A Case Study on Prototyping with Specifications and Multiple Implementations

Xavier Franch

Report LSI-95-9-R
A case study on prototyping with specifications and multiple implementations

Xavier Franch
franch@lsi.upc.es
Dept. Llenguatges i Sistemes Informàtics (LSI)
Technical University of Catalunya (UPC)
Pau Gargallo 5, 08028 Barcelona, Catalunya (Spain)
FAX: 34-3-4017014. Phone: 34-3-4017006

Abstract

This paper presents through an example a software process model addressed to support the existence of multiple type implementations when programming with abstract data types. We use a multiparadigm programming notation that combines an equational style for specifying types and an imperative one for implementing them. Implementations may be selected by means of ad hoc language constructs in the appropriate contexts; another language construct, the abstraction function, allows implicit switching between implementations during execution. Also, the abstraction function allows prototyping of incomplete programs provided that the specification for non-implemented types is operational. As a result, the resulting software process model permits the user to reach the final program as a sequence of intermediate executable prototypes.

1 Introduction

Development of programs in a modular, ADT oriented framework satisfies that algorithms depend only on the specification of the types they use. To obtain a final program, a single implementation is chosen for every type and so their objects (variables, constants, parameters and function results) are bound just to that implementation. This is the case when writing programs in Modula-2, Ada, Pascal, C and other Algol-like languages, where every type should present a single module to define its operations and a different module for each implementation.

A potential drawback of this approach is that implementation independence is not fully achieved because of the need to bind a type to a single implementation. For instance, given two objects $x$ and $y$ of the same type $t$ but designed to have different implementations, $\text{Imp}_x$ and $\text{Imp}_y$, it is necessary to introduce two different types $t_x$ and $t_y$ with the same operations as $t$ but linked to $\text{Imp}_x$ and $\text{Imp}_y$, respectively; this gives rise to unnecessary complexity of software construction, maintenance, reusability and comprehension.

Different implementations do appear in many contexts: efficiency of programs, joining of already existing parts, reusability of already implemented components, etc.; so, a framework to support multiple implementations of types in a program is missed, as well as the necessary mechanisms to handle the interaction of objects of the same types but different implementations.

In this paper, a software process model is proposed to build programs with previous
specification of abstract data types followed by their implementation. A program is obtained by applying a sequence of refinement steps, being able to prototype the product at every stage of development. It is proposed to specify into the initial semantics framework [EM85]. To deal with multiple implementations, we enrich the programming language with an explicit binding between types and objects and their implementations, which may be set at three different levels. To support interaction between objects implemented in different ways, an abstraction function must be provided for every type, being then possible to switch implicitly from an implementation to another when needed. As an extension, prototyping of incomplete programs is possible if we consider algebraic specification of types as naïve implementations which can be executed with a term-rewriting system.

Throughout the paper, we use an ad hoc notation for both specifying and implementing types. Specifications are written with positive conditional equations, while implementations are imperative. Specifications and implementations are encapsulated in specification modules and implementation modules, respectively. The language constructions are the usual ones in both paradigms (use of modules, renaming of symbols, instances of generic modules, etc.), resembling Clear, OBJ and by the like in the equational side, and Ada in the imperative one; we have preferred not to use any particular languages to provide a more general framework.

2 Why Multiple Implementations?

Let's suppose we represent a geographical distribution (a computer network, a railway service, etc.) with a labelled, undirected graph, being the label the cost of the connection (time, money, etc.); let's also suppose there are n nodes in the graph (with n being probably high) and about kn edges, being k a small integer. The graph is managed in the following way: first, the about n edges are inserted on the empty graph with their cost; then, an all-pair shortest path algorithm must be executed to optimise graph traversals; and last, a lot of individual accesses (about n^2) to obtain edge labels are expected to occur (as well as the necessary queries on shortest paths, which are not relevant anymore).

If we denote by ins, apsp and obt the three steps mentioned above, their worst case execution time measured with the Θ notation is summarised in Table 1. It is clear that the result depends on the graph representation: adjacency matrix or adjacency lists. We use a repeated Dijkstra algorithm with a heap in the adjacency lists representation for the all-pairs shortest path problem, and the Floyd algorithm with the adjacency matrix representation, and we suppose that insertion of edges does not test if they already exist.

<table>
<thead>
<tr>
<th></th>
<th>T_{ins}</th>
<th>T_{apsp}</th>
<th>T_{obt}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix</td>
<td>Θ(n^2)</td>
<td>Θ(n^3)</td>
<td>Θ(n^2)</td>
</tr>
<tr>
<td>Lists</td>
<td>Θ(n^2)</td>
<td>Θ(n^2logn)</td>
<td>Θ(n^3)</td>
</tr>
</tbody>
</table>

Table 1.

We can observe from the table that, no matter which the implementation for the graph is, the worst execution time for the algorithm reaches Θ(n^3). However, we can also note that this order of magnitude may be lessened to Θ(n^2logn) if we use the appropriate representation in every step: matrix for insertion and queries, and lists for all-pairs shortest path calculation. The questions left are: how can we switch between both representations? and does this switch really pay?

The answer to the first question is given in section 3; intuitively, some rules are needed to transform the matrix resulting from graph creation to an abstract graph value and some other
ones to represent this abstract value into an adjacency lists representation\(^1\). Concerning efficiency, and given the assumptions made on the graph, the cost of converting a matrix into an abstract value should clearly be \(\Theta(n^2)\), that is the cost of traversing the whole data structure, while the cost of building the \(n\) lists with about \(n\) edges should be \(\Theta(n)\); then, none of both transformation steps goes beyond \(\Theta(n^2 \log n)\), which implies that switching from matrix to lists does not affect the asymptotic execution time of the algorithm.

### 3. Dealing with multiple implementations

In this section we give a brief outline about how can we deal with multiple implementations: we define some language constructs, we introduce two functions for determining the implementation of every object in the program and we finally study when it is necessary to switch from an implementation to another. For more details, see [Fra94].

Let \(V\) be an implementation module. There are three levels to select a type implementation inside \(V\), using a "implemented with" language construct in the appropriate contexts:

- **Module level:** use and instantiation of modules. Every used or instantiated module \(U\) may be selected with a given implementation \(U_{imp}\), making all the types exported by \(U\) to be implemented with \(U_{imp}\) inside \(V\).

- **Type level:** implementation of already existing types. Every existing type \(t\) may be individually selected with a given implementation \(T_{imp}\) inside \(V\), regardless of any existing binding done at the module level.

- **Object level:** declaration of objects. Every object \(x\) of type \(t\) may be declared with a given implementation \(T_{imp}\), regardless of any existing binding done at previous levels.

We introduce a pair of functions to bind types and objects to their corresponding implementation modules at the syntactical level. Let \(U_{imp}\) be an implementation module, let \(\text{types}_{U_{imp}}\) and \(\text{obj}_{U_{imp}}\) be the domain of those types and objects declared inside \(U_{imp}\), respectively, and let \(\text{IMP}_H\) be the set of implementation modules in the hierarchy \(H\). We define:

\[
T_{U_{imp}} : \text{types}_{U_{imp}} \rightarrow \text{IMP}_H,
\]

\[
I_{U_{imp}} : \text{obj}_{U_{imp}} \rightarrow \text{IMP}_H,
\]

These functions bind types and objects to their respective implementations inside \(U_{imp}\). \(I_{U_{imp}}\) fixes the implementation of every object in \(H\).

The rules to determine the value of both functions from the language constructs given above are defined in [Fra94]. As a result, different objects of the same type may in fact be implemented in different ways.

Once \(I_{U_{imp}}\) is fixed, there are four situations that require to switch from an implementation to another during execution of programs, provided that the involved objects have different implementations.

- **T1.** Assignment rule. The assignment \(x := y\) requires to transform \(y\) from its implementation to \(x\)'s one.

- **T2.** Parameter rule. The call \(f(...y...\)) with \(y\) as the actual parameter bound to the formal one \(x\) requires to transform \(y\) from its implementation to \(x\)'s one if the parameter is "in" or "in/out" and, once \(f\) is completed, to transform \(x\) from its implementation back to \(y\)'s one if the parameter is "out" or "in/out".

- **T3.** Function result rule. Calling a function \(f\) in any given context requires to transform from the function result implementation to the required one.

- **T4.** Compatibility rule. The call \(f(...x...y...\)) being \(x\) and \(y\) of the same type \(t\) and being

\(^1\) Other options, as overloading and explicit conversion procedure are discarded; see [Fra94].
A function defined in the same module as \( t \), requires to transform \( y \) or \( x \) from one's implementation to the other's one, or even both of them to a third one, depending on the relationships between the implementation for \( x \) and \( y \) and the one chosen for \( f \).

To carry out implementation switching, we use a pair of functions, the abstraction function and the representation function [Hoa72]. The first one maps an object to an equivalent term at the specification level, while the second one performs the inverse mapping. So, switching of implementation consists of applying the abstraction function on the source implementation and then the representation function on the destination one. Note that the representation function is just the execution of the operations contained in the term into the chosen implementation, while the abstraction function must be explicitly provided for every implementation of a type, using any of both notations of the multiparadigm language. An example is given in section 5.

Specifications may be fit into this scheme if we consider them as naïve implementations (provided that they are operational), executable with the help of a term-rewriting system; the switching between the term-rewriting system and the interpreter is a particular case of implementation switching, considering the quotient-term algebra [EM85] as the implementation attached to any non-implemented specification. The details of the execution procedure are given in [BBF88] and [FB93]; the general ideas appear in section 5.

4. The Software Process Assistant

We describe in this section a software process assistant based on the idea of prototyping with incomplete programs. "Incomplete programs" stands for programs which may contain non-implemented types, and/or objects with no associated implementation. Execution of incomplete programs is interesting in the context of program development through prototyping because the choice of implementation for types and objects may be delayed until the entire behaviour of the module hierarchy has been proved correct. This section proposes to perform this "proof" through testing, by means of the construction of a sequence of executable prototypes.

Let \( H \) be a hierarchy of modules and let \( lib \) be a library of software components. The main capabilities offered by the assistant are:

1. Define the signature of a module: \texttt{define(H, spec_mod)}. During this definition, more concrete orders may be used: \texttt{create(name)} to create the empty module, \texttt{add_type(spec_mod, name)} and \texttt{add_op(spec_mod, op)}; also, reuse from library can be made as explained at point 4.

2. Specify a module: \texttt{specify(H, spec_mod)}. The module can be already defined or not. In order to obtain the equations, the assistant guides the user into the initial semantics framework, asking for:

   2.i Which are the constructors of every type, that is, the minimal set of operations to build any value: \texttt{constructors(spec_mod, set_of_ops)}. Constructors may be introduced as private.

   2.ii Which are the relationships among those constructors: \texttt{congruence(spec_mod, set_of_eqns)}.

   2.iii For each other operation, which are the equations defining its behaviour: \texttt{specify_op(spec_mod, set_of_eqns)}.

In the last step, a default procedure may be applied under request to decompose the parameters of the operation being specified using the constructors of their type. Prototyping may be done at any point during specification; also, specification may be left (temporarily or permanently) incomplete, which may affect prototyping.

3. Implement a defined or specified module: \texttt{implement(H, name_spec, impl_mod)}. In order to allow prototyping of the implementation to occur as soon as possible, the assistant asks for:

   3.i The representation of the type plus its abstraction function plus the implementation
of the constructor operations: \texttt{represent(impl\_mod, name\_type, repr)},
\texttt{abstr(impl\_mod, name\_type, set\_of\_eqns or code)} for the abstraction
function (which may be specified or implemented, see next section) and
\texttt{impl\_op(impl\_mod, name\_op, code)} for each constructor.

3.ii For each other operation, its implementation: \texttt{impl\_op(impl\_mod, name\_op, code)}.

So, prototyping may take place after 3.i and also after every operation implemented in
3.ii. Implementation may be left temporarily incomplete, which may affect prototyping.

4.- Reuse a module from the library: \texttt{use(mod, lib, name\_spec, ren)} and
\texttt{instantiate(mod, lib, name\_spec, assoc, ren)}. That is, the module is used or
instantiated inside a module \texttt{mod} that is being defined, specified or implemented; \texttt{ren}
stands for the symbol renamings made and \texttt{assoc}s stands for the binding between formal
and actual parameters in the instantiation (the fitting morphism in the initial semantics
approach). In case of using or instantiating a module with a given implementation, its
name is added as the last parameter.

5.- Choose an implementation for an module, a type or an object: \texttt{choose\_mod(mod,
name\_spec, name\_impl)} for modules, \texttt{choose\_type(mod, name\_type,
name\_impl)} for types and \texttt{choose\_const(mod, name\_const, name\_impl)},
\texttt{choose\_funct\_result(mod, name\_funct, name\_impl)} and \texttt{choose\_var(mod,
name\_funct, name\_var, name\_impl)} for the appropriate kind of object (the last one
includes parameters); for modules, we may also use the orders in point 4. All the
bindings are made inside the implementation module \texttt{mod}.

6.- Store a module into the library: \texttt{store(H, lib, module)}.

7.- Prototype a function: \texttt{prototype(H, name\_module, name\_function)}. The
assistant asks for the input parameter values and then the required function is executed,
calling either a conventional interpreter if the function is implemented or the term-
rewriting system if not. Input parameter values may be persistent objects as well as input
/ output devices, which are treated as abstract data types. Prototyping is the crucial
functionality of the assistant, and it can be invoked at any moment when specifying or
implementing a module.

Figure 1 shows a diagram of these orders. In the current implementation, a window system
implements this diagram.
5 A case study

We present the development of a program using the assistant introduced in the last section. The goal is to implement a software component to perform the work described in section 2. To illustrate the process, we suppose that graphs do not exist in any available library.

Step 1. Specification of graphs

First of all, we must choose the model and operations of a directed graph. To simplify the problem, we suppose that the set of nodes of the graph is fixed; so, we have an operation empty to create a directed graph with all the nodes and no edges and operations to add a new edge and to obtain the label of an edge. Moreover, we need an operation succ to obtain a list with the successors of a node together with their label.

We suppose that two generic modules, PAIR (A, B: ELEM) and LIST (X: ELEM_=_), already exist in the library of reusable components. ELEM_= is a module defining the formal parameters of lists: a sort elem and an equality operator eq, while ELEM defines just the sort elem.

Figure 2 presents the specification module for directed graphs, that creates a new type list_of_node_and_nat by parameter passing to the specification of LIST; this type is used to fix the signature of succ, once the new type is created, the operations may be declared and the axioms stated.
module GRAPH (Y: ELEM_=) defines

    type graph
    instantiates PAIR (A, B: ELEM) where A.elem is Y.elem, B.elem is nat {parameter passing to PAIR}
    renames pair by node_and_nat {renaming of the new type}
    instantiates LIST (X: ELEM_=) where X.elem is node_and_nat, X.eq is Y.eq {parameter passing to LIST}
    renames list by list_of_node_and_nat {renaming of the new type}
    ops empty: → graph {graph with no edges}
    add: graph elem elem nat → graph
    label: graph elem elem → nat
    succ: graph elem → list_of_node_and_nat

    ... errors g: graph; v, w: elem; x, y: nat
    add(add(g, v, w, x), v, w, y) {no repeated edges}
    equations g: graph; u, v, w, a, b: elem; x, y: nat
    add(add(g, v, w, x), a, b, y) = add(add(g, a, b, y), v, w, x) {not ordered edges}

    ... succ(empty, v) = LIST.empty
    succ(add(g, v, w, x), v) = put(succ(g, v), <w, x>)
    [eq(v, u) = false] ⇒ succ(add(g, v, w, x), u) = succ(g, u)

end module

Figure 2.

So, the prototype APL1 obtained after this first step is defined by the process expression:

APL1 = specify(Ø, ...instantiate(instantiate(...(create(GRAPH), ...),
          lib, PAIR, ..., pair → node_and_nat),
          lib, LIST, ..., list → list_of_node_and_nat),

...).

As soon as the specification for graphs is complete, we could prototype to test its behaviour. It may be done by rewriting terms and properties and checking the results. For instance, we could rewrite succ(add(add(add(empty, v1, w1, 1), v1, w2, 5), v2, w2, 1)) to see what the result is, or we could check whether succ(add(add(add(empty, v1, w1, 1), v1, w2, 3), v2) = empty. This kind of prototyping is useful not only to correct the specification but also to acquire more knowledge about problem's domain.

Step 2. Definition of all-pair shortest path operation

Once graphs are assumed to be correct, we specify the all-pair shortest path operation (see Figure 3); it returns a map from pair of nodes to labels (to simplify the problem, paths are not stored), being MAP a module in the library. We decide not to give a complete, operational meaning to the function since the result would in fact look like an implementation; this is a classical situation in the initial semantics framework and it is the reason why our method does not force to provide a complete specification to data types. So, we only formulate two properties the operation must fulfill: every node has a zero path to itself and the shortest path between two nodes is a symmetric relation; later, we can prototype implementations for graphs to verify that they satisfy these properties.

The prototype APL2 has been obtained from an assistant expression close to the one before.

---

2 Currently, the assistant is being enriched to help the user to perform proof obligations.

3 Even more, we may decide to give no equations at all, in which case a specification universe consists only of the type signature (always required for syntactical checks).
module APSP (X: ELEM_ =) defines
uses GRAPH (ELEM_ =)
instantiates PAIR (A, B: ELEM) where
A.elem is X.elem, B.elem is X.elem
renames pair by 2nodes, first by v1, second by v2
instantiates MAP (A: ELEM_ =, B: ELEM) where
A.elem is 2nodes, B.elem is nat
A.eq(p1, p2) is X.eq(v1(p1), v1(p2)) ∧ X.eq(v2(p1), v2(p2))
renames map by map_2nodes_to_nat
ops
apsp: graph → map_2nodes_to_nat
eqns g: graph; v, w: X.elem
value(apsp(g), v, v) = 0
value(apsp(g), v, w) = value(apsp(g), w, v)
end module

Figure 3.

Step 3. Floyd implementation of all-pairs shortest path

Note that the non-operational specification for apsp prevents the environment to prototype it; so, we immediately implement the function using first the Floyd algorithm for simplicity.

module APSP_FLOYD (X: ELEM_ =) implements APSP (X: ELEM_ =)
function apsp (g: graph) returns map_2nodes_to_nat is
var f: map_2nodes_to_nat; v, w, u: X.elem end var
for all v, w in X.elem do f := MAP.add(f, v, w, GRAPH.label(g, v, w)) end for all
for all v, w, u in X.elem do
if MAP.value(f, v, u) + MAP.value(f, u, w) < MAP.value(f, v, w) then
f := MAP.value(f, v, u) + MAP.value(f, u, w)
end if
end for all
returns f
end module

Figure 4.

The prototype is given by the expression:

APL3 = implement(APL2, APSP, impl_op(create(APSP_FLOYD), apsp, "code for apsp"))

Next, the implementation could be tested with respect to its specification by building some graphs using the operations in GRAPH and executing apsp on them. Although this execution is done by the interpreter (we prototype the implementation of apsp) the term-rewriting system is also involved because of the call to the non-implemented operation label. For instance, if we prototype apsp on the graph add(add(empty, v, w, 3), u, v, 4), the evaluation of label(g, v, w) takes place during the algorithm and then the term-rewriting system is called by the interpreter to execute label(add(add(empty, v, w, 3), u, v, 4), v, w) which results in the value 3 that is returned to the interpreter to continue execution. Note that the interpreter may require later many more term-rewriting steps not only for graphs, but also for maps, because we have not chosen any implementation for them yet4. For more details about this kind of execution see [BBF88].

Step 4. Main module

Now, we focus in the main application.

---

4 However, as far as implementations for MAP do surely exist in the library, the environment may be switched to implicitly assign any of them, until the user fixes the definitive one.
module REAL_APPLICATION defines
ops main: input/output → input/output
end module

Figure 5.

Let's suppose that our main function defines the type of graphs with natural numbers from 1 to 10000 as nodes via parameter passing to APSP. The implementation of this module may use the operations exported by APSP that includes those from GRAPH:

module REAL_APPLICATION_IMPLIM implements REAL_APPLICATION
uses NAT_10000
instantiates APSP (X: ELEM =)
    where X.elem is nat_10000, X.eq is NAT_10000.
renames graph by graph_nats_10000
...
function main (c: input/output) returns input/output is
var g: graph_nats_10000; f: map_2_nodes_to_nat; ...
g := empty
while ... g := add(g, v, w) ... end while {creation of the graph}
f := apsp(g)
... {process g and f }
end module

Figure 6.

Note that nothing has been said about implementations in this context. If we prototype function main the graph may be built and maintained in a term representation ready to be rewritten as in step 3; however, when trying to execute apsp the term-rewriting system fails because its specification is not operational. To solve the problem, we explicitly require APSP to be implemented with APSP_FLOYD by substituting the instantiation clause by:

instantiates APSP (X: ELEM =) implemented with APSP_FLOYD (ELEM =) where
    X.elem is nat_10000, X.eq is NAT_10000.
renames graph by graph_nats_10000

Alternatively, we could temporarily give this information to the execution subsystem, which does not bind subsequent prototype executions to APSP_FLOYD. Even more, the environment may be switched to bind APSP to APSP_FLOYD implicitly, as far as it is the only existing implementation for the specification.

The prototype is given by:

APL4 = implement(define(APL3, add_op(create(REAL_APPLICATION), "main op."))),
impl_op(instantiate(create(REAL_APPLICATION_IMP),
    lib, APSP, "assocs.", "renaming"),
    main, "code for main"))

Step 5. Implementation of graphs

In this state, apsp may in fact be executed, although there is no representation chosen for neither graphs, lists nor maps; variables of these types are maintained as a term and so their operations may be executed by the term-rewriting system. In fact, we could prototype apsp even if no implementation for these types exists. Eventually, however, a particular implementation must be provided to all data in the application to obtain the final program; we focus first on the implementation of graph.

To implement a data type, we begin by choosing a data structure to hold the values of the type. In order to reach the next executable state we must also provide the implementation for the constructors set, {empty, add}, and the abstraction function of the representation. Next we can prototype to test this partial implementation and then we can code the rest of the operations of the type, either all together or not (in either case, prototyping is possible). In our example, we
choose to implement all functions at once.

We decide to build implementations for adjacency lists and for adjacency matrix; so, once both of them are checked they may be placed in the library for future uses. For efficiency purposes, the adjacency list implementation forces lists to be implemented by pointers as we suppose it occurs in \textit{LIST\_BY\_POINTERS}; in the general case the choice of a specific implementation for a type may be delayed, as it is done in the matrix implementation, where we use the specification of lists without regarding what particular implementation is to be used in \textit{succ}.

The prototype is given by:

\begin{verbatim}
APL5 = implement(impl[APl4], impl_op(...(create(MATRIX_GRAPH), ...), GRAPH),
              impl_op(...(abstr(repr(...(instantiate(create(LISTS_GRAPH), lib, LIST,
                                "assocs.", "renaming",
                                LIST\_BY\_POINTERS), ...)...)...
\end{verbatim}

At this point, it is advisable to test if both implementations are correct with respect to the corresponding specification; for instance, we may execute the same terms and properties we used to test the specification. The role of the abstraction function is crucial to find out which value is represented by a particular state of the implementation: we may execute any term \( t \) of type \textit{graph} using its implementation and then abstract the result to check if it equals the result of rewriting \( t \) using the specification of the type. On the other hand, to demonstrate a property \( t_1 = t_2 \) we may proceed in a similar way by executing terms \( t_1 \) and \( t_2 \) and comparing their abstraction.

The abstraction function may be given either by equations or by a program; in any case, auxiliary functions may appear. The first form is more convenient than the second one because the term is built step by step and its manipulation may finish as soon as any stop condition is reached; however, sometimes it is more difficult to express. As an example, Figure 7 shows the abstraction function for directed graphs with adjacency matrix as a program; note the existence of a kind of generic type \textit{term} to store the term under construction:

\begin{verbatim}
abstr (g: graph)
  var v, w: X.elem; t: term end var
  t := empty
  for all v, w in elem do
    if g[v, w] then t := add(t, v, w) end if
  end for all
  returns t
\end{verbatim}

Figure 7.

\textbf{Step 5. Dijkstra implementation of all-pairs shortest path}

At this moment, we could end the application by choosing which implementations are to be used for the types and objects not yet determined. However, before doing this, we decide to build another implementation for \textit{apsp} based on Dijkstra algorithm. The reason is twofold: on the one hand, as we explained in section 2, the resulting program is more efficient; on the other hand, the reusable software components library is enriched for future use.

Figure 8 gives an outline of the module; the process expression is very close to the one for Floyd version. We use a type \textit{pot_map} merging operations of partially-ordered trees and maps, defined in the module \textit{POT\_MAP} that we suppose already exists; to improve efficiency, we require this type to be implemented with a heap. Also, to obtain the most efficient version of the algorithm, graphs are explicitly required to be implemented with adjacency lists; this is a crucial difference with Floyd implementation, where graphs are not bound to any particular representation.
module APSP_Dijkstra (X: ELEM_=) implements APSP (ELEM_=)
  instantiates POT_MAP (Y: ELEM_<=) implemented with HEAP(ELEM_<=) where
  ...
  renames pot_map by remaining_costs
  function apsp (g: graph implemented with LISTS_GRAPH) returns map_2nodes_to_nat is
  ...
end module

Figure 7.

Step 6. Implementing types and objects

Once introduced implementations for graphs with all-pair shortest path algorithm, we fit them into REAL_APPLICATION_IMPL. It is enough with binding APSP_Dijkstra to APSP and implementing graph with MATRIX_GRAPH. Also, implementation for maps is to be chosen.

There are several ways to obtain this result. For instance, with the expression:

APL₆ = modify(APL₅,
  choose_var(
    choose_type(
      choose_mod(obtain(APL₅, REAL_APPLICATION_IMPL),
        APSP, APSP_Dijkstra),
      graph, MATRIX_GRAPH),
    main, g, HASHING))

(the meaning of modify and obtain a module is the intuitive one) which yields to the module of Figure 9.

module REAL_APPLICATION_IMPL implements REAL_APPLICATION
  uses NAT_10000
  instantiates APSP (X: ELEM_=)
    implemented with APSP_Dijkstra (X: ELEM_=)
    where X.elem is nat_10000, X.eq is NAT_10000.=
  renames graph by graph_nats_10000
  type graph implemented with MATRIX_GRAPH end type
  ...
  function main (c: input/output) returns input/output is
    var g: graph_nats_10000
    f: map_2_nodes_to_nat implemented with HASHING; ...
    g := empty
    while ... g := add(g, v, w) ... end while
    f := apsp(g)
  ...
end module

Figure 9.

6 Conclusions

A design methodology to support prototyping has been presented. In this framework, development of applications consists of a sequence of states towards a final implementation; the transition between states is a refinement step, and the whole process is guided by an assistant. Each intermediate state could be fully prototyped before any further refinement. The notion of implementation plays an important role.

The idea of explicitly binding implementations with program objects in the ADT framework is
not new; it is described as a proposal in [Ben87] with the same motivation as the one given here. However, up to know we do not know of any language with the characteristics described in this paper; the closest idea takes place in Pebble [LB88], but the goal being to define a kernel language for formally describing ADT oriented languages.

Other approaches deal with some aspects covered here, but none with all of them. In [WHS89], an extension of ASL is proposed for systematic reuse of specifications. Type implementation is determined with an inductive operator "implement ... by ...". Due to its application field, this work does not address to type implementation conflicts; also, as far as there are no imperative constructs in the language, neither implementation for objects nor mixed execution of programs are studied. In a similar way, [LGZ87] describe a methodology with explicit binding of specifications and implementations, but the last ones are also given by equations and so many problems introduced here do not appear.

On the other hand, many approaches do exist for combining different programming paradigms in the same tool. Most of them extend an existing imperative language (typically Modula-2 or Ada) with logical, Prolog-like features [Bud91, Cra88, Rad90, TC88]. There are also a few of them that combine imperative modules with specified ones, being the Asspegique execution subsystem [CK90] the closest to ours. Asspegique also proposes a prototyping development strategy by gradual Ada-implementation of Pluss-specified modules. However, it does not take profit of the mixed execution scheme to allow different implementation of types as we propose here. Furthermore, another crucial difference is that prototyping requires a previous compilation from Pluss rules to Ada programs, in order to improve execution speed; it is not clear that this improvement does really worth, because compilation time may interfere in the prototyping development process.

Last, comparison with object-orientedness is needed. If A is an implementation for B, we may declare an abstract class CA with most (or all) the operations left virtual, and then another class CB inheriting from CA. With this scenario, polymorphism and dynamic binding allows associating different implementations to the same type; so, programs may be written using the abstract class CA as an ADT yielding to implementation-independent programs and, once a representation is chosen for A objects, they may be converted to it by means of polymorphism; even more, two different objects of class CA may be bound to different heir classes allowing thus multiple implementations to coexist. However, switching of implementations is not supported by inheritance, to avoid a heir exhibiting properties of the other; so, it is necessary to introduce any other scheme (transformation messages or overloading) not as easy to use. As an additional point, most O.-O.-languages do not provide the algebraic specification paradigm (although there are some exceptions), not allowing thus mixed implementation to occur.

References


[Bud91] T.A. Budd. "Blending Imperative and Relational Programming", Software IEEE.


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.

Copies 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