOR Graph;
    FROM ...;
    EFFICIENCY ...
        SPACE(graphAbs) = e; TIME(graphAbs) = po(n, 2); TIME(graphRepr) = po(n, 2);
    ...
    TYPE graph = POINTER TO privgraph; privgraph = ARRAY [1..n, 1..n] OF INTEGER;
    PROCEDURE graphAbs (g: graph): TERM;
    VAR u, v: CARDINAL; t: TERM;
    BEGIN
        t := GraphCreate;
        FOR u := 1 TO n DO
            FOR v := 1 TO n DO IF g^[u, v] <> -99999 THEN t := GraphAdd(t, u, v, g^[u, v]) END;
        END;
        RETURN t;
    END;
    ...
END AdjacencyMatrix.

IMPLEMENTATION MODULE AdjacencyLists FOR Graph;
    FROM ...;
    EFFICIENCY ...
        SPACE(graphAbs) = e; TIME(graphAbs) = e; TIME(graphRepr) = e^n;
    ...
    TYPE requires List IMPLEMENTED WITH ListCardIntByPointers;
        graph = POINTER TO privgraph; privgraph = ARRAY [1..n] OF List;
    PROCEDURE graphAbs (g: graph): TERM;
    VAR u: CARDINAL; t: TERM;
    BEGIN
        t := GraphCreate;
        FOR u := 1 TO n DO AdjListAbs(t, u, g^[u]) END;
        RETURN t;
    END;
    PROCEDURE AdjListAbs (VAR t: TERM; u: CARDINAL; l: List);
    VAR v: CARDINAL; x: INTEGER;
    BEGIN
        ListReset(l);
        WHILE NOT ListEol(l) DO BEGIN
            ListGet(l, v, x); ListNext(l); t := GraphAdd(t, u, v, x)
        END;
    END;
    ...
END AdjacencyMatrix.
As far as their only goal is to support implementation switching, abstraction functions are not mandatory in MODULA-NF programs. However, it should remain clear that the absence of the abstraction function for a type prevents its full compatibility in all contexts.

5.2 Switching Implementations during Execution

Here we fix which situations in a program require a switch from one implementation to another and also which transformations are required when these situations happen. Throughout this section, we suppose that implementation selection functions as introduced in subsection 3.2 are well-defined (that is, they determine at the end a single implementation for each variable, parameter and procedure call in every procedure of every implementation).

Let $t$ be a type introduced in a definition $T$ and let $Timp1$ and $Timp2$ be two different implementations for $T$. Let $\text{IMP}_{\text{VAL,H}}$ be a shorthand of: "$\text{IMP}_{\text{VAR,H}}, \text{IMP}_{\text{PARAM,H}}$ or $\text{IMP}_{\text{CALL,H}}$ depending on the kind of value involved". There are three situations that require to apply the abstraction function:

**T1.** Assignment rule. Given two values $x$ and $y$ ($x$ is a -component of a- variable or parameter; $y$ may be a function result in addition) of type $t$ introduced in a procedure $f$ coded inside an implementation $Uimp$, which are implemented with $Vimp$ and $Wimp$ respectively, that is $\text{IMP}_{\text{VAL,H}}(Uimp)(f)(x) = \{Vimp\}$ and $\text{IMP}_{\text{VAL,H}}(Uimp)(f)(y) = \{Wimp\}$, the assignment $x := y$ requires to transform $y$ from $Wimp$ to $Vimp$:

$$x := y \rightarrow x := \text{repr}_{Vimp,t}(\text{abs}_{Wimp,t}(y))$$

provided that $Vimp \neq Wimp$.

**T2.** Parameter rule. Given a value $x$ of type $t$ used in a procedure $g$ coded inside an implementation $Uimp$ and implemented with $Vimp$, $\text{IMP}_{\text{VAL,H}}(Uimp)(g)(x) = Vimp$, and given a procedure $f$ implemented in a .MODth file $Fimp$ with a formal parameter $y$ of type $t$ implemented with $Wimp$, $\text{IMP}_{\text{PARAM,H}}(Fimp)(f)(y) = Wimp$, calling $f(\ldots x \ldots)$ with $x$ as the actual parameter bound to $y$ requires:

**T2.i.** To transform $x$ from $Vimp$ to $Wimp$ before executing $f$, if the call is by value:

$$f(\ldots x \ldots) \rightarrow f(\ldots \text{repr}_{Wimp,t}(\text{abs}_{Vimp,t}(x)) \ldots)$$

**T2.ii.** To apply T2.i and then to transform $y$ from $Wimp$ to $Vimp$ after executing $f$, if the call is by reference?:

$$f(\ldots x \ldots) \rightarrow \text{aux} := \text{repr}_{Wimp,t}(\text{abs}_{Vimp,t}(x));$$

$$f(\ldots \text{aux} \ldots); x := \text{repr}_{Vimp,t}(\text{abs}_{Wimp,t}(\text{aux}))$$

being $\text{aux}$ of type $t$ implemented with $Wimp$.

provided that $Vimp \neq Wimp$. The rule is applied to every parameter matching this case.

---

6 Let’s suppose that predefined Modula-2 constant values, as 4, 3.12, etc., are special cases of calls to procedures defined in MODULA2.DEFth and implemented in MODULA2.IMPth. The same assumption is made for Modula-2 predefined operators (sums, etc.).

7 In this case, $x$ must be a (component of a) variable or a parameter.
T3. Compatibility rule. Given two values $x$ and $y$ of type $t$ introduced in a procedure $g$ coded inside an implementation $Uimp$ and implemented with $Vimp$ and $Wimp$ respectively, $\text{IMP}_{V,H}(Uimp)(g)(x) = Vimp$ and $\text{IMP}_{V,H}(Uimp)(g)(y) = Wimp$, and given a procedure $f$ defined in $T$ (that is, $f$ may manipulate the type representation of $t$) with two formal parameters $p$ and $q$ of type $t$, calling $f(...x...y...)$ with implementation $Fimp$ and with $x$ and $y$ as the actual parameters bound to $p$ and $q$, such that $Fimp$, $Vimp$ and $Wimp$ are not equal at the same time, requires:

T3.i. To transform $y$ from $Wimp$ to $Vimp$ if $Fimp = Vimp$, or to transform $x$ from $Vimp$ to $Wimp$ if $Fimp = Wimp$. This will be the usual case where an implementation acts as default, and an individual value has been chosen with a different one.

$$Fimp = Vimp \Rightarrow f(...x...y...) \rightarrow f(...x...\text{repr}_{Vimp,t}(\text{abs}_{Vimp,t}(y))...)$$

$$Fimp = Wimp \Rightarrow f(...x...y...) \rightarrow f(...\text{repr}_{Wimp,t}(\text{abs}_{Wimp,t}(x))...)$$

T3.ii. To transform $x$ from $Vimp$ to $Fimp$ and to transform $y$ from $Wimp$ to $Fimp$, if $Fimp \neq Vimp$ and $Fimp \neq Wimp$. Normally, it will happen that $Vimp = Wimp$, being the default implementation, and $Fimp$ will be a .MODTh file where $f$ has any interesting property.

$$Fimp \neq Vimp \land Fimp \neq Wimp \Rightarrow f(...x...y...) \rightarrow f(...\text{repr}_{Fimp,t}(\text{abs}_{Vimp,t}(x))...\text{repr}_{Fimp,t}(\text{abs}_{Wimp,t}(y))...)$$

In any case, the transformation will be done applying the rule T2 with respect to the parameter mode. The rule is easily generalised for more than two parameters.

An example will show how these rules are applied during execution. Given the hierarchy of modules presented at subsection 4.4, let's suppose that $M2$ defines a type $c$ and three operations $f1: \rightarrow c, f2: c \rightarrow c$ and $f3: c \rightarrow c$, the first two defined as functions and the other one as a procedure, PROCEDURE $f1$ RETURNS $c$, PROCEDURE $f2(\ p; \ c)$ RETURNS $c$, PROCEDURE $f3$ (VAR $p; \ c; \ q; \ c$). Also, let's imagine that the main program $MI$ states an efficiency requirement on $M2$ at module level such that $M2I$ is the selected implementation, and also that it declares three variables of type $c$, VAR $x, y; \ c; z; \ c$ IMPLEMENTED WITH $M22$ (so, $x$ and $y$ are implemented with $M21$). With this scenario, we present below a collection of assignments and procedure calls, showing which rule or rules are applied in every case (if any).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Rule(s) Applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := y$</td>
<td>no rule applied</td>
</tr>
<tr>
<td>$x := z$</td>
<td>rule T1; it is interpreted as $x := \text{repr}<em>{M21,c}(\text{abs}</em>{M22,c}(z))$</td>
</tr>
<tr>
<td>$z := f1$</td>
<td>rule T1; it is interpreted as: $z := \text{repr}<em>{M22,c}(\text{abs}</em>{M21,c}(f1))$</td>
</tr>
<tr>
<td>$x := f2(y)$</td>
<td>no rule applied</td>
</tr>
<tr>
<td>$x := f2(z)$</td>
<td>rule T2.i; it is interpreted as: $x := f2(\text{repr}<em>{M21,c}(\text{abs}</em>{M22,c}(z)))$</td>
</tr>
<tr>
<td>$z := f2(z)$</td>
<td>rules T2.i and T1; it is interpreted as: $z := \text{repr}<em>{M22,c}(\text{abs}</em>{M21,c}(f2(\text{repr}<em>{M21,c}(\text{abs}</em>{M22,c}(z)))))$</td>
</tr>
<tr>
<td>$z := M22.f2(z)$</td>
<td>no rule applied</td>
</tr>
</tbody>
</table>
f3(x, y)  no rule applied
f3(x, z)  rule T3.i, using T2.i for z; it is interpreted as:
         f3(x, repr_{M21,c}(abs_{M22,c}(z)))
f3(z, x)  rule T3.i, using T2.ii for z; being aux an object of type c implemented with M21, it is interpreted as:
         aux := repr_{M21,c}(abs_{M22,c}(z))
         f3(aux, x)
         z := repr_{M22,c}(abs_{M21,c}(aux))
M22.f3(x, y)  T3.ii, using T2.ii for x and T2.i for y; being aux an object of type c implemented with M22, it is interpreted as:
             aux := repr_{M22,c}(abs_{M21,c}(x))
             f3(aux, repr_{M22,c}(abs_{M21,c}(y)))
             x := repr_{M21,c}(abs_{M22,c}(aux))

5.3 Preventing Implementation Switching

From the efficiency of the abstraction and representation functions shown in 5.1, which are representative of the general case, it becomes clear that the cost of switching between implementations may affect the global cost of algorithms. If we want implementation efficiency properties to be accurate and thus implementation selection to be reliable, analysis of algorithms should take into account which places may require switching of implementations (as described in 5.2) and then, knowing the cost of abstraction and representation functions, state the final cost\(^8\). In the rest of the section, we enrich the set of operators for stating efficiency properties and efficiency requirements; concerning properties, operators to state them depending on used implementations are introduced; concerning requirements, new constructs to prevent implementation switching if desired are added. Next, implementation selection rules are redefined to fit into these new constructions, as well as the algorithm for checking implementations against requirements.

5.3.1 Efficiency Properties Revisited

Two new operators are added to the language. The operator IMPL(x) gives the name of the implementation associated to x inside the type t or the procedure f whose efficiency is being stated. In the case of types, x may be the name of other types or even definition modules; in the case of procedures, x may be the name of a type of any parameter appearing in f's header, a procedure invoked inside f's body, a parameter appearing in f's header, or even a definition module.

The result of IMPL(x) can be compared to other IMPL(y) or to an appropriate implementation name (which must be quoted) with an IF-THEN-ELSE-END operator used as a NF-expression. Depending on the satisfactibility or not of a boolean expression, the first or the

---

\(^8\) Note that, taking implementation switching into account, efficiency results may be very high values preventing the implementation to be selected in most of its contexts of use.
second NF-expression is selected as the result of whole conditional expression. The boolean expression will generally test implementation relationships using the IMPL operator or else relationships between measure units, while the NF-expressions will reflect final costs depending on whether implementation switching occurs or not in the procedure.

For instance, let's suppose we have a procedure \( f \) inside an implementation module \( M \) using the \texttt{SetTable} module introduced in 2.4 such that \( f \) receives two objects of type \texttt{table}, \( x \) and \( y \), as parameters, and such that \( f \)'s main task consists on calling the procedure \texttt{TableUnion} with \texttt{SetTableImp1} implementation. It is clear that \( f \)'s time cost will depend on the implementation of \( x \) and \( y \), which is unknown when writing \( M \); so, the final time efficiency may be stated as a conditional NF-expression\(^9\):

\[
\text{TIME}(f) = \text{IF} (\text{IMPL}(x) = \text{"SetTableImp1"}) \text{ AND } (\text{IMPL}(y) = \text{"SetTableImp1"}) \\
\hspace{1cm} \text{THEN } n \text{ ELSE } n + \text{TIME(tableAbstr) END}
\]

assuming that \texttt{tableRepr} for \texttt{SetTableImp1} is \( O(n) \), as it appears to be from the constant time of \texttt{TableAdd} (note that the time for the abstraction function is not known in advance because the source implementations for \( x \) and \( y \) are not determined).

The scheme would be very close if the implementation for \texttt{TableUnion} call were not known:

\[
\text{TIME}(f) = \text{IF} (\text{IMPL}(x) = \text{IMPL(TableUnion)} \text{ AND } (\text{IMPL}(y) = \text{IMPL(TableUnion)}) \\
\hspace{1cm} \text{THEN } n \text{ ELSE } n + \text{TIME(tableAbstr) + TIME(tableRepr) END}
\]

In this case, \( \text{TIME(tableRepr)} \) must be stated because the goal implementation is also unknown.

Efficiency properties may look simpler again with the use of new requirement features introduced next.

### 5.3.2 Efficiency Requirements Revisited

Two features are considered when stating efficiency requirements. First, an IMPLEMENTED LIKE operator may be used at object level and procedure call level requirements; it serves to force an object to be implemented as another one, or a procedure call to be executed with the same implementation as a given object. Note that, with this kind of restriction, a procedure call may be executed without any implementation switching, as for instance in:

\[
\text{PROCEDURE } f (...) \text{;} \\
\text{VAR } ... \text{ y IMPLEMENTED LIKE } x \text{;} \\
\hspace{1cm} g(x, y) \text{ IMPLEMENTED LIKE } x; \\
\hspace{1cm} ...
\]

\(^9\) In this case, we have considered that the cost of changing the implementation for the parameter has to be considered in the calling procedure and not in the called one. This is an arbitrary decision; alternatively, implementation switching may be considered as part of procedure execution. In this example, this would imply modification of \texttt{SetTable} efficiency properties.
(A default rule will be introduced in the next subsection to relax the procedure call explicit requirement.)

The second construction is the concept of implementation inheritance, which is used to force implementations to propagate inside the module hierarchy. Being $U$ an implementation, being $M$ the definition being restricted inside $U$ and being $f$ a procedure coded inside $U$, inheritance may be stated at four different levels:

- Module level. The set of implementations associated to $M$ is the intersection of the implementations associated to $M$ at module level in the modules using $U$. Further requirements may be stated at this or lower levels.

- Type level. The set of implementations associated to $M$ inside the type representation area is the intersection of the implementations associated to $M$ inside those modules which type representation area also uses types from $M$, independently of which implementations were associated to $M$ at module level. Further requirements may be stated at this level or at parameter level.

- Procedure level. The set of implementations associated to $M$ inside $f$ is the intersection of the implementations associated to $M$ inside those procedures calling $f$, independently of which implementations were associated to $M$ in upper levels. Further requirements may be stated at this level or at parameter level.

- Parameter level. As before, but applied just to the restricted parameter.

This construction is designed to be used when an implementation uses a definition without efficiency constraints, either because they are not clear, or because the implementation is of wide use and constraints could prevent its use in some contexts; also, it could be used to prevent implementation switching. In all these cases, the final implementation is up to the context of use.

For instance, both features could be used to add a requirement on $Table$ in the implementation $SetTableImp2$ of 2.4: on the one hand, $Table$ is required at module level to inherit implementation from its users but forcing ordered traversal to be lineal:

```plaintext
REQUIRES ON Table (INHERITS); TIME(ord_traversal) = n;
```

Moreover, $SetTable$ operations are required to have equal implemented parameters:

```plaintext
PROCEDURE TableUnion (t1, t2 IMPLEMENTED LIKE t1: table;
VAR t3 IMPLEMENTED LIKE t2: table);
```

(As usual, this low level requirements act on the result of upper levels.)

Note that, strictly speaking, $Table$ procedure calls inside procedure bodies should also be forced to be selected with the same implementations as their parameters. As it would be too tedious, implementation selection rules revisited at the next subsection define a default behaviour that saves this task.
5.3.3 Implementation Selection Rules Revisited

It is necessary to fit implementation inheritance into the implementation selection rules presented at section 3. Mainly, inheritance affects rules in the set of implementations that are intersected with the one coming from restrictions. To define the new rules, many auxiliary domains concerning inherited modules at many different levels are needed; as far as all levels are modified the same way, we just present syntactic entities and selection rule for module level.

\[ \text{InheritModules}_{H} : \text{ImplModules}_{H} \rightarrow \mathcal{P}(\text{DefModules}_{H}) \]
\[ \text{InheritModules}_{H}(U_i) = \{U_d\} / \{U_d\} \text{ are restricted with inheritance at module level in } U_i \]
\[ \text{InheritedModules}_{H} : \text{ImplModules}_{H} \rightarrow \text{DefModules}_{H} \rightarrow \mathcal{P}(\text{ImplModules}_{H}) \]
\[ \text{InheritedModules}_{H}(U_i)(U_d) = \{U'_{i}\} / \{U'_{i}\} \text{ is the set of implementations inherited for } U_d \text{ in } U_i \]

**P76** All modules restricted with inheritance must be really restricted.

\[ \forall U_i : U_i \in \text{ImplModules}_{H} : \text{InheritModules}_{H}(U_i) \subseteq \text{RestrictedModules}_{H}(U_i) \]

**P77** Inherited modules are defined on modules restricted with inheritance.

\[ \forall U_i : U_i \in \text{ImplModules}_{H} : \text{dom(InheritedModules}_{H}(U_i)) = \text{InheritModules}_{H}(U_i) \]

**P78** Inheritance over a definition must refer to its valid implementations.

\[ \forall U_i : U_i \in \text{ImplModules}_{H} : \]
\[ \forall U_d : U_d \in \text{InheritModules}_{H}(U_i) : \]
\[ \text{InheritedModules}_{H}(U_i)(U_d) = \text{DefToImp}_{H}(U_d) \]

Selection rule definition:

\[ \forall U_i : U_i \in \text{ImplModules}_{H} : \]
\[ \forall U_d : U_d \in \text{ImportedModules}_{H}(U_i) \land U_d \in \text{RestrictedModules}_{H}(U_i) \land \]
\[ U_d \in \text{InheritModules}_{H}(U_i) : \]
\[ \text{IMP}_{\text{MOD},H}(U_i)(U_d) = \text{Restrictions}_{H}(U_i)(U_d) \cap \text{InheritedModules}_{H}(U_i)(U_d) \land \]
\[ \forall U_d : U_d \in \text{ImportedModules}_{H}(U_i) \land U_d \in \text{RestrictedModules}_{H}(U_i) \land \]
\[ U_d \notin \text{InheritModules}_{H}(U_i) : \text{IMP}_{\text{MOD},H}(U_i)(U_d) = \text{Restrictions}_{H}(U_i)(U_d) \land \]
\[ \forall U_d : U_d \in \text{ImportedModules}_{H}(U_i) \land U_d \notin \text{RestrictedModules}_{H}(U_i) : \]
\[ \text{IMP}_{\text{MOD},H}(U_i)(U_d) = \text{DefToImp}_{H}(U_i)(U_d) \land \]

68
The value of InheritedModules\textsubscript{\(H\)} is computed by the selection algorithm, as outlined in 5.3.4.

On the other hand, once the problem of implementation switching has been presented, the selection rule for procedure calls becomes inconvenient, in the sense that, when switching is not desired and all of the parameters \(x_i\) belonging to types introduced in the same definition \(U_d\) that the called procedure \(g\) are such that IMP\textsubscript{PARAM,\(H\)}(\(U_i\))(\(f\))(\(x_i\)) = IMP\textsubscript{PARAM,\(H\)}(\(U_i\))(\(f\))(\(x_i\)) but IMP\textsubscript{PARAM,\(H\)}(\(U_i\))(\(f\))(\(x_i\)) \(\neq\) IMP\textsubscript{PROC,\(H\)}(\(U_i\))(\(f\))(\(U_d\)), implementation for \(g\) has to be explicitly selected as IMP\textsubscript{CALL,\(H\)}(\(U_i\))(\(f\))(\(g\)) = IMP\textsubscript{PARAM,\(H\)}(\(U_i\))(\(f\))(\(x_i\)). It seems logical, in this case, to select IMP\textsubscript{PARAM,\(H\)}(\(U_i\))(\(f\))(\(x_i\)) by default, and so the rule for procedure calls is modified:

\[
\forall U_i: U_l \in \text{ImplModules}_{H}:
\forall f: f \in \text{NewProcsImpl}_{H}(U_i):
\forall x: x \in \text{ProcCalls}_{H}(U_i)(f) \land x \notin \text{RestrictedCallsOnProc}_{H}(U_i)(f) \land
\quad (\forall p, q: p, q \in \text{CallParams}_{H}(U_i)(f)(x):
\quad \quad \quad \quad \text{IMP}\textsubscript{VAL,\(H\)}(U_i)(f)(p) = \text{IMP}\textsubscript{VAL,\(H\)}(U_i)(f)(q))
\quad \quad \quad \quad \text{IMP}\textsubscript{CALL,\(H\)}(U_i)(f)(x) = \text{IMP}\textsubscript{OBJ,\(H\)}(U_i)(f)(p)
\quad \land
\quad \forall x: x \in \text{ProcCalls}_{H}(U_i)(f) \land x \notin \text{RestrictedCallsOnProc}_{H}(U_i)(f) \land
\quad \quad (\exists p, q: p, q \in \text{CallParams}_{H}(U_i)(f)(x):
\quad \quad \quad \quad \text{IMP}\textsubscript{VAL,\(H\)}(U_i)(f)(p) \neq \text{IMP}\textsubscript{VAL,\(H\)}(U_i)(f)(q))
\quad \quad \quad \quad \text{IMP}\textsubscript{CALL,\(H\)}(U_i)(f)(x) =
\quad \quad \quad \quad \text{IMP}_{\text{PROC,\(H\)}(U_i)(f)(\text{DefProcToDef}_{\text{H}}(\text{ProcCallToProc}_{\text{H}}(x)))}
\quad \land
\quad \forall x: x \in \text{ProcCalls}_{H}(U_i)(f) \land x \notin \text{RestrictedCallsOnProc}_{H}(U_i)(f):
\quad \quad \text{IMP}\textsubscript{CALL,\(H\)}(U_i)(f)(x) =
\quad \quad \quad \text{RestrictionsOnCall}_{H}(U_i)(f)(x) \land
\quad \quad \quad \text{IMP}_{\text{PROC,\(H\)}(U_i)(f)(\text{DefProcToDef}_{\text{H}}(\text{ProcCallToProc}_{\text{H}}(x)(x)))}
\]

where CallParams\textsubscript{\(H\)} results in the set of variables, parameters or function calls that act as real parameters in a procedure call and IMP\textsubscript{VAL,\(H\)} stands as a shorthand for: "IMP\textsubscript{VAR,\(H\)}, IMP\textsubscript{PARAM,\(H\)} or IMP\textsubscript{CALL,\(H\)} depending on the kind of value involved".

### 5.3.4 Algorithms for Implementation Checking Revisited

The modifications required on the implementation checking algorithm are twofold. On the one hand, procedures SingleModuleChecking and SubstituteTypesAndFunctions must be extended to deal with the new constructors, namely IMPLEMENTED LIKE in the first one and IF-THEN-ELSE-END and IMPL in the second one; such modifications are very simple. On the other hand, as far as efficiency properties and restrictions for an entity may refer to others, implementation checking must be done inside selection rules in an ordered way; the hierarchy of modules is traversed top-down so that inherited modules at every level for every module are
maintained. Then, for every module, levels keep being considered in descending order as required for the rules, viewing type representation and procedure levels equal and also for object and procedure call ones, because requirements for a type may depend on the implementation selected for a procedure and vice versa, and so for objects and procedure calls. In both pairs of levels, a directed graph is built to record dependencies between their entities in such a way that IMPLEMENTED LIKE and IMPL may always be properly evaluated by means of a topological sort traversal on this graph.
6 Translation from Modula-NF to Modula-2

In this section we address the problem of obtaining an executable program from a Modula-NF one. Basically, there are two options: to build a compiler or an interpreter directly for Modula-NF, or else to build a translator from Modula-NF to Modula-2 to make use of its compiler. At this first stage of the project, we have chosen the second option, although the existence of a Modula-NF interpreter could be useful for programs under development.

As a previous remark, we should note that translation will be limited to those implementation modules that are really used in any point of the program; for instance, if there were four implementations for the ADT table but just two of them were used, the other ones would not be translated. However, Modula-2 files generation is a bit more difficult than this: note that translation of an implementation module $M$ may require more than a single Modula-2 resulting file, if $M$ is used by two different objects or procedure calls which require different implementations for a component of $M$; for instance, this would be the case for tables implemented with lists if different list implementations were required at different places. So, given a hierarchy $H$ of modules, the maximum number of Modula-2 files resulting from all possible combinations of implementations is defined as $\text{NM}(H) = \text{NM}(\text{TopModule}(H))$, where:

$$\text{NM}(U_d) = \Sigma_{U_i \in \text{DefToImpl}_{H}(U_d)} (\Pi_{V_d \in \text{ImportedModules}_{H}(U_i)} \text{NM}(V_d))$$

for every definition module $U_d$.

So, in the framework of real programs, the maximum number of Modula-2 resulting files may be very high. Then, the existence of all of these files may not be advisable, and this is the reason why we have chosen a selective file generation.

The translation consists of two steps. First, a directed graph is created with the import-export relationships of .DEFth files inside the hierarchy of modules: nodes are .DEFth files and an edge exists for every .DEFth file using another one. Nodes in the graph also include a list with all .MODth files implementing the corresponding definition (we recall that this information appears explicitly in all Modula-NF implementation modules); moreover, edges are labelled with a name list of those implementations that are really used in the corresponding definitions; so, it will be possible to find out which Modula-2 files must be effectively generated. Also, a parse tree is built for all modules; in the case of implementations, it is decorated with IMP$_{x,H}$ values for every program entity $x$, following the algorithms and rules described in the previous sections. This step succeeds if every variable, parameter and procedure call ends with a single implementation.

The second step generates Modula-2 files from parse trees and the import-export graph. The main idea is that a type must be created for all different implementations of a .DEFth file involved in the program, as explained above; this type is defined with the two usual Modula-2 files, .DEF and .MOD. More precisely, given a .MODth file with name $\text{Imp.MODth}$ implementing a .DEFth file with name $\text{Adt.DEFth}$, two files $\text{Adt.DEF}$ and $\text{Adt.MOD}$, being $n$ a number randomly generated in $[1, \text{NM(Adt)}]$, are generated with the following rules:

- All design information appearing in the Modula-NF files disappear in the Modula-2 ones.
• Every type, procedure and constant symbol $x$ introduced in `Adt.DEFh` is renamed by $x_n$ in the corresponding Modula-2 files.

• Imported modules `ImpAdt_1`, ..., `ImpAdt_k` are imported with names `ImpAdt_{1n_1}`, ..., `ImpAdt_{kn_k}`, being `ImpAdt_{jn_i}` the name corresponding to the chosen implementation $n_i$ for `ImpAdt_j` at module level inside `Imp.MODth`.

• In addition to the procedures explicitly declared in `Adt.DEFh`, the procedures:

  \[
  \text{PROCEDURE } x_n\text{Abs ( } v: x_n ); \text{ TERM;}
  \]

  \[
  \text{PROCEDURE } x_n\text{Aval ( } t: \text{ TERM) : } x_n;
  \]

appear in `Adt.DEF`, for all exported types $x$ in `Adt.DEFh`, provided that the abstraction function for $x$ has been written in `Adt.DEFh`. The first procedure is the abstraction function for the implementation, while the second one is the corresponding concretion function, that is the execution of a term of type $x$ yielding a data structure; this procedure is nothing but a preorder traversal of the term and thus it is automatically generated by the translator. The implementation of $x_n\text{Abs}$ is the translation of the corresponding abstraction function converting direct calls to type functions to build the resulting terms, into parameters of a `MakeTerm` procedure.

• Type representation translation consists of substituting type names in Modula-NF by the new type names in Modula-2. This means that every type $x$ introduced in `Adt.DEFh` is renamed by $x_n$, and that every imported type $y$ used in the representation is renamed by $y_m$, being $m$ the number randomly generated for the chosen implementation for $y$ at type representation level and type component representation levels.

• Procedure code translation (excluding the abstraction function, whose treatment has been already explained) consists of copying the code with name renaming as in type representation translation and taking care of multiple implementations interaction as explained in 5.2.\textsuperscript{10} Objects type and procedure calls must be renamed to the appropriate new name depending on its implementation.

• The modules `TERM.DEF` and `TERM.MOD` defining the type `TERM` together with its operations (`MakeTerm` and other similar ones) are implicitly linked in the application.

We remark that the key point in this translation rules is that Modula-NF symbol (modules, types, procedures and procedure calls) should be correctly renamed depending on its implementation; this renaming is possible from the information stored at the parse tree and from a mapping from implementations to numbers in $[1, NM(M)]$, for every definition module $M$. Also note that private symbols need not be renamed.

In the next figure the Modula-2 files corresponding to implementations for graph presented at 2.4 using list implemented by pointers. To improve the readability of the example, Modula-2 names are made more mnemonic by using names ("ByAdjacencyMatrix", "ByAdjacencyLists",

---

\textsuperscript{10} Be aware that, if the abstraction function is needed somewhere and it has not been provided by the corresponding MODth file, the linker will complain.
"ByPointers") instead of numbers. Note the translation of abstraction functions (coded in 5.1) to Modula-2 files using predefined procedures on terms.

```
DEFINITION MODULE GraphByAdjacencyLists;
   FROM ListCardIntByPointers IMPORT listByPointers;
   EXPORT QUALIFIED graphByAdjacencyLists, GraphCreateByAdjacencyLists,
       GraphAddByAdjacencyLists, GraphDeleteByAdjacencyLists,
       GraphLabelByAdjacencyLists, GraphSuccByAdjacencyLists;

   CONST nByAdjacencyLists = 100;
   TYPE graphByAdjacencyLists;
   PROCEDURE GraphCreateByAdjacencyLists (VAR g: graphByAdjacencyLists);
   PROCEDURE GraphAddByAdjacencyLists
       (VAR g: graphByAdjacencyLists; u, v: CARDINAL; et: INTEGER);
   PROCEDURE GraphDeleteByAdjacencyLists
       (VAR g: graphByAdjacencyLists; u, v: CARDINAL);
   PROCEDURE GraphLabelByAdjacencyLists
       (g: graphByAdjacencyLists; u, v: CARDINAL): INTEGER;
   PROCEDURE GraphSuccByAdjacencyLists
       (VAR g: graphByAdjacencyLists; u: CARDINAL; VAR i: listByPointers);
   PROCEDURE graphByAdjacencyListAbs (g: graphByAdjacencyLists): TERM;
   PROCEDURE graphByAdjacencyListAval (t: TERM): graphByAdjacencyLists;
END GraphByAdjacencyLists.
```

```
DEFINITION MODULE GraphByAdjacencyMatrix;
   FROM ListCardIntByPointers IMPORT listByPointers;
   EXPORT QUALIFIED graphByAdjacencyMatrix, GraphCreateByAdjacencyMatrix,
       GraphAddByAdjacencyMatrix, GraphDeleteByAdjacencyMatrix,
       GraphLabelByAdjacencyMatrix, GraphSuccByAdjacencyMatrix;

   CONST nByAdjacencyMatrix = 100;
   TYPE graphByAdjacencyMatrix;
   PROCEDURE GraphCreateByAdjacencyMatrix (VAR g: graphByAdjacencyMatrix);
   PROCEDURE GraphAddByAdjacencyMatrix
       (VAR g: graphByAdjacencyMatrix; u, v: CARDINAL; et: INTEGER);
   PROCEDURE GraphDeleteByAdjacencyMatrix
       (VAR g: graphByAdjacencyMatrix; u, v: CARDINAL);
   PROCEDURE GraphLabelByAdjacencyMatrix
       (g: graphByAdjacencyMatrix; u, v: CARDINAL): INTEGER;
   PROCEDURE GraphSuccByAdjacencyMatrix
       (VAR g: graphByAdjacencyMatrix; u: CARDINAL; VAR i: listByPointers);
   PROCEDURE graphByAdjacencyMatrixAbs (g: graphByAdjacencyMatrix): TERM;
   PROCEDURE graphByAdjacencyMatrixAval (t: TERM): graphByAdjacencyMatrix;
END GraphByAdjacencyMatrix.
```
IMPLEMENTATION MODULE GraphByAdjacencyMatrix;
FROM ListCreateByPointers IMPORT listByPointers ...;
TYPE graphByAdjacencyMatrix = POINTER TO privgraph;
privgraph = ARRAY [1..nByAdjacencyMatrix, 1..nByAdjacencyMatrix] OF INTEGER;

... PROCEDURE GraphSuccByAdjacencyMatrix
  (VAR g: graphByAdjacencyMatrix; u: CARDINAL; VAR l: listByPointers);
VAR v: cardintByPointers;
BEGIN
  ListCreateByPointers(l);
  FOR i := 1 TO nByAdjacencyMatrix DO
    IF g^*[u, i] <> -99999 THEN FormByPointers(v, i, g^*[u, i]); ListAddByPointers(l, v) END
  END;
END;

PROCEDURE graphByAdjacencyMatrixAbs (g: graphByAdjacencyMatrix): TERM;
VAR t, t1, t2, t3: TERM; u, v: CARDINAL;
BEGIN
  t := CreateTermId("GraphCreateByAdjacencyMatrix");
  FOR u := 1 TO nByAdjacencyMatrix DO
    FOR v := 1 TO nByAdjacencyMatrix DO
      IF g^[u, v] <> -99999 THEN
        t1 := CreateTermCard(u); t2 := CreateTermCard(v); t3 := CreateTermInt(g^[u, v]);
        t := MakeTermS("GraphAddByAdjacencyMatrix", t, t1, t2, t3)
      END
    END
  END
  RETURN t;
END;

PROCEDURE graphByAdjacencyMatrixAval (t: TERM): graphByAdjacencyMatrix;
VAR g: graphByAdjacencyMatrix; t1, t2, t3: TERM; u, v: CARDINAL; x: INTEGER;
BEGIN
  IF LabelId(t) = "GraphCreateByAdjacencyMatrix" THEN GraphCreateByAdjacencyMatrix(g);
  ELSIF LabelId(t) = "GraphAddByAdjacencyMatrix" THEN
    UnmakeTermS(t, t1, t2, t3);
    g := graphByAdjacencyMatrixAval(t);
    u := LabelCard(t1); v := LabelCard(t2); x := LabelInt(t3);
    GraphAddByAdjacencyMatrix(g, u, v, x);
  END;
  RETURN g;
END;
END GraphByAdjacencyMatrix.

Implementation with adjacency lists is very close to the one above and it is not presented here.
7 Conclusions and Future Work

A classical imperative programming language (Modula-2) has been extended with some new constructs related to efficiency issues measured with the $O$ notation (yielding to Modula-NF); these constructs make explicit both the efficiency characteristics and the efficiency requirements of ADT implementations making then feasible the automation of the design process. This automation is carried out by means of a set of rules for selecting implementations which relies on an algorithm able to check implementations against efficiency requirements. If the automation succeeds (that is, a single implementation is associated to each object and procedure call in a program), the Modula-NF program may be translated into Modula-2 files.

Our programming language provides many interesting features. First, designers just establish design requirements and design properties of software; implementations are automatically selected. Second, software is robust with respect to changes on requirements and construction of new implementations, requiring just re-running the implementation selection algorithm and the Modula-2 translator to actualise software. Third, the same ADT may be implemented in more than one way in the same program if its efficiency requirements require it; this is true even if two objects of the same type interact provided that an abstraction function has been defined for implementations. Last, design information in a constituent part of programs, improving thus their comprehension and making easier the communication between designers, implementers and users of modules. On the other hand, there seems to be no drawbacks in our approach, since the usual Modula-2 programming style may be also done with Modula-NF.

As far as we know, there exists no proposal for a programming language with the design constructs presented in this report. Also, as far as we know, the full compatibility of objects of the same type but different implementations, as we get in our proposal, is not achieved by any imperative programming language, in despite of being an attractive characteristic, both for efficiency constraints and for easy integration of pieces of software. About the first issue, some projects do exist aimed to program transformation field [AiA93, SY94]; in both cases, the treatment of efficiency is not as powerful as ours, perhaps because it is not the central point of their work. About the second point, a project do exist for maintaining software reusable components with distinguishable different implementations [WHS89], but it is centred on the algebraic specification field (specifications implementing other specifications) and most of the problems presented here do not arise in their work. On the other hand, inheritance from O-O. partially solves the multiple implementations problem: for an ADT $T$, a class for its definition and a class for every different implementation may be created, and a inheritance relationship from implementation classes to specification ones may be added; then, many objects of the same type but different implementations may exist in the same program; however, as far as abstraction functions do not exist, they may not interact unless some specific methods are explicitly written with different implementations (i.e., classes) for parameters.

Future work has to do mainly with four issues. First, our proposal has to be extended to more powerful languages with genericity and inheritance; this will make implementation selection more difficult because there will exist more relationships between ADTs. Second, a whole process model should have to be defined centred on Modula-NF (or whatever programming language is finally chosen); to do this, also a language for writing specifications could be incorporated to the language, as done in [Fra92]. Third, efficiency of code could be
automatically computed, following the classical rules to analyse program efficiency and being aware that automatic computation may yield worst results that a careful ad hoc analysis. Last, other non-functional criteria may be added: non-asymptotical measurement of space, accuracy in numerical algorithms, etc., and also less conventional issues, as confidence on software correctness, probability of poor behaviour in type representation (hashing, not-balanced binary search trees, etc.), and so on; even more, a meta-language for defining new criteria could be defined.

References


LSI-95-1-R “Octree simplification of polyhedral solids”, Dolors Ayala and Pere Brunet.


LSI-95-3-R “The complexity of searching implicit graphs”, José L. Balcázar.


LSI-95-6-R “Triangles, ruler, and compass”, R. Juan-Arinyo.

LSI-95-7-R “The modifiability factor in the LESD project: definition and practical results”, Nuria Castell and Olga Slavkova.


LSI-95-9-R “A case study on prototyping with specifications and multiple implementations”, Xavier Franch.


LSI-95-12-R “Computer (PC) assisted drawing of diagrams for forecasting soaring weather”, Lluís Pérez Vidal.

LSI-95-13-R “Animats adaptation to complex environments as learning guided by evolution”, Mario Martin, Márkus Garcia, and Ulises Cortés.


LSI-95-16-R “MDCO to B-Rep conversion algorithm”, Dolors Ayala, Carlos Andújar, and Pere Brunet.
LSI–95–17–R. "Augmented regular expressions: A formalism to describe, recognize, and learn a class of context-sensitive languages", René Alquézar and Alberto Sanfeliu.


LSI–95–26–R. "External schemas in object oriented databases" (written in Spanish), José Samos.


LSI-95-40-R. "Parallel approximation of MAX SNP and MAX NP problems", Maria Serna and Fatos Xhafa.

LSI-95-41-R. "Automatically extracting translation links using a wide coverage semantic taxonomy", German Rigau, Horacio Rodriguez, and Jordi Turmo.

LSI-95-42-R. "Belief increasing in SKL model frames", Matías Alvarado and Gustavo Núñez.

LSI-95-43-R. "An incremental algorithm to obtain a Constrained Delaunay Triangulation", Marc Vigo Anglada.


Hardcopies of reports can be ordered from:

Nuria Sánchez
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Pau Gargallo, 5
08028 Barcelona, Spain
secrelsi@lsi.upc.es

See also the Department WWW pages, http://www-lsi.upc.es/www/