The Knowledge Engineering Review, Vol. 11:4, 1996, 303–316

An overview of declarative process modelling using logic programming

PAOLO CIANCARINI

Dip. di Scienze dell'Informazione, University of Bologna, Italy. Email: ciancarini@cs.unibo.it

Abstract

Software process modelling is the activity of formalising the production lifecycle of large software systems. Its aim is to formally describe a software development process, which is then effectively used and possibly enacted by an environment able to support the geographically distributed and coordinated activities involved in the process itself. I show that rule-based languages, especially logic programming languages, are an important technology for the specification, modelling, enactment and coordination of software processes. This is because most routine activities in any development process can be defined by rules. Some initial proposals aimed at simply simulating the software process by a Prolog-like program embedding some development rules. A further step toward the integration of rule-based languages in the software process has been taken using a dynamic knowledge base as project database, and a number of special primitives have been introduced to support process programs. Currently there is a trend toward more complex programming environments, called *process-centred development environments*. I show how some rule-based coordination language have been used to build an environment of this kind.

1 Introduction

Activities involved in the development of large software products are complex, time-consuming and error-prone. Osterweil (1987) proposed to completely specify the development process of software products, in order to gain control of the process and improve the quality of the products.

The result of such a specification is a so-called *software process model*, that is, some kind of description which should precisely define the activities that are carried out during the development of a software project, providing guidance to all parties involved and controlling the overall evolution of the project status (Curtis *et al.*, 1992). Thus, a *software process model* is a description of some integrated software development activities: Osterweil suggested that such a description should be so prescriptive that actually it is a *software process program*.

Software process modelling is a relatively recent research area that aims to study languages and environments which allow one to model, design and enact real software processes (Curtis *et al.*, 1992; Armenise *et al.*, 1993). The main concern of process modelling is the formalisation of a number of customary activities which take place during the production of software objects. A software process includes agents that can be programmers or tools, data that are mostly documents, and cooperation protocols that involve several transformational steps of synchronising events and communication actions. A software process model must allow us to describe all these entities, specifying their role in the actual process, and possible support and enactment by the underlying support environment. In fact, most software development processes usually take place within a software development environment.

I have studied software process models and programs for a number of years; my main idea has been that modelling the process also means modelling its environment, whereas to build an environment means to (implicitly) model a process. Think about "plain" Unix: its features for

supporting projects in which several programmers are involved define, in some way implicitly, the process that has to be used by the programmers themselves.

A *process-centred software development environment* is a project support environment which explicitly allows one to define and enact software development processes (Christie, 1995). That is, a process-centred software environment supports (sometimes enforces) the interactions of the project members, coordinates the use of programming tools, and monitors the evolution of the project documents.

The design of process-centred environments is currently an exploratory and incremental prototyping task, because we know very little about suitable process models and the features which should be offered by related process-centred environments.

I decided to use rule-based languages for this design activity because of their declarative expressive power and support for an exploratory programming style. In fact, I was not alone in such a choice, because other researchers as well used rule-based or even logic languages for process modelling (Huff & Lesser, 1988; Katayama, 1989; Heimbigner, 1989; Peuschel *et al.*, 1992; Jaccheri & Conradi, 1993; Welzel, 1993).

The main lesson I learned was that any programming environment should be designed with the goal of explicitly modelling and coordinating the process it is intended to support. In fact, a natural way to model a software process consists of introducing a suitable abstract machine describing the environment in which the software process itself takes place. Such a description should be written using a suitable notation able to enact and control the software process evolution. The guiding principle is that, in order to specify a software process, one has to clarify the *coordination model* that should be used by all the agents involved in the process, i.e. the communication mechanisms that are at the basis of the interaction protocols used by the participants of a software project must be made explicit (Kraut & Streeter, 1995).

I identify the coordination model with the abstract architecture of the (distributed) environment which supports the execution of the process itself. I explored especially the blackboard model (Nii, 1986) as coordination model. I found that it is quite natural as a support architecture for various cooperation protocols. Thus, I started to use the blackboard concept as the main structuring mechanism for process models and environments.

Another choice I and my colleagues made was to employ for all our experiments logic programming. The use of logic programming techniques and tools in software engineering practice is not widespread (Ciancarini & Levi, 1992). I have chosen logic programming because of some key features I could exploit in my research: declarativeness helped in dealing with abstract data used to model process documents; rules helped in dealing with complex coordination protocols necessary to control the process workflows; most important, the availability of efficient commercial Prolog programming environment supported the design and rapid prototyping of specific process centered tools.

Ideally, a language for software process modelling should be both *declarative*, to specify at a high level what the agents can or cannot do, and *prescriptive*, to state what the agents should do. In this paper, I discuss the use of the logic programming paradigm to model through rules software process models. The paper is organised as follows: section 2 describes the problem we deal with; section 3 overviews a number of proposals in the field of (sequential) rule-based process modelling languages. In section 4 I describe my work on software process modelling using Flat Concurrent Prolog and the coordination language Shared Prolog. I also shortly overview my most recent work on process modelling and monitoring using coordination technology and the World Wide Web to offer support for process enaction.

2 Software process modelling

Modern industrial software development is a dynamic activity in which many cooperating participants act to transform a set of requirements into a validated system whose maintenance phase starts almost immediately after the requirements have been written. Software products are the

result of complex production processes that involve a collection of interrelated activities which should be not only rigorously performed, but also recorded and studied in order to improve the overall quality of both the products and the processes used to develop them.

A concrete definition of the software process is the following (Mi & Scacchi, 1990): *the software process is a collection of related activities, seen as a coherent process subject to reasoning, involved in the production of a software product.* These activities take place inside an environment that should actively enact the cooperation of all the agents involved in the software production process.

The reader can find a complete, paradigmatic example of an informal software process specification in Kellner *et al.* (1991). Such a problem specifies a software development process in natural language; it concerns the development, change and testing of a defective software module. Such a process is decomposed in a number of phases: designing, coding, testing and management of change. There is an authority, CCB (Change Control Board), in charge of the project; there is a programmers' team PT coordinated by a project manager PM. There are documents that have to be produced, there are interaction protocols that have to be enforced.

In Fig. 1, taken from Kaiser *et al.* (1993), a software process fragment is depicted, representing a subprocess of the problem to be specified. The graphical notation is an abstraction of several activity threads cooperating and synchronising.

This kind of picture is difficult to interpret, because what is depicted is a number of concurrent activities with some synchronisation points. This difficulty demonstrates that software process models should be described in some formal notation with a clear operational semantics able to deal with concurrency and non-determinism.

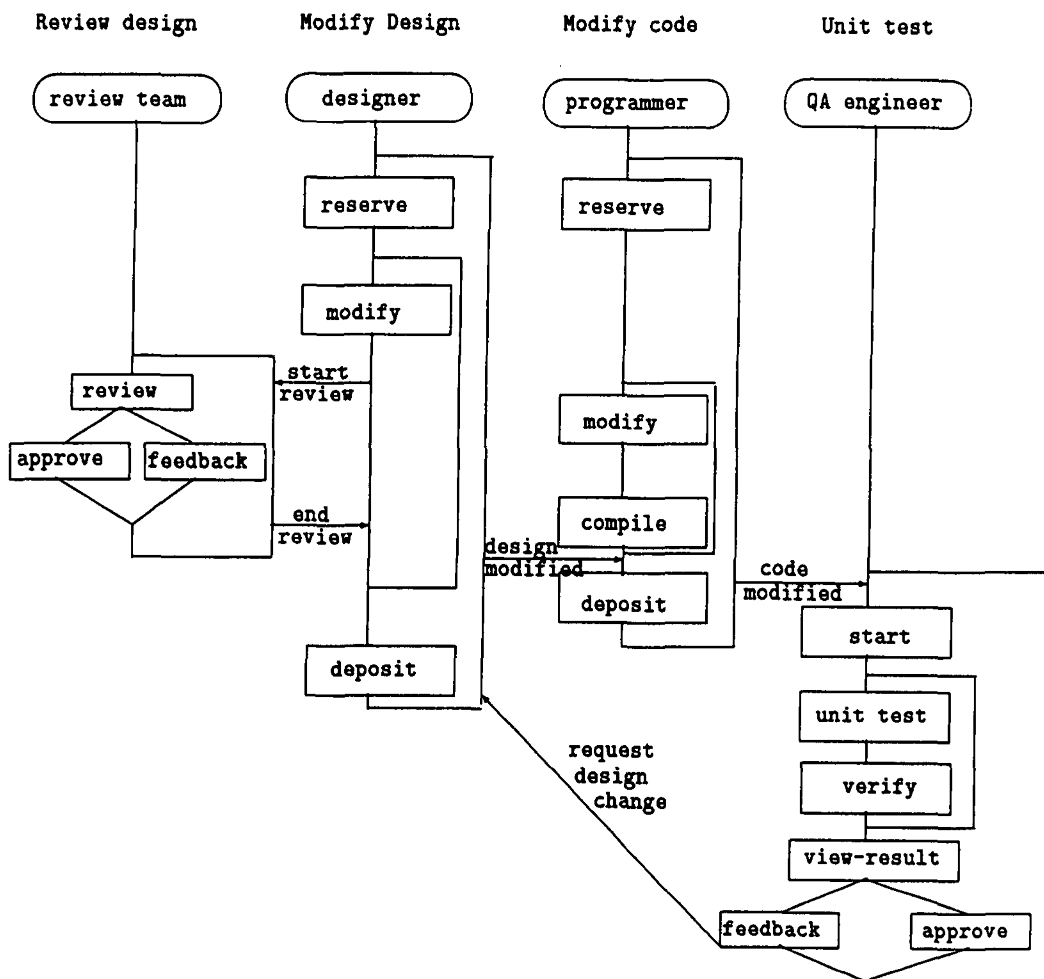

Figure 1 A software process fragment

Here we do not intend to survey and assess the main languages proposed and/or used for software process programming: the reader can find such a survey in Armenise *et al.* (1993). Instead, I will focus on a specific class of software process languages, namely rule-based, declarative languages. This class includes several examples, most of which are directly or indirectly based on Prolog.

3. Some solutions based on rule-based languages

All the approaches we describe exploit two features of rule-based programming, namely declarativeness and flexibility. Usually, rules encapsulate software development activities, thus specifying the preconditions that should be satisfied for performing a given activity, and the effect of such activity (Barghouti & Kaiser, 1990).

Most of the examples we will describe are based on Prolog, because it offers several features that are useful for process modelling. I believe that several researchers used Prolog because it is a widespread rule-based language for which mature programming environments exist.

3.1 Modelling software processes with Prolog

A first proposal for the use of logic programming for software process modelling is found in the SDA project (Kishida *et al.*, 1988), where Prolog was suggested as process programming language (Ohki & Ochimizu, 1989). The idea is that the process designer writes a process program specifying in Prolog the rules that govern the development phases. A process program is then defined by a set of rules specifying all the roles involved in the development process.

Example

```
% a process program in Prolog
designer_interaction(User_Spec,Arch_design):-
  listup(User_Spec,IO_data_and_Transctns),
  threads(IO_data_and_Transctns,Threads),
  write_spec(Threads,Reqs),
  validate(Threads,Reqs,Analysis)
  apply_methods(Reqs,Arch_design),
  propose(User_Spec,Arch_design).
```

This Prolog clause is a rule that states a number of activities that should be executed by the process designer himself to pass from a document stating the user requirements to the document describing the architectural design. The rule includes as substeps: the listing of all I/O data and transactions included in the user specification document; the description of all control threads needed to manage the I/O data and transactions; the writing of requirements using the threads; the validation through the analysis of these requirements; the use of some design method for obtaining an architectural design, and the proposal of an architectural design for implementing the user specification.

Failure of any subtask is easily handled by backtracking to the preceding subtask.

I believe that this kind of process program can be seen more as a memo for project management than as an executable specification of development processes. This approach is also the main idea at the basis of the design of DesignNet (Liu & Horowitz, 1989), an "intelligent", knowledge-based tool written in Prolog, and able to assist a project leader in coordinating a team of software designers.

Another early proposal that suggested a rule-based formalisation of the software process is HFSP (Katayama, 1989). The acronym HFSP stands for "Hierarchical and Functional Software Process"; abstractly, the process is described by a number of functions; here we are interested in the fact that each function is represented by rules that involve typed entities.

Example

I show a simple software process from Katayama (1989). It formalises the process enacted by someone applying the JSP method, which is often used in the development of business information systems. The graphic notation used in Figure 2 is JSP itself.

Now we show the formalisation in HFSP:

```
type
  spec = sequence of word
  formalSpec,analyzedSpec =
    sequence of spectoken
  prog,keywords= sequence of token
  oprs = sequence of operation
activity
  JSP(spec | prog.out) =>
    MakeProgTree(spec | progTree)
    EnumerateOprs(spec | oprs)
    MakeProg(oprs,progTree | prog.mid)
    DoProgInversion(prog.midTree | prog.out)
MakeProgTree(spec | progTree) =>
    AnalyzeSpec(spec | formalSpec)
    MakeInputTree(formalSpec | dataTree.in)
    MakeOutputTree(formalSpec | dataTree.out)
    ComposeTrees(dataTree.in,dataTree.out|progTree)
AnalyzeSpec(spec | formalSpec) =>
    ExtractKeywords(spec | keywords)
    SyntaxAnalysis(keywords,spec | analyzedSpec)
    SemanticAnalysis(analyzedSpec | formalSpec)
.....
```

A software process program in HFSP starts declaring a number of types; then a number of rules called activities are listed. Each rule in its left part (before the arrow) declares an activity, giving its name and its input and output documents as parameters (the bar separates inputs from outputs). In the right part of a rule, a number of subactivities are listed, which have to be performed to complete the activity listed in the left part. The coordination of subactivities inside a rule is governed by backtracking. This means that if an activity for some reason fails, for instance the `SyntaxAnalysis` in `AnalyzeSpec`, control flows back to past activities, to find alternate ways to fulfil the stated task.

More complex examples of the use of such a notation for describing and comparing software process models can be found in Song and Osterweil (1994).

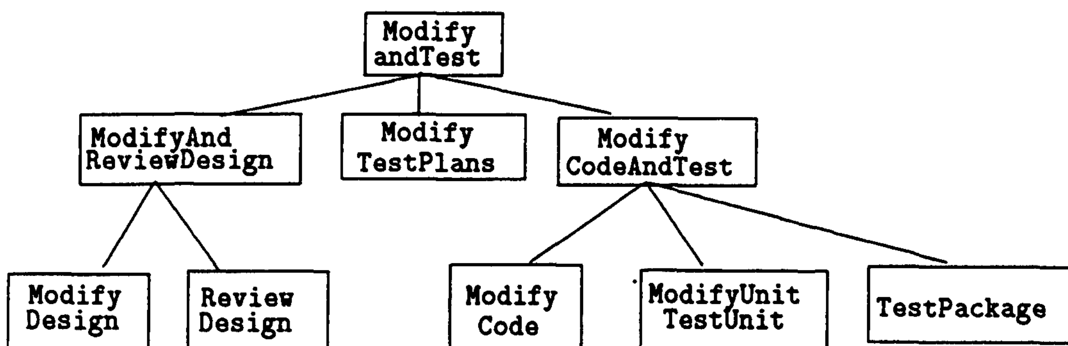

Figure 2 Depicting a software process in JSP

3.2 *Animating a software process*

A direct practical use of Prolog in software process animation is suggested in Christie (1995). Christie uses a graphical notation as notation for software process modelling. The notation is called Pro-Net. Using Pro-Net a software process is depicted by a graph having a precise semantics.

For instance, in Figure 3 it is depicted as a simple software process element: it is a generic activity which has a number of input edges and a number of output edges. Elements can be combined in complex pictures depicting software processes.

The pictures representing the software process can then be animated in Prolog to verify that the process has some specific properties, typically concerning both process and product certification.

More precisely, the graphic specifications are compiled by Prolog for animation. For instance, in Figure 4 we show a very simple Pro-Net and its Prolog counterpart. A square box represents an activity, with some input and output edges. An activity may only be started if some abstract resources are available, or some conditions are met. As a consequence of an activity, exit conditions may become true (or false) and some resources may be generated. Each activity is recorded in a log which can be recorded and replayed back. In fact, the Prolog rule associated to such a picture says: "the activity fires if there is some `i_product`; after the activity some `o_condition` becomes true".

Christie shows how a process program animated in Prolog can be used to formally certify the development process, validating and verifying the development phases it is composed of.

A similar use of Prolog for software process animation is found in SCOPE. Welzel (1993) describes a process representation technique based on rules that are written in a special graphic form and easily translated in Prolog, so that the process can be simulated and verified. This formalism has been adopted by the SCOPE consortium, that includes a number of European software industries, as a tool for software evaluation and certification. The graphic form is a variant of condition-event Petri nets. The basic idea consists of representing and planning processes and project activities as rules which state their causal dependencies.

The notation is based on rules of the following form:

```
IF   | a(A1,...,An),...
     | x(X1,dots,Xn)
-----+-----------------
THEN | y(Y1,...,Yn)
```

meaning that if predicates a,...,x are valid, then the predicate y is also valid. It is immediate to translate these rules into Prolog, to achieve an animation of some allowed processes.

Another early proposal for the use of Prolog in the activity of software process programming can be found in Heimbigner (1989).

3.3 *Toward rule-based process-centred environments*

The use of a sequential rule-based language leaves some questions open: How does the process program invoke standard tools? How does it coordinate a team of programmers? How are the complex data structures produced by the software development process modelled?

An approach which tries to give an answer to these questions consists of defining the software process as an activity that necessarily takes place within a software development environment. Such

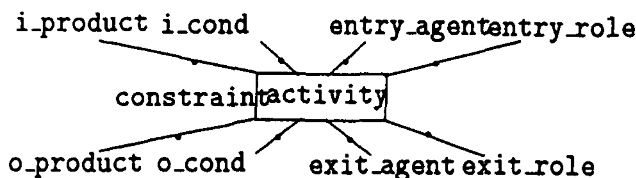

Figure 3 Representation of a software process element

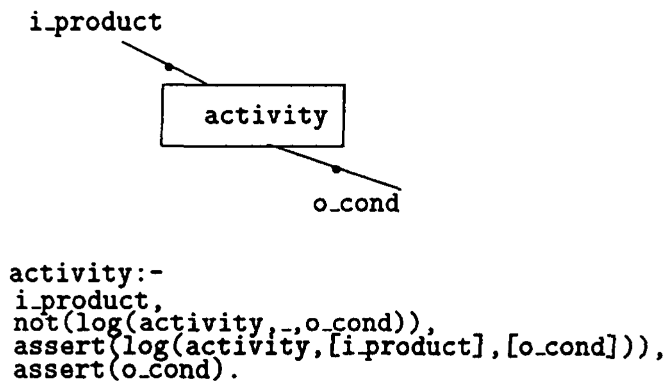

Figure 4 A process element and its Prolog counterpart

an activity involves a number of roles and documents that can be specified by a conceptual language, e.g. using the Entity Relationship model. Then the environment itself that supports the software process is a program: since it specifies the coordination of the users, let us consider it the process program. Thus, building an environment able to offer process programming support is like giving an explicit representation of the software process. This is the rationale underlying the notion of process centered development environments.

For instance, the Darwin project (Minsky & Rozenshtein, 1990) develops a Prolog-based framework to support rule-based software engineering environments, called for *law governed systems*. In Darwin a law is an explicit statement of a rule that must be followed by agents in the environment.

The main assumption about software process modelling made by Darwin designers is the existence of a law governing the evolution of a system, by constraining the messages allowed to be exchanged among the entities that compose the system. Formally, a law is a restricted Prolog program. Darwin itself is a constraint satisfaction system implemented as a prototype written in Prolog.

A different Prolog-based tool is GRAPPLE (Huff and Lesser, 1988). It is a plan-based process assistant, that includes primitive environment operators that correspond to atomic actions within the environment. Some primitive actions correspond to tool invocations, while other primitive actions correspond to predefined scripts. These operators may be combined in complex operators to achieve a given goal in a software process phase. Programmers communicate with the environment, which provides both passive (constraints) and active (plans) assistance. Plans are dynamically built by instantiating operators. The environment recognises user plans and generates system plans. The assistant makes use of explicit knowledge on the software process manipulated by non-monotonic reasoning based on a multivalued logic. GRAPPLE consists of a tool implemented in Prolog: the inferential capabilities of the language were largely used.

The approach that sees the process program as closely related to the software development environment in which it evolves is evident in MERLIN (Peuschel *et al.*, 1992). Such an environment monitors and guides a team of developers and managers that produce software objects. MERLIN is based on an extension of Prolog that combines backward and forward chaining.

Example

A knowledge base stores all the information about a software project. For instance, documents to be manipulated are represented by facts of the form `document(Doc_Type,Doc_Name,Status)`, were Doc_Type is a document type and Doc_Status is the current status of the document. They have attributes of the form `work_on(Doc_Type,Doc_Status,R,W,X)`, where R, W, X are lists describing access rights for read, write, and execution operations, respectively.

```
document(module, m1, to_be_edited).
document(specification,m1,specified).
```

```
document(error_report, m1, reported).
work_on(module, to_be_edited,
  [spec, report, review], [module], ([])).
work_on(review, to_be_reviewed,
  [spec, module], [review], []).
work_on(object_code, to_be_executed,
  [spec, module], [], [object_code]).
```

These tables define the document types that a programmer can manipulate. Other facts define the roles of the people involved in the development process, and their responsibilities with respect to the documents. Then, a number of rules define the operating environment inside which every process participant works. For instance, the following rule initialises the working context of a programmer:

```
do_working_context(R,W,X,Item):-
IF start_working_context
THEN
  CALL(working_context,R,W,X,Doc_Type,Doc_Name,New_Status,Item),
  REMOVE(document(Doc_Type,Doc_Name,_),
  INSERT(document(Doc_Type,Doc_Name,New_Status),
  REMOVE(start_working_context);
IF document(module(Obj_Name,to_be_compiled))
THEN
  CALL(compiler,Obj_Name,Compile_Status),
  REMOVE(document(module(Obj_Name,to_be_compiled)),
  INSERT(document(module(Obj_Name,Compile_Status)).
```

The first component of this rule specifies that after each change of the document status the document database is updated accordingly. The second component specifies the automatic invocation of a compiler, again updating the document database with the new correct attributes.

Even from this simple example, it is evident that MERLIN is a rule-based software environment that can support a large variety of software processes. These are declaratively defined by set of rules whose combination induces cooperation protocols on a set of programmers.

A similar use of Prolog in combination with a database can be found in the EPOS system: a software process is written in a special language called SPELL (Nguyen & Conradi, 1994) which to be executed is translated in Prolog (Conradi *et al.*, 1992). SPELL is defined as a persistent object-oriented language with a reflective architecture (Jaccheri & Conradi, 1993). The idea is to define the software process via an Entity-Relationship conceptual model including classes and meta-classes. The conceptual model shapes documents in a project database. SPELL is a query language for the project database which can be used to create, change, instantiate and enact process models. The SPELL interpreter is invoked by a Prolog predicate as follows:

```
call_proc(?Caller,+Called,+Procedure_Name,+Input_Args,-Output_Args)
```

The software process is defined by several rules which are evaluated by this interpreter with respect to the contents of the project database.

3.4 Using rule-based blackboard coordination

The blackboard model of problem solving is well known in artificial intelligence (Nii, 1986). The model includes as main components:

- the blackboard, a data repository which is shared among a number of knowledge sources;
- a number of knowledge sources, namely agents which share and can modify the contents of the blackboard;
- a scheduler, which coordinates and enables the knowledge sources to modify the blackboard.

If we see the software process as a set of related activities of agents which have to be coordinated in producing the final product, it seems natural to model the environment in which the process takes place as a blackboard; programmers and tools can be seen as knowledge sources which cooperate through a document repository which behaves like a blackboard. A recent and relevant example of this approach is REBUS, where the blackboard is called "whiteboard" (Sutton *et al.*, 1991). REBUS is a process program written in APPL/A, an extension of Ada for software process modelling. The REBUS whiteboard is used through five *ad hoc* primitives that manipulate its contents. The whiteboard is used to coordinate the activities of a number of participants in the software process.

Another environment where we can trace a blackboard coordination architecture is Marvel (Ben-Shaul *et al.*, 1992; Barghouti, 1992). Marvel puts emphasis on designing the interaction protocols used by a multiplicity of users who share a project database: what happens in such a database is similar to what happens in a problem solving environment based on a blackboard architecture.

Marvel relies upon a special rule-based language derived from AI production systems (Kaiser *et al.*, 1993). A software process is specified by three sets of specifications: the project rule set describes the development process; the project type set specifies the project data; and the project tool set defines the interface with external tools.

Example

This is an example of a Marvel rule (Barghouti & Kaiser, 1992):

```
compile[?f:CFILE]:
:
  (?f.compile_status=NotCompiled)
{COMPILER compile ?f.contents
   ?f.object_code ?f.error_msg '-g'}
(and (?f.compile_status=Compiled)
(?f.object_time_stamp=CurrentTime));
(?f.compile_status=Error);
```

Marvel rules have a name and include three sections: a precondition, an activity part, and a postcondition. The above rule has name `compile` and a parameter which is a file containing C source code. The rule precondition is satisfied if such a file has not yet been compiled. The precondition is a condition to be evaluated with respect to the current status of the project database; if it is verified, the activity part is executed, which usually invokes a tool. In the above rule, the tool is the C compiler. At the end of the tool activity, the post-condition specifies the effects of the rule on the project database. In the example, the effects are that the file if flagged "Compiled" obtains a new time stamp, and its compile status is defined by the compiler errors report.

Both the REBUS and Marvel experiences show that a software process can be easily defined and controlled by rules coordinating agents which share a project database containing the documents produced by the process itself.

4 Coordinating a software process

Software processes include several activities running in parallel, so it is natural to model them using parallel languages. Our choice was to investigate the suitability of concurrent logic languages for such a task.

4.1 Using a stream-based parallel logic language

In fact, I and some colleagues developed and animated a software process program in Flat Concurrent Prolog (FCP) (Ambriola *et al.*, 1995). Such a program specifies a simple multi-user

environment in which coordination is achieved by a check-in/check-out protocol for reserving documents stored in a central project database.

Using FCP, the evolution of the environment is abstractly depicted by graphs whose nodes are activities (logic processes) and edges are channels (logic streams). Such an evolution is controlled by FCP clauses whose atoms are the logic processes and the streams are shared variables.

The environment we modelled and tested enforced a simple software process coordinating the activity of several programmers using a shared project database. This experiment showed that a non-sequential language like FCP can be used as a software process programming language to model coordinated activities; however, it lacks some important coordination abilities. For instance, since FCP is clearly geared toward fine-grained concurrency, only dataflow, one-to-many communication is easy to specify. Moreover, since no sequential language component is embedded in the language, linear sequence of atomic activities are not easy to control and coordinate.

4.2 ESP and Oikos

To overcome the shortfalls of stream-based logic languages I designed and developed with A. Brogi a new language, namely Shared Prolog (SP). SP combined the coordination mechanisms offered by blackboards with the declarative and inferential capabilities of Prolog. Initially, only one blackboard was allowed; then to the language were added multiple tuple spaces, and such an extension was called Extended Shared Prolog (ESP) (Ciancarini, 1993).

In ESP a software system is modelled as a set of dataspaces; each dataspace is both a repository of facts in form of logic tuples, and a communication channel (blackboard) shared among agents that can read or write tuples in the dataspace itself. ESP is an extension of Prolog insofar as it includes some primitives to manipulate the contents of a dataspace. In fact, agents execute coordination rules that can include normal Prolog goals. ESP has been used as a specification language for reasoning on a process model: Chen and Montangero (1995) show how to use it to refine very abstract process models into coordination programs enacting some specific cooperation protocol.

In my experience, the combination of the blackboard model with logic programming offered by ESP provides the process specifier with a powerful, rule-based, concurrent framework to specify and prototype distributed software development environments and the related software processes.

As a simple example, we here show how ESP can be used to model a software process which takes place inside a distributed software development environment. The idea is that the software process is explicitly defined by rules which form coordination protocols; these govern the activities inside a distributed environment. The rules specify goals, duties and constraints that the agents involved in the software process have to fulfil. It is the environment which imposes constraints and supports coordination protocols among the users, and that controlling the environment means governing the software process.

The concepts of distributed development environment and rule-based software process as well as their interplay are easily implemented in ESP. In fact, we show who ESP can be used to design simple programming environments corresponding to simple software processes.

Example

A software process designer can easily model part of the software process described in Kellner *et al.* (1991) using ESP. The first step consists of creating three blackboards: one for the change control board CCB, one for the project manager PM, and one for the programmers' team PT. The following goal sets up the initial environment in which the process takes place.

```
?-{tsc(bb_of_CCB),tsc(bb_of_PM{change_and_test(example-unit)}),tsc(bb_of_PT)}
```

Such a goal creates (tsc means "tuple space create") three blackboards. Blackboards called `bb_of_CCB` and `bb_of_PT` are empty, whereas `bb_of_PM` contains a tuple `change_and_` test(example-unit).

For blackboard bb_of_PT, we specify the termination conditions using the following invariant rule:

```
?-{invariant(end_work(Id)@bb_of_PT),invariant(abort_work(Id)@bb_of_PT)}
```

This means that blackboard bb_of_PT will terminate its activities when either tuple end_work(Id) or tuple abort_work(Id) is found.

The tuple initially put in the project manager (PM) blackboard corresponds to an agent executing the following ESP theory, that controls the overall process from the viewpoint of PM:

```
theory develop_change_and_test(Unit):-
eval
  change(requirements(Unit),Id),{authorization(Id)}
  →
  {schedule_and_assign_tasks(Unit),start_work(Unit,Id)@bb_of_PT}
#
  {success(Id)}
  →
  {end_work(Id)@bb_of_PT}
#
  {cancel_change(Id)}
  →
  {abort_work(Id)@bb_of_PT}
```

The theory develop_change_and_test(Unit) includes three rules: the first one describes activation of the process that starts when the CCB issues the authorisation tuple and the specification of the changes for the target unit. The project manager responds to this "stimulus" by starting the schedule_and_assign_tasks phase, which is defined by another ESP theory. The other two rules state the possible terminal conditions of the process.

The theory schedule_and_assign_tasks include one rule only: it states that project plans should be updated and new assignments should be sent by e-mail:

```
theory schedule_and_assign_tasks(Unit):-
eval
  {change(requirements(Unit),Id),project_plans(Plans)}
  →
  update(Plans,NewPlans)
  {project_plans(NewPlans),mail(Assignments)@bb_of_PT,
  change(requirements(Unit),Id)@bb_of_PT}
with
  update(Plans,NewPlans):-... % new scheduling of tasks.
```

A coordination language that offers a rule-based programming style, like ESP, becomes more effective and useful if supported by a fully fledged environment, because a software process program can immediately be instantiated to obtain a process-centred environment. This was a key aspect in the design and development of the Oikos meta-environment (Ambriola *et al.*, 1990; Ambriola & Montangero, 1992b; Montangero & Scarselli, 1994). Oikos is a rule-based meta-environment that supports distributed software processes. It offers a number of standard services giving some basic facilities, like access to databases and private workspaces, remote activation of shells, etc. An example of the use of Oikos for enacting a real software process is found in Ambriola & Montangero (1992a).

4.3 Current work

Since 1993 a new technology has become more and more supportive of groupware environments:

the protocols and tools supporting the World Wide Web. I am currently engaged in using this new software platform as software process monitoring and enaction environment.

In fact, the collection of documents produced during a software development process can be managed as a (versioned) hypertext: this idea was exploited for instance in the Matisse project (Garg *et al.*, 1994). Matisse models process documents, roles and tasks as entities in an information system; relationships among entities can be searched via information retrieval and navigational tools.

I am exploring a similar idea using directly the World Wide Web as a software platform for organising and monitoring process activities. Here at the University of Bologna I and my students are integrating a new coordination language that is a descendant of ESP, called Shade (Castellani *et al.*, 1996), with Web-oriented services so that we can use the coordination language to manage process activities and cooperation, whereas some Web browsers (e.g., Mosaic or Netscape) are used for monitoring and process status.

5 Conclusions

Each activity in a software process can usually be described in terms of input-output behaviour (preconditions and postconditions) and decomposition in subactivities. This explains why the use of rule-based formalisms for software process programming is widely investigated. Moreover, there are clear analogies between rule-based descriptions and graphic descriptions in formal languages like Petri nets, another popular choice for process modelling. For an example of the use of Petri nets in software process modelling, see for instance the SPADE environment and its SLANG language (Bandinelli *et al.*, 1993). Logic programming is at the basis of an evolution of the SLANG language, called LATIN, used in the SENTINEL environment to control process evolution and dynamic changes (Cugola *et al.*, 1995).

However, the use a sequential rule-based language like Prolog for animating a process model is not fully satisfactory. There are the same problems that one meets in using Prolog as a formal specification language, namely the lack of expressivity for coordination features like synchronisation and communication. Moreover, it is difficult to use a process program written in Prolog as a basis for designing and implementing a process-centred distributed environment. In fact, in the MERLIN project Prolog has been augmented with special, extralogic operations suitable for describing process activities.

The use of the ESP coordination language in Oikos is a plus with respect to MERLIN, where sequential Prolog is used, because it is easy to design and implement distributed coordination protocols. Marvel and Oikos are quite similar, yet a logic language seems to us simpler and more expressive for process modelling.

The use of rule-based languages for process modelling is satisfactory for describing at a high level of abstraction the causal dependencies among documents and activities typical of software processes. However, process modelling needs much more, especially because such an activity is connected with the design of process-centred development environments. The evolution of the technology of distributed development environments, which is currently under way, based on the World Wide Web used as a groupware platform, needs the integration of rule-based process languages with coordination technology. It is still unclear which software architectures are better suited to support such an integration.

Acknowledgements

This work has been partially funded by the ESPRIT BRA Project 9102-Coordination, and by the Italian Ministry of University (MURST 40%—Project on Concurrent Software Engineering).

References

Ambriola, V, Ciancarini, P and Corradini, A, 1995. "Declarative specifications of the architecture of a software development environment" *Software: Practice and Experience* **25**(2) 143–174.

Ambriola, V, Ciancarini, P and Montangero, C, 1990. "Enacting software processes in Oikos" *Proceedings ACM SIGSOFT Conference on Software Development Environments. (ACM SIGSOFT Software Engineering Notes,* **15**(6) 12–23.)

Ambriola, V and Montangero, C, 1992a. "Modeling the software development process", In: V Ambriola and G Tortora (eds.), *Advances in Software Engineering and Knowledge Engineering,* 41–72, World Scientific.

Ambriola, V and Montangero, C, 1992b. "Oikos at the age of three", In: J Derniame (ed.), *Proceedings 2nd European Workshop on the Software Process Technology: Lecture Notes in Computer Science 635,* 84–93. Trondheim, Norway. Springer-Verlag.

Armenise, P, Bandinelli, S, Ghezzi, C and Morzenti, A, 1993. "A survey on assessment of software process representation formalisms" *International Journal on Software Engineering and Knowledge Engineering* **3**(3) 401–426.

Bandinelli, S, Fuggetta, A and Ghezzi, C, 1993. "Software process model evolution in the SPADE environment" *IEEE Transactions on Software Engineering* **19**(12) 1128–1144.

Barghouti, N, 1992. "Supporting cooperation in the Marvel process centered SDE" *Proceedings 5th ACM SIGSOFT Conference on Software Development Environments. (ACM SIGSOFT Software Engineering Notes* **17**(5) 21–31.)

Barghouti, N and Kaiser, G, 1990. "Modeling concurrency in rule-based development environments" *IEEE Expert* **5**(6) 15–27.

Barghouti, N and Kaiser, G, 1992. "Scaling up rule-based software development environments" *International Journal on Software Engineering and Knowledge Engineering* **2**(1) 59–78.

Ben-Shaul, I, Kaiser, G and Heineman, G, 1992. "An architecture for multi-user software development environments" *Proceedings 5th ACM SIGSOFT Conference on Software Development Environments. (ACM SIGSOFT Software Engineering Notes* **17**(5) 149–158.)

Castellani, S, Ciancarini, P and Rossi, D, 1996. "The ShaPE of ShaDE: a coordination system" Technical Report UBLCS 96-5, Dipartimento di Scienze dell'Informazione, Università di Bologna, Italy.

Chen, X and Montangero, C, 1995. "Compositional refinements in multiple blackboard systems" *Acta Informatica* **32**(5) 415–458.

Christie, A, 1995. *Software Process Automation,* Springer-Verlag.

Ciancarini, P, 1993. "Coordinating rule-based software processes with ESP" *ACM Transactions on Software Engineering and Methodology* **2**(3) 203–227.

Ciancarini, P and Levi, G, 1992. "What is logic programming good for in software engineering?" In: V Ambriola and G Tortora (eds.), *Advances in Software Engineering and Knowledge Engineering* 109–134. World Scientific.

Conradi, R, Jaccheri, M, Mazzi, C, Aarsten, A and Nguyen, N, 1992. "Design, use and implementation of SPELL, a language for software process modeling and evolution" *Proceedings European Workshop on the Software Process: Lecture Notes in Computer Science 635,* 167–177. Trondheim, Norway. Springer-Verlag.

Cugola, G, Nitto ED, Ghezzi, C and Mantione, M, 1995. "How to deal with deviations during process model enactment" *Proceedings 17th International Conference on Software Engineering (ICSE 17)* Seattle, WA.

Curtis, B, Kellner, M and Over, J, 1992. "Process modeling" *Communications of the ACM* **35**(9) 75–90.

Garg, P, Pham, T, Beach, B, Deshpande, A, Ishizaki, A, Wentzel, K and Fong, W, 1994. "Matisse: a knowledge-based team programming environment" *International Journal on Software Engineering and Knowledge Engineering* **4**(1) 17–59.

Heimbigner, D, 1989. "P4: a logic language for process programming" *Proceedings 5th International Software Process Workshop* Kennebunkport, Maine.

Huff, K and Lesser, V, 1988. "A plan-based intelligent assistant that supports the software development process" *Proceedings 3rd ACM SIGSOFT Symposium on Software Development Environments (ACM SIGSOFT Software Engineering Notes* **13**(5) 97–106.)

Jaccheri, M and Conradi, R, 1993. "Techniques for process model evolution in EPOS" *IEEE Transactions on Software Engineering* **19**(12) 1145–1156.

Kaiser, G, Popovich, S, and Ben-Shaul, I, 1993. "A bi-level language for software process modeling" *Proceedings 15th International Conference on Software Engineering* 132–143, Baltimore, MD.

Katayama, T, 1989. "A hierarchical and functional software process description and its enaction" *Proceedings 11th IEEE International Conference on Software Engineering* 343–252, Pittsburgh, PA.

Kellner, M, Feiler, P, Finkelstein, A, Katayama, T, Osterweil, L, Penedo, M and Rombach, D, 1991. "ISPW-6 software process example" In: M Dowson, (ed.), *Proceedings 1st International Conference on the Software Process* 176–186, Redondo Beach, CA.

Kishida, K *et al.*, 1988. "A novel approach to software environment design and construction" *Proceedings 10th International Conference on Software Engineering* 69–79, Singapore.

Kraut, R and Streeter, L, 1995. "Coordination in software development" *Communications of the ACM* **38**(3) 69–81.

Liu, L and Horowitz, E, 1989. "A formal model for software project management" *IEEE Transactions on Software Engineering* **15**(10) 1280–1293.

Mi, P and Scacchi, W, 1990. "A knowledge-based environment for modeling and simulating software engineering processes" *IEEE Transactions on Knowledge and Data Engineering* **2**(3) 283–294.

Minsky, N and Rozenshtein, D, 1990. "Configuration management by consensus: an application of law-governed systems" In: R Taylor, (ed.), *Proceedings 4th ACM SIGSOFT Symposium on Software Development Environments. (ACM SIGSOFT Software Engineering Notes* **15**(6) 44–55.)

Montangero, C and Scarselli, F, 1994. "Software process monitoring mechanisms in Oikos" *International Journal on Software Engineering and Knowledge Engineering* **4**(4) 481–499.

Nguyen, M and Conradi, R, 1994. "Spell: a logic programming language for process modelling" In: P Ciancarini and L Sterling, (eds.), *Proceedings ICLP Workshop on Logic Programming and Software Engineering* S. Margherita Ligure, Italy.

Nii, H, 1986. "Blackboard systems: the blackboard model of problem solving and the evolution of blackboard architecture" *The AI Magazine* 38–106, Summer.

Ohki, A and Ochimizu, K. 1989. "Process programming with Prolog" *Proceedings ACM Software Process Workshop. (ACM SIGSOFT Software Engineering Notes* **14**(4) 118–121.)

Osterweil, L, 1987. "Software processes are software too" *Proceedings 9th IEEE International Conference on Software Engineering* 2–13.

Peuschel, B, Schaefer, W and Wolf, S, 1992. "A knowledge-based software development environment supporting cooperative work" *International Journal on Software Engineering and Knowledge Engineering* **2**(1) 79–106.

Song, X and Osterweil, L, 1994. "Experience with an approach to comparing software design methodologies" *IEEE Transactions on Software Engineering* **20**(5) 364–384.

Sutton, S, Ziv, H, Heimbigner, D, Yessayan, H, Maybee, M, Osterweil, L and Song, X, 1991. "Programming a software requirements specification process" *Proceedings 1st International Conference on the Software Process*, Redondo Beach, CA.

Welzel, D, 1993. "A rule-based process representation technique for software process evaluation" *Information and Software Technology* **35**(10) 603–610.