Computational Reflection in Software Process Modeling: the SLANG Approach

Sergio Bandinelli Alfonso Fuggetta

Politecnico di Milano* and CEFRIEL

Abstract

Software production processes are subject to changes during their life-time. Therefore, software process formalism must include mechanisms to support the analysis and dynamic modification of process models, even while they are being enacted. It is thus necessary for a process model to have the ability to reason about its own structure. Petri net based process languages have been criticized because of the lack of these reflective features and their inability to effectively support process evolution. In this paper we present the reflective features offered by SLANG, a process formalism based on an high-level Petri net notation. In particular, we discuss the mechanisms to create and modify different net fragments while the modeled process is being enacted.

Keywords: *software process modeling, process languages, computational reflection, process evolution.*

1 Introduction

Recently, many researchers and practitioners have pointed out that software processes need to be analyzed and carefully modeled, in order to support and facilitate their understanding, assessment, and automation (see for example [18, 6]). To address such issues, several research efforts have been launched both in industry and in academia. The ultimate goal of such efforts is to provide innovative means to increase the quality of software production processes and, consequently, of the applications delivered to final users. The first results of these activities are several prototype languages and experimental environments, which provide specific features to create, analyze, and enact software process models [1, 14].

In order to effectively support modeling, analysis, and automation of processes, a *software process language* must exhibit several characteristics:

1. It must be formally defined and based, so that it is possible to automatically analyze and execute (enact) a process model.
2. It must allow the process specifier to describe both the activities which constitute the process, and the results produced during its execution (the process artifacts).
3. Software processes are human-oriented systems, i.e. systems in which humans and computerized tools cooperate in order to achieve a common goal. Therefore, a process formalism must provide means to describe such interaction, by clearly defining, for instance, when and how a task is delegated to a tool or a human, and how to coordinate the operations of different human agents.
4. The target architecture for process enactment must be a concurrent/distributed environment, possibly operating on different, heterogeneous systems.
5. 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 information hiding, abstract modeling, and reuse of process model fragments.
6. Finally, software processes are dynamic entities, that must evolve in order to cope with changes in the software development organization, in the market, or in the technologies and methodologies used to produce software. Accordingly, it must be possible to incrementally refine and change a software process model, *even while it is being enacted.*

*Politecnico di Milano, Dipartimento di Elettronica e Informazione, P.za Leonardo da Vinci 32, 20133 Milano (Italy). E-mail: bandinelli@mailer.cefriel.it, fuggetta@ipmel2.elet.polimi.it. Tel.: +39-2-23993623, Fax: +39-2-23993411.

0270-5257/93 $03.00 © 1993 IEEE

As we said, many approaches have been proposed in the last years, but it does not seem that any existing solution is able to effectively cope with all the abovementioned issues. In particular, the last point on process model evolution seems to constitute one of the most challenging problems which the software process community is facing.

The core of the evolution problem can be summarized as follows. A process model must include the description of the activities and tasks constituting both the actual software process and the *meta-process* used to create and evolve the process model itself [5]. Consequently, the process machine accessing the process model must be able to use it as the "code" to be executed in order to support and automate the software process, and also as the "data" to be edited in order to apply modifications to the process model itself. This interleaving of execution and editing activities must be managed in such a way that it is possible to gracefully migrate from the old process model to the new one, without interrupting or restarting from the very beginning the operations being carried out within the software process, unless this is explicitly required by the meta-process.

The problem of producing a program that is able to "reason" about its structure has been largely discussed within the programming languages community. Languages such as LISP and Prolog allow programs to be manipulated as data (for example a Lisp program is a list that can be executed using the `eval` function, and the `assert` and `retract` rules in Prolog allow modification of the program while it is running). More recent developments in the area of object oriented languages offered new chances and challenges to researchers interested in this issue (see for example [16]). This research topic is very often referred to as *computational reflection* [15].

From the above discussion, it is clear that, in order to effectively address the process model evolution problem, a process formalism must provide reflective features. In some existing systems, this is achieved by building the process model as a rule-based system, where powerful mechanisms are provided to add, modify, or retract rules during the evaluation of a program, using mechanisms quite similar to those provided by LISP and Prolog. However, as we will discuss in Section 5, rule-based systems suffer from other problems that may adversely affect other aspects of process enactment.

SPADE is a research process modeling environment that supports software process analysis, design, and enactment. Within this project, we have defined a language, called SLANG (SPADE LANGuage), that addresses most of the issues discussed above and, in particular, provides language features and execution mechanisms to cope with the process evolution problem. In this paper, we present SLANG characteristics, with a special emphasis on its reflective facilities, that can be used to model the meta-process for managing the modification and evolution of the process model.

The paper is organized as follows: Section 2 introduces SLANG and summarizes its basic features using a simple example process. Section 3 explains the architecture supporting the enaction of a process model and the basic reflective mechanisms of SLANG. Section 4 discusses how the reflective features of SLANG can be used to support process evolution, and provides some hints and examples on how to describe a metaprocess as part of the process model. Related works are surveyed in Section 5. Finally, Section 6 draws some conclusions.

2 An introduction to SLANG

SLANG is a domain-specific language for software process modeling and enactment, based on ER nets[1]. ER nets [8] are a high-level extension of 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 relations to transitions, that describe the constraints on tokens consumed and produced by transition firings.

SLANG addresses all the process language requirements presented in the introduction. Process enactment and interaction with humans and tools is discussed in [2], while problems related to process modularization are treated in [3]. SLANG specifications are hierarchically structured, using the high-level *activity* construct. Activities may be associated with different process engines for their execution. In addition, SLANG provides a way to deal with time in process specification and supports ways to formally reason about the temporal evolution of a process. In this paper we will focus on the SLANG facilities to support process evolution and, in particular, on the SLANG mechanisms to manipulate process models as data.

In order to present SLANG, we will refer to an example process throughout the paper. The example, inspired by the ISPW example [13], is presented in the next section.

[1] A SLANG specification may be given semantics by a translation to the formal ER net model.

2.1 An example process

The example refers to the modification of a software unit, triggered by requirements change. It is composed of the following steps:

- The "design" step involves the modification of the design document. It uses as input the requirements change document.
- The "coding" step corresponds to the implementation of the modifications. It includes the editing of the source code and its compilation. The "coding" step uses as input the requirements change document and the modified design. However, it may be initiated even if the design has not been completed. The coding has to be finished within 5 days from its beginning, otherwise, the process manager has to be notified by e-mail.
- The "prepare test package" step involves the preparation of the test package for the unit. The test package contains the software to drive and evaluate the coded unit. This step takes as input the unit design. Its output is the requested test package.
- The "testing" step involves the application of the test package on the object code produced by the "coding" step. Therefore, it cannot start before both the "coding" and the "prepare test data" steps have been completed. It takes as inputs the object code and the test package. If the test results are OK, the test step terminates successfully. Otherwise, the code has to be revised: the "coding" step is reinitiated using as input the test feedback, and then the module has to be retested.

2.2 Basic SLANG features

A SLANG process specification is a pair of sets:

$$SLANGSpec = (ProcessTypes,\ ProcessActivities)$$

ProcessTypes is a set of type descriptions organized in a type hierarchy. Each type is a class description in an object oriented style. The hierarchy contains a predefined type called *Activity.* The second component of a SLANG specification is *ProcessActivities* which is a set of instances of type *Activity.*

The *ProcessTypes* hierarchy is described in Figure 1. The root of this hierarchy is *ProcessData,* the ancestor of all types used in SLANG. All the elements of a SLANG specification are defined by the *ProcessTypes* hierarchy. Since SLANG is based on a Petri net formalism, the hierarchy includes the definition of the components of a Petri net. *Place, Transition,* and *Arc* respectively describe the types of places, transitions, and arcs.

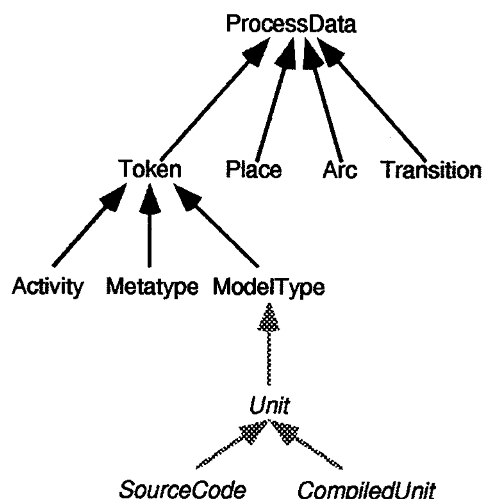

Figure 1: The *ProcessTypes* hierarchy

Token describes the tokens of the net and has three subtypes *ModelType, Activity,* and *MetaType. ModelType* the root of all types needed for the description of the objects manipulated within a software process, such as modules, documents, resources, ...(with the exception of activities and types). The subhierarchy whose root is *ModelType* depends on each particular process, and thus may vary from one SLANG specification to another. Moreover, it may change during process enactment. *Activity* is the type that defines a SLANG activity as a Petri net composed of places, transitions and arcs. The *Metatype* type defines how types are described. There is one *Metatype* instance for each subtype of type *Token,* including *Metatype* itself. Each instance is a tuple *(TypeName, TypeDescr, TypeVersion),* where *TypeName* is the name of a type, *TypeDescr* is its description, and *TypeVersion* is a version number incremented each time *TypeDescr* is changed.

While all subtypes of *ModelType* are user modifiable, the types *Metatype, ModelType* and *Activity* are not. Thus, the instances of *Metatype* representing types *Activity, ModelType,* and *MetaType* itself cannot be changed; their definitions are built-in. Changes to other instances of *Metatype* will cause a corresponding modification in the *ModelType* sub-hierarchy. Note

that, since *MetaType* and *Activity* are subtypes of type *Token*, activities and types are also tokens. This fact plays a fundamental role in the reflexive mechanism of SLANG, as we will see later on.

Example 2.1: Type definitions As an example of a type definition, consider the types *Unit*, *SourceCode* and *CompiledUnit*, shown in Figure 1 (operations are omitted). These types, as all user-defined types, are subtypes of type *ModelType*.

```
Unit is a subtype of ModelType;
    name: string;
SourceCode is a subtype of Unit;
    source_code: text;
CompiledUnit is a subtype of Unit;
    n_compilation_errors: positive_integer;
    source_code: text;
    object_code: code;
```

□

An *activity definition* includes an *interface* and an *implementation* part. The activity interacts with the rest of the process through its interface. The interface is composed of:

- a set of interface transitions,
- a set of interface places,
- a set of arcs, connecting interface places to interface transitions and vice versa.

Interface transitions are partitioned in two subsets: the set of transitions representing *starting events* (SE), and the set of transitions representing *ending events* (EE). An activity begins executing when one starting event occurs and finishes with the occurrence of an ending event.

Interface places are classified in three disjoint sets. The set of *input places*, that are input for any of the starting events; the set of *output places*, that are output places for any of the ending events; and the set of *shared places*, that can be input or output places for any transition in the implementation part. Basically, input and output places play the role of formal parameters, while shared places are similar to traditional global variables. Any other place internal to the implementation part (not belonging to the interface) is called a *local place*. The implementation part of an activity describes how inputs are transformed into outputs.

Example 2.2: Activity definition. Figure 2 shows the activity that describes the coding step of the example process. It gives the user two possibilities for the editing of the unit code: with and without access to the approved design. The code is then compiled and the process is iterated until no errors are obtained during compilation. If the compiled unit is not ready within 5 days, the project leader is notified via e-mail. The interface of the activity is identified by the entities external to the dashed rectangle, while the implementation part is represented by all the entities internal to it. In this example, SE = {*Restart Coding*, *Begin Coding*}, and EE = {*End Coding*}; *Test Feedback* and *Requirement Change Document* are input places, *Coded Unit* is an output place. Place *Units To Be Edited* (UTBE) is of type *SourceCode*; *Compiled Unit* (CU) and *Ready Object Code* (ROC) have type *CompiledUnit*. □

Since *TokenType* is the root of the hierarchy that contains token definitions, every token is instance of a subtype of *TokenType* (hence also instance of *TokenType*). In a given SLANG specification, we will denote with *Tok* the set of all (potential) instances defined by the sub-hierarchy whose root is *TokenType*.

Each place has a name and a type. A place works as a token repository that may only contain tokens of its type or any subtype of it. The only way a place may change its contents is by the firing of a transition connected to it. A particular kind of places, called *user interface* places may change their contents by human intervention. User interface places are depicted with a double line and are used to communicate events provoked by humans to the system .

Transitions represent events taking a negligible amount of time to occur: the occurrence of an event corresponds to the firing of a transition. Each transition is associated with a guard and an action. The transition's guard is a predicate on input tokens and is used to decide whether an input token tuple enables the transition (an input tuple satisfying a transition guard is called enabling tuple). The dynamic behavior of a transition is described by a *firing rule*. The firing rule states that when a transition fires, tokens satisfying the guard are removed from input places and the transition's action is executed. As a result of the action execution, an output tuple is inserted in the output places of the fired transition.

Example 2.3: Guards and actions. Consider transitions *Compilation OK* and *Compilation not OK* of the coding activity implementation of Figure 2.

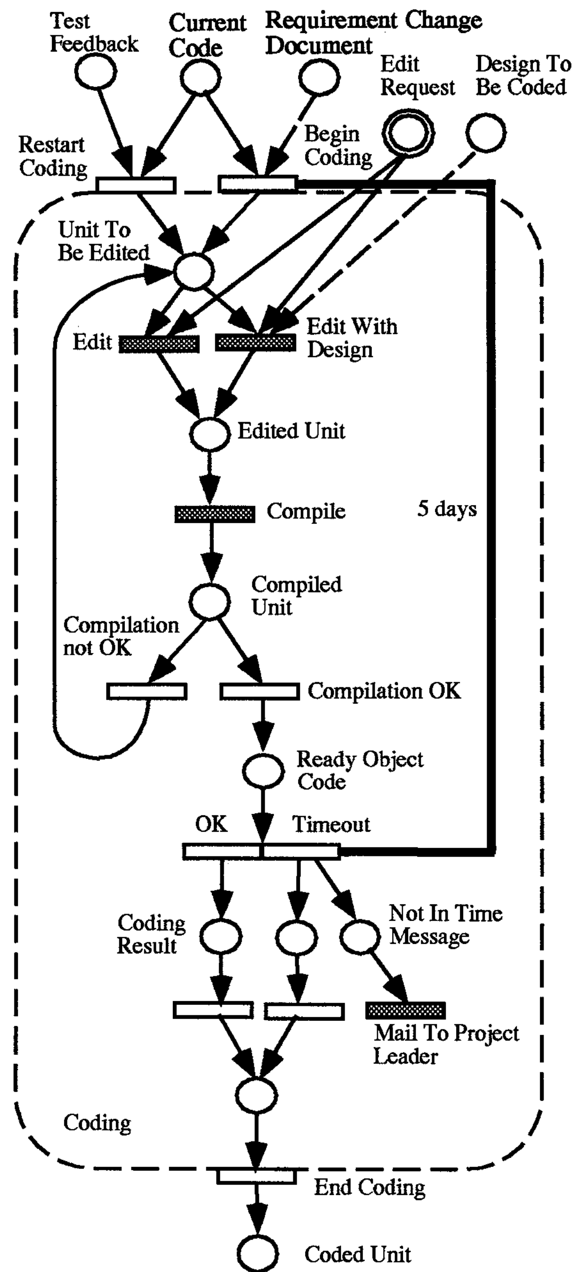

Figure 2: SLANG implementation of the "coding" activity.

- *Compilation OK.* It takes as input a *Compiled Unit* (CU); if no errors occurred in compilation, it produces a *Ready Object Code* (ROC).
- *Compilation not OK.* It takes as input a *Compiled Unit* (CU); if any error occurred during compilation, it produces as output a *Unit To Be Edited* (UTBE), so that more editing can be done.

Guards and actions may be specified in SLANG as follows.

```
Event CompilationOK (CU; ROC)
Guard
    any CU from Place(CU)
    such that CU.n_compilation_errors = 0
Action
    ROC.name := CU.name
    ROC.source_code := CU.source_code
    ROC.object_code := CU.object_code
Event CompilationNotOK (CU; UTBE)
Guard
    any CU from Place(CU)
    such that CU.n_compilation_errors > 0
Action
    UTBE.name := CU.name
    UTBE.source_code := CU.source_code
```

□

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 has been replaced by a call to 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 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.

Example 2.4: Black transitions. In Figure 2, transitions *Edit*, *Compile*, and *Mail To Project Leader* are examples of black transitions. □

Arcs are weighted (with default weight 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. In the latter case, the arc weight is known only at run-time. 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* (depicted

using a dashed line) and *overwrite* (they are depicted with a double arrow). A read-only arc may connect a place to a transition. The transition can read and use token values from the input place in order to evaluate the guard and the action, but no token is actually removed or modified. An overwrite arc may 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 token(s) 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.

An activity implementation may contain *calls* to other activities. Considering the different activities in a SLANG specification it is possible to define the relation *uses*. We say that A *uses* B iff A's implementation contains a call to 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 calls are not allowed in SLANG, and thus, the graph is a DAG (Direct Acyclic Graph). This graph does not change as long as the process specification is not changed. For this reason we call it *activity static DAG* (ASD). In a SLANG specification, there is one "main" activity at the top of the ASD. This activity is called *root activity* and is launched by a specialized boot program to start the process execution.

Example 2.5: Activity call. Figure 3 shows the activity *modify unit* that contains calls to other activities that implement the different steps of the example process. This high-level view gives a general abstract idea of the relationships among the different steps. □

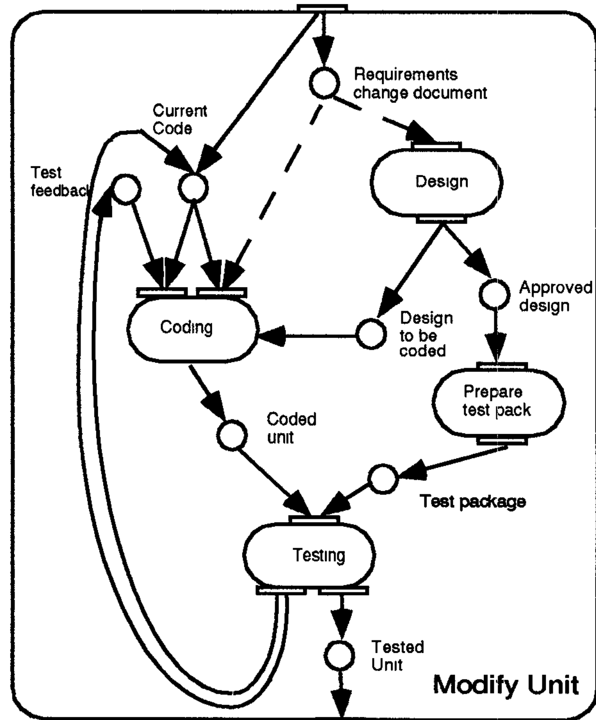

Figure 3: High-level specification of the example process.

3 SLANG reflective features

In a process centered environment, the process model plays the role of the "code" to be executed in order to control and monitor the process. The *SLANG interpreter* is the tool that executes a SLANG specification. We use the name *process engine* to refer to each running instance of the SLANG interpreter.

The dynamic behavior of a SLANG activity is defined operationally by the firing rule that describes possible state changes. The process engine evaluates the guards associated with the transitions and analyzes their time constraints. From all enabled transitions (feasible events), one is chosen (automatically or with user intervention) and it is fired. The firing removes tokens from the input places, executes the corresponding action, and inserts the produced tokens in the output places. This produces a state change that may enable new firings. The procedure is iterated and this yields a firing sequence that is the net execution trace.

In order to have an efficient execution and favor distribution, a SLANG specification is concurrently interpreted by several process engines that communicate and are synchronized via interface places, as discussed next.

3.1 Activity execution

If during the execution of an activity A, one of the starting events in the interface of activity B used by A is selected to fire, the following steps are executed:

- First, the process engine executing A creates a new copy of B, called an *active copy* of B. An active copy of an activity is obtained by creating a new instance of the activity. Thus, each new active copy of an activity has its own local places. Interface places are shared by the different active copies and the calling activity.

- A's process engine extracts the tokens enabling the starting transition from the input places. Then, a new process engine is assigned to execute the newly created active copy of B.

- This process engine receives as input parameters the name of the starting transition that has caused the invocation of B, and the related enabling tuple. Then, it executes the action of the specified starting transition with the given enabling tuple.

- A's process engine continues the execution of activity A, while, in parallel, B's new active copy is executed by the new process engine.

When one of the transitions corresponding to the ending events of an active copy is selected to fire, the active copy terminates by executing the following steps.

- The process engine in charge of the active copy execution synchronizes with other active copies in order to access the output places of the caller activity and insert the produced tokens.

- The process engine of the ending active copy deallocates all local places, whose contents are therefore lost, and then terminates its execution.

SLANG guarantees that the ending transitions of an activity are not enabled as long as all the active copies that have been launched by the current active copy are not terminated. Namely, if an active copy A_{active} has instantiated several active copies a_1, a_2, etc., during its execution, A_{active} will not be enabled to terminate until a_1, or a_2, etc. terminate their own processing.

The issue of persistence of places and tokens requires a few comments. The first step of the procedure used to create an active copy includes the creation of local places for each new active copy that has to be launched. These places and their contents are deallocated when the active copy terminates. This means that a place and its contents exist as long as the active copy is running. Thus, in general, the contents of local places are persistent only while the associated active copy is running. The only true persistent places in a SLANG specification are the interface places of *root activity*, since they are not local to any other activity.

The activity invocation schema that has been outlined before can be described in terms of a lower level net using only events and one black transition (see Figure 4). This is based on the fact that there is no restriction about the tool that may be invoked by a black transition. Since activities are also tokens of the net, it is possible to call a tool that operates on an activity. In particular it is possible to call the SLANG interpreter. The interpreter takes as input a net representing an activity and a corresponding input tuple and then the net is executed.

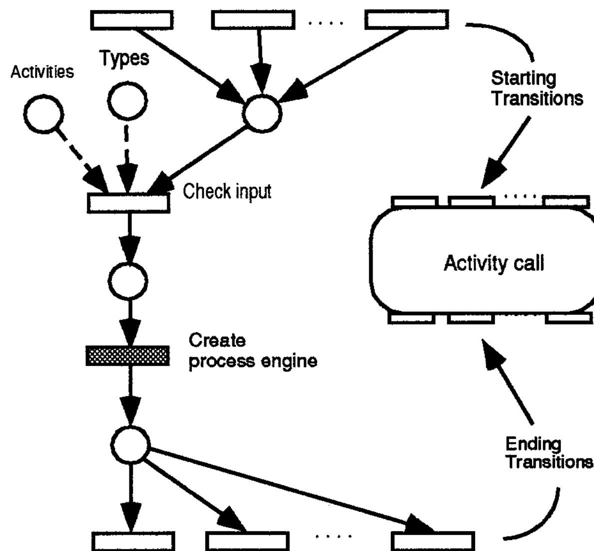

Figure 4: Activity invocation in SLANG.

Since the token representing an activity is only read, and the execution of the black transition is asynchronous, it is possible to launch different, independent active copies of the same activity in parallel.

In this approach, it is also possible to access a token describing an activity in order to modify it, even if there exist several active copies of the same activity. In particular, it is possible to design an *editing* activity which reads the token of the activity to be modified from place *Activities* and passes it to a tool supporting activity editing. This tool produces a new version of the token (i.e. of the activity) that will become available as soon as it is stored back in the *Activities* place, superseding the old one. In this way, the new definition of the activity will be used when a new active copy of that activity is created: the existing active copies are not affected.

A similar mechanisms is adopted to manage type definitions. In Figure 4 a place called *Types* is introduced to contain all the tokens of type *Metatype*, describing the types used in the SLANG specification. It is possible to access any token in this place and modify it much the same way as we did to modify activity definitions. The new version of a *Metatype* token becomes "visible" when it is accessed to create new places of that type.

3.2 State of an enacted SLANG specification

When a SLANG specification is accessed by one or more process engines, we say that it is being enacted and we call it an *enacted SLANG specification.* We characterize the state of an enacted SLANG specification as the union of the states of all the active copies of activities in *ProcessActivities* at a given time, as detailed in the following.

Given a SLANG specification $Spec = (ProcessTypes,\ ProcessActivities)$, we will denote with $\mathcal{ACT}_t$ the set of active copies of all activities in *ProcessActivities* at time t. The active copies of $\mathcal{ACT}_t$ are related to each other through the relationship *has invoked*, which links an active copy with all the active copies that it has created so far. This relationship defines a tree in which each node is an active copy, and each arc connects an active copy with one of the active copies it has launched. This tree is modified when an active copy is created or destroyed, for this reason we call it *activity dynamic hierarchy.* Notice that this is a tree and not a DAG, since each time an activity is invoked, a new active copy is created.[2]

The root of this hierarchy is the (unique) active copy of the *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 (as we said, it is created by a specialized boot program). Moreover, it is the first active copy to start and the last one to terminate.

The state of an active copy of an activity is defined by the marking of the corresponding Petri net. The *state of an enacted SLANG specification* at time t is defined by the union of all markings of the active copies in $\mathcal{ACT}_t$.

In general the firing of a transition causes a change in the state of the enacted SLANG specification.

1. When a normal (white) transition fires, it provokes a change in the marking of the corresponding active copy. In this case, the state of an active copy is modified, but the set $\mathcal{ACT}_t$ remains untouched.

2. If the firing of a black transition or of a transition belonging to the starting or ending set of an activity occurs, the process state changes, due to a modification of set $\mathcal{ACT}_t$.

Notice that if the contents of place *Activities* or of place *Types* is modified, the change of the state of the enacted SLANG specification causes also a change in the SLANG specification itself.

[2] This hierarchy is the dynamic counterpart of the activity static DAG (ASD), discussed in Section 2.

3.3 Types and process enactment

As a consequence of the dynamicity of the SLANG binding rule we have:

1. Every definition is bound at run-time. Thus, type checking is completely dynamic.

2. Different versions of a type definition can be used at the same time in different active copies, even for the same activity definition. Changes in type and activity definitions are effective only when they are used. In other words, there is a possible latency time between the change of a definition and use of the new definition in the enacted process.

3. Places *Activities* and *Types* are predefined as shared places of *root activity.* Their definition cannot be changed.

The resulting behavior supports a controlled and smooth migration from an existing process specification to its modified version, as discussed in more detail in Section 4.

4 Process model evolution

One of the key aspects of the enacting scheme presented in Section 3 is the invocation (or instantiation) mechanism, and its relation with the editing of an activity. As we said, tokens describing activities are kept in the *Activities* place and therefore may be read and modified by transitions. This means that, for example, it is possible to launch several process engines executing multiple instances of the same activity, and at the same time change its definition. The termination of the editing session (which causes the new token to be stored in the *Activities* place) does not affect the active copies of the modified activity which are being executed, since the related process engines operates on a copy of the activity implementation net. Therefore, if new invocations of the modified activity occur before the older instances are terminated, it happens that different versions of the same activity are concurrently executed.

This is a general and flexible scheme, since it supports a gradual or "lazy" migration from the older process specification to the new one. In an "eager" approach, changes to type or activity definitions have

an immediate effect on all involved active copies and tokens. The latter strategy can be explicitly modeled in SLANG, for example, by adding to an activity an ending transition enabled by a shared place. A token in this place represents a request to the activity to terminate immediately. When a new version of an activity or type has been made available, a token is inserted in the above-mentioned shared place to force the termination of all the active copies of that activity. The choice of having the lazy strategy as the basic mechanism in SLANG gives more flexibility to the language and, as we saw, permits the implementation of other strategies.

Consider Figure 5 as an example of changing an activity. On the left-hand side, a black transition is used to model the invocation of a SLANG editor. If no consistency checks are done, the token is directly put back in the *Activities* place. On the right-hand side the modification of the activity is performed by a complex activity. This may correspond to the case of modifying the interface of an activity which may have an impact on other activity definitions where the modified activity is used. Thus, the change activity may involve a sequence of editing and check steps, before producing a definitive result. The modifications to an activity are visible when a new active copy of the activity is created.

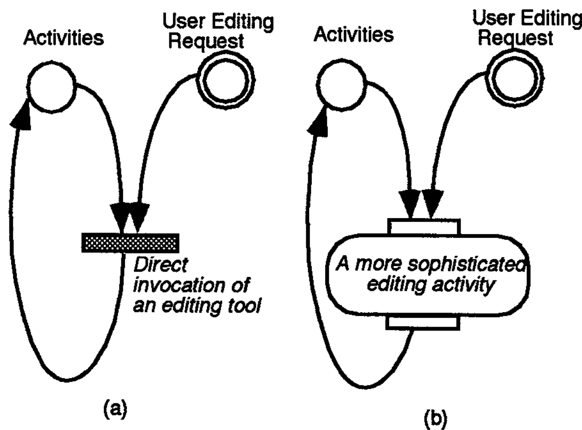

Figure 5: Editing of tokens representing activities.

A very last comment concerns the relationship between the SLANG constructs and the portion of the process model used to modify the process model itself. To support reflection in SLANG we have tried to enrich our basic formalism with the minimum set of features that are sufficient to develop a reflective system. Therefore, we have provided only basic mechanisms to support reflection, leaving the task of creating specific modification policies to the process specifier.

Summing up, the reflective features described above allow us to create process models that includes the capabilities of modifying themselves. In particular, it is possible to change a SLANG specification by the firing of a transition that changes the contents of place *Activities* and/or place *Types*, dynamically modifying already existing tokens or creating new ones. Dynamic evolution of a specification is not the only application of reflective features. In effect, reflective features may be used for analysis purposes or for defining measurement policies within the process model. For instance, a SLANG specification may include an activity that collects data about the enactment of the specification itself.

5 Related work

Most of the currently available systems providing some form of support to process model evolution exploit the rule-based approach to process modeling (e.g., MARVEL, Merlin, OIKOS, EPOS, ALF/MASP). A non rule-based system which provides specific functionalities to deal with process evolution is MELMAC, which is based on the extended Petri nets formalism called FUNSOFT Nets.

In FUNSOFT Nets 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 agencies internal to *sn* are terminated before the editing of the subnet is enable 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) [9]. Notice that, since the body (the actions) associated with each transitions are specified in the C language, "on-the-fly" modifications can only concern the topology of the subnet. If the "code" associated to a transition is changed, it has to be recompiled. This means that the whole system must be stopped and relinked [10]. In SLANG, the modification mechanism is more flexible since several instances of an activity (similar to FUNSOFT's subnets) can be executed in parallel with the editing of the activity itself. This is possible since SLANG associates a new process engine with each new activity instance. Moreover, the new process engine operates on a (virtual) copy of the implementation net of the activity. Finally, the actions

associated with a SLANG transition are interpreted, thus providing a more effective approach to process model change.

In MARVEL one may include consistency predicates to decide whether an evolution step is permissible 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 carried out, or are independent of them [4]. This limits the possible evolutions of a process, but these limitations may turn out to be too strong. For example it may be not permitted 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 object base. For these cases, a mechanism to perform a lazy transformation of the object base should be added, in order to accommodate the data to the new constraints.

In ALF [17], a software process is described by a hierarchy of MASPs. Each MASP describes a set of objects, the relationships existing among objects, and a set of inference rules. MASPs are instantiated to create IMASPs (Instantiated MASPs) that are enacted by the IMASP server. Even if the dynamic instantiation of MASP guarantees a high degree of flexibility, it is not clear how it is possible to ensure or check that the object base is still congruent after the modification to a MASP is applied. In fact, as in MARVEL, a MASP includes inference rules defining triggers on the object base, and it seems reasonable that, if something changes in a MASP definition, they should be applied to all objects in order to verify that the object base is still consistent.

In EPOS [12] activities are described through a hierarchy of task types. To execute a task, the EPOS planner instantiates from the task definition (a task type) the set of lower level tasks to be executed, and then activates the tasks Execution Manager. The distinction between task types and task instances is 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.

MERLIN [19, 11] is a rule-based language supporting forward and backward chaining, and specific command to alter the set of rules and facts being interpreted by the MERLIN executor, using a mechanism similar to Prolog `assert` and `retract` rules. From the available published literature, however, it is not clear how dynamic changes of the process model can be accommodated and managed when multiple users access and/or execute the same process rule set.

6 Conclusions

In this paper we have presented the basic features provided by SLANG to support the enactment and, in particular, dynamic evolution of a process model. We have emphasized the well-known idea that a process formalism aiming at supporting such functionalities must provide reflective features to support the integration of the meta-process as part of the process model itself.

In the last years, several process formalisms have been proposed. In order to cope with the evolution problem, most of them have been built as rule-based systems, since within this approach it is possible to provide simple and effective features to assert and retract rules during program execution. In the SPADE project, which is based on an operational formalism, we have been able to support evolution by providing:

- Concurrent interpretation of different cooperating process model fragments.
- Late-binding.
- Mechanisms to build re-entrant process model fragments.

In SLANG, we added these features to the basic mechanisms offered by Petri nets. In this way, we are able to retain the advantages deriving by the adoption of a formal, graphical, state-oriented notation, and at the same time provide new reflective features to support the process model evolution problem.

Acknowledgments

The authors wish to thank Carlo Ghezzi for his continuous support and supervision.

References

[1] Pasquale Armenise, Sergio Bandinelli, Carlo Ghezzi, and Angelo Morzenti. Software Process Representation Languages: Survey and Assessment. In *Proceedings of the 4th International*

Conference on Software Engineering and Knowledge Engineering, pages 455–462, Capri (Italy), June 1992. IEEE.

[2] Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, and Sergio Grigolli. Process Enactment in SPADE. In *Proceedings of the Second European Workshop on Software Process Technology*, Trondheim (Norway), September 1992. Springer-Verlag.

[3] Sergio Bandinelli, Alfonso Fuggetta, and Sandro Grigolli. Process Modeling-in-the-large with SLANG. In *Proceedings of the 2nd International Conference on the Software Process*, Berlin (Germany), February 1993.

[4] Naser S. Barghouti and Gail E. Kaiser. Scaling up rule-based software development environments. In Axel van Lamsweerde and Alfonso Fuggetta, editors, *Proceedings of ESEC 91–Thrid European Software Engineering Conference*, volume 550 of *Lecture Notes on Computer Science*, Milano (Italy), October 1991. Springer-Verlag.

[5] Reidar Conradi, Christer Fernström, Alfonso Fuggetta, and Bob Snowdon. Towards a reference framework of process concepts. In *Proceedings of the Second European Workshop on Software Process Technology*, Trondheim (Norway), September 1992.

[6] Mark Dowson, Brian Nejmeh, and William Riddle. Fundamental Software Process Concepts. In *[7]*, pages 15–37.

[7] Alfonso Fuggetta, Reidar Conradi, and Vincenzo Ambriola, editors. *Proceedings of the First European Workshop on Software Process Modeling.* AICA–Italian National Association for Computer Science, Milano (Italy), May 1991.

[8] Carlo Ghezzi, Dino Mandrioli, Sandro Morasca, and Mauro Pezzé. A Unified High-level Petri Net Formalism for Time-critical Systems. *IEEE Transactions on Software Engineering*, February 1991.

[9] Volker Gruhn. *Validation and Verification of Software Process Models.* PhD thesis, University of Dortmund, 1991.

[10] Volker Gruhn and Rüdiger Jegelka. An Evaluation of FUNSOFT Nets. In *Proceedings of the Second European Workshop on Software Process Technology*, Trondheim (Norway), September 1992.

[11] H. Hünnekens, G. Junkermann, B. Peuschel, W. Schäfer, and J. Vagts. A Step Towards Knowledge-based Software Process Modeling. In *Proceedings of the First Conference on System Development Environments and Factories*, London (UK), 1990. Pitman Publishing.

[12] Letizia Jaccheri, Jens-Otto Larsen, and Reidar Condadi. Software Process Modeling and Evolution in EPOS. In *Proceedings of SEKE '92–Fourth International Conference on Software Engineering and Knowledge Engineering*, pages 574–581, Capri (Italy), June 1992. IEEE Computer Society Press.

[13] M. Kelner et al. Ispw-6 software process example. In *Proc. of the 6th. International Software Process Workshop*, Hakodate (Japan), October 1990.

[14] Chunnian Liu and Reidar Conradi. Process Modeling Paradigms: An Evaluation. In *[7]*, pages 39–52.

[15] Pattie Maes. Concept and experiments in Computational Reflection. In *Proceedings of OOPSLA '87*, pages 147–155, 1987.

[16] Norman Meyrwitz, editor. *Proceedings of OOPSLA '89*, New Orleans (Lousiana), October 1989. ACM.

[17] Flavio Oquendo, Jean-Daniel Zucker, and Philip Griffiths. The MASP Approach to Software Process Description, Instantiation and Enaction. In *[7]*, pages 147–155.

[18] Leon Osterweil. Software processes are software too. In *Proceedings of the Ninth International Conference on Software Engineering.* IEEE, 1987.

[19] Burkhard Peuschel and Wilhelm Schäfer. Concepts and Implementation of a Rule-based Process Engine. In *Proceedings of the 14th International Conference on Software Engineering*, pages 262–279, Melbourne (Australia), May 1992. ACM-IEEE.