Software Process Modelling as Relationships between Tasks

Xavier Franch
franch@lsi.upc.es
Universitat Politècnica de Catalunya
Jordi Girona 1-3, 08034 Barcelona
Catalonia (Spain)
FAX: 34-3-4017014. Phone: 34-3-4016965

Josep M. Ribó
josepma@eup.udl.es
Universitat de Lleida
P. Víctor Siurana 1, 25003 Lleida
Catalonia (Spain)
FAX: 34-73-702162. Phone: 34-73-702000

Abstract

Systematic formulation of software process models is currently a challenging problem in software engineering. We present here an approach to define models covering the phases of specification, design, implementation and testing of software systems in the component programming framework, taking into account non-functional aspects of software (efficiency, etc.), automatic reusability of implementations in systems and also prototyping techniques involving both specifications and implementations. Our proposal relies on the identification of a catalogue of tasks that appear during these phases which satisfy some relationships concerning their order of execution. A software process model can be defined as the addition of more relationships over these tasks using a simple, modular process language. We have developed also a formal definition of correctness of a software development with respect to a software process model, based on the formulation of models as graphs.

1. Introduction

It is widely recognised that one of the most challenging problems in the field of software engineering is the systematic formulation of software process models in a such a way that it can be said that software processes are software too [13]. A lot of research has been done in this field and, as a result, some proposals have been defined, which can be characterised by the kind of language or formalism used to represent the model: imperative programs [18, 2], transformation rules over specifications and programs [12], composition of inference rules [14], multiview approaches [17] and so on; [7] gives a presentation of many projects recently developed (including exhaustive reference lists). Some methods address to the whole software process, while others focus on a subset of phases of the life-cycle (usually, specification and design); anyway, the existence of those proposals is a step beyond the uniform treatment of products (software) and processes (developments) in software engineering, as we think it should be.

In this paper, we are going to present a framework to formulate software process models for component programming [10, 15], assigning a prominent role to the management of operational aspects of software (as efficiency or reliability); we also emphasise prototyping and reusability of (implementation of) components. More precisely, we consider a subset of the whole software process composed by specification, design, implementation and testing phases, and we identify a catalogue of tasks that arise during these activities, stating some precedence relationships between them. We define then a process language characterised by the statement of new relationships between tasks; the language has been designed with the goal of simplicity and modularity in mind and, so, it seems to be easy to learn and use. Also, we provide a formal definition of the concept of correctness of a software development with respect to a process model defined with this language.

2. The framework

We are interested in software systems as a hierarchy of software components. A component is defined by means of a specification, which includes two parts: the functional one, stating how does the component behave, and the non-functional one, that declares additional requirements referred to some operational attributes (as efficiency); these attributes are defined in property modules, imported in non-functional specifications. Once the specification is complete, many implementations may be built for this component, all of them satisfying the properties stated in both parts of the specification; implementations include a description of their non-functional behaviour, which determines the values that the operational attributes declared in the non-functional specification take in the implementation, possibly stating some additional constrains on implementations of imported components.

Up to now, our method has been defined over an *ad hoc* language called *Merlí*. Merlí includes features to build functional and non-functional specifications and implementations. The reason of working with Merlí is twofold. On the one hand, it will be not necessary to develop new tools when considering concrete specification and implementation languages, except from a translator from them to Merlí. On the other hand, we have defined an execution tool over Merlí [1] able to prototype systems that combine specifications and programs provided that some conditions hold. However, it must remain clear that the methodology we are going to propose does not depend of the languages used to specify components and to implement them (provided that they have similar characteristics to the ones adopted in Merlí); so, we are not requiring to learn Merlí to adopt our proposal.

A more detailed description of Merlí may be found at [4, 6]; we give here just the highlights to understand the framework of our proposal.

2.1. Functional specifications

We consider two kinds of functional specifications:

- Model-oriented specifications. As in Z [16] or VDM [11], 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 [8] or OBJ3 [9], the specification consists of a set of equations. We are particularly interested in the possibility of using different semantics (initial and behavioural, as Larch does) to interpret the equations.

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 they 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, ...-) and string (e.g., programmer name)[1], and they can be attached to single operations or to whole modules (so, we can talk about response time of individual operations or about full portability of a whole module). It is possible to declare what we call *measurement units*, which represent problem domain sizes (e.g., number of books in a library) and that may be used as constant values, mainly when stating efficiency.

Once NF-properties have been selected, non-functional specifications state restrictions (*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 must have a response time not exceeding one second".

2.3. Implementations

As it has become usual in the component programming field, we have chosen the object-oriented programming paradigm to code the implementations. Concerning non-functional behaviour, it includes: on the one hand, assignment to all the NF-properties declared in the non-functional specification; on the other hand, requirements stated over the implementations of imported components to make sure that the assigned values really hold. So, it is possible to state 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".

2.4. An example

We present in the next page four figures that show the modules for a *NETWORK* component, which represents topological networks (directed graphs) with nat numbers as nodes, and unlabelled connections (edges) between them. In fig. 1, we outline both model-oriented and algebraic specifications (one of them should be chosen); in the second case, the keyword "behavioural" before *top_sort* breaks the default rule of interpreting the last equation with initial semantics. Fig. 2 gives a non-functional specification, which attach the NF-properties declared in some property modules appearing in fig. 4 to modules and operations, and adds some additional properties; the measurement units stand for the number of nodes and connections in the network. Last, fig. 3 gives a behaviour module for an implementation *IMPL_NETWORK* of *NETWORK*; the NF-requirement over *LIST_NAT* must be satisfied by the implementation selected for this component inside *IMPL_NETWORK*.

3. Catalogue of process tasks

We describe in this section a set of process tasks aimed at supporting component programming with prototyping, and allowing the automatic selection of implementations from their non-functional characteristics. In the general case, prototyping could involve both functional specifications and implementations; we have explored in previous works [1, 3, 4] the conditions that should be fulfilled in order to have successful prototyping. As we said in the introduction, the tasks identified in this catalogue act as primitives of our process language, introduced in section 4.

[1] We have also a special kind of domain for measuring efficiency, the domain of the asymptotic notations, that we do not introduce here for the sake of brevity.

```
functional specification module NETWORK
  imports LIST_NAT
  type network = V: set_of(nat) x E: set_of(nat x nat)
  invariant g: network; v, w: nat
        not (v, v) in g.E     -- not reflexive edges
        (v, w) in g.E => (v in g.V) and (w in g.V)
  operations
      ...
      add, remove (network, nat, nat) returns network
      ..
  behaviour
      ... pre and post conditions for the operations
      {m <> n and m in g.V and n in g.V}
         g' := add(g, m, n)
      {g'.E = union(g.E, singleton((m, n)))}
      ...
end module
functional specification module NETWORK
  imports LIST_NAT
  type network
  operations
      ...
      behavioural top_sort (network) returns list_nat
  equations
      ... equations for the component
      [belongs(succ(d, m), n)] =>
                       before(top_sort(d), m, n) = true
end module
```

Fig. 1: Two alternative functional specifications for a NETWORK component.

```
non-functional specification module NETWORK
    imports PORTABILITY, EFFICIENCY,
              PROGRAMMER, RELIABILITY
    module level fully_portable, programmer_name,
                    external_programmer,
    operation level time, space, reliability
    measurement units nbnodes, nbconns
    requirements
       nbconns <= pow(nbnodes, 2)
       not fully_portable =>
                    reliability(ops(NETWORK)) <> high
        external_programmer and not fully_portable =>
                    reliability(ops(NETWORK)) = low
end module
```

Fig. 2: Non-functional specification of NETWORK.

```
behaviour module for IMPL_NETWORK
      fully_portable; not external_programmer
      programmer_name = "Smith"
      time(succ) = nbnodes; ...
      reliability(ops(NETWORK)) = high...
      requirements on LIST_NAT: fully_portable
end module
```

Fig. 3: Behaviour module for a NETWORK implementation

```
property module PORTABILITY
   properties
      boolean fully_portable
end module
property module PROGRAMMER
   properties
      boolean external_programmer
      string programmer_name
end module
property module EFFICIENCY
   properties
      numerical time, space
end module
property module RELIABILITY
   properties
      enumer reliability = (high, medium, low)
end module
```

Fig. 4: Some property modules declaring NF-properties.

Tasks are module-oriented; this is to say, all of them are referred to one or more modules from all kinds: functional specification, non-functional specification, implementation, behaviour and property modules. The tasks presented below may be left temporally incomplete while executing other ones, or some of them may be executed simultaneously, provided that relationships between tasks are not violated (see 3.2). Also, some of the tasks may be performed just by doing nothing (for instance, an implementation may be left untested).

3.1. The tasks

- Tasks for building modules. We have identified: *define(M)*, to declare the interface of a software component; *declare(P)*, to introduce the name and type of one or more NF-properties in a property module, and optionally some NF-requirements over them; *Fspecify(M)*, to build the functional specification of a component *M*; *FNspecify(M, Mnf)*, to build a non-functional specification *Mnf* of a component *M*; *implement(M, I)*, to build an implementation *I* of a component *M*; and *state_behaviour(I, Inf)*, to state the behaviour *Inf* of an implementation *I*. These tasks may reuse modules from the library with the tasks introduced below.
- Library management tasks. We consider the existence of libraries to store specifications, implementations and property modules. Then, we have defined the following tasks:
  ◊ Reusing a component specification composed by *M* and *Mnf* (functional and non-functional parts) from the library *L* in a component specification *M'*. This kind of reuse may be of three different kinds:

i *import(M', L, M, Mnf)*: to reuse the module *M* without any modification except some optional renaming.

ii *instantiate(M', L, M, Mnf)*: to obtain a concrete component from a generic one (e.g., sets of books from generic sets), with optional renaming.

iii *inherit(M', L, M, Mnf, ...)*: to obtain a new component by (possibly multiple) inheritance from other ones.

◊ Reusing a component implementation composed by *I* and *Inf* (code and behaviour module) from the library *L* in a component implementation *I'*. In this case, only *import* and *instantiate* are allowed.

◊ Importing a property module *P* from the library *L* in another property module *P'*: *import(P', L, P)*.

◊ Importing a property module *P* from the library *L* inside a non-functional specification *Mnf*: *attach(Mnf, L, P)*.

◊ Storing a module *X* into the corresponding library *L*: *store(L, X)*. In the case of specifications and implementations, *X* include both the functional and the non-functional parts.

- Operational tasks. We include here prototyping tasks, implementation selection tasks and validation tasks.

◊ Prototyping a functional specification *M*: *test_spec(M)*. Also, there exists *test_impl(I)*, to prototype an implementation *I*. Both tasks may eventually involve mixed execution (combining specifications and code) as explained in [1, 4].

◊ Testing if the behaviour module *Inf* attached to an implementation satisfies the NF-requirements stated in the corresponding non-functional specification *Mnf*: *NFvalidate(Inf, Mnf)*. The success of this task is necessary to consider the implementation correct.

◊ Testing if an implementation *I* of a component *M*, with NF-behaviour *Inf*, satisfies the NF-requirements stated over *M* inside another behaviour module *Inf'*: *NFtest(Inf , M, Inf)*. The success of this task is necessary to consider *I* as a valid implementation of *M* in the context represented by *Inf'*.

◊ Selecting manually an implementation *I* of a component *M*, with NF-behaviour *Inf*, inside another implementation *I'*: *NFmanselect(I', I, Inf)*.

◊ Selecting automatically an implementation of a component *M* which satisfies the NF-requirements stated over *M* inside another implementation *I'* with NF-behaviour *Inf'*: *NFautoselect(Inf', M)*.

3.2. Precedence graphs

It is clear that the tasks identified above 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 three different kind of graphs, that we call *precedence graphs*, referred to specifications, implementations and property modules. Each graph is bound to concrete modules of the appropriate type; so, relationships are module-oriented, as well as tasks.

Fig. 5 presents the precedence graph for a specification with functional part *M* and non-functional part *Mnf*, *SpecGraph(M, Mnf)*. Interface definition should precede both functional and non-functional specifications of the module. To carry out prototyping, functional specification should be complete. Once the specification is complete, it may be stored in the library in order to be retrieved for its future use in other components.

Fig. 7 shows the precedence graph for an implementation *I* with NF-behaviour *Inf*, *ImplGraph(I, Inf)*. It is stated that an implementation (its code) should be built once its specification is complete, and afterwards its NF-behaviour should be stated, and also prototyping of the code may be carried out. From the non-functional specification and the NF-behaviour, implementation validation is possible and it must precede storage in the library.

Last, fig. 6 shows the graph for a property module *P*, *PropGraph(P)*. As the ones before, it is stated that a module should be completed before storing it in the library, and then it may be imported by other property modules or it may be attached to a particular non-functional specification.

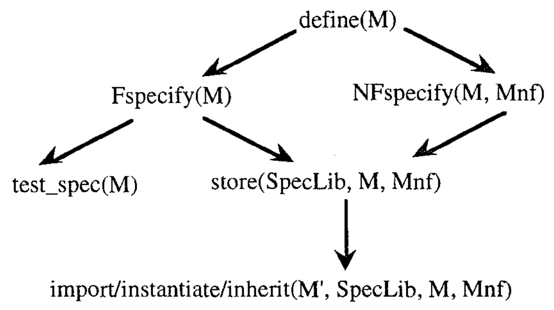


Fig. 5: Precedence graph at the specification level.

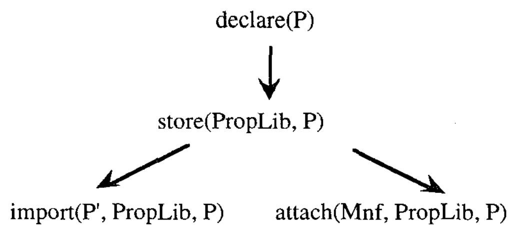


Fig. 6: Precedence graph at the property level.

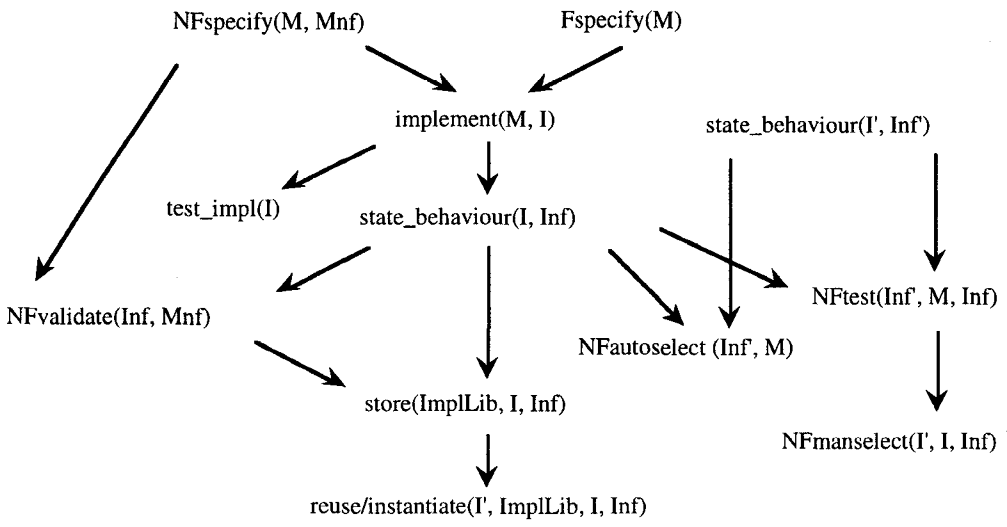


Fig. 7: Precedence graph at the implementation level.

4. Software process models

Once we have defined the catalogue of existing tasks and the relationships they should follow in system development, we focus in the problem of how to define concrete software process models. As said in the introduction, we focus in four particular phases of software process: specification, design, implementation and testing; however, we will continue using the term "software process" as a shorthand for these phases.

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 in 4.1 and 4.2.

4.1. Rules of precedence

A software process model is mainly characterised by some particular 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's remark that tasks are defined at module level; as a result, rules will be parameterised by the modules appearing in tasks.

We define the following elements to write rules:

- Identifiers to represent module names.
- Tasks, parameterised by modules.
- A quantifier of the form:

  for all x **in** set of modules: rule(x)

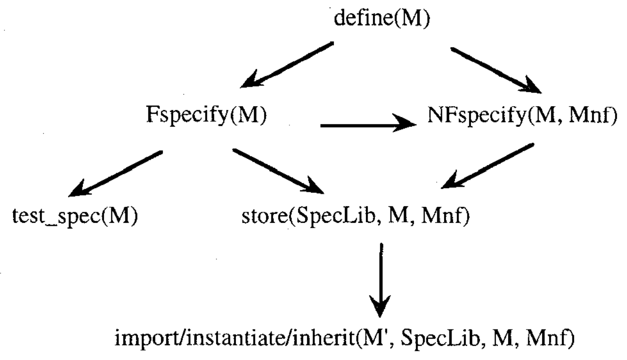
  meaning that the rule holds just for the specified set of modules.
- Some built-in functions to obtain sets of modules related somehow with a given one.

Fig. 8 shows an example. 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. The first strategy module forces functional specification of components to be carried out before non-functional one. This is a rule that add edges in single graphs, the ones for component specifications, as we show in the first graph of fig. 8. The second strategy determines a kind of bottom-up specification strategy: before specifying a module *M*, it is necessary to specify all the modules used by *M*; so, many graphs are involved, and we show in fig. 8 the specification graphs for a system with three specifications (including functional and non-functional parts) (*A*, *Anf*), (*B*, *Bnf*) and (*C*, *Cnf*) such that (*A*, *Anf*) uses the other ones. Last, the third strategy module combines the previous ones, yielding to a kind of bottom-up specification development strategy that gives precedence to the functional part. We omit variable declarations.

```
strategy FUNCTIONAL_BEFORE_NON_FUNCTIONAL
  Fspecify(M) -> NFspecify(M)
end  module
```

```
strategy BOTTOM_UP_SPECIFICATION
      for all Z in used_in(X):
          Fspecify(Z) -> Fspecify(X), NFspecify(X, Xnf)
          NFspecify(Z, Znf) -> NFspecify(X, Xnf)
end  module
```

```
strategy BOTTOM_UP_WITH_FUNCTIONAL_BEFORE
      combines
          FUNCTIONAL_BEFORE_NON_FUNCTIONAL,
          BOTTOM_UP_SPECIFICATION
end  module
```



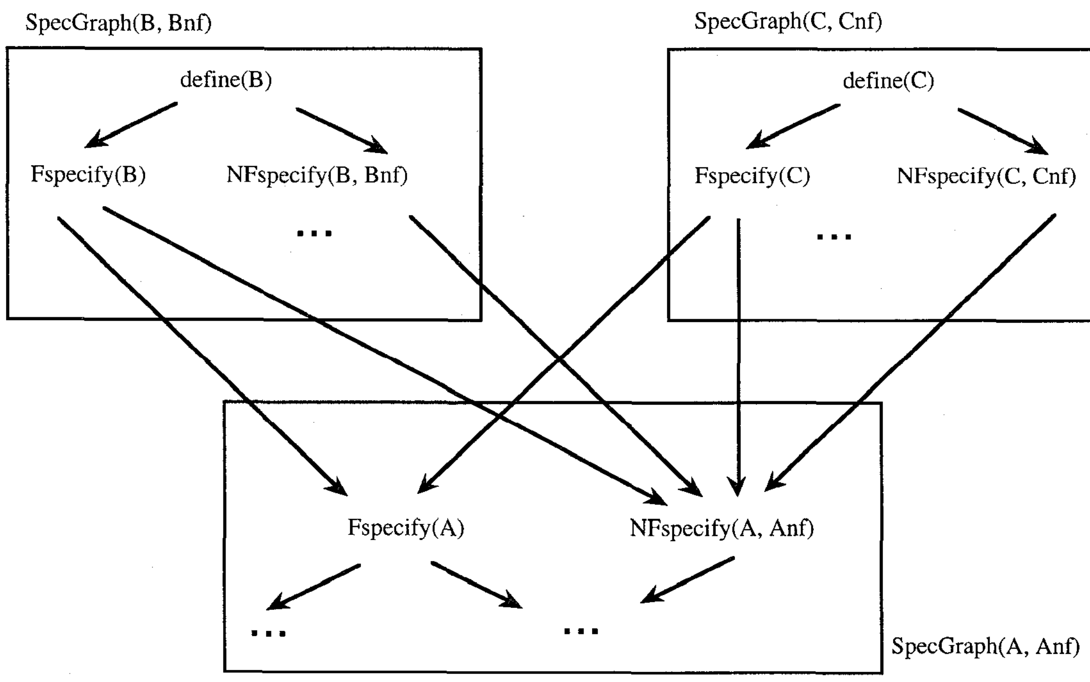


Fig. 8: Three strategy modules and the specification graph resulting from the first two of them.

4.2. Grouping of tasks

We introduce here some notation to cover the need of grouping some related tasks, all of them usually referred to the same module. 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. Tasks in a grouping may refer to a subset of modules, using the same quantifier as rules.

Note that grouping does not state nothing about 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). Last, we remark that 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. 9 shows two examples of grouping and its combination with rules. The first strategy module forces functional and non-functional specification of a component to be performed as a whole. As the specification graph of the component does not include any precedence relationship between these two tasks, any order of execution and state of completion is possible. But if we consider a development strategy combining this grouping and the *FUNCTIONAL_BEFORE_NON_FUNCTIONAL* strategy (see fig. 8), the result is a new strategy that requires, when specifying every component in the system, to complete the functional part and immediately the non-functional one; note the difference with the strategy *FUNCTIONAL_BEFORE_NON_FUNCTIONAL* alone, that allows to carry out other tasks between functional and non-functional specifications. Last, the strategy *SPECIFICATION_OF_USED_MODULES* shows the use of quantifiers in grouping; the module states that all functional specifications of imported components must take place indivisibly; this module could combine with *BOTTOM_UP_SPECIFICATION* to form a new strategy.

```
strategy WHOLE_SPECIFICATION
        (Fspecify(M), NFspecify(M, Mnf))
end  module

strategy WHOLE_SPECIFICATION_FUNCTIONAL_1ST
    combines
        FUNCTIONAL_BEFORE_NON_FUNCTIONAL,
        WHOLE_SPECIFICATION
end  module

strategy SPECIFICATION_OF_USED_MODULES
        (for all Z in used_in(M): Fspecify(Z))
end  module
```

Fig. 9: Two new strategy modules with grouping.

5. Correctness of software developments

In this section, we are going to define formally the notion of correctness of a system software development with respect to a software process model[2]. First, we propose a model for system software developments as a sequence of tasks.

Definition 1. Software system development.
A software system development is a sequence of tasks such that there are not repeated tasks:

SSdev = (task)* / $\forall s \in$ SSdev: $s = t_1 \ldots t_k$: $i \neq j \Rightarrow t_i \neq t_j$

where "$t_i \neq t_j$" means that t_i and t_j cannot be the same task applied to the same module(s). ◊

Next, we formalise the notion of software process model as a pair of sets, a set for rules and a set for groupings. Then, we define the graph bound to a software system development as a graph including as many subgraphs as modules appear in *s* (see 3.2) and incorporating directed edges between nodes given both the set of rules and the set of groupings of the process model, as explained in section 4. Last, we formulate the notion of correctness of a software system development with respect to a process model in terms of a topological sort over the resulting graph.

Definition 2. Software process model.
A software process model *g* is a pair, *g* = (*Sr, Sg*), such that *Sr* is a set of rules and *Sg* a set of groupings:

$$\text{Sr} \in \mathcal{P}((\text{task})^* \times (\text{task})^*) \wedge \text{Sg} \in \mathcal{P}(\mathcal{P}(\text{task})) \quad ◊$$

Definition 3. Graph induced by sets of modules, rules and groupings.
Let *S*, *Sr* and *Sg* be sets of modules (in the case of specifications and implementations, pairs of modules including functional and non-functional parts), rules and groupings, respectively. We define the graph induced by *S*, *Sr* and *Sg*, Graph(*S*, *Sr*, *Sg*), as the minimum graph satisfying:

- $\forall$M, Mnf: (M, Mnf)$\in$ S $\wedge$
    M and Mnf form a specification:
        SpecGraph(M, Mnf) $\subseteq$ Graph(S, Sr, Sg)
- $\forall$I, Inf: (I, Inf)$\in$ S $\wedge$ I and Inf form an implementation:
        ImplGraph(I, Inf) $\subseteq$ Graph(S, Sr, Sg)
- $\forall$P: P$\in$ S $\wedge$ P is a property module:
        PropGraph(P) $\subseteq$ Graph(S, Sr, Sg)
- $\forall$r: r$\in$ Sr $\wedge$ r = (l -> r):
    $\forall$x, y: x$\in$ l $\wedge$ y$\in$ r:
        $\forall$assignment α of the modules of x and y
            with values from S, x[α] and y[α]:
        the edge (x[S] -> y[S]) is in Graph(S, Sr, Sg)
- $\forall$g: g$\in$ Sg $\wedge$ g = (t_1, ..., t_k):
    $\forall$i, j: $1 \leq i, j \leq k \wedge i \neq j$:
        $\forall$assignment α of the modules of t_i and t_j
            with values from S, $t_i[\alpha]$ and $t_j[\alpha]$:
        ($\forall$x,y: the edges (x -> $t_i[\alpha]$) and ($t_j[\alpha]$ -> y)
            are in Graph(S, Sr, Sg)):
            the edge (x -> y) is in Graph(S, Sr, Sg)

where *SpecGraph(M, Mnf)*, *ImplGraph(I, Inf)* and *PropGraph(P)* are defined as in 3.2. We assume that quantifications implicitly expand to sets of rules and groupings. ◊

Definition 4. Correctness of a software system development with respect to a software process model.
Let $s = t_1 \ldots t_k \in$ SSdev be a software system development and let *g* = (*Sr, Sg*) be a software process model. We say that *s* is correct with respect to *g* if *s* follows a valid topological sort traversal of the graph induced by *g* and the modules of *s*:

$s \in$ TopSort(Graph(Modules(s), Sr, Sg)),

where Modules(*s*) gives the set of modules introduced in *s* and TopSort(*f*) gives the set of valid topological sort traversals over the graph *f* ◊

[2] We do not focus here on functional and non-functional correctness of the system itself, which may be studied through classical proof obligations. Also, we do not address here to completeness of software systems developments, defined as the existence of implementations enough to obtain a completely implemented system satisfying all the requirements stated in behaviour modules.

6. Conclusions

We have presented a proposal to formulate software process models in the component programming framework. This proposal relies on the existence of a catalogue of tasks to build the components, to prototype them, to select the appropriate implementations for them and to store them and to retrieve them to/from libraries; this tasks present precedence relationships between them. Software process models are encapsulated into strategy modules, which consists of a set of rules (new precedences between tasks) and a set of groupings (tasks that must be considered as a whole). Finally, it has been formally introduced the notion of correctness of a software development with respect to a software process model, based in a graph representation of process models.

There are many aspects of our work that have not been included in the paper. First, the notion of component redevelopment, which requires redefining slightly the definition of software system development. Also, we do not include the complete definition of correctness, taking into account functional and non-functional correctness. Last, we have not shown the decomposition of tasks into subtasks. Another interesting point is the use of the development sequence as a script to analyse the software process and, eventually, to replay it in the future [5, 14].

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

- The process language consists of very few elements to make it ease 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 (quantification and predefined 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 may say that our process model language falls into component programming at the process level and, so, the benefits in this field also apply to our proposal.
- A formal notion of correctness has been defined. We believe that correctness in the process level is as important as correctness in the product level. Our work aims at treating both levels uniformly.
- Non-functional requirements of software are taken into account during software development. This aspect has not been studied in detail here, but is a basic one in our project [4, 6]: we believe that non-functionality is as important as functionality and this requires explicit treatment in the process model.
- Although the proposal has been presented for an *ad hoc* notation, it does not really depend on it; so, the proposal may be adapted for every (functional) specification and programming languages with similar characteristics to the ones presented here: model-oriented or algebraic specifications and object-oriented programming.

References

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

[2] E. Dubois, A. van Lamsweerde. "Making Specification Processes Explicit". In *4th International Workshop on Software Specification and Design*, Monterey (U.S.A.), 1987.

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

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

[5] X. Franch, P. Botella. "Prototipado de Programas usando Especificaciones Funcionales y No Funcionales" (written in spanish). In *Actas de las Primeras Jornadas de Trabajo en Ingeniería del Software*, Sevilla (España), 1996.

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

[7] A. Finkelstein, J. Kramer, B. Nuseibeh. *Software Process Modelling and Technology*. J. Wiley & sons, 1994.

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

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

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

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

[12] B. Krieg-Brückner (ed.). *Program development by Specification and Transformation*. LNCS 680, Springer Verlag, 1993.

[13] L. Osterweil. "Software Process are Software Too". In *Proceedings of 9th ICSE*, Monterey (U.S.A.), 1987.

[14] M. Sintzoff. "Expressing Program Developments in a Design Calculus". In *Procs. of the Intl. Summer School on Logic of Programming*, NATO ASI Series, Vol F36, Springer Verlag, 1987.

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

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

[17] M. Saeki, K. Wenyin. "Specifying Software Specification and Design Methods". In *Proceedings of 6th CAiSE*, Utrecht (The Netherlands), LNCS 811, 1994.

[18] D. Wile. "Program Developments: Formal Explanation of Implementations". In *New Paradigms for Software Developments*, IEEE Computer Society Press, 1986.