Software Process Model Evolution in the SPADE Environment

Sergio C. Bandinelli, Alfonso Fuggetta, *Member, IEEE*, and Carlo Ghezzi, *Member, IEEE*

***Abstract*— Software processes are long-lived entities. Careful design and thorough validation of software process models are necessary to ensure the quality of the process. They do not prevent, however, process models from undergoing change. Change requests may occur in the context of reuse, i.e., statically, in order to support software process model customization. They can also occur dynamically, while software process models are being executed, in order to support timely reaction as data are gathered from the field during process enactment. In this paper, we discuss the mechanisms a process language should possess in order to support changes. We illustrate the solution adopted in the context of the SPADE environment and discuss how the proposed mechanisms can be used to model different policies for changing a software process model.**

***Index Terms*— Object-oriented databases, object-oriented languages, Petri nets, process-centered software engineering environments (PSEE's), process enactment, process evolution, process modeling, reflective languages, software process.**

I. Introduction

RECENTLY, it has been pointed out that software processes[1] need to be analyzed and carefully modeled in order to support and facilitate their understanding, assessment, and automation (see for example [1], [2]). To address these issues, several research efforts have been undertaken both in industry and in academia. The ultimate goal of such efforts is to provide innovative means to increase the quality of software development processes and, consequently, the quality of the applications delivered to final users. The results of these activities are several prototype languages and experimental environments, called process-centered software engineering environments (PSEE's), which provide specific features to create, analyze, and execute software process models [3], [4].

In order to effectively support the goals of modeling, analysis, and automation, an ideal *software process language* must exhibit several characteristics:

- It must be formal, so that process models may be automatically analyzed and executed (enacted).
- It must allow the process modeler to describe both the activities that constitute the process, and the results produced during its execution (i.e., the process artifacts).
- Software processes are human-oriented systems, i.e., systems in which humans and computerized tools cooperate in order to achieve a common goal. A process formalism must provide means to describe such interaction, by clearly defining, for instance, when and how a task is assigned to a tool or a human, and how to coordinate the operations of different human agents.
- The target architecture for process enactment must be a heterogeneous, concurrent/distributed environment, supporting cooperation and interaction of several human agents. A process language should provide mechanisms to support process model execution on such architectures.
- Software process models are often very large and it is therefore mandatory to enrich a process formalism with effective constructs supporting modeling-in-the-large concepts, such as abstraction, information hiding, and reuse of process and model fragments.
- The process language should support various kinds of analysis. Extensive static analysis could be performed before enactment and after any major change to verify certain properties (e.g., reachability of desirable states within a given time bound). Dynamic analysis through simulation would provide a complementary assessment of a process model by—say—generating sample behaviors and checking whether they match the expected behaviors.
- The language should support process evolution. Software process models are dynamic entities, that must evolve in order to cope with changes in the enacting process, due to changing requirements or unforeseen circumstances, in the software development organization, in the market, in the technologies, and in the methodologies used to produce software.

In recent years, several process languages have been proposed, but still no solution effectively copes with all the aforementioned issues. In particular, the issue of process model evolution represents one of the most challenging problems faced by the software process community.

During its lifetime a software process model undergoes changes that can be caused by a variety of reasons and needs [5]. We mention here three significant categories of changes:

1) It is usually impossible to define the entire software process model from the very beginning, i.e., before the actual software development activity starts. Several details of the process might (or have to) be left initially unspecified. They can be detailed later when additional or more precise information on the process have been

Manuscript received November 1, 1992; revised August 1, 1993. Recommended by N. M. Madhavji and M. H. Penedo. This work was supported in part by the Italian National Research Council (CNR). S. Bandinelli was supported in part by Digital Equipment Corporation.

The authors are with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piza Leonardo da Vinci 32, 20133 Milano, Italy.

IEEE Log Number 9213498.

[1] We will use the expressions "process" and "process model" to respectively mean "software process" and "software process model."

collected and analyzed. For example, testing procedures can be designed only after the quality requirements for the application to be developed have been specified. One may view this *incremental definition* as a kind of change which adds new parts to an existing process model. The newly added parts should not interfere with the existing model; otherwise, it may be necessary to change the old part as well.

2) Software processes are long-lived entities that are carried out within highly dynamic environments. A software development organization may change in response to *changes in the environment* in which it operates, or *changes in the organization* at the corporate level. Such changes may be caused, for example, by poor performance, by new tools acquired by the company to support its software development staff, changes in the marketing strategy or in customers' expectations and requirements. Thus, an existing software process model has to be modified or extended to reflect the evolution of the environment and/or internal changes.
3) It must be possible *to customize a software process model* in order to allow process agents (i.e., the humans operating the process) to dynamically select the most effective solution for a given problem. For instance, a design activity can be conducted bottom-up or top-down, depending on the characteristics of the software system to be designed, and the software engineer's judgement.

There is a large spectrum of policies that can be adopted to apply changes to a software process model. At one extreme, under a purely static policy, changes are applied only to the specification of the software process, without affecting ongoing development activities. The effects of a purely static modification become visible only when a new process that follows the modified model is launched. At the other extreme, under a fully dynamic policy and modification to the specification of a software process is immediately propagated to ongoing activities. Between these two approaches a variety of policies exists, and it must be possible to model and instrument them within the process support environment.

In order to model evolution, a software process can be described as the interaction of two (sub)processes: the *software development process* and the *software meta-process.* The former includes all the activities, roles, procedures, rules, tools, and information related to the development of a software product. The latter is in charge of maintaining and evolving software processes; i.e., the data it manipulates are process models and process states. The meta-process is a process too and should be designed, implemented, and evolved as any other process. This might be done by a meta-meta-process. In general, we might create several levels of meta-processes as needed. One possible solution would be to keep all these levels separate, and to define a separate model for each level. This solution, however, freezes the number of levels, and does not allow changes to the software process to be specified as resulting from observing both the development process and the meta-process itself. On the other hand, some process metrics can be collected only by jointly observing both processes. The solution we adopted, discussed below, does not freeze the boundaries between levels and uses the same language to define both process and meta-process features.

The problem of process and process model evolution has been given different solutions by existing systems. It is possible to identify the following main approaches.

1) The meta-process is not modeled as part of the software process. The process formalism does not provide specific features to manipulate the model during its execution. The meta-process is thus described separately, with no formal relationship between the two models, or it is not formalized at all. This seems to be the approach used by imperative languages, whose best-known example is APPL/A [6].
2) The meta-process is modeled as part of the software process. To do so, the process formalism is augmented with specific, ad hoc features that allow only partial modifications of the software process model. This approach is used, for example, by FUNSOFT nets [7], which allow the topology of a Petri net describing an activity to be changed. Another example is HFSP [8], where it is possible to specify alternative sequences of operations to be selected according to the state of the process. In these approaches, the possible evolutions of a process model have to be in some way *anticipated* and (at least partially) hard-coded in the initial specification of the process model itself. Thus, the degree of flexibility offered by these approaches is still limited, since it is not possible to anticipate from the very beginning all of the future changes that might occur in the process model.
3) Finally, to support the modeling of the meta-process as part of the software process, several process formalisms are based on a fully *reflective language.* In this scheme, a process model can be accessed both as code to be executed and as datum to be modified. This is the approach used, for example, by EPOS [9] and IPSE 2.5 [10].

In the SPADE project,[2] we have defined a language, called SLANG (SPADE LANGuage), whose goal is to address all of the linguistic issues discussed so far. In particular, it provides execution mechanisms to cope with the evolution problem. SLANG is a fully reflective language built over a high-level extension of Petri nets.[3] Its main reflective features can be summarized as follows.

- A SLANG process model is partitioned in different modules (called *activities*) that can be executed in parallel by different SLANG interpreters called *process engines.* Process engines are dynamically created during enactment.
- Activity definitions and activity states are tokens of the net, and can be manipulated by transitions as any other token.
- SLANG provides dynamic type-checking and late binding mechanisms.

[2] SPADE stands for software process analysis, design, and enactment.

[3] In this paper, we assume that the reader knows the basic concepts of Petri nets. Otherwise, the reader may refer to [11]–[13].

A first prototype of SPADE has been implemented and is currently under assessment. The prototype uses an object-oriented database to store a process model and software artifacts. Meanwhile, SLANG has been used to model several real-life examples, including the process used by a large telecommunication company to maintain defense software.

The paper is organized as follows. Section II introduces SLANG and summarizes its basic features using a simple example process. Section III explains the basic mechanisms supporting the enaction of a process model and the reflective features of SLANG. Section IV discusses how these features can be used to support process evolution, and provides some hints and examples on how to describe a meta-process as part of the process model. Related work is surveyed in Section V. Finally, Section VI draws some conclusions and outlines future work.

II. Process Modeling in SLANG

SLANG [14]–[16] is a language for software process modeling and enactment. It is based on high-level Petri nets, and is given formal semantics in terms of a translation scheme from SLANG objects into ER nets. ER nets [17], in turn, are a mathematically defined class of high-level Petri nets that provide the designer with powerful means to describe concurrent and real-time systems. In ER nets, it is possible to assign values to tokens, and associate transitions with relations that describe the constraints on tokens consumed and produced by transition firings. The Appendix contains a brief introduction to ER nets.

SLANG offers features for software process modeling, enaction, and evolution that will be presented in detail in the following sections. In addition, SLANG may interact with external tools and humans in a uniform manner. The language provides constructs to deal with time in process models. It is possible to specify time constraints on the occurrence of an event or impose a maximum duration between the occurrence of two events by defining a timeout. It then supports ways to formally reason about the temporal evolution of a process. In this paper, we illustrate SLANG informally through examples; for simplicity, we also ignore time issues. A detailed overview of the language can be found in [14]. A complete formal definition of the language is in preparation.

In Section II-A we discuss a process example that will be used throughout the paper to illustrate SLANG. Section II-B provides an overview of SLANG main features and shows how a SLANG process model may be hierarchically structured, using the high-level *activity* construct.

A. A Running Example

The example is centered on a hypothetical quality assurance activity; in particular, it deals with testing a collection of modules. Each single module of the product is treated separately. Integration test is done afterwards and is not part of the example. The process, based on a white-box testing strategy, may start as soon as the source code of the module is provided. The source code must be compiled with the corresponding libraries, which include the stubs, drivers, and other necessary information for performing the test. If errors are found during compilation no testing is performed.

Once compilation terminates successfully and the test cases are generated, the compiled module is run on the test cases, one after the other. The errors that may be discovered in each execution are accumulated in an error report. The test continues until all test cases have been run on the module. If no error is found during test, the module is accepted. Otherwise the module is rejected and a report is produced with a list of all the failures occurred duringtest.

B. Overview of SLANG Features

A SLANG process model is a pair of sets:

$$SLANGModel = (ProcessTypes, ProcessActivities)$$

ProcessTypes is a set of type definitions organized in a hierarchy, according to an object-oriented style. *ProcessActivities* is a set of activity definitions. An activity definition is basically a high-level Petri net where each place, arc, and transition has been augmented with additional information, as specified below.

Process Types: A large amount of data is produced, used, and manipulated in a software process, from specification documents to executable code and test data. In SLANG, all process data are typed, and types are defined in an object-oriented style. Type definitions are organized in a hierarchy, defined by an *is_a* (or *subtype*) relationship. A subtype T_s of type T is any of the direct or indirect descendants of T in the *is_a* hierarchy, including T. The root of the hierarchy is the type `ProcessData`. Types inherit the attributes and operations of the ancestors. SLANG supports multiple inheritance.

Activities are defined in SLANG using a Petri net based language. Process data are therefore represented as tokens of the net. In order to be able to define SLANG activities supporting process model evolution, activity definitions, type definitions for process data, and process states must be accessible as any other data, i.e., as tokens of the net. Four predefined types (`Token`, `Place`, `Arc`, and `Transition`) are thus provided (they are subtypes of `ProcessData`). Three predefined subtypes of `Token` are also introduced: `Activity`, `Metatype`, and `ActiveCopyType`. `Activity` is the type whose instances are activity definitions. `Metatype` is the type whose instances are type definitions. `ActiveCopyType` is the type whose instances are enacted instances of activity definitions, as explained in Section III. These types are part of the SLANG definition: they are not process dependent and cannot be modified by process modelers. Fig. 1 shows this built-in portion of the SLANG type hierarchy. Fig. 2 contains some of the corresponding type definitions.

Type definitions for process data, which characterize the particular process being modeled, are represented by a subtree rooted at `ModelType` (a subtype of `Token`).

Example II.1: Type definitions: Fig. 3 shows the *is_a* type hierarchy for some of the types of our running example; the corresponding textual type definitions are given in Fig. 4. Type `NamedModule` defines a general module, with a

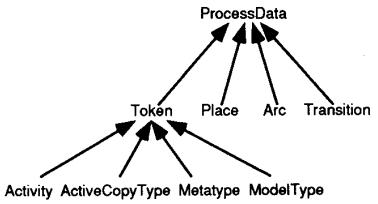


Fig. 1. A part of the ProcessTypes hiearchy corresponding to SLANG built-in types.

```
Activity subtype of Token;
     ActivityName: string;
     Places: set of Place;
     Transitions: set of Transitions;
     Arcs: set of Arcs;
     ...

Metatype subtype of Token;
     TypeName: string;
     TypeDescriptor: Descriptor;
     ...

ActiveCopyType subtype of Token;
     Act: Activity;
     Typeset: set of Metatype;
     State: Marking;
     Caller: ActiveCopyType;
     Start/EndEvent: Transition;
     Input/OutputTuple: TokenTuple;
     ...
```

Fig. 2. Some built-in SLANG types.

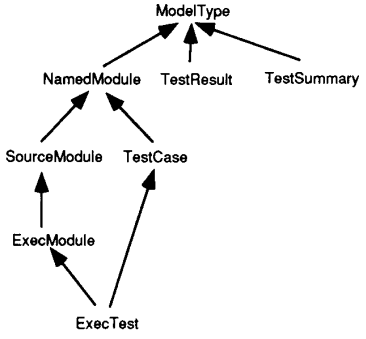


Fig. 3. User-defined type hierarchy for the running example.

name and an author. `SourceModule` represents a module of source code and inherits from `NamedModule`. `ExecModule` defines executable modules (i.e., modules that have been compiled and linked with drivers and stubs) inheriting from `SourceModule`. `TestCase` is a module that contains a test case for a module. The information needed to apply a test case on a module is described by the type definition `ExecTest`. Each test result is described by `TestResult` and the complete result of a series of tests is described by `TestSummary`. Type definitions may also include operations on the defined data (in this example operations have been omitted for simplicity). □

Process Activities: *ProcessActivities* is a set of activity definitions. Each activity definition is an instance of type `Activity` and is specified as a high-level Petri net, i.e., as a set of places P, a set of transitions T, and a set of arcs A. The

```
NamedModule subtype of ModelType;
     Name: string;
     Author: Person;

SourceModule subtype of NamedModule;
     SourceCode: Text;

ExecModule subtype of SourceModule;
     Compiler: string;
     ExecCode: binary;

TestCase subtype of NamedModule;
     TestData: text;

ExecTest subtype of ExecModule, TestCase;

TestResult subtype of ModelType;
     UsedTest: TestCase;
     Failures: list of Failure;

TestSummary subtype of ModelType;
     TestingModule: ExecModule;
     Results: set of TestResult;
     #PendingTests: integer;
```

Fig. 4. Type definitions of some of the types of the running example.

activity state is given by the net marking, i.e., by an assignment of tokens to places. Events are represented by transitions. The occurrence of an event (represented by the firing of the corresponding transition) modifies the activity state by removing tokens from the input places of the transition and adding tokens to the output places. The net topology describes precedence relations, conflicts, and parallelism among events and activities. Each activity corresponds to a logical work unit, that may include local events, invocation of other activities, and interaction with the environment (tools and humans). According to the principles of information hiding, an activity definition has an *interface* and an *implementation* part. The activity interacts with the rest of the process model through its interface; the implementation part remains hidden. The interface is composed of

- a set of interface places $P_{int} \subset P$,
- a set of interface transitions $T_{int} \subset T$,
- a set of interface arcs $A_{int} \subset A$, connecting interface places and interface transitions.

Events that initiate and terminate the execution of an activity are represented by interface transitions. Interface transitions are partitioned in two sets: SE (called *starting events*) and EE (called *ending events*), such that $SE \cap EE = \emptyset$, and $SE \cup EE = T_{int}$. The activity is said to start (terminate) as soon as any transition in SE (EE) fires. Interface places in P_{int} are classified into three sets.

- *Input places:* They are input places for any of the starting events.
- *Output places:* They are output places for any of the ending events.
- *Shared places:* They are shared by different activities and can be input or output places for any transition in the implementation part.

Interface places play the role of formal parameters of an activity. Any other place internal to the implementation part (not belonging to the interface) is said to be a *local place.* The implementation part details how the activity is performed by

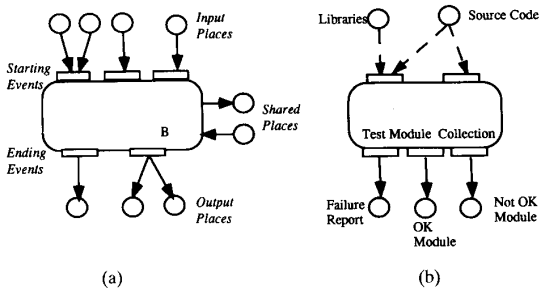


Fig. 5. (a) Graphical representation of activity interface. (b) Interface of activity `Test Module Collection`.

showing the relationships among the events that may occur during its execution. An activity interface is graphically represented by a box surrounded by interface transitions, places, and arcs, as shown in Fig. 5(a). For example, Fig. 5(b) describes the interface of activity `Test Module Collection`, which is invoked by a larger Quality Assurance activity. The implementation of `Test Module Collection`, shown in Fig. 6, contains invocations to other activities (`Compile For Test`, `Generate Test Cases`, and `Run Tests`). Activity invocations are graphically represented according to the same syntax as for activity interfaces. An activity invocation must match the corresponding definitions in terms of interface places, transitions, and arcs. For simplicity, in this paper we assume that the binding is specified by using the same names for (actual) place and transition names of activity invocation and (formal) place and transition names of activity interfaces. The precise semantics of the invocation is detailed later.

Example II.2: Activity definitions: Fig. 7(a) shows the interface of activity `Run Tests`, which is invoked by activity `Test Module Collection`. `Run Tests` has a single starting transition and a single ending transition, respectively, called `Start Test` and `End Test`. It has two input places (`Executable Module` and `Ready Test Cases`), one output place (`Final Test Results`) and no shared places. In Fig. 7(b), we provide a possible implementation for activity `Run Tests`. The precise meaning of all the elements presented in Figs. 6 and 7 is explained later. Intuitively, Fig. 7(b) shows that an executable module is repeatedly executed, each time on a test case. The results of each execution are compared to the expected results and the outcome of the comparison is added to the cumulative test results. When all tests have been executed, the `Run Tests` activity terminates. Note that the implementation of `Run Tests` is entirely defined in terms of primitive events, without involving other activities. □

Activity Static DAG: An activity implementation may contain *invocations* of other activities. It is thus possible to define the relation *uses* among activity definitions, such that A *uses* B iff A's implementation contains an invocation of B. The *uses* relationship may be represented by a graph, where nodes represent activities. There is a directed arc from A to B iff A uses B. Recursive invocations are not allowed in SLANG, i.e., the relation is a hierarchy, and thus the graph is a DAG (direct acyclic graph). We require *uses* to be a hierarchy,

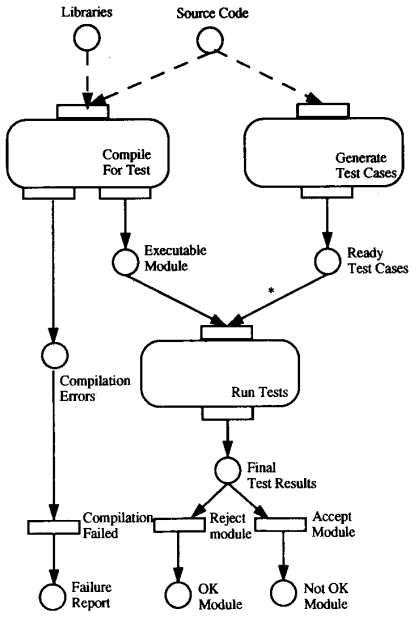


Fig. 6. Implementation of activity `Test Module Collection`.

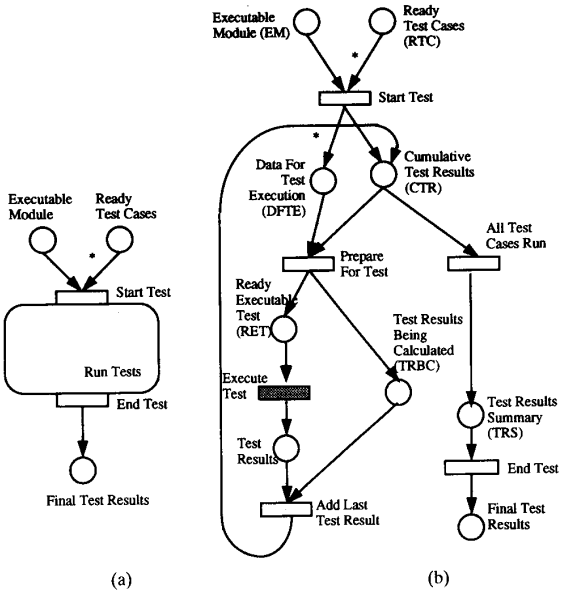


Fig. 7. (a) Interface of activity `Run Tests`. (b) Implementation of activity `Run Tests`.

according to the principles discussed by [18]. This assumption also simplifies the language and its implementation. The graph representing the uses relationship does not change as long as the process model is not changed. For this reason we call it *activity static DAG* (ASD). In a SLANG process model, there is one "main" activity, not invoked by other activities, that is at the top of the ASD. This activity is called *root activity* and is spawned by a specialized boot program to start process enactment.

Places and Tokens: Each place has a name and a type. A place behaves as a token repository that may only contain tokens of its type or of any subtypes. Places may change their contents through transition firings. The only exception to this rule are *user interface* places, which are a special kind of places that may change their contents as a consequence of human intervention. They are used to transfer external events caused by humans within the system. User interface places are graphically represented by a double circle.

The type of a place must be one of the subtypes of type `Token` (see Fig. 1), contained in the *ProcessTypes* set. In particular, places can be of type `Activity` (i.e., they may contain tokens whose value is an activity definition), `Metatype` (i.e., they may contain tokens whose value is a type definition), and `ActiveCopyType` (i.e., they may contain tokens whose value is an enacted instance of an activity definition). Consequently, activity definitions, type definitions, and process states can be created and manipulated as any other value. An in depth discussion of the implications of having process models as values is delayed to Sections III and IV, where we also discuss how activity states are manipulable.

Transitions: Transitions represent events whose occurrence takes a negligible amount of time.[4] Each transition is associated with a guard and an action. The guard is a predicate on tokens belonging to the transition input places and is used to decide whether a tuple of input tokens enables the transition (an input tuple satisfying a guard is called an enabling tuple). The dynamic behavior of a transition is described by the *firing rule.* The firing rule states that when a transition fires, tokens satisfying the guard are removed from input places and the transition action is executed. As a result of executing the action, an output tuple is inserted in the output places of the fired transition.[5]

A software development process involves the activation of a large variety of software tools. Tool invocation is modeled in SLANG by using *black transitions.* A black transition is a special transition where the action part is the invocation of a non-SLANG executable routine (e.g., a Unix executable file). When the black transition "fires," the routine is executed *asynchronously.* This means that other transitions may be fired while the black transition is still being executed. It is also possible to fire the black transition itself many times with different input tuples, without waiting for each activation to complete.

Arcs: Arcs are weighted (the default weight is 1). The weight indicates the number of tokens which flow through the arc at each transition firing. It can be a statically defined number or it may be dynamically computed (this is indicated by a "*"). In the latter case, the arc weight is known only at run-time and may vary at each firing occurrence. This is useful to model events requiring, for example, *all tokens* that verify a certain property.

Besides "normal" arcs, SLANG provides two other special kinds of arcs: *read-only* (represented by a dashed line) and *overwrite* (represented by a double headed arrow). Read-only and overwrite arcs are shorthand notational devices that do not add to the semantic power of the language, but improve its usability. A read-only arc may be used to connect a place to a transition. The transition can read token values from the input place in order to evaluate the guard and the action, but no token is actually removed. An overwrite arc may be used to connect a transition to a place. When the transition fires, the following atomic sequence of actions occurs. First the output place is emptied of all its tokens. Then, the tokens produced by the firing are inserted in the output place. The overall effect is that the produced tokens *overwrite* any previous content of the output place.

A bidirectional arc between a place and a transition stands for a pair of arcs: one from the place to the transition and one from the transition to the place.

Example II.3: Transitions, guards, actions, and arcs: Transition `Execute Test` in Fig. 7(b) is an example of a black transition. It represents the call to a tool that executes the program under test with the given test data. In order to provide an example of how guards and actions are written in SLANG, assume that in Fig. 7(b) place `Executable Module` has type `ExecModule`, place `Ready Test Cases` has type `TestCase`, `Data For Test Execution` and `Ready Executable Test` have type `ExecTest`, and finally places `Cumulative Test Results`, `Test Results Summary`, and `Test Results Being Calculated` have type `TestSummary`. The guards and actions associated with transitions `Prepare For Test`, `All Test Cases Run`, and `Start Test` are provided in Fig. 8. Variable names in capital letters (corresponding to the initials of place names) are used to represent both the tokens that are removed from input places and the tokens that are inserted in output places when a transition fires. The type of these variables coincides with the corresponding place type, except for places that are connected with a "*" arc. In this latter case, the variables are of set type, since the number of tokens that may be removed/inserted may be greater than one. For example, transition `Start Test` removes from place `Ready Test Case` all the tokens representing the test cases that refer to a given module. For each event, input variables are separated from output variables by a semicolon. Some predefined operations on sets are used in the example: `forall` ··· : ··· is a quantifier on the elements of a set which returns true iff the predicate following the colon is true on all elements of the set; `for`···`in` is an iterator on all elements of the set; `Empty ( )` always returns the empty set; `set`(···) returns a set containing the elements between brackets, `union` is the union between sets (repeated elements appear only once); and `count`(···) returns the cardinality (integer value) of the set between brackets. In the action part, variables denoting input (sets of) tokens are automatically initialized with the corresponding elements in the input tuple. Variables denoting output (sets of) tokens

[4] Local time constraints and timeouts can also be specified in SLANG. Both affect the firing rule. As we mentioned, time issues are ignored in this paper for simplicity.

[5] This is the normal case. A transition may be connected to an input place by a read-only arc; it may be connected to an output place by an overwrite arc. In such cases, the firing rule changes as discussed below.

```
Event Prepare For Test (DFTE: ExecTest, CTR: TestSummary;
                        RET: ExecTest, TRBC TestSummary)
Guard
    CTR.#PendingTests > 0
Action
    RET = DFTE; TRBC.TestingModule = CTR.TestingModule;
    TRBC.Results = CTR.Results;
    TRBC.#PendingTests = CTR.#PendingTests -1;
-----------------------------------------------
Event All Test Cases Run (CTR: TestSummary;
                          TRS: TestSummary)
Guard
    CTR.#PendingTests = 0;
Action
    TRS = CTR;
-----------------------------------------------
Event Start Test (EM: ExecModule, RTC: set of TestCase;
                  DFTE: set of ExecTest, CTR: TestSummary)
Local test: TestCase, data: ExecTest;
Guard
    forall test in RTC: test.Name = EM.Name
Action
    DFTE = Empty();
    for test in TC {
        data.Name = EM.Name; data.Author = EM.Author;
        data.SourceCode = EM.SourceCode; data.Compiler = EM.Compiler;
        data.ExecCode = EM.ExecCode; data.TestData = test.TestData;
        DFTE = DFTE union set(data);
        }
    CTR.TestingModule = EM;
    CTR.Results = Empty();
    CTR.#PendingTests = count(RTC);
```

Fig. 8. Guards and actions for events `Prepare For Test`, `All Test Cases Run` and `Start Test`

must be explicitly assigned a value. This value is copied in the corresponding output tuple when the action terminates.

An example of read-only arcs is shown in Fig. 6, to describe the fact that, during `Test Module Collection`, libraries and source code are available to other activities (not shown in the figure). □

III. Mechanisms Supporting Process Evolution

In process-centered environments, the process model plays the role of the code to be executed in order to provide automatic support and guidance to the people involved in the process. In SLANG, the process model may include not only the description of the software development process, but also the specification of the software meta-process. The reflective nature of SLANG makes it possible to manipulate the process model and the process model state in the same way as other process data are manipulated. Thus, the mechanisms supporting process enactment in SPADE also support process evolution.

This section presents the *enaction mechanisms of SLANG*, with emphasis on those that make it possible to support process model evolution. For the sake of readability, we illustrate such features in a stepwise fashion, by progressively enriching them as new motivations are presented. Section IV will then discuss how to use the basic mechanisms to define *higher-level policies*. These policies are implemented as part of the process model and combine the above basic mechanisms to obtain the desired behavior.

A. SLANG Interpreter

The *SLANG interpreter* is responsible for the enaction of SLANG process models. In this section we provide an initial intuitive description of how it works. A complete description will be provided in Section III-C.

Each time an activity has to be executed, a new instance of the SLANG interpreter is created. We use the term *process engine* to refer to each running instance of the SLANG interpreter. The root activity of a SLANG process model is executed by the initial process engine which is spawned by a specialized boot program. In any other case, process engines are created using black transitions in which the invoked tool is the SLANG interpreter. These black transitions are executed as part of other activities (initially, just the root activity). This scheme is similar to the mechanism adopted in the Unix operating system to manage process creation. The first Unix process is created by an ad hoc boot program, while any other process is generated using the `fork` service, executed by some running process.

The black transition describing the invocation of the SLANG interpreter has one input and one output place, both of built-in type `ActiveCopyType` (see Fig. 2). Instances of this type are called *active copies* of an activity. Each active copy contains all the necessary information to execute an activity. In particular, it contains the activity definition together with all the type definitions used by the activity, the active copy state, and a reference to either the starting or the ending event along with the enabling or the output tuple, respectively. The active copy state is defined by the contents (marking) of all places of the active copy.

When a process engine starts its execution, it receives an active copy as input. The activity and type definitions define the code to be executed, the state is taken as the initial execution state, and execution begins with the firing of the transition representing the specified starting event with the corresponding enabling tuple. The behavior of a process engine is described here in the case where no activity invocation occurs in the current active copy. Activity invocation is discussed in Section III-B.

Starting from the given initial state, the process engine evaluates the guards associated with the transitions to check whether an input tuple enables some transitions. Then, an enabled transition is chosen (automatically or with user intervention) and it is fired. The firing removes tokens from the input places (or simply reads the values if the input place is connected with a read-only arc), executes the corresponding action, and inserts the produced tokens in the output places (overwriting the existing place contents in case of an overwrite arc). Transition firing is an atomic action, i.e., no intermediate state of the firing is made visible to other process engines that may be executing. In the case of a black transition, however, the firing corresponds to spawning a non-SLANG process which is executed asynchronously. Tokens are removed from the input places when the process is spawned. Upon termination, results are copied into the output places of the black transition.

The occurrence of an event (transition firing) produces a state change that may enable new firings and/or disable previously enabled transitions. An activity terminates if one of its ending events eventually fires. At this point the interpreter execution terminates and the resulting active copy is produced

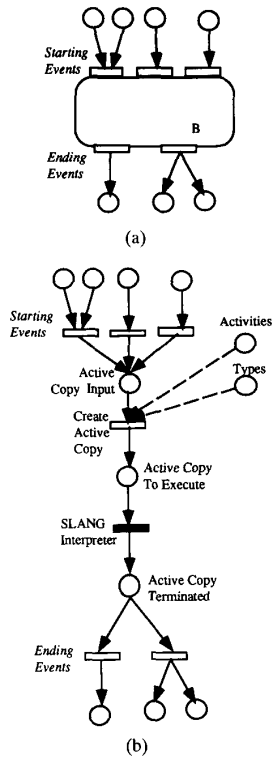


Fig. 9. (a) Invocation of activity B within activity A. (b) Corresponding series of steps in basic SLANG.

in the output place of the black transition that spawned the process engine. The resulting active copy contains the definitions of the activity and its types, the final state, and the ending event, along with the output tuple produced by its firing.

B. Activity Invocation

Activity invocation is a SLANG construct. Along with a few other constructs, however, it is a shorthand notation that does not add to the semantic power of the language, but simply aims at improving its usability. The meaning of activity invocations may thus be described in SLANG using more basic constructs. For example, Fig. 9(a) represents an invocation of activity B within—say—activity A; Fig. 9(b) represents the invocation in terms of SLANG basic constructs.

According to Fig. 9(b), we have the following.

1) Execution of activity B starts when one of its starting events is selected to fire by A's process engine.
2) B's starting event and its enabling tuple are collected as a token value in place `Active Copy Input`.
3) The firing of transition `Create Active Copy` reads from place `Activities` (of type `Activity`) a token representing B's definition, from place `Types` (of type `Metatype`) the definitions of the types used in B's definition, and from place `Active Copy Input` the starting event and its enabling tuple. This information is used to create a token representing the new active copy for activity B. This token is stored in place `Active Copy To Execute`. In particular, the firing of the transition `Create Active Copy` fills in fields `Act`, `Typeset`, `State`, `Start/EndEvent`, and `Input/OutputTuple` (see Fig. 2) of the newly created token. Field `State` is set to the empty marking (i.e., no tokens are initially stored in net places).
4) The black transition `SLANG Interpreter` takes as input a token from place `Active Copy to Execute`. The execution of the black transition yields a process engine for the new active copy of B.
5) When one of the ending events of B's active copy occurs, execution terminates, and thus the black transition in the caller that spawned it terminates as well. The black transition generates a token in place `Active Copy Terminated`. This token has type `Active-CopyType` and stores information on the ending event, together with the resulting output tuple and the active copy state.
6) The value of the output tuple is then used by the ending events to set the contents of the output places of the current invocation. The active copy is destroyed and, thus, its final state is lost.

We emphasize some important issues deriving from the above execution scheme:

- An activity invocation is dynamically bound to the corresponding activity definition (and to the types used within such definition) by the firing of transition `Create Active Copy`. This interpretation mechanism supporting late binding provides the basis for our solution to process evolution, as we will see shortly.
- The actual execution of an activity definition is carried out through a black transition that calls the tool `SLANG Interpreter`. That is, the mechanism used by the language to invoke the interpreter is the same used for invoking any other external tool.
- Activities are executed asynchronously, i.e., the calling activity (A, in our example), continues executing while the called activity (B) is enacted by a newly spawned process engine. It is also possible to have many simultaneous executions of the same activity, by calling the SLANG interpreter several times with the same activity definition.
- The process engine executing B may access shared places during its execution and, at the end, it accesses the output place of the black transition. Consequently, process engines must be synchronized in order to discipline access to shared and output places in mutual exclusion.

An activity can terminate execution if all invoked activities have already terminated. SLANG guarantees that the ending transitions of an activity are not enabled as long as all the (sub)activities that have been spawned are still active.[6] For example, the ending events of activity A will be enabled only if no active copy of B is currently in progress.

[6]This feature is reminiscent of Ada taking. It does not require a semantic extension to SLANG. It may be implemented in SLANG by adding a counter place that holds information on the number of spawned active copies that are still running. This place is in input to each ending event.

We can now define the *activity dynamic tree,* which is the dynamic counterpart of the activity static DAG, presented in Section II-B. Given a SLANG process model *SLANGModel* = (*ProcessTypes, ProcessActivities*), $\mathcal{ACT}$ denotes the set of active copies of all activities in *ProcessActivities* at any time. The relation *is _invoked_by* may be defined on $\mathcal{ACT}$, such that b *is_invoked_ by* a iff b has been created by a. This relation defines a dynamic hierarchy that we call *activity dynamic tree* (ADTree). Each node of an ADTree is an active copy, and each arc connects an active copy with an active copy that created it. The ADTree is modified each time an active copy is created or terminated. The root of the ADTree is the (unique) active copy of *root activity.* There is only one active copy of *root activity* because the static graph is acyclic, and thus *root activity* cannot be invoked by any other activity (it is created by a specialized boot program). It is the first active copy to start and the last to terminate.

C. Accessing an Active Copy State

The mechanisms presented so far provide the ability, within the process model, to manipulate and execute fragments of process models. The late binding schema presented in Section III-B makes it possible, for example, to instantiate an activity with the latest definition provided for it. In order to support dynamic evolution policies, it is also useful to provide mechanisms to manipulate active copies, i.e., the instances of activity definitions that are created during the enactment of the process model.

In order to manipulate an active copy, it is first necessary to suspend its execution. Once suspended, an active copy can be manipulated as any other value represented by a token in the net. A new process engine may then be spawned to restart execution of the modified active copy. The mechanisms needed to suspend and restart an active copy are represented in SLANG by enriching the activity invocation schema described in Fig. 9(b) with further features, as shown in Fig. 10. Moreover, activity definitions, as defined by the user in SLANG, are automatically augmented with a set of places and transitions through a standard syntactic transformation. Fig. 11 shows how B's definition would be augmented through the transformation. In particular, there is a new input place, (`Start`), a new starting event (`Start Active Copy`), new ending events (`out_events`), and a new output place (`Output Info`).

The complete algorithm followed by the SLANG interpreter to manage also suspension and restart of active copies can be described by showing the extensions to the original algorithm of Section III-B. Note that the places that appear with the same name in different nets (such as `Dyn Tree`) represent shared places.

To create an active copy, transition `Create Active Copy` (Fig. 10) initializes the token in `Active Copy To Execute` as follows (refer to the definition of `Active-CopyType` in Fig. 2).

- Field `Act` is set to the extended definition of B, as specified in Fig. 11.
- The transition removes one token from place `Dyn Tree` and one token from place `Active Copy Input`. These tokens are used to create a token to be stored in place `Start` of the active copy being created (see Figs. 11 and 12). Thus, field `State` of the active copy being created is

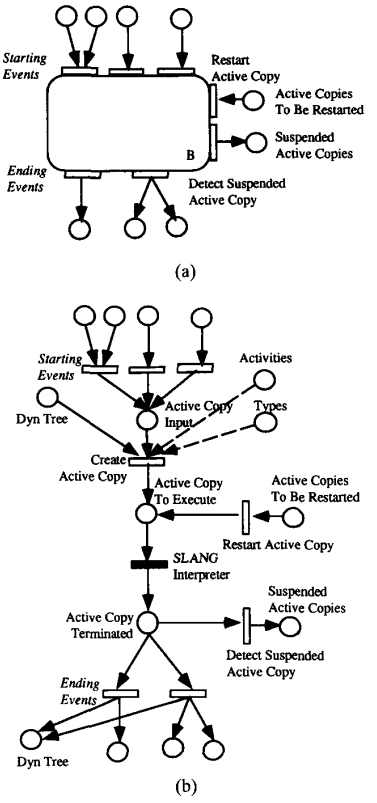


Fig. 10. Enhancement of Fig. 9 to show suspension and restart. (a) Invocation in SLANG. (b) Corresponding subnet in basic SLANG.

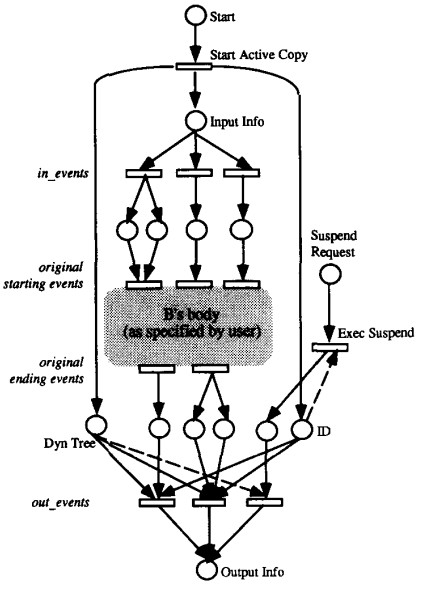


Fig. 11. Augmented definition of an invoked activity in order to manage suspension.

```
I/O Info subtype of Token;
     DT: Tree; -- represents the dynamic tree
     S/E: Transition; -- represents the starting or ending events
     I/O: TokenTuple; -- represents the input or output tuple
     ...
```

Fig. 12. Type definition for type `I/O Info`.

not empty as in the original algorithm of Section III-B, but includes this token. Note that this token holds the information that in the original algorithm was stored in fields `Start/End Event` and `Input/Output Tuple` of the token representing the instantiated active copy. Thus, in this extended algorithm, the two fields are no longer necessary, and the definition of type `ActiveCopyType` should be changed accordingly.

Let `a` be the token removed by the firing of the black transition `SLANG Interpreter` of Fig. 10. The newly created process engine executes `a.Act` (i.e., the net shown in Fig. 11) starting from the initial marking described by `a.State`. In the initial marking, only transition `Start Active Copy` is enabled since the only token in the net is the token stored in place `Start`. The action associated with `Start Active Copy` generates a new unique identifier for the active copy and stores it in place `ID`. It also generates a token in shared place `Dyn Tree` representing the value of the ADTree after the invocation. Such ADTree is obtained by adding the newly generated unique identifier to the ADTree removed from place `Start`. Finally, it generates a token with the enabling tuple and the starting event in place `Input Info`. This token is used by one of the `in_events` transitions to reconstruct the enabling tuple for the starting event to be fired.

At any time during its execution, the active copy may be suspended by the firing of transition `Exec Suspend`. `Exec Suspend` is enabled when a token containing the unique active copy identifier is inserted in place `Suspend Request`. The active copy can also terminate when one of the activity ending events (labeled "original ending events" in Fig. 11) fires. In both cases, one of the `out_events` fires and produces a token of type `I/O Info` (defined in Fig. 12) in place `Output Info`. If an `out_event` fires as a consequence of the firing of one of the original ending events, the tokens stored in places `Dyn Tree` and `ID` are removed, and a token is produced in place `Output Info` with the following information.

- Field `DT` contains the dynamic tree, where the active copy unique identifier has been removed.
- Field `S/E` contains the identifier of the original ending event which fired.
- Field `I/O` contains the output tuple produced by the firing of the original ending event which fired.

If termination occurs becuse of the firing of transition `Exec Suspend`, the only difference is that the dynamic tree is not removed from place `Dyn Tree`. Its value is read in field `DT`. Moreover, there is no output tuple (the active copy has not produced a result yet).

The firing of any of the `out_events` transitions causes the termination of the process engine. As a result, a token is created in place `Active Copy Terminated` of the caller. This token (of type `ActiveCopyType`) contains in field `State` the final marking of the terminated active copy. In particular, place `Output Info` contains information on the cause of termination, as explained above. Depending on this information, either transition `Detect Suspended Active Copy` fires (in which case the active copy is inserted in shared place `Suspended Active Copies`, or one of the ending events fires (see Fig. 10). In the latter case, the output tuple produced by the invoked activity is stored in the corresponding actual places of the caller and the dynamic tree is stored back in place `Dyn Tree`.

When an active copy has been suspended, the token representing it in place `Suspended Active Copies` can be manipulated as any other token. To resume the execution of a previously suspended active copy, it is necessary to move the (possible updated) token describing it into the shared place `Active Copies To Be Restarted`. The `Restart Active Copy` transition moves the token to place `Active Copy To Execute` and then the normal processing is follows. Notice that, in this case, the process engine receives an active copy in an intermediate processing stage, whose marking thus depends on the operations executed before suspension and the operations performed on it while it was suspended.

As a final remark, note that the ADTree is explicitly represented in the net as a token stored in place `Dyn Tree`, which is shared among all active copies. As we will see in Section IV, we need to have the ADTree as a token in the net to support changes to existing active copies. It is easy to realize that the mechanisms shown in Figs. 10 and 11 guarantee that the ADTree is updated in an atomic way. When an active copy is created, the ADTree is extracted by transition `CreateActiveCopy` from shared place `Dyn Tree`. This operation is atomic since it is accomplished through the firing of a single transition. The ADTree is then stored in a field of the token created in place `Active Copy To Be Executed`. This place is local and thus replicated in all active copies. Therefore, the ADTree cannot be accessed by any other process engine other than the one who is currently spawning the new active copy. The updated ADTree is then stored back into place `Dyn Tree` by the process engine of the newly created active copy through the (atomic) firing of transition `Start Active Copy` (see Fig. 11). A similar procedure is followed when an active copy terminates.

D. Lifetime and Visibility Issues

All SLANG process data (process development data and meta-process data) are represented by tokens. The lifetime of tokens depends on the lifetime of the places in which they are contained. When an active copy is created, it allocates database space for token containers for each local place in the activity definition. The (normal) termination of an activity execution causes all local token containers to be deallocated and thus the lifetime of the contained tokens comes to an end.

According to this scheme, only the results of an activity are kept when the active copy terminates. Intermediate results can be explicitly placed in shared places if their lifetime must

extend beyond the lifetime of the active copy. One may also explicitly extend an object's lifetime by using a tool (called via a black transition) to export data to some external database management system. This mechanism, however, takes the data out of the process control (hence out of the SLANG repository), and thus the designer is responsible for managing it correctly.

E. Changing Activity and Type Definitions

Activity and type definitions may be manipulated by firing transitions that remove tokens from, and produce new tokens into, places `Activities` and `Types`, respectively. For example, modification of a type `alpha` consists of removing the token whose value of field `TypeName` is `alpha`, changing the field `TypeDescriptor`, and inserting a new token back into `Types`, with value `alpha` in field `TypeName`, and the modified descriptor in field `TypeDescriptor` (see Fig. 2).

New activity and type definitions become visible only if a new active copy is generated. An existing active copy is not affected by the change since, in its fields `Act` and `Typeset` (see Fig. 2) it keeps the definitions which were valid when the active copy was created. In particular, changes to type definitions stored in place `Types` do not affect existing instances of the modified types.

Activity and type definitions may be changed also locally, by modifying fields `Act` and `Typeset` of a suspended active copy, respectively. When a type is changed, the active copy's places may contain data that were generated according to the old type definition. It is therefore necessary to devise a migration policy of existing data from the old to the new type. This point is further discussed in Section VI-B.

IV. Implementing Evolution Policies

SPADE supports two main classes of change. It is possible to modify tokens whose values are activity and type definitions. It is also possible to modify active copies while they are suspended. In most cases, one first modifies a definition and, later, the effect of a definition change is made visible in a running active copy. It is useful to distinguish between two times: *change definition time* (CDT) and *change instantiation time* (CIT). CDT is the time at which changes are applied to activity and type definitions; CIT is the time at which a change in a definition becomes visible in a running active copy.

SPADE provides the basic mechanisms to apply changes to activity and type definitions, and to active copies. It does not provide any built-in policy. Policies have to be designed by the process modeler and built as part of the process model. As significant examples, we can consider the following policies. A first policy (*lazy*) does not propagate the modification of activity and type definitions to existing active copies. When new active copies are created, the new definitions are used. A second policy (*eager*) is based on the immediate propagation of activity and type definition changes to all the existing active copies. Notice that a whole spectrum of policies exists from fully lazy to fully eager. Finally, a third policy can consist of a local modification of a suspended active copy without modifying any activity or type definition (as already mentioned in Section III-E). This would make only the active copy affected by the change, with no effect on future creations of new active copies of the same or other activities. All these policies can be defined as part of the process model and can vary depending on process requirements. That is, *change is a process too and thus it is modeled as any other part of the process.*

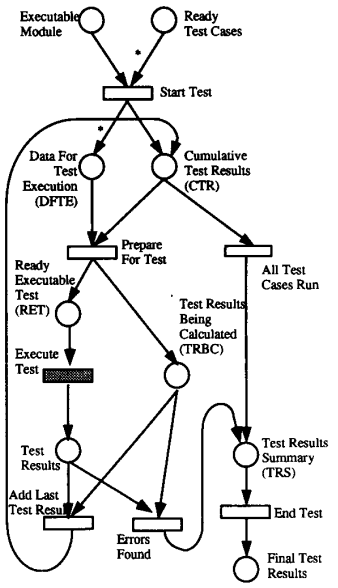


Fig. 13. Modified definition of activity `Run tests`.

A. A Process Evolution Example

In this section, we illustrate a process model evolution example in SLANG. This is just a possible meta-process, not "the" solution. The example of Section II performs the `Run Tests` activity by executing the module under test on all available test cases. We wish to modify the process in such a way that the `Run Tests` activity terminates as soon as a test case generates a failure, instead of using *all* available test cases. The process model must evolve accordingly, in order to capture the new requirements. A new definition of activity `Run Tests` that reflects the changes in the process requirements is shown in Fig. 13.

The activity definition of Fig. 13 can be generated by a meta-process that accesses places `Activities` and `Types`, to edit the required definitions. The time at which editing terminates on a set of definitions defines the CDT of such definitions. The corresponding CIT depends on the policy one wishes to adopt. Fig. 14 shows a fragment of a meta-process that manages changes in our example. It defines two simplified variants of the lazy and eager policies discussed above. For the sake of simplicity, we consider only modifications to activity definitions; the extension of type definitions is straightforward.

The process defined by the net fragment of Fig. 14 can be informally described as follows.[7]

[7]We provide here only an informal textual description. SLANG code can be easily derived from it.

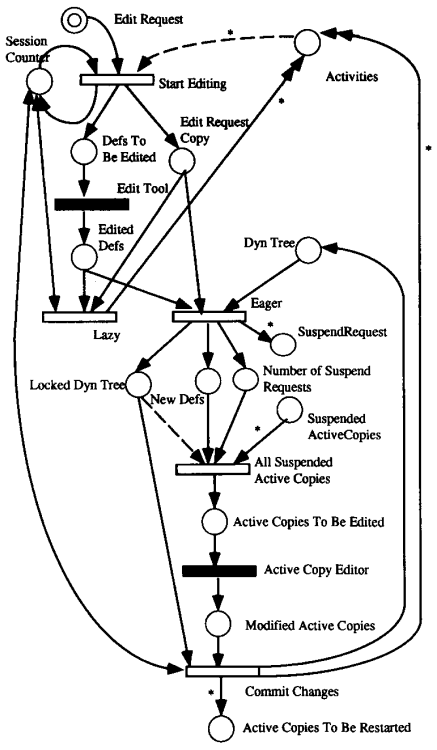


Fig. 14. Meta-process model for modifying activity definitions and active copies.

- The net uses some of the places that were introduced in Figs. 10 and 11. The intended meaning is that places with the same name in different net fragments denote the same place (they are shared places).
- The process designer issues a request to edit the definitions of activities by placing a token in `Edit Request`. If no other editing session is being performed (the value of the token in `Session Counter`[8] is equal to zero), transition `Start Editing` is enabled to fire. The token in `Edit Request` specifies if the update has to be accomplished according to a lazy or an eager policy.
- When `Start Editing` fires, it makes a copy of all the tokens stored in `Activities` (i.e., all activity definitions), and creates a single token in `Defs To Be Edited` containing the set of all the existing definitions. In addition, the session counter is incremented by 1, and a copy of the request is placed in `Edit Request Copy`. Notice that we have used the "*" weight for the arc from `Activities` to `Start Editing` to specify "all the tokens in the place." Moreover, the arc is read-only, because we do not want to remove the tokens from the place. In this way, other active copies can be created and enacted in parallel with the editing activity.
- The firing of the black transition `Edit Tool` causes the invocation of the editing tool that receives as input a set of activity definitions. When the process designer quits from `Edit Tool`, a set of new activity definitions is produced in place `Edited Defs`.
- If the change activity has to be managed according to a lazy policy, transition `Lazy` is enabled to fire. When it fires, the session counter is decreased by 1, the new set of activity definitions substitutes the old set (see the overwrite arc from transition `Lazy` to place `Activities`, and the token in `Edit Request Copy` is consumed.
- If the eager policy is adopted, transition `Eager` fires. The token representing the ADTree is removed from `Dyn Tree` and stored in `Locked Dyn Tree`. In this way, it is impossible for any process engine to start or terminate the execution in an active copy. Transition `Eager` also creates requests to suspend all of the active copies derived by modified activity definitions. Finally, the set of changed definitions is moved to `New Defs`.
- All of the active copies are effectively suspended when the number of tokens in `Suspended Active Copies` equal the value of token in `Number Of Suspend Requests`. It is therefore possible to produce the token in `Active Copies To Be Edited` containing the set of active copies to be edited and the new definitions from `New Defs`.[9]
- When `Active Copy Editor` fires, a tool to manipulate active copies is spawned. When the `Active Copy Editor` terminates, a token containing the new activity definitions and the updated active copies is stored in `Modified Active Copies`.
- Eventually, transition `Commit Changes` overwrites the contents of place `Activities` with the updated activity definitions and inserts in `Active Copies To Be Restarted` as many tokens as the number of active copies to be restarted.

The simple meta-process of Fig. 14 can be extended in several ways.

1) Type definitions may also be modified, by means of a net fragment accessing place `Types`. A few comments on this issue are given in Section IV-B.
2) Instead of accessing all of the tokens in `Activities` at the same time a SLANG meta-process fragment can be defined where activity definitions are edited one at a time and a consistency checker is invoked to check consistency.
3) Other policies can be implemented. For example, instead of suspending all of the active copies affected by the modification of an activity definition, we might decide that the active copies that were created before a given data must not be suspended.[10] This means that all the active copies that have been operating for a long time are terminated according to the old definition, while those that have been recently started are suspended and modified.

[8] Session Counter acts as a sempahore allowing only one editing session at a time.

[9] The new definitions are needed in order to allow the editor to perform consistency checks.

[10] The creation date can be stored in the corresponding node of the dynamic tree when it is created.

This example shows that in SPADE meta-processes are like any other process, and can thus be specified, refined, changed, and improved. In particular the policies chosen to make definition changes visible in the running active copies are fully specificable in the meta-process.

B. Further Issues in Process Change

SLANG is a powerful, but low-level, process language. As we explained, it provides mechanisms to support process change through late binding and ability to manipulate types, activity definitions, and active copies as any process data. In Section IV-A we saw how certain change management policies can be specified in SLANG. It has been a deliberate design decision not to freeze such policies in the language, but rather to allow the process engineer to specify them in SLANG. If experience shows that only certain predefined change policies are needed in practice, we might consider hard-coding them in the language. Presently, however, we feel it would be premature.

As we mentioned in Section III-E, changing a type of definition in a suspended active copy is critical if there are places containing data that were generated according to the old type definition. In our present SLANG implementation, we encapsulate the solution to this problem into the active copy editing tool, which prompts the user to provide a transformation function to support data migration from the old type to the new type. This is a very simple solution that has been adopted for the current prototype. Other solutions are being investigated, including default migration functions.

In order to support reliable change policies it is important to assist the user in understanding and managing the impact of changes. This may be incorporated in the meta-process. Just as an example, the meta-process of Fig. 14 may be extended so that the results produced by `Edit Tool` (i.e., the new definitions stored in place `Edited Defs`) are checked for consistency by an analyzer, before being stored back into place `Activities`. It is therefore possible to catch mismatches between activity definitions and invocations. If types were also changed by `Edit Tool` the analyzer could be applied to detect type mismatches. The analyzer can be implemented as a tool to be invoked through a black transition taking its inputs from `Edit Defs` and providing data for transitions `Lazy` and `Eager`.

V. Related Work

Many research efforts are currently active in the software process modeling area. The results of such efforts are language definitions and prototype environments that are rapidly evolving as research advances. In the following, some the most significant approaches to process model evolution will be discussed and compared with SLANG.

To facilitate the comparison, we will refer to the following set of features.

- F1 It is possible to change (fragments of) a process model during enactment.
- F2 It is possible to change the state of an enacted process model fragment while other fragments are being enacted.
- F3 The process of change can be designed by the process modeler. This means that the mechanisms to support features F1 and F2 can be composed to build different change policies, possibly using the process modeling formalism itself.

SLANG supports F1, F2, and F3.

In FUNSOFT nets [7], a transition can be used to model the editing of a subnet sn (i.e., sn is a part of the whole net executed by the MELMAC interpreter). In order to perform such modification, no further tokens are passed to sn (i.e., no new instances of sn can be started), and all the transitions (agencies in FUNSOFT terminology) internal to sn are terminated before the editing of the subnet is enabled to start. When the editing operation terminates, the new version of sn replaces the old one, and it is enabled to receive new tokens (i.e., sn's transitions are enabled to fire). Notice that, since the body (the actions) associated with transitions are specified in C language, "on-the-fly" modifications can only concern the topology of the subnet. If the "code" associated with a transition is changed, it has to be recompiled. This means that the entire system must be stopped and relinked (see [19, p. 209]). In SLANG, the modification mechanism is more flexible since several active copies of the same activity (similar to FUNSOFT's subnets) can be executed in parallel with the editing of the activity definition. This is possible since SLANG associates a new process engine with each new active copy. In conclusion, MELMAC supports some modification of a process model fragment before that fragment is enacted. This means that MELMAC partially supports F1. It does not support F2, since it is not possible to change the state of an enacting subnet sn. In fact, "all agencies of sn that are currently executed are finished $\cdots$ no enabled agencies are started $\cdots$ [and eventually] sn can be modified by the software developer." This means that is not possible to access the state of an enacting subnet. Finally, the change policy ("algorithm" in [7, p. 91]) is predefined and embedded in the FUNSOFT nets semantics. Therefore, MELMAC does not support F3.

In MARVEL [20], one may include consistency predicates to decide whether an evolution step is allowed or not. A modification to the rule set describing the process is permitted only if the consistency implications after the evolution step are either weaker than the implications before the evolution step is performed, or are independent of them. This constrains the possible evolutions of a process, but the limitation may turn out to be too strong. For example it may be forbidden to further constrain an existing model, since it is not possible to statically ensure that the new constraints are verified by all the instances stored in the MARVEL objectbase. Notice also that any modification to a MARVEL process model can occur only after the system has been stopped. I fact, in [21, pp. 179–180] it is explicitly said that the `evolver`, i.e., the tool supporting process evolution, can be invoked only if the MARVEL server is inactive. This seems to mean that modifications can only be applied off-line. Therefore, MARVEL supports some form of process evolution, but it seems that it supports neither F1, F2,

nor F3, since modifications can be accomplished only when the process model is not being enacted.

In EPOS [9] activities are described through a hierarchy of task types. To execute a task, the EPOS planner instantiates from the task definition (i.e., a task type) the set of lower level tasks to be executed, and then activates the task Execution Manager. The distinction between task types and task instances, and the availability of these entities as objects of the system, are the basic reflective feature in EPOS. In this way, it is possible to apply modifications to the process model even during enactment, since a task definition can be manipulated as any other object in the system. Multiuser support is achieved by synchronizing all the operations performed by the Execution Manager through a centralized, versioned object-oriented database. Thus, EPOS supports feature F1. As far as feature F2 is concerned, when a task definition is modified, EPOS requires existing enacting task instances to be restarted. Restart "reinitiates the task state and possibly backtracks the actions performed by the task" [22, pp. 26–28]. This implies that the execution state and intermediate results are lost. Moreover, it is not clear how certain undoable actions (such as "send a mail") can be backtracked. In general, EPOS embeds a specific policy to evolve enacting process model fragments. It seems, therefore, that EPOS supports features F2 and F3 only partially.

MERLIN [23], [24] is a rule-based language supporting forward and backward chaining. Specific constructs are provided by MERLIN to modify the set of rules and facts being interpreted by the MERLIN executor, using a mechanism similar to Prolog `assert` and `retract` rules (feature F1). From the available published work, however, it is not clear how dynamic changes of the process model can be accommodated when multiple users access and/or execute the same process rule set. Moreover, it is not clear if features F2 and F3 are fully supported. In [25, p. 102], it is said that "the process engineer will be supported by an environment supporting the development and instantiation of software process model. ··· [the environment] will also support the change of existing software processes ···".

A notable example of a process modeling formalism supporting process model evolution is the PML language (and the related environment PSS), developed within the IPSE 2.5 project [10].[11] PML is an object-oriented, reflective language in which it is possible to describe the whole software process (the development process and the meta-process) as an integrated model. Moreover, it provides the process modeler with a set of predefined model fragments (*roles* in the PML terminology) implementing the basic operation of the meta-process. These predefined roles constitute the so-called PMMS paradigm, and are the starting point to implement/extend the meta-process model in order to tailor it to specific needs. Thus, from a conceptual point of view, PML and SLANG have similar functionalities, since the reflective features offered by both languages make it possible to create, enact and maintain the meta-process model as any other fragment of the entire software process model. An interesting and detailed example of meta-process model and process model change using PML is presented in [26]. Summing up, PML supports features F1 and F3. However, it does not seem it can support feature F2, since the state of an enacted process model fragment cannot be accessed during enactment.

The problem of dynamically changing executable code while it is being executed is also addressed in other research areas. In particular, there are several interesting ideas and approaches in the domain of distributed, nonstop systems. For example, [27] discusses a model to dynamically accommodate changes to a distributed system. To apply changes, it is first necessary to "passivate" a component, i.e., to inhibit the invocation of the functionalities offered by the component. Eventually, all of the ongoing operations are terminated, and the component reaches a quiescent state in which it is possible to apply the modification. When this modification is completed, it is possible to consistently restart the component. This approach has many similarities with SPADE functionalities to support the editing of active copies. In fact, an active copy can be changed only after it has been suspended, and can be restarted when the change operation is terminated.

VI. Concluding Remarks

In this paper, we discussed the basic features provided by SLANG to support enactment and, in particular, dynamic evolution of a process model. We emphasized the principle that a process modeling language supporting flexible and powerful change mechanisms should provide reflective features to support the integration of the meta-process as part of the process model itself.

The main advantages of SLANG, and its approach to process evolution, can be summarized as follows.

- SLANG is based on formal foundations of Petri net theory. It is defined in terms of ER nets, a class of high-level Petri nets.
- It provides the process modeler with the basic mechanisms needed to describe the meta-process and, in particular, to describe process evolution as part of the process. These mechanisms can be combined to support different policies to apply changes to a process model.
- It supports the modification of both the process model and the process state (i.e., active copies) during process enactment.

At present, SLANG has the following weaknesses.

- The whole software process, which includes the meta-proces, may become quite large and complex. Complexity may be partly managed by using the modularization construct provided by the language. One may also think that different access rights should be given to different classes of users (e.g., the software developer, the process engineer), so that certain parts of the process (e.g., the meta-process) are only visible to certain classes of users having specific roles.
- SLANG does not tolerate any deviation from the specified model, unless it is explicity introduced as a change in the process model. In real cases, it might be necessary to temporarily tolerate slight deviations by accommodating

[11] This environment is now distributed by ICL as Process Wise.

on-the-fly, light-weight changes or to bypass the process rules by introducing an exception. This would require additional features to be introduced in the language.

- The features described in this paper make SLANG quite a powerful language. As for most powerful languages, however, fully automatic and formal analysis procedures do not exist in general. For example, reachability analysis is undecidable in the general case. It is therefore necessary to derive approximate analysis procedures (including model testing) that would provide partial support, but still would detect errors before enactment. (For a discussion of related issues and possible solutions, see [17], [28].)
- The features provided by SLANG are rather low-level. We view SLANG as the kernel formalism of a process-centered environment. As such, it provides the basic mechanisms of the abstract machine supporting the environment. In this paper, we have shown how various change policies may be represented by suitable SLANG meta-process fragments. As experience is gained in the use of the language, however, we fell that a number of standard change policies will be identified. Such policies could then be introduced as built-in constructs in SLANG, without requiring the process modeler to define them in SLANG. As a result, process models would become simpler and easier to understand.

The SPADE project is currently being developed at CEFRIEL and Politecnico di Milano. A first prototype of the SPADE environment, called SPADE-1, has been implemented and is currently being assessed. The design of SPADE-1 is centered on the principle of separation of concerns between process model interpretation and user interaction. Following this principle the SPADE-1 architecture is structured into two main components: the *process enactment environment* and the *user interaction environment.* These two components communicate through a third component called the *filter* [29].

The *process enactment environment* includes facilities to execute a SLANG process model, by creating and modifying process artifacts. These process artifacts are stored and managed using the object-oriented database O_2 [30].

The *user interaction environment* manages the interaction with users and tools. It is based on an enhanced version of DEC FUSE [31], a product that provides service-based tool integration. In DEC FUSE, a tool is viewed as a set of services that can be invoked through a programmatic interface. This programmatic interface defines a protocol to manage tool cooperation. The protocol is based on a standard set of messages that are exchanged using a multicast mechanism.

The *filter* manages the communication between the two environments above. In particular, it converts messages generated by tools in the user interaction environment into tokens to be managed by the SLANG process model being executed. In the same way, operations accomplished by the SLANG interpreter that affect the user interaction environment are converted by the filter into messages to specific tools.

The basic idea underlying this architecture is that the paradigm used to model the process and support its enactment, and the paradigm used to guide the interaction with the user can be reasonably kept distinct. This is useful to support distribution, evolution and improvement of the environment, and the integration of different types of paradigms in the same process-centered environment.

The SPADE-1 prototype is written in C++ and O2C and runs on DEC 5000 workstations under Ultrix 4.2 and Sun workstations running SunOS.

APPENDIX

ER NETS

ER nets are a class of high-level Petri nets similar to Pr/T nets [32] and Colored nets [33]. They have been introduced in [17] as a formalism that integrates data, control, functionality, and timing aspects. The language was designed as a formal kernel for higher-level specification languages. Its main motivation was in the area of reactive, real-time systems [28], [34]. Here we provide a brief and informal introduction to the formalism.

The tokens of ER nets carry information, and their transitions are augmented with predicates and actions. Predicates constrain the tokens that enable transitions. Actions describe how the tokens produced by a firing depend on the tokens removed. Information attached to tokens is described in terms of variables and associated values. Predicates and actions refer to the information attached to tokens by means of the variables' and places' names.

Fig. 15 shows a sample ER net. The predicate associated with transition t requires the value of variable x, associated with the token in place $p_1(p1.x)$, to be less than the value of variable y, associated with the token in place p_2 $(p2.y)$. The action associated with transitions t states that the token produced in place p_3 by the firing of transition t is associated with variables y and z. The value of variable y of the produced token is given by the sum of the values of $p1.x$ and $p2.y$. The value of variable z of the produced token is any value between $p1.x$ and $p2.y$. For instance, if place p_1 contains two tokens:

$$\tau_1' : \{x \rightarrow 0\}$$
$$\tau_1 : \{x \rightarrow 7\}$$

and place P_2 contains the token

$$\tau_2' : \{y \rightarrow 6\}$$

transition t is enabled by the pair $\langle \tau_1', \tau_2' \rangle$. In fact, the value (0) of variable x associated with the token in place p_1 is less than the value (6) of variable y associated with the token in place p_2, and thus the predicate associated with transition t is satisfied. On the contrary, the pair of tokens $\langle \tau_1, \tau_2' \rangle$ does not satisfy the required predicate and therefore it does not enable transition t. The firing of transition t removes the enabling tokens from places p_1 and p_2 and produces a new token in place p_3 associated with two variables: y and z. The value of variable y in place p_3 is 6. The value of variable z can be any value between 0 and 6, e.g., 4.

In order to model timing, no extensions are needed for ER nets. Time can be represented by simply adding to the set of variables associated with the tokens a variable *time* representing the time at which the token has been created. The

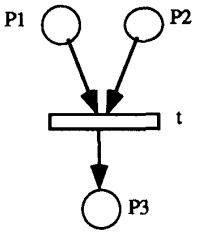


Fig. 15. An example of ER net. The predicate and action associated with transition t are predicate(t): $p_1.x < p_2.y$ action(t): $p_3.y = p_1.x + p_2.y$ and $p_1.x < p_3.z < p_2.y$.

value of variable *time* is determined by predicates and actions associated with the transitions as any other variable associated with the tokens. The value of variable *time* associated with the tokens produced by a firing represents the *firing time* of the transition (we assume that all of the tokens produced by a firing are associated with the same value for variable *time*). Actions can represent any relation between the values of the variables associated with the tokens removed by the firings (including the values of variables *time*) and the values of the variable produced by the firings, thus giving a general model for time. For instance, ER nets can specify the case where the firing time of a transition, representing the transmission of a packet on a net, can depend on the length of the packet. In [17] ER nets are shown to be more general than the timed Petri net formalisms introduced in the literature. Suitable axioms constrain variable *time* in order to ensure the representation of an intuitive concept of time, which is non decreasing with respect to sequences of firings. Depending on the set of axioms, two different time semantics can be given: *weak* and *strong* time semantics. The main difference between the two semantics is that in weak time semantics an enabled transition *may* fire within its time frame, as specified by the action. The transition, however, is not forced to fire. If time moves beyond the maximum possible firing time for a transition, the transition looses its right to fire. According to strong time semantics, and enabled transition *must* fire within its time frame, unless it is disabled by the firing of another conflicting transition.

ACKNOWLEDGMENT

The anonymous reviewers and the editors provided very valuable comments and suggestions that helped us improve the paper. Several friends and colleagues contributed to this work. In particular, we are indebted to L. Lavazza for his constructive criticisms and to CEFRIEL students who did a superb job in developing the prototype as part of their Master's Thesis. We also thank the software engineers from the DEC Engineering Center in Gallarate for their support.

REFERENCES

[1] L. Osterweil, "Software processes are software too," in *Proc. Ninth Int. Conf. Software Engineering,* IEEE, 1987.

[2] M. Dowson, B. Nejmeh, and W. Riddle, "Fundamental software process concepts," in *Proc. First European Workshop Software Process Modeling,* A. Fuggetta, R. Conradi, and V. Ambriola, Eds., Milano, Italy, AICA—Italian National Association for Computer Science, May 1991, pp. 15–37.

[3] P. Armenise, S. Bandinelli, C. Ghezzi, and A. Morzenti, "Software process representation languages: Survey and assessment," in *Proc. 4th Int. Conf. Software Engineering and Knowledge Engineering,* Capri, Italy, IEEE, June 1992, pp. 455–462.

[4] C. Liu and R. Conradi, "Process modeling paradigms: An evaluation," in *Proc. First European Workshop Software Process Modeling,* A. Fuggetta, R. Conradi, and V. Ambriola, Eds., Milano, Italy, AICA—Italian National Association for Computer Science, May 1991, pp. 39–52.

[5] N. H. Madhavji, "Environment evolution: The prism model of changes," *IEEE Trans. Software Eng.,* vol. 18, pp. 380–392, May 1992.

[6] D. Heimbigner, S. M. Sutton, and L. Osterweil, "Managing change in process-centered environments," in *Proc. 4th ACM/SIGSOFT Symp. Software Development Environments,* Dec. 1990; in *ACM SIGPLAN Notices.*

[7] V. Gruhn, "Validation and verification of software process models," Ph.D. dissertation, Univ. Dortmund, 1991.

[8] M. Suzuki, A. Iwai, and T. Katayama, "A formal model of re-execution in software process," in *Proc. 2nd Int. Conf. Software Process,* Berlin, Germany, Feb. 1993.

[9] L. Jaccheri, J.-O. Larsen, and R. Conradi, "Software process modeling and evolution in EPOS," in *Proc. SEKE '92—Fourth Int. Conf. Software Engineering and Knowledge Engineering,* Capri, Italy, IEEE Computer Spciety Press, June 1992, pp. 574–581.

[10] R. Bruynooghe, J. Parker, and J. Rowles, "PSS: A system for process enactment," in *Proc. First Int. Conf. Software Process,* IEEE Computer Society Press, 1991.

[11] W. Reisig, *Petri Nets—An Introduction.* New York: Springer-Verlag, 1982.

[12] J. Peterson, *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[13] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. IEEE,* vol. 77, pp. 541–580, Apr. 1989.

[14] S. Bandinelli, A. Fuggetta, C. Ghezzi, and S. Grigolli, "Process enactment in SPADE," in *Proc. Second European Workshop Software Process Technology,* Trondheim, Norway, Springer-Verlag, Sept. 1992.

[15] S. Bandinelli, A. Fuggetta, and S. Grigolli, "Process modeling in-the-large with SLANG," in *Proc. 2nd Int. Conf. Software Process,* Berlin, Germany, Feb. 1993.

[16] S. Bandinelli and A. Fuggetta, "Computational reflection in software process modeling: The SLANG approach," in *Proc. 15th Int. Conf. Software Engineering,* Baltimore, MD, IEEE, May 1993.

[17] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzé, "A unified high-level petri net formalism for time-critical systems," *IEEE Trans. Software Eng.,* Feb. 1991.

[18] D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Trans. Software Eng.,* vol. 5, pp. 128–138, Mar. 1979.

[19] V. Gruhn and R. Jegelka, "An evaluation of FUNSOFT nets," in *Proc. Second European Workshop Software Process Technology.* Trondheim, Norway: Springer-Verlag, Sept. 1992.

[20] N. S. Barghouti and G. E. Kaiser, "Scaling up rule-based software development environments," in *Proc. ESEC'91—Third European Software Engineering Conf.,* A. van Lamsweerde and A. Fuggetta, Eds., vol. 550 of *Lecture Notes in Computer Science,* Milano, Italy, Springer-Verlag, Oct. 1991.

[21] Marvel Group, *Marvel 3.1 Manual,* Columbia Univ., 1993.

[22] M. L. Jaccheri and R. Conradi, "Techniques for process model evolution in EPOS," Politecnico di Torino, Tech. Rep. DAI/SE/92/1, Oct. 1992.

[23] B. Peuschel and W. Schäfer, "Concepts and implementation of a rule-based process engine," in *Proc. 14th Int. Conf. Software Engineering,* Melbourne, Australia, ACM–IEEE, May 1992, pp. 262–279.

[24] H. Hünnekens, G. Junkermann, B. Peuschel, W. Schäfer, and J. Vagts, "A step towards knowledge-based software process modeling," in *Proc. First Conf. System Development Environments and Factories.* London, England: Pitman, 1990.

[25] B. Peuschel, W. Schäfer, and S. Wolf, "A knowledge based software development environment supporting cooperative work," *Int. J. Software Eng. Knowledge Eng.,* vol. 2, pp. 79–106, Jan. 1992.

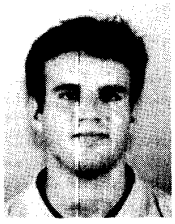
[26] R. Snowdon, "An example of process change," in *Proc. Second European Workshop Software Process Technology,* Trondheim, Norway, Springer-Verlag, Sept. 1992.

[27] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Trans. Software Eng.,* Nov. 1990.

[28] M. Felder, C. Ghezzi, and M. Pezzé, "Formal specification and timing analysis of high-integrity real-time systems," in *Proc. NATO Advanced Study Institute Real-Time Computing,* St. Marteen, Oct. 1992.

[29] S. Bandinelli, M. Braga, A. Fuggetta, and L. Lavazza, "The architecture of the SPADE-1 process-centered SEE," to be presented at the *Proc.*

Third European Workshop Software Process Technol., Grenoble, France, Feb. 1994.
[30] O. Deux, "The O_2 system," *Commun. ACM,* vol. 34, Oct. 1991.
[31] Digital Equipment Corp., Maynard, MA, *DEC FUSE Handbook,* Version 1.1, Dec. 1991.
[32] H. J. Genrich, "Predicate transition nets," in *Advances in Petri Nets 1986 LNCS 254-255.* Springer-Verlag, 1987.
[33] F. Jensen, "Coloured Petri nets," in *Advances in Petri Nets 1986 LNCS 254-255.* Springer-Verlag, 1987.
[34] M. Felder, C. Ghezzi, and M. Pezzé, "High level timed nets as a kernel for executable specifications," *J. Real-Time Syst.,* May 1993.



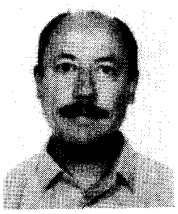
Sergio C. Bandinelli received the Dr. degree in computer science from the University of Lujan (ESLAI), Argentina, in 1989.

He has been an instructor at the University of Buenos Aires, Argentina, and at the Politecnico di Milano, Italy. He is currently a Ph.D. student at the Politecnico di Milano. He is also a research assistant at CEFRIEL, a research and educational institute established in Milano by universities, industries operating in the information technology market, and local public administrations. His research interests are in process modeling, software engineering environments, and transaction management.

Alfonso Fuggetta (M'83) received the Dr.Eng. degree in electrical engineering from Politecnico di Milano, Milano, Italy.

He now holds the position of Associate Professor of Computer Science at the same university. He is also Senior Researcher at CEFRIEL. His research interests are in software process modeling, executable specifications, and architectures of advanced software engineering environments.

Dr. Fuggetta is Chairman of the Steering Committee of the European Software Engineering Conference (ESEC) and member of the Steering Committee of the European Workshop on Software Process Technology (EWSPT). He is member of ACM, and of the board of directors of AICA, the Italian National Association for Computer Science.



Carlo Ghezzi received the Dr.Eng. degree in electrical engineering from Politecnico di Milano, Milano, Italy.

He now holds the position of Full Professor of Computer Science at the same university. Prior to that, he taught at the Universities of Padova (Italy) and North Carolina at Chapel Hill. He also spent sabbatical periods at UCLA and UCSB. His research interests are in software engineering specification languages, support environments, formal process modeling, and programming languages. He is coauthor of over 100 scientific papers and 8 books.

Dr. Ghezzi has been the Program Chair of the 2nd European Software Engineering Conference (ESEC 89), Program Co-Chair of the 6th IEEE International Workshop on Software Specification and Design, and Program Co-Chair of the 14th International Conference on Software Engineering (ICSE-14). He will be Program Chair of the 9th International Software Process Workshop (ISPW-9). He is a member of the editorial board of IEEE TRANSACTIONS ON SOFTWARE ENGINEERING and *Trends in Software.*