ComProLab: A Component Programming Laboratory

Xavier Franch[1], Pere Botella[1], Xavier Burgués[1], Josep M. Ribó[2]
franch@lsi.upc.es, botella@morfeo.upc.es, diafebus@lsi.upc.es, josepma@eup.udl.es
Dept. Llenguatges i Sistemes Informàtics (LSI)
[1]Universitat Politècnica de Catalunya (UPC), Barcelona (Catalonia, Spain)
[2]Universitat de Lleida (UdL), Lleida (Catalonia, Spain)

Abstract

We present here an approach to component programming which defines languages and tools at both the product and the process levels. At the product level, we allow the use of already existing languages to write functional specifications and implementations of components; also, we provide a notation to state their non-functional specifications, which involve operational attributes as efficiency. Functional specifications can be employed to perform prototyping in a mixed execution framework, which allows the combination of algebraic specifications and imperative code, while non-functional specifications are used to select automatically the best implementation of every component appearing in a software system. At the process level, we have introduced a set of basic program development tasks and we have defined a process language to formulate software process models as particular combinations of these tasks. A process assistant can be used to guide software development following any model defined with this language.

1 Introduction

Component programming is a strategy to develop software systems as the combination of individual software components, many of them coming from standard software libraries. A component consists of two sections, a specification and an implementation. The specification describes the relevant properties of the component in an abstract manner, and it includes two parts: the functional one, stating how does the component behave, and the non-functional one, which declares additional requirements referred to some operational attributes (as efficiency). The implementation provides executable code for the component, usually written in an imperative or object-oriented programming language, and it must satisfy all the properties stated in the specification. Different software components with the same specification may differ in their implementation; this happens because given a functionality, normally it is not possible to develop one implementation that is the best in all operational respects.

For the time being, there are many projects addressing to component programming in the software community. We may cite: the STL approach [16] that extends C++ with a standard library of templates; RESOLVE [21], which provides a framework adaptable to many programming languages, as Ada and C++; and the Eiffel programming language and environment [18, 19], which includes a large number of public libraries. In this paper, we are going to present ComProLab, a new project in the component programming area. ComProLab can be viewed as structured in three parts that are presented in the next three sections:

- The product level. We have adopted and defined a set of languages to specify and implement software components.
- The process level. We have defined both a catalogue of tasks relevant to the development of systems, and a process language to combine them in order to formulate particular software process models.
- The tools. We have designed many tools for assisting software development in a variety of ways, emphasising prototyping and reusability of (implementation of) components.

For lack of space, and being our main goal an overall presentation of the project, we have decided to describe the components of ComProLab mainly through a general description followed by examples, without providing formal definitions. We refer to previous papers [1, 7, 8, 9, 10] to look into some additional details of our approach.

2 The product level

We present here the languages used to specify and implement software components in our laboratory, and we start by determining the organisation of components into modules. Fig. 1 shows the existence of four modules. Two of them correspond to the specification,

distinguishing the functional part from the non-functional one. The other two refer to the implementation, including the implementation itself (the code) and also a module to record the non-functional behaviour of the module (for instance, the efficiency of its operations, its degree of maintainability, etc.). The reason for keeping functional and non-functional information in separated modules will become clear later on.

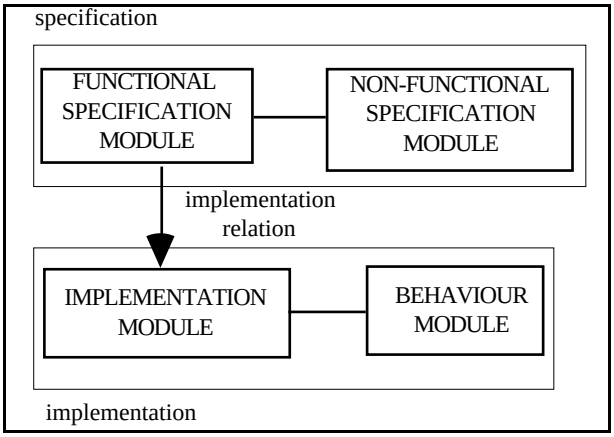

Fig. 1: Organising software components into modules.

Next, we address to the languages adopted. A main goal was to obtain an open environment at the product level, so that it could be adapted to arbitrary specification or programming languages. Even more, we have decided to allow a single system to be specified and/or implemented in more than one language (improving thus reusability of components, easiness of specification -using different formalisms in different places [23]-, etc.). To achieve these goals, we have defined an intermediate language called *Merlí* that includes features to build functional and non-functional specifications and implementations. The tools of our system work on Merlí specifications and implementations; so, in order to use a particular existing language in ComProLab it is enough to develop a translator from it to Merlí, being not necessary to adapt the existing tools to this language, which would be too time-consuming. The effort of translation to Merlí depends just on how far from it the source language is. An attractive result of this approach is that ComProLab does not require to learn new languages to be used (although it is possible to work directly with Merlí, as we are currently doing).

2.1 Functional specifications

For the time being, we consider two different kinds of functional specifications:

- Model-oriented specifications. As in Z [22] or VDM [17], where a model of the component is stated and the specification is expressed mainly by means of pre and post conditions over the model.
- Algebraic specifications. As in Larch [11] or OBJ3 [12], the specification consists of a set of equations. We are particularly interested in the possibility of using different semantics to interpret the equations (currently, we are using initial and behavioural semantics, as Larch does).

In fig. 2, we show the functional specification of a *NETWORK* software component, which represents networks (graphs) with a set of natural numbers as nodes, and unlabelled connections (edges) between them. *NETWORK* imports *LIST_OF_NATURAL* to form lists of nodes. We outline the algebraic specification. First, it introduces the type and the operations. Then, the equations for the operations appear; note that they may be conditional. By default, we interpret these equations into the initial semantics approach [5]; with this meaning, the equations can be seen as rules that manipulate expressions of type *network* until no further simplifications are possible, obtaining a minimal expression (its normal form) which involves a minimal subset of the operations of the component (in our case, *empty* and *add*). However, in the case of *top_sort*, the key word "behavioural" preceding its declaration breaks this default rule and so the last equation is interpreted in the behavioural framework.

```
functional specification module NETWORK
        imports LIST_OF_NATURAL
        type network
        operations ...as before, except for
        behavioural
            top_sort (network) returns list_of_natural
        equations ...
          remove(empty, m, n) = empty
          [x<>m or y<>n] =>
          remove(add(g, m, n), x, y) =
          add(remove(g, x, y), m, n)
          remove(add(g, m, n), m, n) = remove(g, m, n)
          ... rest of equations for the component
          [belongs(succ(d, m), n)] =>
          before(top_sort(d), m, n) = true
end module
```

Fig. 2: An algebraic specification of the NETWORK component.

2.2 Non-functional specifications

Non-functional specifications declare which operational attributes (what we call NF-properties) are relevant to the component being specified. NF-properties are really introduced in property modules and then non-functional specifications just import them. NF-properties may be of many different kinds, depending of the domain of their

values: boolean (e.g., full portability), numerical (e.g., degree of reliability), real (e.g., response time), by enumeration of values (e.g., kind of user interface -icons, menu, command language, ...-), string (e.g., programmer name) and asymptotic (time and space efficiency of types and operations, measured with the asymptotic big-Oh notation as defined in [2]). The set of asymptotic NF-properties bound to a component is fixed from its interface. It is possible to declare what we call measurement units, which represent problem domain sizes and that may be used as constant values when stating efficiency.

Once NF-properties have been selected, non-functional specifications state restrictions (called NF-requirements) over the implementations of the component. So, it is possible to formulate NF-requirements such as "implementations must be fully portable and user interface must be by means of icons" or "operations should not waste auxiliary space". Also, NF-requirements may appear in property modules to state universal facts about their NF-properties.

Fig. 3 gives a non-functional specification for *NETWORK*, which imports three property modules and states many NF-requirements over the NF-properties declared in them; other NF-requirements appear in the property modules themselves. The measurement units stand for the number of nodes and the number of connections in the network. In addition to the NF-properties that appear in these property modules, asymptotic properties such as time(*succ*) and space(*network*) also come into existence without explicit declaration. We remark that while the last two NF-requirements are up to the specifier, the first one state an universal relationship between the number of nodes and connections. We remark also the possibility to form hierarchies of property modules, and this fact (together with the possibility of declaring derived NF-properties, see below) allows to create structured catalogues of NF-properties.

In fact, the non-functional part of Merlí, called NoFun, presents more features that the ones shown here. For instance, it is possible to define derived NF-properties, which values are computed from other ones, that may also be derived as well. So, it is possible to create hierarchical libraries of NF-properties according to standards as [14, 15], defining general software attributes (maintainability, efficiency, etc.) in terms of more specific ones. Also, we mention the possibility to bind NF-properties to all the components in a software system at once, avoiding then explicit import in every component. We think that these and other features make NoFun powerful enough to be interesting in its own, and this is why we are using it in our laboratory. In fact, unlike the functional case, it is expected that most ComProLab users will build non-functional specifications with NoFun, because of the lack of notations offering the same capabilities in the component programming framework (see [20] and [3] as alternative proposals in this direction).

```
property module PORTABILITY
   properties
      boolean fully_portable
end module
property module PROGRAMMER
   properties
      string programmer_name
      boolean external_programmer
end module
property module RELIABILITY
   imports PORTABILITY, PROGRAMMER
   properties
      enumerated reliability = (high, medium, low)
   requirements
      not fully_portable => reliability <> high
      external_programmer and not fully_portable =>
                  reliability = low
end module
non-functional specification module NETWORK
 imports PORTABILITY, PROGRAMMER, RELIABILITY
       measurement units nbnodes, nbconns
       requirements
         nbconns <= power(nbnodes, 2)
         reliability <> low
         time(top_sort) < power(nbnodes, 2)
end module
```

Fig. 3: Non-functional specification of NETWORK using many property modules.

2.3 Implementations

As it have become the standard in component programming, we have chosen the object-oriented paradigm to implement components; also, imperative languages may be used. Almost all the constructions that Merlí provides are the classical ones in this paradigm (genericity, inheritance, etc.); however, a particular requirement must be pointed out: every implementation should include an abstraction function [13] to map a program object to the value it represents The existence of this function is required by the mixed execution tool we will present in the section 4.

We present next an abstraction function for an implementation of networks. It is characterised by the type of return, *TERM*(*network*), which is an instance of a generic (and predefined) type *TERM* to represent expressions of any kind. The operations over this type are the ones defined in the signature of the component. Note that the abstraction function appears in recursive form.

While at first glance it could seem that it is not efficient, section 4 will show that it is often the other way round.

```
type network = ^node end type
private type node = record
                        from, to: integer; next: ^node
end type
function abs_net (g: network) returns t:TERM(network)
        if g = NIL then t := empty
        else t := add(abs_net(t^.next), t^.from, t^.to)
        end if
end function
```

Fig. 4: An abstraction function for NETWORK.

2.4 Non-functional behaviour

Non-functional behaviour of implementations includes: on the one hand, assignments to all the NF-properties declared in the non-functional specification; on the other hand, NF-requirements stated over the implementations of imported components to make sure that the assigned values really hold. So, it is possible to express things as: "the response time of the operation *list_books* will not exceed one minute provided that the sorting algorithm for the set of books is not quadratic over the size of this set". As an alternative, an NF-requirement may be a selection of an implementation directly by its name.

Fig. 5 gives a behaviour module for an implementation *ImpNETWORK* of *NETWORK*; the NF-requirement over *LIST_OF_NATURAL* must be satisfied by the implementation selected for this component inside *ImpNETWORK*.

```
behaviour module for ImpNETWORK
    fully_portable; reliability = high
    programmer_name = "Franch"; not external_programmer
    time(succ) = nbnodes; ...
    requirements on LIST_OF_NATURAL:
        fully_portable and time(put) = 1
end module
```

Fig. 5: Behaviour module for an implementation of the NETWORK component.

In the general case, a behaviour module may include a list of NF-requirements over every imported component; NF-requirements in the list are considered in order of appearance (which corresponds to the usual case of having requirements with different degrees of importance). For instance, a NF-requirement over *LIST_OF_NATURAL* in *ImpNETWORK* could be: first, implementation must be as reliable as possible; next, the cost of the operations to build a list and their auxiliary space must be as fast as possible (i.e., O(1) in the big-Oh notation); last, the implementation should be fully portable. The list of NF-requirements stating this constraint will be:

```
requirements on LIST_OF_NATURAL:
    max(reliability);
    time(empty, put) = 1 and
        space(ops(LIST_OF_NATURAL)) = 1;
    fully_portable
```

3 The process level

We describe in this section a set of process tasks aimed at supporting system development in component programming, emphasising prototyping and component reusability. In the general case, prototyping could involve both functional specifications and implementations. The tasks identified in this catalogue act as primitives of a process language, which is also outlined here.

3.1 A catalogue of tasks

We have identified a set of tasks which cover all the aspects of component programming we are dealing with in ComProLab. The tasks are module-oriented; this is to say, all of them are referred to one or more particular modules from all kinds: functional specification, non-functional specification, implementation, behaviour and property modules. Many tasks will involve also standard libraries of components. They may be left temporally incomplete while performing other ones, or some of them may be executed simultaneously, provided that relationships between tasks are not violated (see below); also, many tasks may be completed by doing nothing (for instance, a component implementation may be left untested). Actually, tasks are organised into more specific subtasks, but we are not going to address this issue in the paper. The tasks are of three different kinds: for building functional specifications, for building non-functional specifications and for building implementations.

It is clear that these tasks satisfy some precedence relationships that must be followed in order to develop a correct design for a software system. To modelise these relationships, we have defined *precedence graphs*, bound to particular modules (i.e., relationships are module-oriented, as well as tasks), such that there is a node for every task in the catalogue and there is an edge from task *u* to task *v* whenever task *u* must be performed before than task *v*[1]. The process language presented in the next subsection will allow to establish relationships between tasks associated to different components.

[1] In fact, these relationships between subtasks can be slightly modified with relationships between subtasks.

3.2 Software process modelling

Once we have defined a catalogue of development tasks and the relationships they should follow, we focus in the problem of defining particular software process models as valid combinations of these tasks.

Given the modelisation of precedence relationships using graphs, we can consider a development strategy as a set of new edges binding nodes of these graphs. Sometimes, edges will relate tasks (nodes) in the same graph, to say things like "the functional specification of a component must be developed before the non-functional one"; however, in the general case, edges will involve tasks appearing in graphs bound to different modules, as in "it is necessary to specify all the components imported by a component *M* before any implementation of *M* is built". Also, we define a kind of grouping mechanism to allow the statement of facts as "functional and non-functional specification of a component must take place as a whole". As a result, we identify two different elements to formulate development strategies: rules and groupings, which are introduced below.

3.2.1 Precedence rules

We characterise software process models by adding new precedence relationships between tasks. We define these relationships as a pair (called *rule* hereafter) *left -> right*, where *left* and *right* are sets of tasks. The meaning of the rule is: if the tasks appearing in *left* have been completed, then all the tasks appearing in *right* can start to be executed; in other words, the rule is adding an edge from every task (node) appearing in *left* to every task appearing in *right*. Once again, let us remark that tasks are defined at the module level; as a result, rules will be parameterised by the modules appearing in tasks.

Software process models are encapsulated in *strategy modules*. It is possible to combine existing strategy modules to form new ones, adding optionally new rules and groupings. This property supports incremental development of strategies as combination of simpler ones, and improves understandability and reusability of the modules. Fig. 6 shows a module that determines a kind of bottom-up specification strategy: before specifying a component *M* (with the tasks *Fspecify* and *NFspecify*), it is necessary to specify all the components used by *M*; so, many (component-bound) precedence graphs are involved. As an example, we show in the figure the precedence graphs for a system with three components with specifications (*A*, *Anf*), (*B*, *Bnf*) and (*C*, *Cnf*) (functional and non-functional parts) such that (*A*, *Anf*) imports the other two. Note the use of predefined language constructs ("for all") and predicates ("used_in").

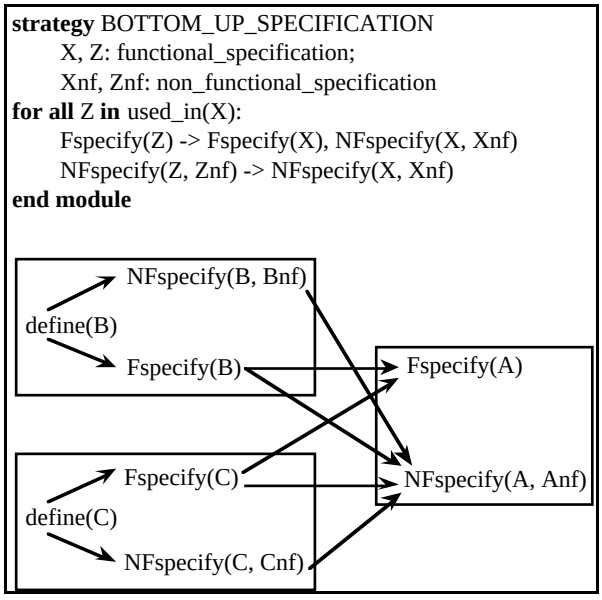

Fig. 6: A strategy module and some new relationships resulting from it.

3.2.2 Grouping tasks

We introduce here some notation to cover the need of grouping some related tasks, all of them usually referred to the same component. This grouping is expressed by enclosing the set of tasks between parenthesis, *(task1, ..., taskn)*. The meaning of this grouping is: once a task from *task1, ..., taskn* is started, development must complete all of them before starting any other task. This kind of mechanism is a mean to form new tasks as the joint of existing ones.

Note that grouping does not state nothing about the order of execution of these tasks (this is done using rules); also, note that grouping does not oblige neither to complete a task before starting others of the group nor the other way round (for instance, the *n* tasks may be simultaneously in execution if rules allow this situation). In fact, groupings can be formulated in terms of tasks: a grouping *(task1, ..., taskn)* adds an edge from every predecessor of every task in *task1, ..., taskn* to every successor of every task in *task1, ..., taskn*; so, the *n* tasks must infallibly be carried out as a whole.

Fig. 7 shows two examples of grouping. The first strategy module forces functional and non-functional specification of a component to be performed altogether. As component precedence graphs do not include any precedence relationship between these two tasks, any order of execution and state of completion is possible. In the second one, we show the use of quantifiers in grouping; the module states that all the functional specifications of imported components must take place as a whole. Finally,

we include a third strategy module that shows how strategies may be built in an incremental manner.

```
strategy WHOLE_SPECIFICATION
        M:functional_specification;
        Mnf:non_functional_specification
          (Fspecify(M), NFspecify(M, Mnf))
end module

strategy SPECIFICATION_OF_USED_MODULES
        M, Z:functional_specification;
          (for all Z in used_in(M): Fspecify(Z))
end module

strategy BOTTOM_UP_WHOLE_SPECIFICATION
    combines BOTTOM_UP_SPECIFICATION,
                WHOLE_SPECIFICATION
end module
```

Fig. 7: Three new strategy modules with groupings.

4 The tools

We present in this section the tools of ComProLab. We focus on those we think are the most relevant ones: a mixed execution system, to support prototyping with specifications and implementations; an implementation selection algorithm, to select automatically the best implementation of components in their context of use; a tool for computing the efficiency of programs; and a process development assistant, parameterised by the particular process model being used. The first three tools, which work at the product level, manipulate Merlí specifications and programs; software written in others languages requires thus a previous translation step.

4.1 Mixed execution

A point that is often argued against formal specifications is the gap between them and the final software system. Some approaches try to fill this gap by introducing transformation techniques that automatically derive correct implementations from specifications. Currently, most of these approaches present some limitations that difficult their actual success in the industrial community.

We propose here a different framework that allows to create preliminary prototypes that combine algebraic functional specifications and implementations in an arbitrary manner. This is to say, an initial prototype may be defined consisting of components for which just a specification exists, and then they may be implemented one by one, obtaining at any stage an executable prototype; furthermore, prototyping is possible even with components partially implemented, making thus possible to implement and test operations one by one. On the other hand, there is no need to specify all the components in the system; some of them may be directly implemented, and prototyping will also be possible in this case. We have explored in previous works [1, 6, 7] the conditions that should be fulfilled in order to have successful prototyping.

The mixed execution framework combines a term rewriting system to work with specifications and an interpreter to deal with implementations. The equations of the algebraic specification are interpreted as rewrite rules oriented from left to right. The crucial point during execution is value passing from the rewriting system to the interpreter and vice versa. The abstraction function of implementations play a crucial role in this communication. Let *A* and *B* be two components such that *A* is only specified while *B* has both specification and implementation, and such that *B* introduces a type *t* which is represented somehow in the implementation, including an abstraction function *abs*. If *A* contains a rewriting rule *F(...x...) --> K*, where *x* is a subexpression of type *t*, and given a particular data structure *str* built with the implementation of *t* in *B*, the rewriting rule is applicable over *str* if *abs(str)* matches *x*. On the other hand, if we have an expression *x* of type *t* coming from a manipulation done with the rewriting system, it is enough to execute this expression using the implementation of *B* to obtain a data structure.

Fig. 8 shows an example of mixed execution. We assume a partial implementation of *NETWORK*, providing a type representation by adjacency lists, the abstraction function and the implementation of *empty* and *add*. Prototyping in this situation may involve the execution of *remove(x, 1, 2)*, being *x* a data structure. Since *remove* is not implemented, *x* is converted into an equivalent expression with the abstraction function, and the resulting term is rewritten using the equations shown in fig. 2, obtaining its corresponding normal form that can be transformed into a data structure again by executing the operations contained in the nodes of the term using the partial implementation of *NETWORK*.

It is clear that mixed execution is a time-consuming technique. However, it should be noted that it takes place only during prototyping (not in the final version of the system); in this stage, we are worried about testing both system functionality and implementation issues (absence of errors in the code, user-interface, etc.), not about efficiency. On the other hand, we have defined some mechanisms to improve execution time. We remark a kind of lazy evaluation to find out if a rewriting rule may be applied or not; if the abstraction function is provided in recursive form, it may be evaluated partially just to find a rewriting rule matching the piece of term built. In fig. 8, just an application of the abstraction function is necessary to apply the appropriate rewrite rule.

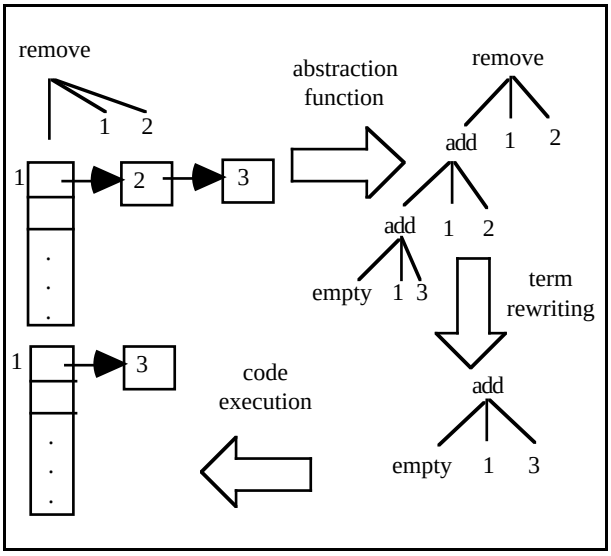

Fig. 8: An example of mixed execution.

4.2 Automatic selection of implementations

Non-functional specifications and non-functional behaviour may be used not just as additional information of components, but also in an operative way to select the best implementation for a component in a context of use. So, we have built an algorithm for selecting implementations of software components with a NF-behaviour that satisfy the requirements put on them.

Given an implementation *Imp*, the algorithm proceeds by comparing the NF-requirements stated in *Imp* over the imported components *C1*, ..., *Ck* with the NF-behaviour appearing in the implementations of these *C1*, ..., *Ck*. In order to be able to perform these comparisons, relationships between the measurement units should be stated; then, it is possible to test if conditions such as "nbnodes * nbconns < power(nbnodes, 2)" are satisfied. These relationships may appear in any place of the software system, but it is advisable not to include them in generic components as *NETWORK*, because they may depend on the context where components appear. So, two different software systems using *NETWORK* may state different relationships between the number of nodes and the number of connections of their networks; for instance, a system may state "nbconns = nbnodes" to modelise a ring computer network, while the other may declare "nbconns = power(nbnodes, 2)" (the maximum value allowed by the non-functional specification of *NETWORK*) to represent fully connected networks.

Note that, in fact, the algorithm will not bind single implementations to software components but trees of implementations, because an implementation may import other components for which the algorithm must also proceed. For instance, if we suppose that the implementation *ImpNETWORK* (see fig. 5) for *NETWORK* is selected inside the implementation *ImpMAIN* of a component *MAIN* that imports *NETWORK*, then an implementation for the component *LIST_OF_NATURAL*, imported by *NETWORK*, should also be selected inside *ImpNETWORK*; all these selections are made following the NF-requirements appearing in the corresponding NF-behaviour modules. More than this, as far as the software system is a hierarchy of software components, NF-requirements in upper modules must be taken somehow into account in lower ones [9]. For instance, the NF-behaviour of *ImpMAIN* (see fig. 9) could also state some NF-requirements over *LIST_OF_NATURAL*, which must be combined with those ones appearing inside the behaviour module of *ImpNETWORK*. By default, requirements are combined and then the implementation of *LIST_OF_NATURAL* inside *ImpNETWORK* must satisfy the constraint "time(put) = 1 and fully_portable and dynamic_storage", although there are some constructs in the language that allow to change this behaviour.

```
behaviour module for ImpMAIN
  behaviour
  requirements
     on NETWORK: time(top_sort) <= power(nbnodes, 2)
     on LIST_OF_NATURAL: dynamic_storage
end module
```

Fig. 9: A module for stating NF-behaviour of a component using NETWORK.

Finally, it must be noted that the algorithm may select a component with different implementations in different places of the system. This situation may be problematic if two objects of the same type interact during program execution, as it could happen if we try to concatenate a list implemented with an array to a list implemented by pointers. The abstraction function could be a mean to transform one of the values from one implementation to another, but as this would take a lot of execution time in the final version of the system, we have currently preferred to reject this possibility (although this decision could change in the future).

4.3 Computing values of NF-properties

We address now to the computation of the (asymptotic worst-case) execution time of all the operations of components, the space wasted in representing the values of all their types, and the auxiliary space needed for executing the operations.

Efficiency can be computed from three kinds of

information. On the one hand, we have defined some semantic rules attached to the grammar of Merlí to provide a default computation. For instance, the semantic rule bound to (a restricted form of) the conditional instruction, "**if** <expr> **then** <instr> **else** <instr> **end if**", is:

time(<conditional>) =

time(<expr>)+time(<instr>-1)+time(<instr>-2)

(The sum must be interpreted in the asymptotic big-Oh notation as [2] does -union of set of functions-. The "-1" and "-2" are the mean to distinguish between repeated appearances of the same non-terminal symbol in a grammar rule.)

In many situations, however, efficiency cannot be established just from program syntax. The most obvious case has to be with loops: the execution time of a loop depends on the number of times it is really performed, and this information is not inferable from the syntax of the loop. This implies that the semantic rule for the loop "**while** <expr> **do** <instr> **end do**" will include an additional item of information, *NB_TIMES*, such that:

time(<loop>) =

NB_TIMES * (time(<expr>) + time(<instr>))

Every loop in the program, then, will include an assignment to this item, possibly involving one or more measurement units, as in "NB_TIMES = nbnodes". These assignments will appear as annotations in the code of programs, and the translation to Merlí is required to keep them without any modification, to be used by the tool.

Even in this situation, it may happen that the complexity of a piece of code cannot be determined exactly with semantic rules. For instance, fig. 10 presents two nested loops to visit all the connections of a network *N*; the operations of *LIST_OF_NATURAL* are the usual ones to get the elements of a list one after the other. The assignments to *NB_TIMES* reflect the worst-case situation, in which every node is connected with the others, the lists returned from *succ* being then of length *nbnodes*. As a result, the semantic rule computes the complexity of this piece of program as power(*nbnodes*, 2)) * *k*, being *k* the cost of manipulating the current node in *lsucc*, and assuming that operations over lists take constant time and that *all_nodes* and *succ* are as fast as they can be (i.e., proportional to the number of involved items). However, there are some implementations of *NETWORK* which carry out this loop in a time proportional to the number of connections; if this number is smaller than power(*nbnodes*, 2), the complexity obtained is not accurate anymore.

Fig. 11 shows the solution to this problem. We introduce a third kind of non-functional information, what we call *component skeletons*, which define patterns of use of components with their complexity and that are declared in skeleton modules, imported in non-functional specifications. Skeletons are pieces of programs involving function calls, variables and non-terminal symbols (as "<sent>") which can be instantiated. Note that the program of fig. 10 matches with the skeleton.

```
lnodes := NETWORK.all_nodes(N)
LIST_OF_NATURAL.reset(lnodes)
while not LIST_OF_NATURAL.end(lnodes) do
        -- NB_TIMES = nbnodes
        lsucc := NETWORK.succ(N, current(lnodes))
        LIST_OF_NATURAL.reset(lsucc)
        while not LIST_OF_NATURAL.end(lsucc) do
           -- NB_TIMES = nbnodes
           manipulation of current(lsucc)
                (* taking O(k) time *)
           LIST_OF_NATURAL.next(lsucc)
        end while
        LIST_OF_NATURAL.next(lnodes)
end while
```

Fig. 10: A loop over networks.

```
skeleton module network_traversal for NETWORK
   lnodes := NETWORK.all_nodes(N)
   LIST_OF_NATURAL.reset(lnodes)
   while not LIST_OF_NATURAL.end(lnodes) do
             lsucc := NETWORK.succ(N, current(lnodes))
             LIST_OF_NATURAL.reset(lsucc)
              while not LIST_OF_NATURAL.end(lsucc) do
                <sent>
               LIST_OF_NATURAL.next(lsucc)
              end while
             LIST_OF_NATURAL.next(lnodes)
   end while
end skeleton module
```

Fig. 11: A skeleton for NETWORK.

Implementations must give values to skeletons in their NF-behaviour modules, as if they were normal asymptotic NF-properties; the assigned values will usually involve the efficiency of the non-terminal symbols appearing in the skeleton, which will be unknown as it may be different in different instances of the skeleton. So, an adjacency-list representation of *NETWORK* would assign (*nbnodes* + *nbconns*) * time(<*sent*>) to the NF-property *time*(*network_traversal*), while an adjacency-matrix one would assign power(*nbnodes*, 2) * time(<*sent*>), requiring in both cases constant cost in list operations.

Once an instance of a pattern skeleton is found (applying a kind of pattern-matching technique similar to the one involved in term-rewriting), its complexity is determined from that stated in the NF-behaviour of the involved implementations. In the code of fig. 10, the final cost in the adjacency-list case would be

(*nbnodes* + *nbconns*) * *k*, better than before when it holds that *nbconns* < *nbnodes*; using the adjacency-matrix representation, however, the complexity remains the same. We remark that is precisely this ability to obtain different values for the same piece of program what makes components skeleton attractive.

4.4 The process model assistant

The process model assistant is designed to help developers in building systems using a strategy defined with the process language introduced in section 3. In other words, the assistant is parameterised by the particular process model used to develop the system. To achieve this goal, it offers at every moment the tasks that do not violate the precedences and groupings stated in the strategy module that define the process model. It is worth to point out that various tasks may be carried out in parallel respecting this correctness condition.

A particular point of interest is system redevelopment. When it is necessary to modify one or more components in the system, (part of) the tasks corresponding to these components must be replayed, but this may require to replay tasks bound to other components. In the example of fig. 6, if the specification of the component *B* is modified, the tasks *Fspecify(A)* and *NFspecify(A, Anf)* must be performed again to avoid the violation of precedence rules. In this case, the assistant allows the automatic execution of these tasks as they were executed before.

5 Conclusions

We have presented ComProLab, a laboratory for component programming that defines languages and tools at both the product and process levels.

Concerning languages, we allow the use of arbitrary specification and programming languages in the component programming framework and we define an intermediate language, Merlí, used as a common notation which specifications and programs are translated to; Merlí provides features to state non-functional information of components. Also, a language is introduced to define software process models as the composition of many development tasks establishing some precedence and grouping relationships between them.

Concerning tools, the laboratory includes an execution module to prototype systems combining imperative code with algebraic specifications in an arbitrary manner; a tool to select the best implementation of components in systems from the non-functional information stated in components; a tool to compute the efficiency of implementations; an a process model assistant able to deal with any model defined with the process language. Many of these components have been presented in more detail in previous works, but this is the first time we have integrated all of our lines of research in a single project.

We think that the most interesting points of our approach are the following:

- The product and the process levels are integrated in an unified framework. This results in an homogeneous approach, where languages, tools and methods in both levels are closely related.
- Due to the open architecture of ComProLab, the laboratory may be used with arbitrary specification and programming languages in the component programming framework; this feature is not usual in component programming projects, that normally work with a fixed (small) set of languages. Thus, we could use Z, Larch and OBJ3 to specify components, and C++, Ada95 and Eiffel to implement them, just by building a translator from these languages to the intermediate language Merlí. We defined Merlí complete enough to support these translations from a wide class of those languages.
- Non-functional specifications and non-functional requirements of software are taken into account in the laboratory. This is a point that makes our approach attractive, since we know of very few other approaches addressing to non-functional issues of software in the component programming field, although it is widely recognised that they are as important as functional ones. In fact, [20] is the only approach that we know in this field that treats a particular non-functional feature (efficiency) together with the functional ones.
- Use of formal specifications is supported by a prototyping tool able to execute programs at intermediate stage of development, combining code and specifications in an arbitrary way. The tool is aimed at covering the gap that exists for the time being between the initial specification of the problem and the final delivered system. Other projects support this kind of execution, but we only know [4] providing almost identical features from a compilation point of view.
- We have defined a process language consisting of very few elements to make it easy to learn and use: a small catalogue of tasks with well-defined relationships, two mechanisms to relate tasks (rules and groupings) and a few additional constructions (as quantification and many built-in functions).

- Software process models may be defined incrementally, from the combination of small strategy modules, each one of them addressing to particular points of the model. We believe that this modularity increases understandability, maintenance and reusability of strategy modules.

Acknowledgments

We would like to thank the anonymous referees for their valuable comments.

References

[1] X Burgués, X. Franch. "Evaluation of Expressions in a Multiparadigm Framework". In *Proceedings of 7th Programming Languages: Implementation, Logics and Programs (PLILP)*, Utrecht (The Netherlands), LNCS 982, Springer Verlag, 1995.

[2] G. Brassard. “Crusade for a Better Notation”. SIGACT News, 16(4), 1985.

[3] D. Cohen, N. Goldman, K. Narayanaswamy. "Adding Performance Information to ADT Interfaces". In *Proceedings of the Interface Definition Languages Workshop*, ACM SIGPLAN Notices 29(8), 1994.

[4] C. Choppy, S. Kaplan. "Mixing abstract and concrete modules: specification, development and prototyping". In *Proceedings of XII International Conference on Software Engineering* (ICSE), Niça (France), 1990

[5] H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics*. EATCS Monographs on Computer Science, Vol. 6, 1985.

[6] X. Franch, X. Burgués. "A Case Study on Prototyping with Specifications". In *Proceedings of ERCIM Workshop on Development and Transformation of Programs*, Nancy (France), 1993.

[7] X. Franch, X. Burgués. "Incremental Component Programming with Functional and Non-Functional Information". In *Proceedings of XVI International Conference of Chilean Computing Science Society* (SCCC), Valdivia (Chile), 1996.

[8] X. Franch, P. Botella. "Supporting Software Maintenance with Non-Functional Information". In *Proceedings 1st EUROMICRO Conference on Software Maintenance and Reengineering* (to be published), Berlin (Germany), 1997.

[9] X. Franch. "Combining Different Implementations of Types in a Program". In *Proceedings Joint of Modular Languages Conference* (JMLC), Ulm (Germany), 1994.

[10] X. Franch. "Including Non-Functional Issues in Anna/Ada Programs for Automatic Implementation Selection". In *Proceedings of Ada-Europe'97. International Conference on Reliable Software Technologies* (to be published), London (United Kingdom), LNCS, Springer-Verlag, 1997.

[11] J.V. Guttag, J.J. Horning. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science, Springer-Verlag, 1993.

[12] J.A. Goguen *et al*. "Introducing OBJ3". Draft Report, SRI International, 1993.

[13] C.A.R. Hoare. “Proof of Correctness of Data Representations”. In *Programming Methodology*, Springer-Verlag, 1972.

[14] IEEE Computer Society. *IEEE Standard for a Software Quality Metrics Methodology*. IEEE Std. 1061-1992, Institute of Electrical and Electronical Engineers, New York, 1992.

[15] International Standards Organization. *Software Product Evaluation - Quality Characteristics and Guidelines for their Use*. ISO/IEC Standard ISO-9126, 1991.

[16] M. Jazayeri. "Component Programming - a Fresh Look at Software Components". In *Proceedings of 5th ESEC*, Barcelona (Catalonia, Spain), 1995.

[17] C.B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1990.

[18] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[19] B. Meyer. *Reusable Software. The Base Object-Oriented Component Libraries*. Prentice-Hall, 1995.

[20] M. Sitaraman. "On Tight Performance Specification of Object-Oriented Components". In *Proceedings 3rd International Conference on Software Reuse*, IEEE Computer Society Press, 1994.

[21] M. Sitaraman (coordinator). "Special Feature: Component-Based Software Using RESOLVE". ACM Software Engineering Notes, 19(4), Oct. 1994.

[22] J.M. Spivey. *The Z Notation*. Prentice-Hall, 1993.

[23] P. Zave, M. Jackson. "Where do Operations come from? A Multiparadigm Specification Technique". IEEE TSE, 22(7), Jul. 96.