

SPADE: An Environment for Software Process Analysis, Design and Enactment

Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, Luigi Lavazza
CEFRIEL – Politecnico di Milano



GOODSTEP ESPRIT-III project No. 6115

GOODSTEP Technical Report No. 020
March, 1994

Abstract

The SPADE project aims at developing an environment to define, analyze, and execute process models defined in the SLANG process modeling language. This paper presents SLANG through a case-study and highlights the main features of the first prototype implementation of the SPADE concept, called SPADE-1. It illustrates the facilities provided by SLANG to support process evolution and shows how the SPADE-1 architecture supports integration of the process interpreter with external tools available in the environment.

SPADE: An Environment for Software Process Analysis, Design, and Enactment*

Sergio Bandinelli ^{◊* †}
Carlo Ghezzi [◊]

Alfonso Fuggetta ^{◊*}
Luigi Lavazza ^{*}

[◊]Dip. di Elettronica e Informazione

Politecnico di Milano

Piazza Leonardo da Vinci, 32

I-20133 Milano

Fax: +39-2-23993587

e-mail: {fuggetta, ghezzi}@ipmel2.elet.polimi.it

{lavazza, bandinelli}@mailer.cefriel.it

^{*}CEFRIEL

Politecnico di Milano

Via Emanuelli, 15

I-20126 Milano

Fax: +39-2-66100448

Abstract

The SPADE project aims at defining and developing a software process-centered environment to describe, analyze, and enact software process models. These models are specified in SLANG (Spade LANGUAGE), a process modeling language that provides process specific modeling facilities. This paper presents SLANG through a case study which illustrates SLANG main features, including process modularization, interaction with tools, and process evolution. In addition, it describes the architecture of the first implementation of SPADE, called SPADE-1, showing how the environment supports concurrent enactment of process activities, integration of external tools, and storage of persistent data in an object-oriented database.

Keywords and phrases: software process, process-centered environment, computer-supported cooperative work, process language, Petri net, object-oriented database.

*This work has been partially supported by ESPRIT projects GoodStep and Promoter, and by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo

[†]Sergio Bandinelli is partially supported by Digital Software Engineering Center, Gallarate, Italy

1 Introduction

Software production is accomplished through a set of activities that are concurrently carried out by interacting agents. Agents cooperate to develop software applications, support their evolution, deliver new releases, produce and maintain reusable components, etc. These activities are performed by managers, software designers, programmers, end-users, etc. with different backgrounds, roles, objectives, and views of the process. The software process is the common framework within which these people interact. Despite its importance, however, the process is usually left implicit, incomplete, or ambiguous and little or no attention is paid to make it clearly understood by the people involved in software development activities.

By providing an explicit process description, flaws in the process can be found and corrected. Communication and cooperation can also be improved, resulting in an overall improvement of the process effectiveness. In addition, if a formal operational notation is used for modeling, it is also possible to automate (parts of) the process, to support interaction and cooperation among people, and to provide them with guidance during enactment.

The SPADE project [BFGG92, BFG93b, BF93] aims at designing and developing a process-centered software engineering environment. The project is carried out at Politecnico di Milano and CEFRIEL and its ultimate goal is to provide a software engineering environment to support Software Process Analysis Design and Enactment. The environment is based on a process modeling language, called SLANG (SPADE Language), which is a high-level Petri net based formalism¹. SLANG offers features for process modeling, enactment, and evolution. In addition, it describes interaction with external tools and humans in a uniform manner. The main features of SLANG modeling facilities can be summarized as follows:

- Process models can be structured in a modular way using the *activity* construct. Activities (i.e., process fragments) can be dynamically instantiated.
- Activities can be manipulated as data by other activities; i.e., SLANG supports computational reflection.
- Process artifacts, including process models, are kept and maintained in an object-oriented database.

¹We assume here that the reader knows the basic concepts of Petri nets and of high-level nets. For a general background, the reader can refer to [Rei85]

It has been a deliberate decision in the design of SLANG to provide the necessary mechanisms for process modeling, without freezing any process-specific policy in the language. The result is the structuring of the language definition in two levels. Kernel SLANG is a low-level and very powerful formal notation, which provides the basic constructs and mechanisms for process modeling. Kernel SLANG is thus a minimum set on top of which new process specific constructs may be defined (by giving semantics in terms of primitive SLANG constructs) and incorporated to the language. The language augmented with these new constructs is called Full SLANG (or SLANG).

A first implementation of the SPADE environment, called SPADE-1, has been completed by mid 1993. SPADE-1 includes a process interpreter, which is able to enact process models written in SLANG. The environment architecture is based on the principle of separation of concerns between process model enactment and user interaction. The process enactment environment includes facilities to execute a SLANG specification, by creating and modifying process artifacts. The user interaction environment manages the interaction between users and tools in the environment. It is based on an enhanced version of DEC FUSE [DEC92]². Software artifacts manipulated during process enactment are stored in a central repository managed by the O_2 object-oriented DBMS [Tec92]. Tools in the user environment may also access the database and share data among them and with the process.

This paper is an overview of the SPADE Project. It provides a guided tour through SLANG and presents the mechanisms needed for process enactment, and the main features of SPADE-1 architecture. Section 2 gives a brief assessment of existing approaches to process modeling and puts SPADE in the context of current efforts. Section 3 presents the case-study that is used through the paper as a running example. Section 4 introduces basic constructs of SLANG, which constitute Kernel SLANG. Section 5 defines Full SLANG process-specific constructs and facilities. Section 6 briefly describes SPADE-1 architecture. A complete process model of the running example, using Full SLANG, is presented in Section 7. Section 8 discusses evolution issues. Finally, Section 9 draws some conclusions and outlines future work.

²DEC FUSE is 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 that defines a protocol to manage tool cooperation. The protocol is based on a standard set of messages that are exchanged using a multicast mechanism.

2 Related Work

Much research effort is being carried out in the software process field, yielding a number of different approaches to process representation and execution. These efforts range from the use of full-fledged programming languages [HSO90] to active database extensions [BEM91] and abstract specification formalisms, including Statecharts [Kel91], rule based languages [PS92, BK91], attribute grammars [SIK93], and Petri nets. An introduction and comparison of existing approaches can be found in [ABGM92]. A number of different approaches are also described in this volume.

Several process modeling efforts have been based on languages derived from Petri nets. FUNSOFT nets [Gru91] are based on PrT nets, a class of high-level Petri nets. PROCESS WEAVER [Fer93], developed by Cap Gemini Innovation in the context of the Eureka Software Factory project, provides a set of tools to add process support to UNIX-based environments. In PROCESS WEAVER a process is described as a hierarchy of activity types. Each activity is associated with a procedure that specifies how the activity is carried out. Procedures are described by transition nets, a data-flow notation similar to Petri nets.

A software process can be viewed as a set of interacting software engineering activities that are carried out in parallel by multiple cooperating agents and must be scheduled in order to meet some global time constraints. Building on this view and on past research done in the area of formal specifications of real-time systems, SLANG has been formally based on a high-level timed Petri net formalism, called ER nets [GMMP91].

One of the main concerns in designing SLANG is to offer expressive and powerful constructs that can be used for process modeling in-the-large. A SLANG process model can be hierarchically structured as a set of activities, each described by a net that may include invocations to other (sub)activities.

Another distinguished feature of SPADE/SLANG is the mechanism supporting process evolution. Petri net based process languages have been criticized because of their inability to support process evolution. SPADE incorporates reflective mechanisms to support changes to process definitions and process states, even while the process is being enacted. The problem of supporting process evolution is addressed also by other process languages, such as EPOS [JLC92] and IPSE 2.5 [BPR91], in the context of other formalisms.

3 A running example

In this paper, SLANG is illustrated through the example proposed for the 6th and 7th International Software Process Workshops [KFF⁺91]. The example deals with the modification of a program unit caused by changes in the global specification of a system. For each unit that has to be modified a change unit specification is written. The design is changed according to the new specification, and a new design unit together with a new unit interface is produced.

In general, changes in the implementation of a unit U do not affect other units. Changes in the interface of U , instead, need to be agreed upon by the designer responsible for U and those responsible for all units which “use” U . Thus, once the new design of U has been completed, the new interface is proposed to all U ’s users, who are asked to either accept the new interface or to reject it. If no reply is sent back within the given deadline, it is assumed that there are no objections to changing the unit interface. If any U ’s user rejects the proposed interface, the unit’s designer makes a new proposal, which is in turn reconsidered by all users. This process is iterated until all U ’s users agree on the interface. Only after this condition is satisfied, U ’s coding and generation of test cases for the unit may start.

Coding implies editing the unit and compiling it. Preparation of data for unit test proceeds in parallel with coding. The unit is then linked together with drivers and stubs in order to obtain an executable program. The unit is then 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. Only if no error is found, the unit is accepted. Otherwise the unit is rejected and a report is produced with a list of all the errors found.

4 Kernel SLANG

This section describes the basic constructs and mechanisms for process modeling defined in Kernel SLANG. It first presents the Kernel SLANG type system; and then, Kernel SLANG nets and their interpretation, since Kernel SLANG is the language understood by the SLANG interpreter

4.1 Kernel SLANG type system

Kernel SLANG offers a rich type system to model the variety of data produced, used, and manipulated in a software process. These data may be of very different kinds, including, for example, graphical software specifications, failure reports, test data, and executable code. The representation of such information may require complex data structures, containing many relationships of different nature (e.g., abstract syntax trees, program dependency graphs, etc.). In addition, the process model may need to access data at different levels of granularity. For example, it may refer to an entire system that has to be validated, or to a unit that has to be compiled, or even to a single procedure that has to be removed.

All process data in Kernel SLANG are typed. Type definitions follow an object-oriented style, organized in a type hierarchy, defined by an “is-a” relationship. Each type defines an abstract data type, i.e., a data structure that may only be accessed by the exported operations. A type definition contains the type name (which uniquely identifies the type), the specification of the types from which the type inherits, the list of attributes, and the list of operations to access instances of the type³.

Kernel SLANG offers a built-in set of basic types, including `integer`, `real`, `char`, `string`, `boolean`, `text`, `bitmap`, etc. Types may be combined using the type constructors `tuple`, `set`, `list`, etc.. The attribute list is in fact a tuple that defines the structure of the type. The structure can be made arbitrarily complex by combining basic types and other user-defined types via type constructors.

Being Kernel SLANG process models based on a high-level Petri net notation, process data are represented by tokens. Each token is a typed object, upon which it is possible to apply the operations defined in its type description or inherited from its super types. The SPADE type hierarchy provides a predefined type `Token`. All tokens in a Kernel SLANG process model are of type `Token`, or of one of its subtypes. Process specific user-defined types, which characterize the particular process being modeled, are represented by a subtree of `Token` in the “is-a” hierarchy rooted `ModelType`.

Example: Type definitions. Figure 1 shows the “is-a” type hierarchy for some of the types used in the case-study. The textual type definitions are given in Figure 2. Type `Unit` represents a general source code unit.

³SLANG type definitions correspond to O_2 classes.

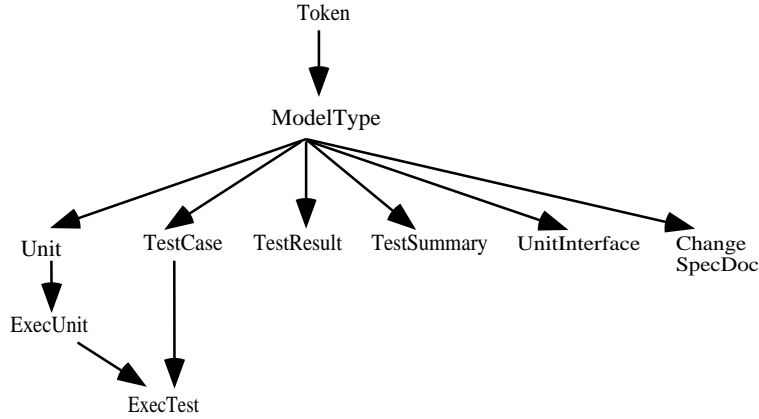


Figure 1: Type hierarchy for some of the types of the running example.

ExecUnit defines executable units (i.e., a unit that has been compiled and linked with drivers and stubs). **TestCase** defines test cases for unit testing. The information needed to apply a test case to a unit is described by the type definition **ExecTest**. Test results are described by **TestResult** and the complete report 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).

4.2 Kernel SLANG nets

This section provides the definition of Kernel SLANG nets and the interpretation algorithm used to execute them. A Kernel SLANG net SN is defined as a triple:

$$SN = (P, T, A)$$

P is the set of *places*, T is the set of *transitions*, and $A \subseteq (P \times T) \cup (T \times P)$ is the set of *arcs*. The standard notation used in Petri net literature will be followed. In particular, $\bullet t$ denotes the preset of transition t (the set of input places of t) and t^\bullet denotes the postset of t (the set of output places of t). A Kernel SLANG net satisfies the following properties:

$$P \neq \emptyset, T \neq \emptyset, A \neq \emptyset$$

$$\forall t \in T, \bullet t \neq \emptyset, t^\bullet \neq \emptyset$$

```

class Unit inherit ModelType
type tuple (public name: unitName,
            public authorName: personName
            public language: string,
            public sourceCode: Text)
end;
-----
class ExecUnit inherit Unit
type tuple (public compilerUsed: string,
            public execCodePath: pathName)
end;
-----
class TestCase inherit ModelType
type tuple (public name: unitName,
            public testData: Text)
end;
-----
class ExecTest inherit ExecUnit, TestCase
end;
-----
class TestResult inherit ModelType
type tuple (public usedTest: moduleName,
            public failures: list (Failure))
end;
-----
class TestSummary inherit ModelType
type tuple (public testingUnit: unitName,
            public results: set (TestResult),
            public pendingTestsNb: integer)
end;

```

Figure 2: Type definitions of some of the types of the running example.

Places. Each place has a name and a type. A place is a token repository that may only contain tokens of its type or of any subtypes. Contents of places change through transition firings. The only exceptions to this rule are *user* places, which are a special kind of places that change their contents as a consequence of an event occurring in the user environment. They are used to communicate external events caused by humans or tools to the system. While normal places are graphically represented by a single circle, user places are represented by double circles. The type of a place must be **Token**, or one of its subtypes.

Transitions. Transitions represent events whose occurrence takes a negligible amount of time. Each transition is associated with a guard and an action. The guard is a predicate on tokens belonging to the transition's input places. The action specifies how the output token tuple is computed as a function of the input token tuple. Black transitions are a special kind of transitions in which the action part includes the asynchronous invocation of a non-SLANG executable routine (e.g., a Unix executable file).

Arcs. Arcs are weighted (the default weight being 1). The weight indicates the number of tokens flowing along 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 weight is known only at runtime and may vary at each firing occurrence. This is useful to model events requiring, for example, *all tokens* that satisfy a certain requirement.

An *active copy* of a Kernel SLANG net SN is a net definition associated with a state. The net state is given by a marking, i.e., an assignment of tokens to places.

An active copy can be executed by a tool, called the *SLANG interpreter*. The SLANG interpreter works in the following way. Starting from an initial state, it evaluates the guards associated with the transitions to check whether an input tuple of tokens enables the transition. Then, one enabled transition is chosen (automatically or with user intervention) and it is fired. The firing of a transition t is accomplished by removing the enabling input tuple from $\bullet t$, executing t 's action and, as a result, producing an output tuple to be inserted into $t\bullet$. The number of tokens removed and inserted depends on the weight of the corresponding arc. Transition firing is an atomic action, i.e., no intermediate state of the firing is made visible to the outside (i.e., to other process engines).

The firing of a black transition represents the invocation of an external tool. The tool is executed *asynchronously*. This means that other transitions may be fired while the tool associated with the black transition is still being executed. It is also possible to fire the same black transition many times with different input tuples, without waiting for each activation to complete.

The occurrence of an event (transition firing) produces a state change that may enable new firings and/or disable previously enabled firings. The execution cycle is then repeated and produces a firing sequence, which represents the execution history of the net.

4.3 Kernel SLANG example

The Kernel SLANG model of the test phase example (originally presented in [BFG93a]) is shown in Figure 3. The Kernel SLANG net shows that an executable unit and test cases for that unit must be available to start testing. All of the test cases developed for the unit must be run (this is represented by the label “*” on the arc from `Test cases` to `Start test`). Test cases are then taken one at a time and executed. This is represented by the black transition `Execute test`. The test results obtained in place `Test results` are then added to the results in `Cumulative test results`. When there are no more pending tests, `All tests cases run` fires and the test phase for that unit is completed.

With reference to the type definitions of Figure 2, in the Kernel SLANG net of Figure 3, place `Exec unit` has type `ExecUnit`, place `Test cases` has type `TestCase,Data for test execution` and `Executable test` have type `ExecTest`, `Test results being computed` has type `TestResult`, and finally places `Cumulative test results` and `Test results summary` have type `TestSummary`.

Each of the transitions of the example has a guard and an action attached. Figure 4 describes the guard and action code associated with events `Prepare test`, `All test cases run`, and `Start test`. 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 type “set”, since the number of tokens that may be removed/inserted may be greater than one. For example, transition `Start test` removes from place `Test cases` all tokens representing the test cases that refer to a given unit.

For each event, input variables are separated from output variables by a semicolon. Some predefined self-explaining operations on sets are used in the example.

5 Full SLANG

Kernel SLANG is the language executed by the SLANG interpreter. With respect to such kernel language,

Full SLANG introduces a set of predefined types to support process evolution (as explained in Section 8) and new constructs that do not modify the semantic power of the language, but provide a set of shorthand notations to improve the language usability⁴.

A Full SLANG process model is a pair of sets:

$$SLANGModel = (ProcessTypes, ProcessActivities)$$

where *ProcessTypes* is a set of type definitions and *ProcessActivities* is a set of activity definitions.

Type definitions are given using the Kernel SLANG type system described in Section 4.1. The *ProcessTypes* set includes user defined process specific type definitions. In order to support process evolution it also includes a number of predefined type definitions that become part of Full SLANG. The reasons for this are clarified later on in this section.

Activities are Kernel SLANG nets that encapsulate a set of logically related process operations. Each activity definition has an interface part and an implementation part. The activity interface contains a set of *starting events*, and a set of *ending events*. When one of the starting events occurs, an active copy of the activity is dynamically instantiated and its execution starts. Eventually, if the activity is well designed, one of the ending events occurs and the active copy execution terminates. The activity interface also contains a set of *input places*, that are the preset of starting events; a set of output places, that are the postset of ending events; and a set of *shared places*, that are connected directly to non-interface transitions.

An activity may include invocations of other activities. A Full SLANG process model is thus structured as a hierarchy of activities. This hierarchy has a unique root, which is a sort of “main activity” of the model. The root activity is not called by other activities and is automatically activated

⁴In the following, when it is clear from the context, we write only SLANG to refer to Full SLANG.

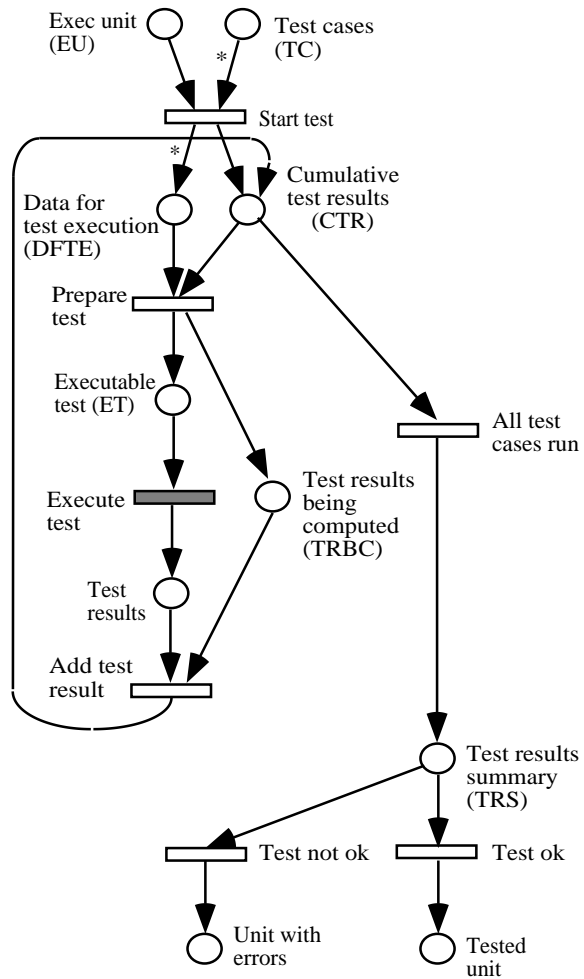


Figure 3: Test phase model using Kernel SLANG.

```

event StartTest (EU: ExecUnit, TC: set (TestCase);
                DFTE: set (ExecTest), CTR: TestSummary)
local test: TestCase, data: ExecTest;
guard
  forall test in TC: test->name = EM->name
action {
  DFTE = set();
  for (test in TC) {
    data = new ExecTest;
    data->name = EU->name;
    data->authorName = EU->authorName;
    data->sourceCode = EU->sourceCode;
    data->compilerUsed = EU->compilerUsed;
    data->execCodePath = EU->execCodePath;
    data->testData = test->testData;
    DFTE = DFTE + set(data);
  }
  CTR->testingUnit = EU->name;
  CTR->results = set();
  CTR->pendingTestsNb = count(TC)
}
end
-----
event PrepareTest (DFTE: ExecTest, CTR: TestSummary;
                  ET: ExecTest, TRBC TestSummary)
guard
  CTR->pendingTestsNb > 0
action {
  ET = DFTE;
  TRBC->testingUnit = CTR->testingUnit;
  TRBC->results = CTR->results;
  TRBC->pendingTestsNb = CTR->pendingTestsNb - 1
}
end
-----
event AllTestCasesRun (CTR: TestSummary; TRS: TestSummary)
guard
  CTR->pendingTestsNb == 0
action {
  TRS = CTR
}
end

```

Figure 4: Guards and actions for events Start test, Prepare test, and All test cases run.

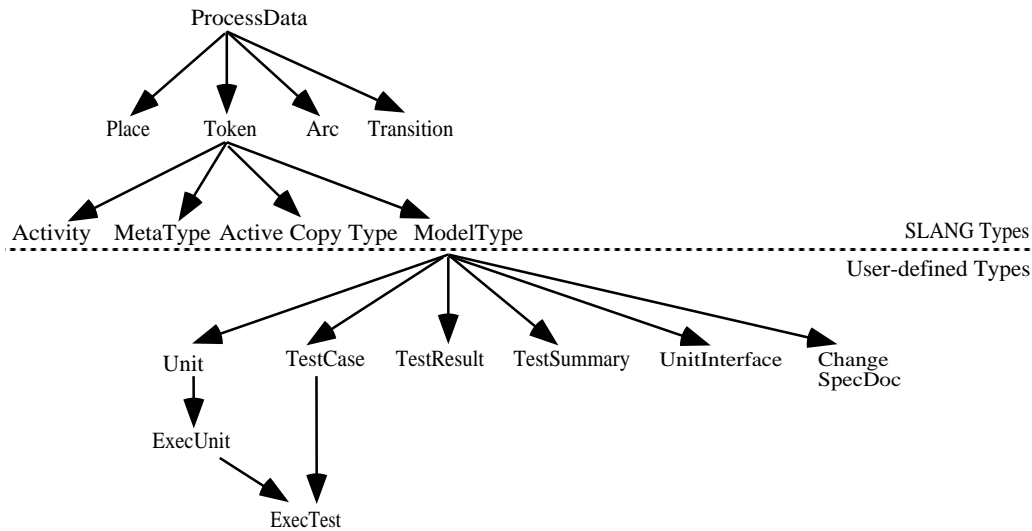


Figure 5: SLANG type hierarchy, including predefined SLANG types and user defined ones.

when the model is enacted. By using the activity construct, a process model can be viewed at different levels of abstractions. At each level, unnecessary details are hidden, so that the complexity of the process can be mastered. In addition, defining activities facilitates the reuse of process model fragments in different contexts.

The term “meta-process” is generally used to denote the process of changing a process model. To support modeling of the meta-process, process types must not only describe the data manipulated by the process, but also process models (i.e., data manipulated by the meta-processes). In order to support process evolution, SLANG provides reflective features that allow the process modeler to view and manipulate type definitions, activity definitions and active copies as any other data in the model, i.e., as tokens. This is achieved in SLANG by providing types `Activity`, `Metatype`, and `ActiveCopy` as subtypes of type `Token` in the *ProcessTypes* set, that respectively describe activity definitions, type definitions, and active copies. All these types are part of the hierarchy shown in Figure 5.

5.1 Full SLANG constructs

Full SLANG provides all Kernel SLANG constructs; in addition, it provides special features (special arcs and activity invocation, see below) whose semantics can be described using Kernel SLANG’s basic constructs.

5.1.1 Special arcs

Besides “normal” arcs, Full SLANG provides two other special kinds of arcs: *read-only* (represented by a dashed line) and *overwrite* (represented by a double arrow). A transition can read token values from an input place connected by a read-only arc in order to evaluate the guard and the action, but no token is actually removed. On the other hand, when an overwrite arc is used to connect a transition to an output place, the transition firing causes the following atomic sequence of actions. First, all tokens are removed from the output place; 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 contents of the output place.

5.2 Activity invocation

In Kernel SLANG, the process is described as a flat net. In order to model large processes, in Full SLANG it is possible to structure the process model as a hierarchy of activities. The execution of an activity may be invoked by another activity. An activity invocation basically consists of the invocation of the SLANG interpreter on the called activity. The SLANG interpreter is a tool and thus its invocation is modeled in Kernel SLANG by a black transition. Activities are executed asynchronously. When an activity invocation has to be executed, the calling activity spawns a new *process engine* to enact the called activity. That is, the child active copy runs concurrently with its parent, and with other sibling active copies. These active copies can share places, that are used for communication and synchronization purposes.

Full SLANG provides an *activity invocation* construct that involves a series of actions that have to be done when an activity is invoked. This construct is designed as a box that hides the activity implementation, showing only its interface. In the following we provide some hints on the implementation (in Kernel SLANG) of the activity invocation construct and discuss its benefits. For a thorough explanation the reader is referred to [BFG93a].

Suppose that activity *A* contains an invocation of activity *B*. Figure 6 shows how the activity call is expanded in a Kernel SLANG net, obtaining

the following behavior when activity B is invoked from a A 's active copy.

- An active copy of the activity B is created.
- A process engine for the new active copy of B is spawned. This operation is accomplished by means of a black transition that asynchronously executes the SLANG interpreter on B 's active copy.
- When B terminates (i.e., when an ending event fires) the process engine stops, yielding a result in the output place of the black transition in A 's implementation part. This result is then stored in the corresponding places of the calling activity A .

Let us emphasize some important aspects of the above execution scheme:

- An activity invocation is dynamically bound to the corresponding activity definition (and to the types used within such definition). This interpretive mechanism supports late binding and provides the basis for our solution to process evolution, as discussed in Section 8.
- The execution of an activity definition is carried out through a black transition invoking the *SLANG Interpreter* tool. That is, the mechanism used by the language to invoke the interpreter is the same used to invoke any other tool.
- Activities are executed asynchronously, i.e., the calling active copy continues executing while the newly created active copy is enacted by a process engine.
- The process engines executing different active copies may access shared places and the output places belonging to the parent activity. 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 (sub)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 active⁵.

⁵This feature is reminiscent of Ada's tasking. It does not require a semantic extension; it may be implemented in Kernel 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.

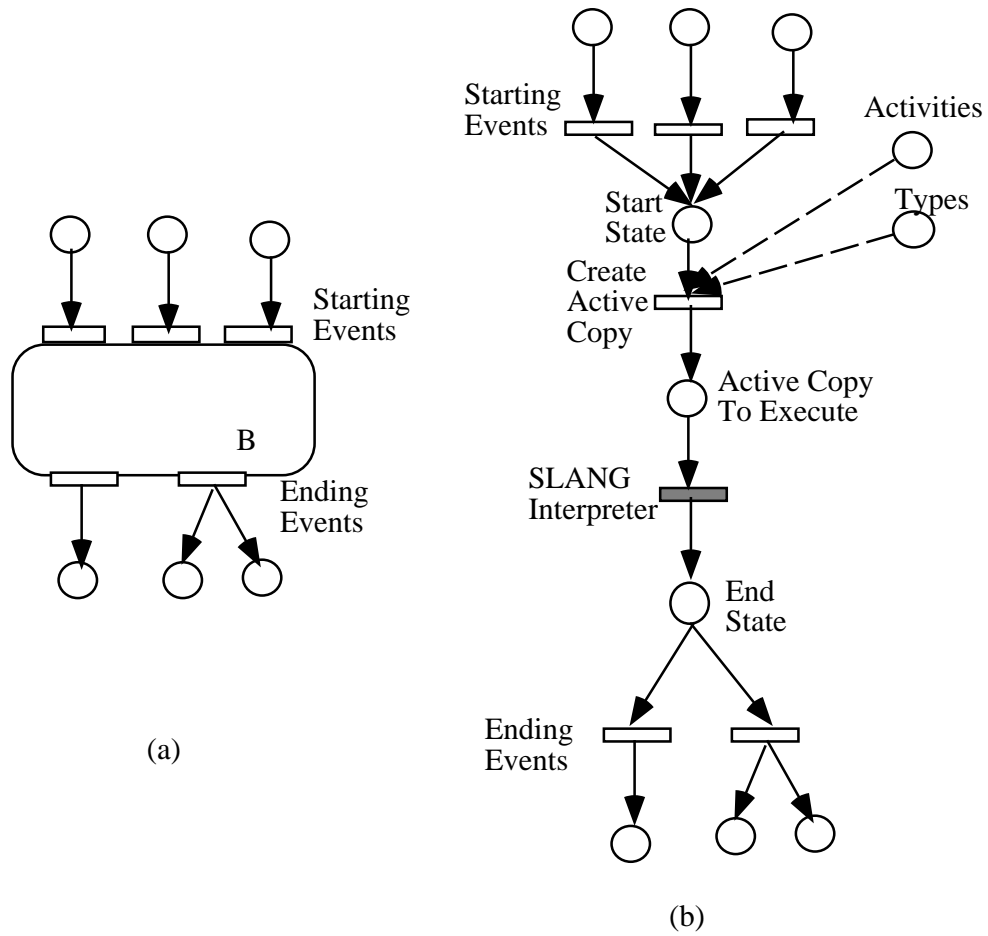


Figure 6: (a) Invocation of activity *B* within activity *A*; (b) corresponding series of steps in Kernel SLANG.

6 SPADE architecture

This section describes the architecture of the first prototype based on the SPADE concept, called SPADE-1. For a detailed description the reader can refer to [BBFL94]. 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 architecture of SPADE-1 is structured in three main components: the *user interaction environment*, the *process enactment environment*, and the *filter*, as shown in Figure 7:

- The user interaction environment is responsible for performing the interaction with users through tools in the environment.
- The process enactment environment is responsible for executing the process model. The execution is concurrently performed by a set of process engines that are dynamically created during enactment.
- The filter performs the communication between these two environments.

SPADE-1 supports multiple users distributed in a network of workstations. Each user interacts with the process through a set of integrated tools. Two classes of tools may be distinguished: black-box tools and structured tools. Black-box tools are viewed by the process engines as functions performed on some input and producing some output. The process has no control on the tool while it is working. A structured tool, instead, is decomposed into fragments that are visible through a programmatic interface. This makes it possible to achieve a fine-grained tool integration in the process model.

The interaction with tools and users is modeled in SLANG through black transitions and user places. Black transitions represent the asynchronous execution of a tool. For example, they may model the execution of a C compiler, or of a “send message” tool, that sends a message to a structured tool (e.g., ask an editor to open a file).

User places are used to capture relevant events that may occur in the external environment. These events — generated by the users by using, for example, a tool in the environment — are captured and transformed into tokens stored in user places. Depending on how the process is modeled, the presence of the token may produce the firing of one or more transitions in the net.

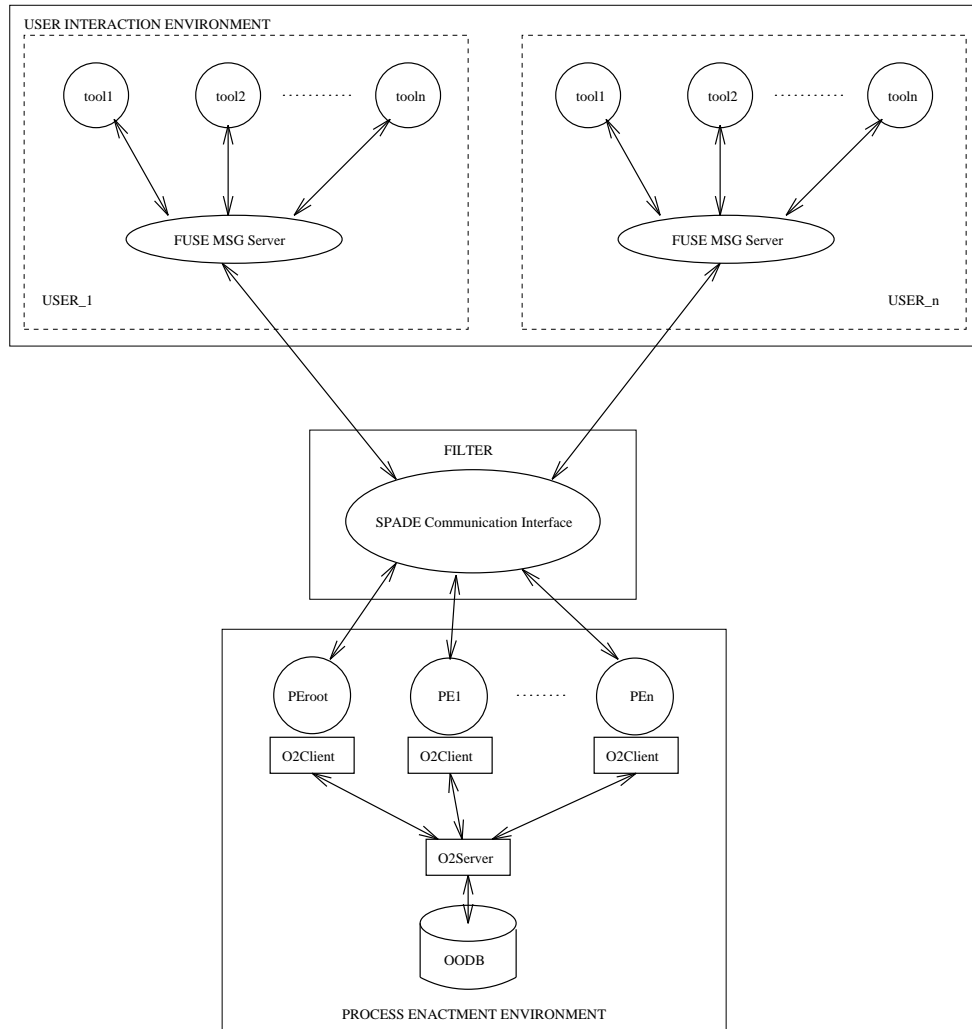


Figure 7: SPADE architecture.

The user interaction environment of SPADE-1 is based on a commercially available tool integrated environment, called DEC FUSE [DEC92]. DEC FUSE provides a set of structured tools to support basic development activities (editing, compiling, configuration building, etc.) In addition, DEC FUSE offers a utility, called EnCASE, to integrate new tools in the environment.

For data storage and management, SPADE-1 relies on the O_2 object-oriented database system [Tec92]. O_2 has a client-server architecture and provides a programmatic interface that is suitable to support SPADE evolution features. Each process engine is a client of the centralized object-oriented database server.

7 An example using Full SLANG

This section presents a SLANG process model for the running example of Section 3. Figure 8 represents a high-level view of the whole example. The part inside the dashed box corresponds to the implementation of activity **Change unit**, while places and transitions outside the dashed box represent the activity interface.

- **Design change** represents the production of a new design for the unit, according to the given specifications. It produces a new unit design and a unit interface definition.
- **Propose interface change** represents the activity of proposing the unit interface to unit users for approval. It is invoked only if the unit interface has been modified. Otherwise, the unit interface is considered to be automatically approved by transition **Interface not new**. Note that a copy of the interface definition is also produced as input to the test case generation.
- **Unit coding**, **Compile**, **Generate test cases**, and **Run tests** represent the usual activities of converting design into code, compiling, generating test cases and testing. In particular the testing activity definition is given by the Kernel SLANG net described in Figure 3.

Figure 9 represents activity **Propose interface change** definition. This activity has one starting event, represented by transition **Produce interface change proposal**, which is enabled as soon as the modified unit interface is available in place **Unit interface**. The firing of this transition produces a

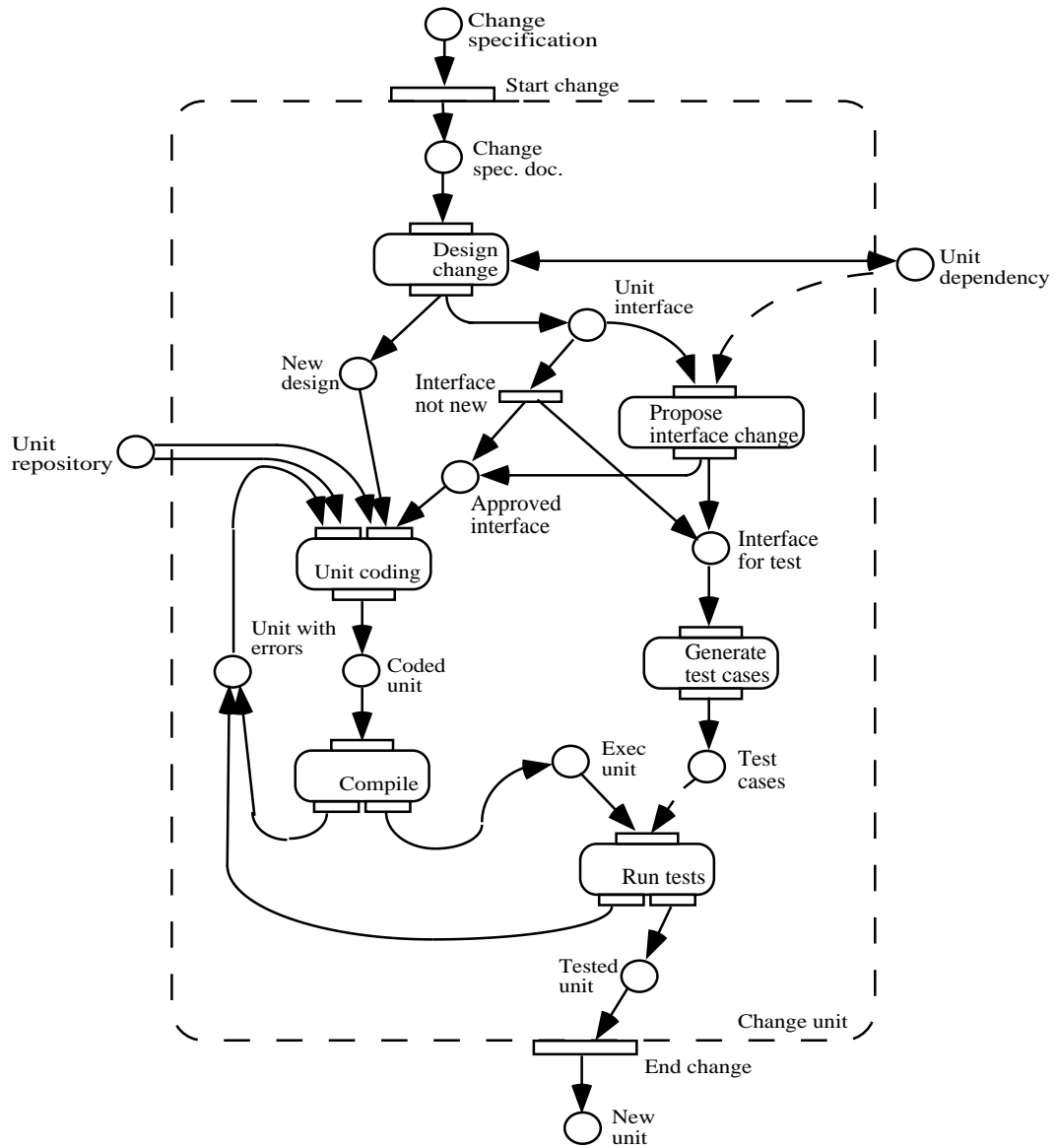


Figure 8: The SLANG definition of activity Change unit.

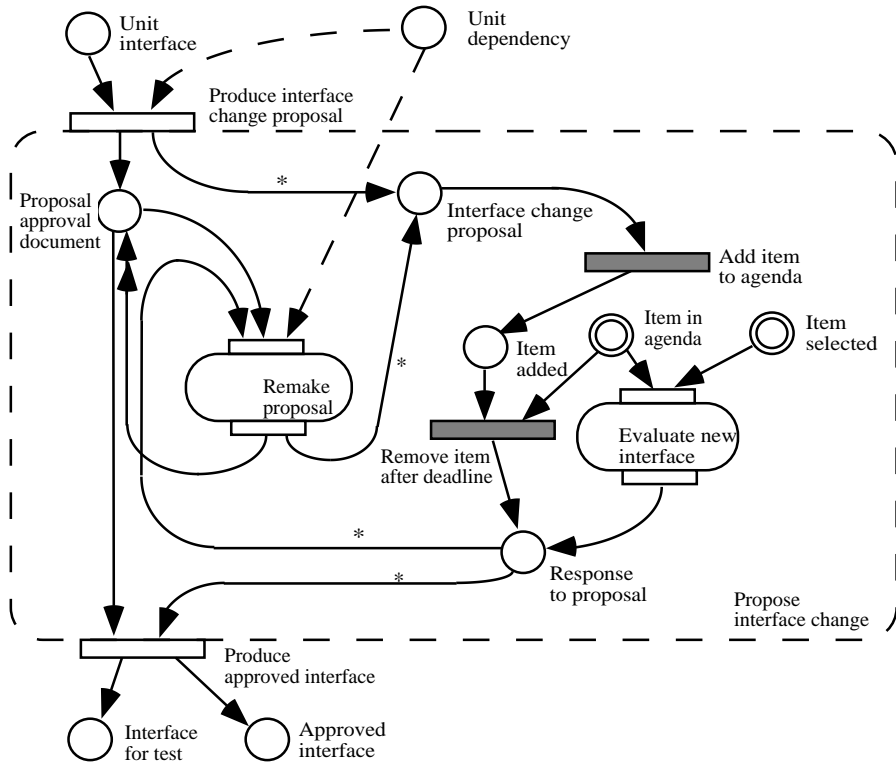


Figure 9: The SLANG definition of activity `Propose interface change`.

token containing the proposal description and the list and number of clients that are expected to approve the proposal in place `Proposal approval document`. It also produces in place `Interface change proposal` as many tokens as there are users of the unit (since this number is not known statically, a dynamically weighted arc is used). Each of these tokens represents an interface proposal document that is submitted to each of the unit users.

The rest of the activity is based on the assumption that the participants in the process interact with the process interpreter through a tool called agenda. This is a service level integrated tool that manages a to-do list that is displayed on the screen. The tool provides two main services:

- *Add item* inserts a given task description (passed as a parameter) into the to-do list.

- *Remove item* removes a given task description (passed as a parameter) from the to-do list.

In addition, the agenda tool produces state change notifications when some relevant event occurs:

- *Item selected* produced each time an item is selected by a user from the to-do list.
- *Item in agenda* produced each time an item has been inserted in the to-do list.

Black transition **Add item to agenda** represents the execution of a tool that invokes the agenda service *Add item* with parameter “Examine interface change proposal”. This transition fires once for each token in **Interface change proposal** and each firing causes a new task to be inserted in the corresponding user’s agenda. As soon as the service has been invoked, the black transition execution terminates producing a token in place **Item added**. After the task has been effectively inserted in the agenda, the tool reports this fact and a token is inserted in the user place **Item in agenda**.

When a user selects such task from the list displayed in his/her agenda, a token is produced in the user place **Item selected**. Such a token enables activity **Evaluate new interface**. An instance of such activity is created for each user of the changed interface: its task is to communicate the proposal to the unit clients and collect the responses (either acceptance or motivated reject).

At this point of enactment, we have an active copy of **Change unit**, an active copy of **Propose interface change**, and as many active copies of **Evaluate new interface** as the number of interface clients, running in parallel.

Answers from the users are accumulated in **Response to proposal**, until their number equals the number of users stored in the place **Proposal approval document**. In such a case there are two possibilities:

1. All the answers indicate acceptance: the ending event **Produce approved interface** is enabled.
2. There is at least one reject and thus activity **Remake proposal** is enabled. Such activity is concerned with the revision of the new interface based on the comments of the users. It is performed by the designer who is responsible for the unit interface and produces the same outputs as the starting event.

8 Process evolution in SPADE

Processes are by no means static entities. Since processes are long lived entities, they must be allowed to evolve, even during enactment, in order to cope with a variety of change requests. Changes may arise, for example, because better ways are found for doing things, because of restructuring in the organization, or because of technological improvements.

Process evolution is a process too, and thus it can be modeled and enacted as any other process. SLANG provides reflective language features which allow process models to be treated as process data and thus process evolution can be described as part of the process. In particular, type descriptions, activity definitions and active copies are tokens in a SLANG net, of type `Metatype`, `Activity`, and `ActiveCopy`, respectively. In this scheme, process models and process states can be accessed both as code to be executed and as data to be modified (i.e., as regular tokens).

For example suppose that while activity `Propose interface change` of Figure 9 is being enacted, a new client decides to use the unit whose interface is being discussed. According to the specified process, the new client would ignore that the interface is currently being discussed, and would not be asked to approve or reject the proposed change. To overcome this problem, we would like to redefine activity `Propose interface change` as illustrated in Figure 10. In the new definition the ending event `Produce approved interface` and the starting event of activity `Remake proposal` are enabled only if the list of users recorded in the `Proposal approval document` corresponds to the current list of users (deduced from `Unit dependency`). If the lists differ, the approved interface is suspended until the new user also approves it. This is accomplished by transition `Make proposal for new user`, that compares the present list of users of the unit, taken from `Unit dependency` with the list of users that was copied in place `Read depend and last check time` when the `Produce interface change proposal` activity was initiated. If there are new users, they are also sent the interface proposal, in place `Interface change proposal`, and the counter in the `Proposal approval document` is incremented. If there are no new users, the activity ends with an ending event `Produce approved interface`, that produces the approved interface token in places `Approved interface` and `Interface for test`.

All activity definitions in a SLANG process model are contained in a place called `Activities` (of type `Activity`) and all type descriptors, in a place called `Types` (of type `Metatype`). We assume that the new definition of

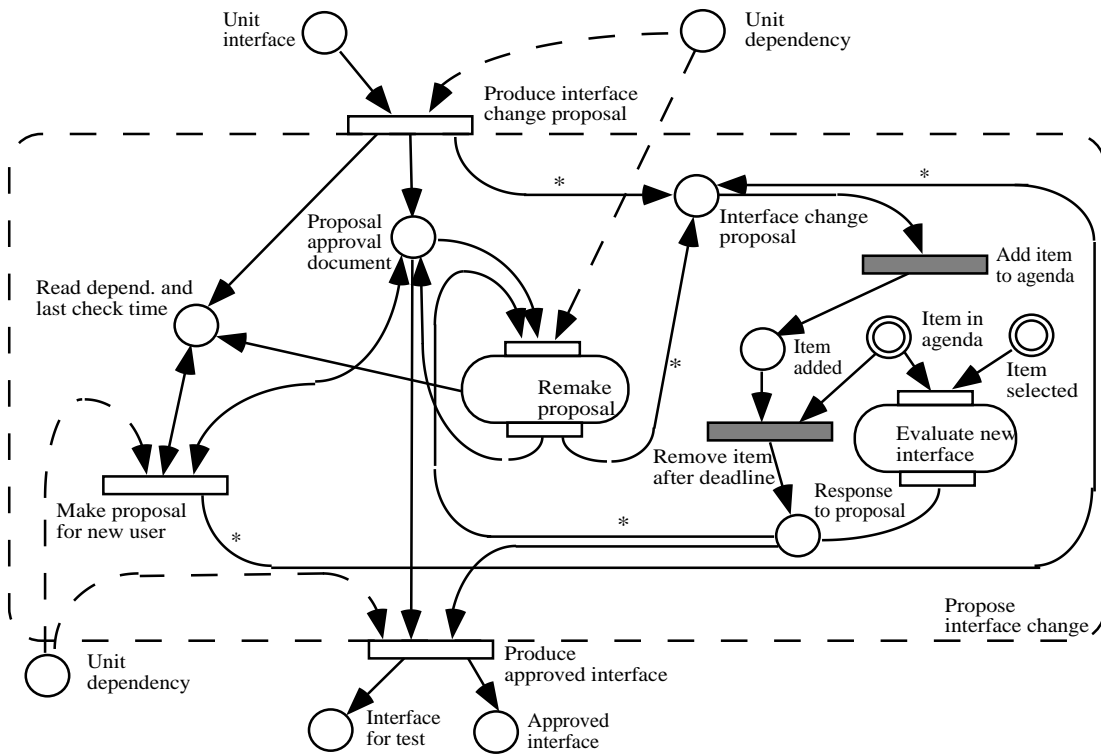


Figure 10: A new definition of activity `Propose interface change`.

activity `Propose Interface Change` (Figure 10) has been edited and generated as a token by a SLANG process (not described here). We call the time at which the editing of an activity definition terminates the *Change Definition Time* (CDT). This time may be different from the *Change Implementation Time* (CIT) that is the time at which the change is applied to the running process.

Several different strategies may be identified to apply a change to a running process. On one extreme, we identify a *lazy strategy* in which the new activity definition does not affect running active copies of the activity, but is only used in new instantiations of the activity. On the other extreme, an *eager strategy* may be necessary. According to the eager strategy, active copies under enaction have to be suspended, producing tokens of type `ActiveCopy`, that contain the activity state. Then the activity definition is changed and the state has to be modified consistently. Finally, when new

active copies are ready, their execution is restarted. The environment provides the basic mechanisms to support this “on the fly” change, but no rules or methodology are provided on how this state adaptation has to be done. In fact, the way this process state migration is done strongly depends on the specific characteristics of the process and of the modification applied.

Other policies can be implemented. For example, instead of suspending all the active copies affected by the modification of an activity definition, the process manager might decide that the active copies that were created before a given date must not be suspended. 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. A SLANG meta-process can involve the modification of type definitions as well. Types are modified by means of a net fragment accessing place `Types`.

In summary, 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 specifiable in the meta-process.

9 Conclusions and future work

This paper provides an overview of the SPADE project. In particular, it presents a case study that highlights the basic features provided by SLANG to support modeling, enactment, and dynamic process evolution. The architecture of the first prototype of SPADE, called SPADE-1, is also briefly described.

The main features of SPADE are the following:

- It provides a process modeling language.
- It provides an interpreter based on O_2 , an object-oriented database for the storage of process data (including software artifacts and process models).
- It uses DEC FUSE, a message-based integrated tool environment, to achieve fine grained tool integration in the process.
- It supports distributed development activities.
- It supports specification and enactment of the software process as well as meta-process.

In parallel with the design of SPADE-1, the practical use of SLANG has been investigated by modeling a large portion of a real software production environment. The modeling activity has provided valuable insight into the process and has shown inconsistencies and incompleteness of existing process definitions [Pic93]. SPADE-1 has been used to run a version of the case study reported in this paper.

Future activities include:

- Improving the user interface, by adding user oriented tools such as a graphical editor, an enactment monitor, a fully functional agenda, etc.
- Continuing the experimentation with real-life test cases.
- Introducing the possibility of writing the activity definitions in a declarative style.
- Adding measurement-oriented features to the language.

Acknowledgements

We wish to thank the DEC Engineering Center in Gallarate supporting this work. In particular, we are indebted to Marco Braga for his constructive help. The implementation of SPADE-1 is mainly due to the work of Paolo Battiston, Guido Galli de' Paratesi and Marco Signori.

References

- [ABGM92] P. Armenise, S. Bandinelli, C. Ghezzi, and A. 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.
- [BBFL94] S. Bandinelli, M. Braga, A. Fuggetta, and L. Lavazza. The Architecture of the SPADE-1 Process-Centered SEE. In *3rd European Workshop on Software Process Technology*, Grenoble (France), February 1994.
- [BEM91] N. Belkhatir, J. Estublier, and W.L. Melo. ADELE 2 - An Approach to Software Development Coordination. In Alfonso

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

- [BF93] S. Bandinelli and A. Fuggetta. Computational Reflection in Software Process Modeling: the SLANG Approach. In *Proceedings of the 15th. International Conference on Software Engineering*, Baltimore, Maryland (USA), May 1993.
- [BFG93a] S. Bandinelli, A. Fuggetta, and C. Ghezzi. Process Model Evolution in the SPADE Environment. *To appear in IEEE Transactions on Software Engineering, Special Issue on Process Evolution*, December 1993.
- [BFG93b] S. Bandinelli, A. Fuggetta, and S. Grigolli. Process Modeling-in-the-large with SLANG. In *Proceedings of the 2nd International Conference on the Software Process*, Berlin (Germany), February 1993.
- [BFGG92] S. Bandinelli, A. Fuggetta, C. Ghezzi, and S. Grigolli. Process Enactment in SPADE. In *Proceedings of the Second European Workshop on Software Process Technology*, Trondheim (Norway), September 1992. Springer-Verlag.
- [BK91] N. Barghouti and G. Kaiser. Scaling up rule-based software development environments. In Axel van Lamsweerde and Alfonso Fuggetta, editors, *Proceedings of ESEC 91-Third European Software Engineering Conference*, volume 550 of *Lecture Notes on Computer Science*, Milano (Italy), October 1991. Springer-Verlag.
- [BPR91] R.F. Bruynooghe, J.M. Parker, and J.S. Rowles. PSS: A system for Process Enactment. In *Proceedings of the First International Conference on the Software Process*. IEEE Computer Society Press, 1991.
- [DEC92] DEC. *DEC-FUSE manual*. Digital Equipment Corporation, 1992.

- [Fer93] C. Fernström. PROCESS WEAVER: Adding Process Support to UNIX. In *Proceedings of the 2nd International Conference on the Software Process*, Berlin (Germany), February 1993.
- [GMMP91] 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.
- [Gru91] V. Gruhn. *Validation and Verification of Software Process Models*. PhD thesis, University of Dortmund, 1991.
- [HSO90] D. Heimbigner, S. M. Sutton, and L. Osterweil. Managing change in process-centered environments. In *Proceedings of 4th ACM/SIGSOFT Symposium on Software Development Environments*, December 1990. In ACM SIGPLAN Notices.
- [JLC92] L. Jaccheri, J. Larsen, and R. 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.
- [Kel91] M. Kellner. Software Process Modeling Support for Management Planning and Control. In *Proceedings of the 1st. International Conference on the Software Process*, Redondo Beach CA (USA), October 1991.
- [KFF⁺91] M. Kellner, P. Feiler, A. Finkelstein, T. Katayama, L. Osterweil, M. Penedo, and D. Rombach. ISPW6 Software Process Example. In *Proc. of the 1th. International Conference on the Software Process*, Redondo Beach CA (USA), October 1991.
- [Pic93] G. Picco. Modeling a real software process with slang. Internal Report RI93059, CEFRIEL, Via Emanuelli, 15 - 20126 Milano (Italy), June 1993.
- [PS92] B. Peuschel and W. 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.

- [Rei85] W. Reisig. *Petri Nets: an Introduction*. Springer Verlag, New York, 1985.
- [SIK93] M. Suzuki, A. Iwai, and T. Katayama. A Formal Model of Re-execution in Software Process. In *Proceedings of the 2nd International Conference on the Software Process*, Berlin (Germany), February 1993.
- [Tec92] O₂ Technology. *The O₂ manual*. 1992.