POLITECNICO DI MILANO
Facoltà di Ingegneria
Dipartimento di Elettronica e Informazione

# The Design and Implementation of SPADE-1 2.0

Relatore:
**Prof. Alfonso Fuggetta**
Correlatori:
**Prof. Carlo Ghezzi**
**Dott. Sergio Bandinelli**

Tesi di Laurea di:
**Antonio Carzaniga**
matr. n° 592572
**Giovanni Vigna**
matr. n° 595296

Anno Accademico 1993-1994

# Acknowledgements

*First, we want to thank Prof. Carlo Ghezzi for his experience and his confidence. He has made our work possible.*

*Moreover, we want to thank Prof. Alfonso Fuggetta for his constant support and guidance during the development of our work. His ideas and his contributions to Software Process research are the basis of our work.*

*Furthermore, we want to thank Ing. Gigi Lavazza for his useful advices and for providing an original point of view in every circumstance.*

*A very special tribute goes to Dott. "Sir" Sergio Bandinelli for the invaluable help and the contributions to our work. He is an authority in the Software Process field, but, what is most important, he is a great friend to us.*

*This thesis work has been developed during the 6th Master in Information Technology at CEFRIEL. We thank everyone at CEFRIEL for resources, technical support and a friendly environment.*

*We could never forget the companions of the SE group. Piero "Zank", Elia "brauser", Gian Pietro "GP", Edoardo "il nonno", Giorgio "Gio", Beppe "Peppe", Gianluca "op shutdown", Alessandro "Parimba", Rosamaria "rm" e Giulio "gallo", not to mention Toti, AD, Ago, Walter, Genius, Piarullo, Rocco, Mimma, Ger, Zang and all the other guys at CEFRIEL. We shared heavy work and light fun.*

*A special thank from Giovanni to Luisa for the loving support, and, from both of us, to Sebastiano for proof-reading of this thesis.*

# Contents

# Chapter 1

# Introduction

A *software product* is a complete set of computer programs, procedures and associated documentation and data designated for delivery to a user [ISO91]. Software products are the result of cooperative processes, involving several persons, in which a series of software engineering activities are carried out with the help of various software tools. The set of such activities, persons and tools, together with rules, policies, and used resources (including software artifacts, documents, etc.) constitutes a *software process*.

Despite its fundamental role, little attention is paid to the assessment and improvement of the software process. The process is often left implicit, ambiguous or incomplete [ABGM92].

Many efforts in the software process research field aim at providing formalisms to *model* the process. An explicit process representation provides a general framework in which persons can cooperate and communicate effectively, improving the process and therefore the software product quality. The process representation includes technical and management activities. Both are needed to produce quality products in a predictable time with limited resources [GJM91].

In order to model a software process, an adequate notation is required. If this notation is based on formal grounds, it is possible to analyze and verify the process, using rigorous reasoning. Moreover, if the notation is executable, it is possible to use the process representation as the "heart" of a Process-centered Software Engineering Environment (PSEE), providing automatic guidance and support to the software engineering activities involved in software development. The process model is "enacted" by the PSEE interpreter, supporting the cooperation among individuals, monitoring process state, and automating some activities. The PSEE also keeps all data of the process in a persistent repository. The concurrent access to these data must be regulated by the process model.

## 1.1 SPADE

The SPADE project aims at developing an environment for Software Process Analysis, Design, and Enactment [BFGG92, BBFL94]. SPADE defines and implements a formal process modeling language and an integrated process-centered environment.

The process language of the SPADE environment is called SLANG (*S*PADE *LANG*uage) [tea93]. SLANG is based on a high-level Petri net formalism, called ER nets [GMMP91]. SLANG offers features for process modeling, enactment, and evolution [BFG93b]. 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 [BFG93a]. A SLANG process model can be hierarchically structured as a set of activities. An activity encapsulates a set of logically related process operations and it is described by a net that may include invocations to other (sub)activities.

Process data (documents, code, plans, test cases, etc.), are represented by tokens which are structured in an object-oriented fashion [BBFL93]. Places are viewed as distributed persistent object containers. Places are typed, and every place may contain only tokens of its associated type. Transitions represent events. A transition fires when its *guard* (a boolean expression) is satisfied by tokens in its input places. The firing causes the execution of an *action*. The topology of the net describes precedence relationships among events, parallelism, and conflict scenarios.

SLANG process models are dynamic entities. They can undergo changes. Moreover, SLANG has reflective features (activities are themselves data of the model), that make it possible to describe SLANG process evolution by means of SLANG models [BF93].

The interaction with process agents is achieved using special transitions and places in the net. Process artifacts, including process models, are kept and maintained in an object-oriented database.

SPADE-1 is an implementation of the SPADE concept. The design of SPADE-1 has been centered on the principle of separation of concerns between process model interpretation and the user interaction environment. Actually, SPADE-1 architecture is structured in three different layers: the Process Enactment Environment (PEE), the User Interaction Environment (UIE), and the SPADE Communication Interface (SCI).

The PEE includes facilities to execute a SLANG specification. SLANG activities are concurrently interpreted by SLANG Interpreters. Process artifacts (including the process model itself) are stored and managed using the $O_2$ OODBMS [Deu91].

The UIE is responsible for performing the interaction with users through tools in the environment. The UIE is viewed as a set of service-based tools which provide a programmatic interface. Entities in the PEE can request services to tools in the UIE.

Tools can notify the PEE of relevant events in the user environment.

The SCI is a filter between the PEE and the UIE. It provides communication mechanisms based on the message passing paradigm, together with services like integrated service-based tools invocation and dynamic configuration of filtering operations.

## 1.2 Related work

Much research effort is being carried out in the software process field. A number of different works concerning both process modeling and model execution have been presented. We can classify these projects according to the formalisms underlying the process description.

One approach uses programming languages such as APPL/A [HSO90]. This formalism allows a precise description of the "control flow". The process can be modeled as a procedural program. The use of functional languages can also emphasize the hierarchical relationships among activities. However, in these process definitions, it is difficult to express concurrency and non-determinism.

Others adopt the opposite approach. In particular, rule based systems [PS92, BK91] do not define the sequence of actions. The process consists of a set of goals and some rules and constraints. Rules represent actions that require some *pre-conditions*, the execution of a rule asserts new facts (*post-conditions*). The enactment system tries to satisfy the goals by applying some rules, that fulfill part of the goals with their post-conditions. The pre-conditions of these rules become new goals, so the system tries to satisfy them recursively (*backward chaining*). If the pre-conditions of a rule are satisfied by the state of the process, then the action associated to the rule is executed, the post-conditions are asserted and, recursively, the rules that have been enabled by the post-conditions are fired (*forward chaining*). This approach aims at describing a process in terms of its tasks. The process modeler defines "what" the process should do, instead of specifying "how". In this view, it is difficult to track down the execution threads. Thus, it is difficult to understand the whole process model. Moreover, concurrency is not an explicit part of the model.

Several process modeling efforts have yielded 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. Although Petri nets extensions are different from each other, they all deliver readable descriptions. The concurrency and synchronization issues are straightforward in such models.

Other studies concerned active database extensions [BEM91], abstract specification formalisms, including state charts [Kel91] and attribute grammars [SIK93]. An introduction and comparison of existing approaches can be found in [ABGM92].

## 1.3 Contribution of this thesis

Figure 1.1 presents SPADE-1 history. The SPADE project started in 1991 at CEFRIEL. First experiences and feasibility studies based on database technologies were carried out in 1991-1992 [Fer92]. In 1992-1993 a first prototype of the SPADE environment, called SPADE-1 1.0, was developed at CEFRIEL [BdPS93]. The prototype had several restrictions, regarding architectural and SLANG interpretation issues. In the same period, the SLANG language was refined [Lip93] and used in modeling examples of industrial processes [Pic93]. Tool integration problems were analyzed in [Par94]. In 1993-1994 the work on SPADE has evolved along three main threads. One thread focused on the development of tools to be integrated in the SPADE-1 environment [GZ94]. Another thread continued the experience with industrial processes [Mas93].

Our work concentrated on the analysis of the original SPADE-1 concepts and general requirements, in order to improve and extend the environment. In particular, we focused on re-designing the SPADE-1 architecture, we added integration facilities and reformulated and enriched the SLANG language definition. The result of our work is a new pre-industrial prototype, called SPADE-1 2.0, with full, extended integration facilities and improved SLANG interpretation mechanisms.

Taking into account the background of the previous works, we designed and developed the whole system *from scratch*, with openness and modularity in mind.

*This thesis presents the design and implementation of SPADE-1 version 2.0, a complete, usable Process-centered Software Engineering Environment which satisfies the SPADE project principles, providing efficient mechanisms for concurrent interpretation of fully featured SLANG process models, a distributed and configurable architecture, and powerful mechanisms for both data integration and control integration of tools.*

In particular, our work addresses the following issues.

**Distributed architecture** We assume process agents work in a local area network. Thus, SPADE-1 2.0 has a distributed architecture that controls SLANG process models execution and the interaction with tools running on different machines. In order to

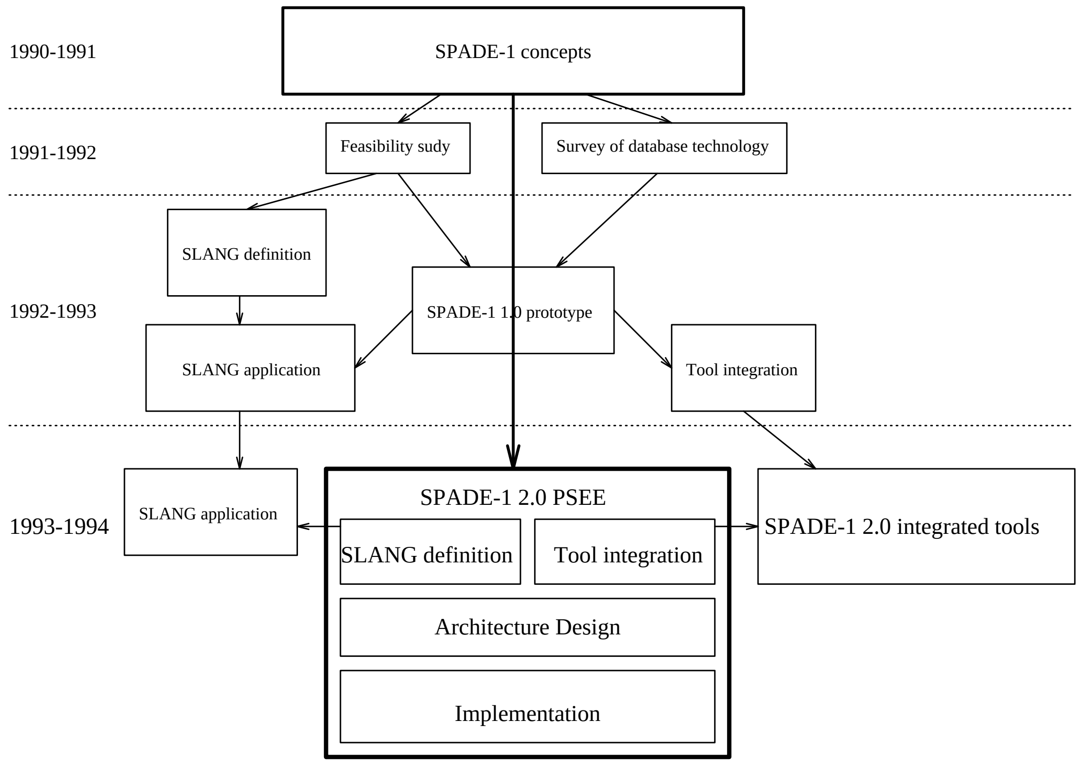


Figure 1.1: SPADE-1 history.

support SPADE-1 2.0 distributed architecture, some general, high-level communication mechanisms have been developed. The whole environment can be configured in order to optimize computational resources. Any process of the system can be allocated on a different host all over a network of workstations. SPADE-1 2.0 provides a graphic interface towards the system administrator, which allows to monitor process model execution.

**SLANG Interpretation** In SPADE-1 1.0 SLANG interpretation algorithm is carried out in a single transaction in the $O_2$ OODBMS, greatly reducing parallelism. In addition, each SLANG interpreter instance is an $O_2$ client. Thus, the maximum number of concurrently executed interpreters is limited by the number of $O_2$ clients (licenses) available.

In SPADE-1 2.0 multiple SLANG interpreter instances concurrently access the process data. Critical sections have been reduced. SPADE-1 2.0 poses no limitations to the number of concurrent SLANG interpreter instances. The system can be configured to run even with a single $O_2$ client.

In SPADE-1 1.0 SLANG interpretation is achieved using run-time queries to the $O_2$ data base. This leads to lack of performance and dynamicity.

In SPADE-1 2.0 the interpretation algorithm uses *compiled* code, improving performance and expressiveness.

**Tool integration** SPADE-1 1.0 permits the integration of service-based tools in a customized version of the DEC FUSE environment. Integrated tools can exchange only atomic data.

SPADE-1 2.0 permits integration of tools out of any particular integration environment, providing high-level, easy-to-use libraries for integrated tools development. Mechanisms have been developed to allow tools to exchange complex ($O_2$) objects, providing effective data integration. SPADE-1 2.0 is no more bound to FUSE. It can integrate different Software Development Environments like Sun Tooltalk or HP SoftBench, providing the appropriate filter.

In SPADE-1 1.0, message filtering between the process enactment environment and the user environment is limited and quite static.

SPADE-1 2.0 implements a multi-cast mechanism for notification messages, allowing dynamic configuration of the filtering mechanism by means of patterns on source and type of the message.

## 1.4 Structure of this thesis

This thesis is structured as follows. Chapter 2 presents the SPADE principles, and the requirements that stem from such principles. Chapter 3 describes the main features of the SLANG language. Chapter 4 presents the complete SPADE-1 2.0 architecture, following an architectural description framework. Chapter 5 presents the Process Enactment Environment, in which the process model is executed. Chapter 6 describes in detail the interpreter of the SLANG language. Chapter 7 describes how communication between the users and the process is achieved, presenting the SPADE Communication Interface. Chapter 8 describes the User Interaction Environment, and the tool integration mechanisms which allow control and data integration in SPADE-1 2.0. Chapter 9 summarizes the work done and outlines future research issues.

# Chapter 2

# The SPADE environment

The SPADE project aims at defining and developing a Process-centered Software Engineering Environment (PSEE). Process models in SPADE are specified via a process modeling language, called SLANG. The SPADE environment supports software engineering activities by enacting the software process model, which defines human tasks and the interaction between humans and software development tools.

The SPADE project is based on some basic principles. First of all, the process specification is formal, in order to allow process analysis based on rigorous grounds and enactment of the process model. Second, process evolution can be described as part of the process itself. Process data and the process model are described and stored in an homogeneous way. Third, SLANG, the SPADE language, provides large spectrum modeling capabilities. SLANG offers high-level constructs (such as the activity construct) to model the process in the large and also allows the process model to describe fine-grained process, at the level of tool integration. Fourth, there is a clear distinction between mechanisms and policies. SPADE offers mechanisms for the description and the enactment of the process model without embedding any particular policy in the environment. Policies, instead, have to be defined as part of the process model. Thus, SPADE is an open environment: software development as well as resource management can be defined using the SLANG basic constructs. Fifth, SPADE uses and assesses new technologies, including OODBMS, environments for tool integration, and software development environments.

These principles inspired the first implementation of the SPADE environment, called SPADE-1. In addition to the general principles, SPADE-1 is based on a set of functional and non-functional requirements, which are presented in this chapter.

## 2.1 Functional requirements for SPADE-1

### 2.1.1 The SLANG modeling language

**Petri nets** A formalism based on ER-nets (a special kind of high level Petri nets) has been chosen as the basis of a SLANG activity definition. Places are data container, transitions correspond to actions. In addition to the plain ER-net entities, some process-oriented constructs are supported by SLANG.

**SLANG activities and abstract data types** The SPADE language must provide facilities for defining both activities and data types manipulated in the process. SLANG abstract data types (ADT) are organized in a class hierarchy with a "is-a" relationship. Each class defines the data structure and a set of methods. Any object manipulated by the process is an instance of a SLANG ADT, being Token, a pre-defined class, the root of the class hierarchy.

**Reflective features** In order to cope with process model evolution, SLANG provides powerful reflective features. SLANG defines activities and data types (the model definition) like any other model data. In this way model definitions can be manipulated during the enactment. Thus the process describing the changing of the model (the meta-process) can be specified just like any other process.

**Interaction with process agents** As far as the interaction with process agents is concerned, SLANG must deliver some I/O primitives. These results in some very few orthogonal mechanisms. Their combination must tackle the problems of control and data integration, manage asynchronous events and provide access to low granularity services. No particular tool-dependent protocol must be "hard-wired" in the language, the forementioned mechanisms must accept tool descriptions or service-dependent information as data.

**Process modeling "in the large"** SLANG must support primitives for the definition and composition of activities which allow process abstraction and process modeling in the large. In addition it must be powerful and expressive in order to specify complex relationships with a rich and clear semantics.

**Process modeling "in the small"** The process modeling activity must support the description of detailed actions, namely transition actions, in a straightforward manner. There is a trade-off between the good features obtained by modeling with SLANG nets, and simplicity of a usual programming language. The process modeler should be able to choose how much of the "model" is programmed by the net and how much the single actions do.

**Time constraints** A software process model is a real time system. Even though they are not necessarily critical, a software process language must deliver some mechanisms for describing time constraints.

### 2.1.2 Tool integration

**Tool set** SPADE-1 must be an open system. It must be able to integrate new tools specifically developed for the environment as well as existing tools available from the market.

The level of integration depends on the tool characteristics and on the specific process model. By level of integration we mean the granularity of the services provided by the tool. From this perspective we distinguish two main classes of tools. *Black-box* tools implement one service, the service correspond to the entire tool execution. `vi` and `cc` are black-box tools. *Service-based* tools export a set of services directly accessible by the process model through a programmatic interface. DEC FUSE (DEC Friendly Unified Software Environment) is an integrated tool environment for software development [Dig91b]. DEC FUSE offers a set of integrated service-based tools.

SPADE-1 provides a set of simple elementary mechanisms that can be combined to obtain the desired level of integration. Both the control dimension and the data dimension of integration are tackled.

**Tool integration** Control integration is granted by a message passing mechanism. Messages originated by the process model are service requests addressed to a particular tool. Events of any interest generated by tools or users are notified by means of messages.

Data integration is achieved by sharing complex objects in the data repository. The mechanism supporting data integration does not depend on the control integration granularity.

### 2.1.3 Process Enactment

**Process interpreter and SLANG** The SLANG interpreter is responsible for the execution of SLANG activities. Each interpreter instance executes one activity. At a given time several threads may concurrently execute different interpreter instances. SPADE-1 must provide a run-time support to manage activity instances and common persistent data.

**Dynamicity of types and activity definitions** The key issue for a system supporting process evolution is the ability of binding the very latest versions of types and activity definitions during execution. It is then important to make the process engine independent from any particular activity or type. Both activity and type definitions must be parameters for the SLANG interpreter.

**Persistency of model data** It is the SPADE-1 environment which is made responsible for keeping data in a stable storage. Every object manipulated by the process is implicitly persistent. The persistency mechanism is transparent for the process modeler.

## 2.2 Non-functional requirements

### 2.2.1 Using state of the art technology

The SPADE project aims at using state of the art technologies in the software engineering field. Examples are the $O_2$ OODB and the development environment DEC FUSE.

**$O_2$ data base** The SPADE-1 system relies on the $O_2$ object oriented data base for all the data management and persistency support. The SPADE-1 process engine is built on the top of $O_2$.

**DEC-FUSE** DEC-FUSE is a tool integration environment based on message passing. Tools already integrated in DEC FUSE may be used in SPADE-1 by exporting towards SPADE-1 the control messages.

### 2.2.2 System distribution

There are several good reasons for a distributed system. A natural approach in building a process environment is to replicate the physical structure of a software development environment, which is inherently distributed. Secondly a central process engine would be a performance bottleneck, while in a distributed scenario, computational resources can be much better exploited.

### 2.2.3 Configuration and system management

**Installation and configuration** SPADE-1 must be configured to run on a local network of workstations. Additional configuration parameters may be tuned to match the hardware and software configuration of each user. In particular it is necessary to configure the number of available $O_2$ licenses.

**Monitoring** In order to have an overall view of what's going on in the process for management purpose. The state of the running activities and the system resources allocation should be presented to the process supervisor. A global monitor should display the running active copies, the computational resources and the invocation dynamic tree, a local monitor should display the evolution of one single activity, showing the firing sequence.

### 2.2.4 Performances

The environment should react reasonably fast to user actions. An obstacle on the way to fast execution is the need for high dynamicity of models. The more the system is dynamic with respect to data and activity definitions, the more run-time type checking and interpretation must be performed. And these are time consuming tasks.

# Chapter 3

# SLANG language

SLANG (Spade LANGuage) is the Process Modeling Language (PML) of the SPADE-1 environment. It has been designed to formally describe software processes.

SLANG is based on a high-level Petri net formalism called ER nets [GMMP91]. The process modeler is provided with a set of process-oriented constructs, built on top of this formalism.

SLANG features can be summarized as follows:

- Process models can be structured in a modular way using the *activity* construct. An activity may include invocations to other activities.
- Process data are described in an object-oriented style, using classes organized in a generalization hierarchy.
- Basic mechanisms to deal with time issues in the process are provided[1].
- Interaction with tools can be uniformly described as part of the process.
- In order to support process evolution, SLANG provides reflective features, e.g., activities can be manipulated as data by other activities. This allows the meta-process to be specified as part of the process.

A process model in SLANG has two parts: one describing types and the other describing activities:

$$SLANGModel = (ProcessTypes, ProcessActivities)$$

[1]A complete discussion of time issues, including time constraints on process execution will be provided in future versions of the language.

*ProcessTypes* is a set of abstract data type (ADT) definitions organized in a generalization hierarchy, following an object-oriented style. The ADT definitions are written using the SLANG object-oriented type system, presented in Chapter 3.1, and provide the description of all data used in the modeled process.

*ProcessActivities* is a set of activity definitions. An activity definition is basically a high-level Petri net where each place, arc, and transition has been augmented with additional information. Activity definitions are introduced in section 3.3. An activity definition may contain *invocations* of other activities. The invocation relationship among activities defines a hierarchy of activity definitions, which may be seen as an activity breakdown structure. At the top of the hierarchy there is one activity, called *root*, which is not invoked by other activities.

In order to enact a process model, in addition to *ProcessTypes* and *ProcessActivities*, it is necessary to provide an initial state for the the active copy of the root activity. This state represents the initial state from where enactment starts.

In short, the type definitions in the *ProcessTypes* set provide the data structure and operations for all process data, the activity definitions in the *ProcessActivities* set specify how these data are manipulated by the process, and the root active copy provides the initial state for process model enactment.

## 3.1 SLANG types

SLANG adopts the $O_2$ data model for the description of abstract data types. Each abstract data type is thus specified as an $O_2$ *class*. Class instances are called *objects*. A class has a unique name, a *type*, and a list of operations, also called *methods*, that may be applied to objects of the class. The *type* describes the data structure template for all the class instances.

Types may be atomic or structured. Atomic types are: `integer`, `char`, `boolean`, `string`, and `bits`. Structured type constructors are: `tuple`, `set`, `unique set`, and `list`.

Tuple values are constructed using attribute names and values. For example `tuple (filename: "test.c", lines: 328, ok: false)` is a tuple value. Tuple attribute values are accessed using the dot operator (`.`).

Sets are constructed from values of the same type. The keyword `set` represents the concept of "bag", where repeating elements makes a difference, while `unique set` represents the set as usually defined in mathematics.

Lists are similar to sets, except for the fact that the order of elements is significant and an element may occur several times.

## 3.2 Defining abstract data types by means of $O_2$ classes

SLANG abstract data types (SLANG ADTs) are defined using the $O_2$ class constructor. A class specification has four parts:

- a class name, that uniquely identifies the class;
- the class position in the generalization hierarchy, that is the class or classes from which the class inherits;
- the type description;
- the list of methods.

Classes may be defined by specializing existing classes in the generalization hierarchy. A subclass inherits attributes and methods from the superclass and can augment them with new ones. The type of the subclass must be a subtype of the type of the superclass. A subtype of a tuple is defined as a tuple containing the same or more attributes than the base class tuple; a subtype of a set or a list type is such that the type of the elements of the set or list are of a subtype of the base type. Note that, with this definition, a type is subtype of itself. A subclass may override the definition of a method or attribute inherited from the superclass, by redefining it. Method redefinition follows the covariance rule, i.e., parameters and result of the method in the subclass must be subtypes of the corresponding items in the base class. Multiple inheritance is supported, but it may generate name clashes when methods with the same name are inherited through different inheritance paths. In general, these problems are solved by renaming the conflicting methods. For a detailed discussion of these issues, the reader may refer to [O292].

### 3.2.1 Tokens and Objects

In SLANG, process model activities are described using high-level Petri nets. Basically, Petri nets have places, representing data containers, transitions, representing events, and arcs, connecting transitions to places and places to transitions. Process data are represented in Petri nets by tokens. Tokens are objects, whose structure and applicable operations are described by SLANG ADTs. Therefore, each token is typed and carries structured information that can be accessed through the methods defined for the corresponding object.

The most general and simple token type corresponds to the predefined SLANG ADT `Token`. All other more specialized tokens are instances of subclasses of `Token`.

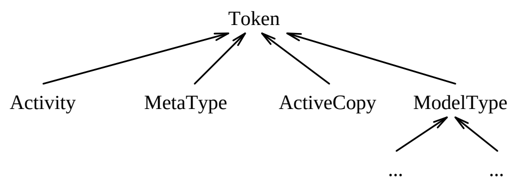


Figure 3.1: Predefined SLANG generalization hierarchy.

SLANG provides other predefined abstract data types that are useful for process modeling purposes. Some of these SLANG ADTs are part of the language definition; they are process-independent and thus are present in all process models. Other SLANG ADTs are process-specific and thus may vary from one process

Process-independent SLANG ADTs include type `Activity` whose instances are activity definitions; type `Metatype` whose instances are type definitions; and type `ActiveCopy` whose instances represent the state of an activity. In order to support process evolution in SLANG, activity definitions, type definitions, and activity states are accessible as any other data, i.e., as tokens of the net. Therefore `Activity`, `Metatype`, and `ActiveCopy` are defined as subclasses of `Token`.

Process-specific SLANG ADTs, which characterize the particular process being modeled, are represented by a subtree of `Token` in the generalization hierarchy rooted `ModelType`. Thus, all new user-defined ADTs will inherit from `ModelType`. Figure 3.1 shows the generalization hierarchy containing some of the predefined SLANG abstract data types.

## 3.3 SLANG activities

The *ProcessActivities* part of a SLANG process model is a set of activity definitions. SLANG activities are based on Petri nets, thus the syntax includes a graphical part as well as a textual part. The SLANG activity semantics follows the standard Petri net semantics, with some extensions that will be explained in the following. Actually, SLANG semantics is given in terms of a lower level Petri net based language, called Kernel SLANG. SLANG process model are translated into Kernel SLANG and then executed. Kernel SLANG, for example, does not support activity invocations. The semantics of Kernel SLANG is given formally in terms of a formal Petri net model, called ER nets [GMMP91].

## 3.4 Activity definition

Each activity corresponds to a logical work unit. The activity definition specifies the starting events, the ending events, and other relevant events of the activity. Petri net transitions represent events, while places behave as data (token) containers. The Petri net topology describes precedence relations, conflicts, and parallelism among events. An activity definition may also include invocations of other activities and interaction with the user environment.

The activity is a static entity, it has no state. During enactment an activity may be invoked, generating an active copy of the activity. Active copies are the dynamic counterpart of activities. At a given time during enactment, one activity may have zero, one, or more active copies associated with it. Each active copy has its own state.

A SLANG activity definition is a graph, with three kinds of nodes (places, transitions, and invocations) and two kinds of arrows (arcs and links). Formally, a SLANG activity *Act* is given by a 5-uple

$$Act = (P, T, A, I, L)$$

where $P$ is the set of *places*, $T$ is the set of *transitions*, $A$ is the set of *arcs*, $I$ is the set of *activity invocations*, and $L$ is the set of *links*.

An arc connects a place to transition or a transition to a place:

$$A \subseteq (P \times T) \cup (T \times P)$$

A link connects a place to an invocation and vice-versa:

$$A \subseteq (P \times I) \cup (I \times P)$$

Places are depicted with circles, transitions with sharp-angle rectangles, arcs with thin arrows, and links with thick arrows. Invocations are structured nodes, represented by rounded-angle boxes with rectangles (transitions) on the upper and lower border. Intuitively, the transitions on the upper and lower border of the invocation correspond respectively to the starting and ending events of the invoked activity.

**Example: Activity definition.** Figure 3.2 shows the graphical part of the definition of an activity. The dotted frame separate the interface of the activity from its implementation, the following section explains these concepts. The figure shows also other graphical elements for particular kind of places, transitions and arcs. They will be better specified later. □

## 3.5 Activity interface and implementation

According to the principles of information hiding, an activity definition has an *interface* and an *implementation* part. The activity interacts with other activities through its in-

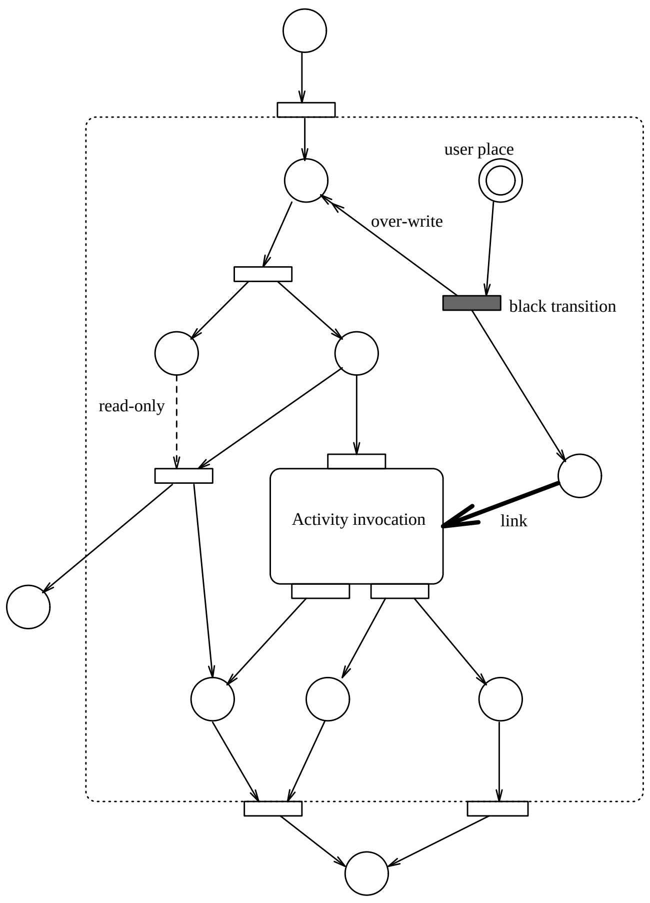


Figure 3.2: SLANG activity definition.

terface, while the implementation part remains hidden. An activity interface is composed of:

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

Events that initiate and terminate the execution of an activity are represented by interface transitions. Interface transitions are partitioned in two disjoint and non empty sets: SE (called *starting events*) and EE (called *ending events*). In addition, starting and ending events of an activity cannot be part of an invocation in the activity implementation. In other words, the same transition may not represent a starting event (or more than one ending event) of more than one activity. Activity execution starts when one of the transitions in SE fire and terminates with the firing of one of the transitions in EE. Interface places in $P_{int}$ are classified into three sets:

- *Input places*: they belong to the preset of any of the starting events,
- *Output places*: they belong to the postset of any of the ending events,
- *Shared places*: they are connected to the invocation through a link, and are shared by the calling and the called activities.

Input places play the role of formal parameters of an activity, output places can be considered the value returned by the activity, and shared places are a sort of input/output variables that may provide further input, during activity execution or may act as preliminary result containers, before activity terminates. Any other place internal to the implementation part (not belonging to the interface) is said to be a *local place*. The implementation part details how the activity is performed, by showing the relationships among the events that may occur during its execution.

### 3.5.1 Activity invocation

The implementation part of an activity definition may contain invocations of other activities. An activity invocation must match the corresponding activity definition interface in terms of interface places, transitions, and arcs.

**Shared and global places.** Shared places are connected to the invocation box by links. A link from a shared place to the invocation box means that the place can only be used in the preset of transitions in the activity implementation. Conversely, a link from the invocation box to a shared place indicates that the place can only be used in the postset of transitions of the activity implementation. A double link permits the use of the share place in both the pre and postset of transitions. Sometimes it is useful to share a place among all activities in a subtree of the ASD. A place with this characteristics is called a *global place*. A global place is local for the activity in which it is defined and is shared among all the activities in the subtree whose root is the activity defining the place. In order to distinguish global places from normal places, global places are depicted with a thicker line. Note that the global place notation is only a shorthand to avoid the repetition of a double link for a shared place in each activity invocation.

### 3.5.2 Root activity.

In a SLANG process model, there is one "main" activity, not invoked by other activities. This activity is called *root*. In the root activity there is no distinction between interface and implementation. This activity has only to specify a non empty set of ending events. The occurrence of one of these ending events indicates the end of the process model execution.

The root activity must include the definition of some places that are needed for activity invocation and evolution. The process modeler must guarantee the presence of these places and their appropriate contents in the root active copy[2]. These places are global and have the following names and types: place `Activities` of type `Activity`, place `Types` of type `Metatype`, place `DynTree` of type `DynamicTree`, place `DynTreeLock` of type `Token`, `SuspendAC` and `ACToBeRestarted` of type `ActiveCopy`, `SuspendingRequest` of type `ACIdentifier`.

Initially place `Activities` must contain the set of all activities defined in the process model, `Types`, the set of all SLANG ADTs defined as subtypes of `ModelType`, `DynTree` a token representing a tree with only one node with the root active copy, and `DynTreeLock`, an anonymous token, `SuspendAC`, `SuspendingRequest` and `ACToBeRestarted` concern the suspension mechanism. Note that, since these places are defined as global in the root activity, they are automatically shared by all the activities.

### 3.5.3 Places

A place defines a template for a persistent repository of tokens. Each place has a name, which is a unique identifier within the activity, and a type. A place can only contain tokens of its type (or of any of its subtypes). The type of a place must be one of the subtypes of the predefined SLANG ADT `Token` (recall that a type is a subtype of itself an thus the place type may also be `Token`). There are two kinds of places: *normal places* and *user places*. The contents of a repository associated with a normal place may only change because of a transition firing. Repositories associated with user places, instead, change their contents as a consequence of an event occurring in the user environment. Actually, an event in the user environment may produce a new token with the information about the occurred event in the repository associated with a user place. User places may only belong to the preset of transitions. While normal places are graphically represented by a single circle, user places are represented by double circles. The way to establish the correspondence between events in the user environment and user places is explained in Chapter 3.6.

### 3.5.4 Arcs

An arc from a place to a transition indicates that the occurrence of the event associated with the transition depends on the contents of that place. Vice-versa, an arc from a

[2]These places are not required if the root activity does not contain invocation of other activities.

transition to a place means that the occurrence of the associated event changes the place contents. SLANG offers different kinds of arcs, with different semantics: *normal arcs*, represented by solid lines; *read-only arcs*, represented by dashed lines and *over-write arcs*, represented by lines with double headed arrows. A normal SLANG arc corresponds to the usual definition of arc in a Petri net, so when a normal arc goes from a place to a transition, tokens that enable the firing are consumed, i.e., they are removed from that place, while a normal arc from a transition to a place causes tokens produced by the firing to be added to that place. A bidirectional (normal) arc between a place and a transition can be used as a shorthand for a pair of normal arcs: one from the place to the transition and one from the transition to the place. Read-only arcs may only connect places to transitions. In this case, the transition may use the place contents in read-only mode. That is, it can read token values from the input place in order to evaluate the guard and the action, but no token is actually removed. An overwrite arc may only connect transitions to places. If a transition is connected with a place by an overwrite arc, when the transition fires, the produced tokens *overwrite* the previous contents of the place. The overall effect is that the output place is first emptied and then, the tokens produced by the firing are inserted.

Arcs are weighted. The weight indicates the number of tokens flowing along the arc at each transition firing. It can be a statically defined number (the default weight being 1) or it may be dynamically computed (in this case the weight is indicated by a "*"). Weight and kind are orthogonal features and all combinations are possible.

### 3.5.5 Transitions

Transitions represent events. Events taking place in the process-enactment environment are represented by *white transitions*. When the event involves a tool execution (which has an effect in the user environment), the event is represented by a *black transition*. Transition execution is atomic, in the sense that no intermediate state of the transition execution is visible outside the transition. White transitions are executed synchronously (one after the other) by the SLANG Interpreter. The execution of a black transition, instead, may overlap with the execution of other transitions (see Chapter 6 for further details on this issue).

Each transition is associated with a guard and an action. The guard is a boolean expression on tokens belonging to the transition's input places. An input token tuple that satisfies the guard is said to be an enabling tuple for the transition. A transition having an enabled token tuple is said to be enabled. An enabled transition may fire. The firing consists in the execution of the transition's action. The action specifies how the output token tuple is calculated as a function of the enabling token tuple. Black transitions specify special actions which include the execution of a tool.

### Variables

Variables may be used in the specification of a transition guard and action. Given a transition, we distinguish different groups of variables:

- place variables, which can be further classified in input place variables and output place variables, and
- local variables.

Place variables correspond to input and output places of the transition, they are implicitly declared according to the input and output places of the transition, the arcs connecting the places and the transition, and the type of the corresponding places. A place variable name (its *identifier*) coincides with the corresponding place name. When a place is both input and output for a transition, the corresponding variables are named `in_`*place-name* and `out_`*place-name* respectively[3]. A place variable type *type* depends on the type of the corresponding place and the weight of the arc connecting the place and the transition. If the place type is `T`, and the arc of weight one, the variable type is also *T*. If the arch weight is greater than one or is a dynamic weight arc, the variable type is `set(`*T*`)`.

Local variables have to be explicitly declared in the transition. The syntax for a declaration is:
`o2` *typeId varId* ;
A complete example with all the variable of a transition is given in appendix A.2

### Guards

A guard is a boolean expression involving place input variables. Local variables may be used if bound by quantifiers. All quantifiers must be placed as a prefix of the expression and followed by predicate that does not contain quantifiers. Thus, a guard syntax is:
((`forall` | `exists`) *varId* `in` *varId*:)* *expression*
The *expression* is any ($O_2$C) boolean expression.

**Example of guard expression** If we have a transition with an input place called `P` containing persons, we may write:

```
P->name = "James"
```

[3]This ambiguous case apply only to actions because output variables are not defined for guards. So the name of the variables for a place called `X` will be `in_X` and `out_X` for the action and simply `X` for the guard.

which is enabled if there is one person called James. Here the weight of the arc is 1. If it were 3 we could write:

```
forall x in P: x->age >= 18
```

In this case the guard selects 3 persons older than 18. □

Transition guards are conditional expressions that have the input place names as free variables. An input tuple of a transition is a tuple where each component is a token or an nonempty subset of the input place contents. Actually, the cardinality of each tuple component depends on the weight of the arch that connects the input place and the transition. If the weight is 1, the corresponding tuple component may be any token belonging to the input place contents. If the weight is a fixed number $n$ with $n > 1$, then the tuple component may be any token subset of the input place contents, with cardinality $n$. If the arch weight is dynamic (indicated with *), the tuple component may be any nonempty susbset of the input place contents.

**Guard semantics** Guard evaluation causes free input variables to be bound to tokens satisfying the guard expression. For each place the number of tokens is determined by the weight of the arc connecting the place; such a set of tokens is called an enabling tuple. There may be no enabling tuples, in this case the transition is not enabled and can not fire, or there may be even more than one, in this case one is chosen nondeterministically.

The evaluation implicitly maximizes the number of tokens when this is not statically defined, that is when dynamic arcs are present in the preset. In other words, in such circumstances an enabling tuple is the set with maximum cardinality that satisfies the guard.

A formal description of the guard semantics is in appendix A.1.

**Example of guards with dynamic arcs** Refer to figure 3.3).

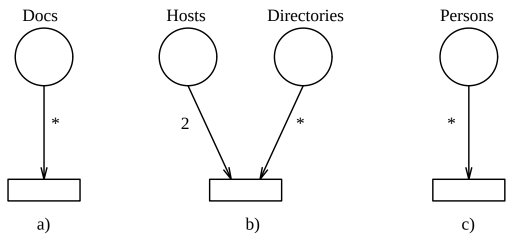


Figure 3.3: Examples for guard semantics

**a)** This first example is very simple. We have one input place containing documents connected with a “star” arc to the transition. Each `document` has a boolean attribute called `reviewed`. We want to extract the reviewed documents. Provided the declaration of one local variable `d` of type `document`, the guard is:

```
forall d in Docs: d->reviewed
```

in this case *all* the reviewed document are bound to the enabling tuple.

**b)** In this example we have a place of type `host` containing machines id of a LAN. Suppose that each host mounts some directories of a distributed file system. The class `host` has an attributes called `fstab` which is the set of all the directories visible to that host. Each directory is identified by a string. The other place is a container for `dir` tokens, the class `dir` has a string attribute called `name`. Suppose this is the marking of the net: place *Hosts* contains these three tokens:

```
{ hostId: "rossini" , fstab: set("/usr/bach","/special/fx") }
{ hostId: "bellini" , fstab: set("/mnt/X","/home/sweet/home") }
{ hostId: "bach" , fstab: set("/usr/bach","/home/sweet/home") }
```

while place *Directories* contains these tokens:

```
{ name: "/usr/bach"} { name: "/special/fx"}
{ name: "/mnt/X"} { name: "/home/sweet/home"}
{ name: "/se1/gioia"}
```

We want to take two machines and all the files accessible by them. Then the SLANG guard that makes the selection is:

```
exists m in hosts: forall d in directories: (d->name in m->fstab)
```

Again, `d` and `m` must be declared in local variables. This guard takes two hosts because the arc weights 2. Among all the combinations, the interpreter chooses `bellini` and `rossini` with `"/usr/bach"`, `"/special/fx"`, `"/mnt/X"` and `"/home/sweet/home"`. The reason for this choice is that taking `bach` would in any case allow just three directories. Instead `bellini` and `rossini` bind four directories.

**c)** Suppose we are modeling an activity that requires a group of persons working together very well (the holidays problem). The starting transition for such activity must extract a self contained group of persons from a global place called *Persons*. Each person has a name and a set of *friends*. The guard should make sure that everyone in the group gets on well with each other. The SLANG expression for this condition is:

```
forall p1 in Person: forall p2 in Person: \
 (p1->name in p2->friends) && (p2->name in p1->friends)
```

Evaluating this guard the interpreter chooses the biggest group of persons that satisfies the constraint[4].

□

## Actions

An action is an $O_2$C code which processes tokens from the enabling tuple and produces new tokens for output places. $O_2$C is an extension of the C language. It provides all the standard control flow instructions and all the operators and expressions of C. In addition to this, it comes with some operators to access complex objects. (for a detailed specification of the $O_2$C language, refer to [O292]).

In the action, input variables are initially bound to the enabling tuple, each variable refers to the token or the set of token extracted from the corresponding place. This initial value may be changed during action execution. In any case, these variables assignments are lost when the action terminates. Output variables are bound to initial default values. If their type is `set(...)` the initial value is the empty set, else they refer to new objects of the proper class[5]. Thus SLANG programmer can refer immediately to output tokens without having to create them. Local variables, declared explicitly, are initialized in the following way: Atomic default values are: `0` for integers, `0.0` for reals, `false` for booleans, `'\0'` for characters and `""` for strings. Structured default values are `list()` for lists, `set()` for sets and `unique set()` for unique sets. No initialization is provided for objects, thus their initial value is `nil`.

There are some differences between white and black transitions in the action specification, as explained in the following.

## White transition actions

The action associated with a white transition is made of one single action body. The input for this code is the enabling tuple, the output is the output token tuple. The scope of every variables include all the action code. The execution takes one single step.

**Example: white transition.** Consider the transition of figure 3.4, place Orders is of type `Order`, place Items is of type `Item`, place TotItems is of type `TotItem`. These are the

[4]This is a particular version of the problem of the complete sub-graph (CS), a well known NP-complete problem. The SLANG interpreter gives the solution in one transition firing! This is not to say that we found the efficient solution to NP-complete problems; it should point out the intrinsic complexity of guard evaluation with this semantics.

[5]The object class may have an `init` method. In this case, this method is executed when the new object is created. In SLANG, the init method may not have parameters.

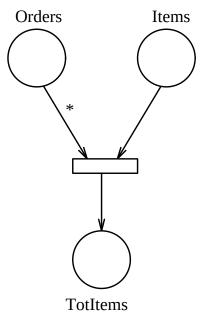


Figure 3.4: Example of white transition.

classes:

```
class Item inherit ModelType
public type tuple
 {
   descr: string
 }
end;

class Order inherit ModelType
public type tuple
 {
   ItemDescr: string,
   quantity: integer
 }
end;

class TotItem inherit ModelType
public type tuple
 {
   ItemDescr: string,
   TotQuantity: integer
 }
end;
```

Let us specify the following local variables:

```
o2 Order cursor;
```

and the guard for the transition:

```
forall cursor in Orders: cursor->ItemDescr == Items->descr
```

This guard binds one `Item` and all the orders related to that item. The following action computes the total amount of ordered items and stores one token with the result. Here is te action code

```
TotItems->ItemDescr = Items->descr;
TotItems->TotQuantity = 0;
for( cursor in Orders )
   TotItems->TotQuantity += cursor->quantity;
```

□

### Black transition actions

The black transition action contains an external action (a tool execution). The action body is actually split in two distinct parts called *prologue* and *epilogue*, which are executed in two different steps.

**Prologue** : is the code that sets up all the conditions for the external action. Usual input and local variables are declared and assigned like in white transition actions. Moreover two other variables are pre-defined:

- `extAction`: is of type string, which represents the external action to be executed. This external action corresponds to the execution of a tool. The format of the strings that may be assigned to `extAction` is discussed in section 3.6.
- `parList`: is a list of objects used to pass $O_2$ parameters along with a tool invocation. In the tool does not require parameters in the form of $O_2$ objects, the value of `parList` is simply ignored.

Both variables values are assigned during prologue execution, and may vary from firing to firing.

**Epilogue** : is the code executed just after the external action termination. It defines output, input and local variables. While output tokens are initialized as usual, input tokens are restored with the last assignments of the prologue. Since the scope of local variables is restricted to a single step, their contents resulting from the prologue are lost. Thus local variables take usual default values. A special variable called `extResult` collects the results of the external action and makes them visible to the epilogue code. User may define the type of `extResult` which in any case must be an object. The default type is `Object`. `extResult` is `nil` in any case of UNIX external action.

## 3.6 SLANG in the SPADE-1 2.0 environment

SLANG provides special constructs to support communication between the Process Enactment Environment and the User Interaction Environment. These constructs are *black transitions* and *user places*. Black transitions are a special kind of transitions whose firing produces an effect in the user environment, namely through the execution of a tool. User places, on the other hand, are special places where tokens are inserted each time a relevant event for the process environment occurs in the user environment.

A black transition can invoke a tool (black-box tool) or it can request a particular service from a service-based tool that is already running. In SPADE-1, interaction with black-box tools is achieved by invoking the tool in a black transition action. The tool name and the necessary parameters for the execution must be assigned to the predefined variable `extAction`. Additional parameters, in the form of $O_2$ objects may be specified in the predefined variable `parList`. In order to interact with service-based tools, SPADE-1 provides a special tool called `ServiceRequest`. This tool requires a service to a tool and finishes execution when the service has been provided. Thus, a black transition may be used to require a service to a tool by specifying in variable `extAction` the tool `ServiceRequest` followed by the necessary parameters (the tool identifier, the service name, etc.).

In order to capture the relevant events that occur in the user environment, the filter, managing interaction between enactment and user environment (SPADE Communication Interface, see chapter 7), must be appropriately configured. SCI configuration is done dynamically during enactment and consists in the registration (or de-registration) of an active copy to receive a notification of the occurrence of a particular kind of event in a user place.

The rest of this chapter discusses in detail the interaction with black-box tools and service-based tools in SLANG. For this discussion it is useful to further distinguish among tools that access the $O_2$ database (and thus may interchange data with the process in the form of $O_2$ objects) and tools that are integrated in message-based software development environments, such as DEC FUSE.

## 3.7 Interaction with black-box tools

Interaction with black-box tools is achieved through black transitions, assigning to the predefined variable `extAction` a string with the command line (as if it were written at a shell prompt) to launch the tool. The string includes the tool name (with the full path if the tool is not in the current path of execution) and its parameters. Thus, the syntax of the string assigned to variable `extAction` is:

```
<toolname> [<parameters>] [-O2]
```

The arguments between "[" and "]" are not compulsory. The `-O2` flag indicates that the invoked tool is an $O_2$ application and that it may have $O_2$ objects as further parameters specified in the variable `parList`. It is thus necessary to include `-O2` as it were a final parameter for $O_2$ applications. `SLANGInterpreter` is the only $O_2$ tool that does not require the `-O2` parameter.

Depending on the tool characteristics a result may (or may not) be returned. The type of the result (if it exists) also depends on the invoked tool. In the case of a returned result, it is made available to the black transition epilogue in the predefined variable `extResult`, whose type must be declared in the black transition according to the expected type for the result.

**Example: Interaction with black-box tools.** Suppose that in a SLANG process model the editor `emacs` has to be launched in a black transition action using display contained in variable `user` on the file `myfile.txt`. Thus, the black transition should include a statement like this one:

```
extAction = "/usr/local/bin/emacs -display " + user + " myfile.txt";
```

In this case, no result is expected in variable `extResult`.

Another example could be the tool `SLANGInterpreter`, which receives an object of type `ActiveCopy` as parameter and returns and object of type `ActiveCopy` as result. In this case, variable `extResult` must be declared of type `ActiveCopy`. Suppose that variable `ac` contains the active copy to be executed. The preamble of the black transition action invoking tool `SLANGInterpreter` should be:

```
extAction = "SLANGInterpreter";
parList = ac;
```

When the `SLANGInterpreter` execution finishes, the resulting active copy is contained in variable `extResult`.

□

## 3.8 Interaction with service-based tools

Interaction with service-based tools is carried out in two ways:

1. service request,

2. event notification.

Service requests are generated by the process enactment environment, by calling tool `ServiceRequest` from within a black transition. The request must specify the tool instance to which the service is requested, the requested service, and, optionally a list of parameters (which may also include $O_2$ objects). All service requests produce some result. The `ServiceRequest` tool execution finishes when results are returned to the black transition that invoked it. Thus, service requests are a two way communication: parameters are passed from the process environment to the tool and then the service result is passed back from the tool to the process environment.

Event notifications are generated spontaneously by tools in the user environment, while they are running. For example, an editor may notify the event that it saved a file successfully, or a compiler may notify the event that it has found a syntax error. Not all events that occur in the user environment are of interest to the process environment. Actually, the process environment dynamically declares which are the events of interest by configuring the SCI. The SCI behaves as a filter that communicates to the process environment only the events of interests. Note that, since SCI configuration is done dynamically, the set of events of interest may change during enactment. The tool `ServiceRequest` is also used for SCI configuration, specifying service `ConfigSCI` and including as parameters the name of the event of interest and the user place where the event information has to be inserted. The effect of a configuration is that the SCI registers the active copy to receive a notification each time the event of interest occurs. The notification will appear in the form of a token in the user place specified in the configuration. A configuration is valid until the process model enactment de-registers the active copy (with a procedure analogous to the registration, i.e., using tool `ServiceRequest`) or until the active copy to which the user place belongs finishes execution.

### 3.8.1 Tool `ServiceRequest`

This section provides a detailed description of tool `ServiceRequest` and of its different uses. The tool syntax is the following:

`ServiceRequest` *address service* [*parameters*] [`-O2`]

*address* is a string identifying the tool instance to which the request has to be sent. The address consists of either one number or a couple of numbers separated by a dot. *service* is the name of the requested service. *parameters* is a string containing a list of command line parameters for the service requested. The `O2` option is used to indicate that the requested service is provided by an $O_2$ tool and thus it may have some parameters in the form of $O_2$ objects. These objects must be assigned to the variable `parList`.

Depending on the value of *address* the service request is managed by different tools. If the value of *address* is `0` the request is managed directly by the SCI. The SCI offers two services: `StartTool` and `ConfigSCI`. If the value of *address* is a single number (different from 0), the service request is managed by the tool instance identified by that number. Finally, if value of *address* is a couple of numbers, the service is to be managed by tools integrated in a separate environment (e.g., DEC FUSE). The first number of the address must identify a bridge tool that connects the SCI with the integrated environment an and the second number identifies a tool instance, within this environment, that provides the requested service.

### Service requests managed by the SCI

**Service `StartTool`.** The service `StartTool` requires as parameter the host machine, the name of the tool followed by the command line parameters. The `StartTool` service terminates when the tool is up and running. An object of the SPADE class `Message` is returned by the external action in the variable `extResult`. This message contains the address of the invocated tool in the field `addr`. In this case, the syntax of the service request is:

```
ServiceRequest 0 StartTool <toolname> [<parameters>]
```

`<toolnane>` is the name of a tool. `<parameters>` is an optional string of parameters for the service execution.

**Service `ConfigSCI`.** The service `ConfigSCI` takes as parameter a configuration pattern for the SCI. A configuration pattern adds or removes a particular filter to the SCI. The syntax for this service request is:

```
ServiceRequest 0 ConfigSCI  <pattern>
```

where a `pattern` has the following syntax:

$$\begin{array}{rcl} \textit{pattern} & \rightarrow & (\texttt{+}|\texttt{-})\ \textit{identifier msgName placeName} \\ \textit{identifier} & \rightarrow & \textit{integer}(.\textit{integer})^*(\texttt{*}\ |\epsilon)|\texttt{*} \\ \textit{msgName} & \rightarrow & \texttt{*}\ |\ \textit{string}\texttt{*} \\ \textit{place} & \rightarrow & \textit{string} \end{array}$$

The meaning of a pattern is straightforward: the plus and minus sign specify weather the filter must be added or removed from the SCI. A filter states that every message matching both tool id and message name (the "*" matches any value) is forwarded to the place of the active copy that introduced that filter. The service terminates when the SCI has been configured. As a result, an object of the class `Message` is returned in the variable `extResult`.

### Service requests managed by tools directly connected to the SCI

When a service request is issued, it is assumed that the addressed tool instance is already running. The tool services the request and always produces a result. For tools linked to $O_2$, the type of the result depends on the service requested. If the result is an $O_2$ object, the type of the result is the type of the returned object. Otherwise, the type of the result is `Message`. In all the cases, the result is stored in the variable `extResult`, whose type has to be declared according to the expected result type.

The syntax of the service request is:

```
ServiceRequest <address> <servicename> [<parameters>] [-O2]
```

In order to identify the tool, `<address>` has to be known to the process model. The process model may get to know a tool address in two ways. If the tool has been launched by the process using the `StartTool` service, the tool address has been returned as a result of the service. Otherwise, the tool has been launched directly by a user. In this case, it is the tool responsibility to communicate its identity to the process and the process responsibility to be able to capture this event (by previously having configured the SCI). The other arguments are similar to those of service `StartTool`. `<parameters>` is an optional string of parameters for the service execution. The `O2` option must be included if the tool is an $O_2$ application and, in this case, additional $O_2$ parameters may be specified in the variable `parList`.

The execution of a service may imply the termination of tool execution. In this case, the tool instance is no more able to receive new service requests. The process model should guaranteed that no requests are issued to terminated tools.

### Service requests managed by tools from an integrated environment

Service requests may also be addressed to tools belonging to tool environments based on message passing (like DEC FUSE), which are integrated in SPADE-1. In this case, the integration is achieved through a tool, called bridge, which is in charge of forwarding the service requests to the tool environment and passing back the service results to the SCI. The bridge guarantees that each request receives an appropriate reply. Each instance of a tool environment requires a different bridge instance to connect with the SCI. Thus, the bridge address identifies one tool environment instance (e.g., one DEC FUSE instance). If the service request is directed to a particular tool instance within the tool environment, a second number is necessary to identify a tool instance within the tool environment. If this second number is not present, the service request is not directed to any particular tool and should be managed by the message server of the tool environment.

Thus, service requests sent to tools in a tool environment may have one of the following forms:

```
ServiceRequest <bridgeaddr>.<tooladdr> <servicename> [<parameters>][-O2]
```

or

```
ServiceRequest <bridgeaddr> <servicename> [<parameters>][-O2]
```

The argument `<bridgeaddr>` is like any other `<address>` managed by the SCI. In any case, when the service is completed, there is always a replay to the service request. If the tool shares the $O_2$ schema, then the result is always `Message`. For tools that access $O_2$ data base, the type of the result depends on the service requested. If the result is an $O_2$ object, the type of the result is the type of the returned object. Otherwise, the type of the result is `Message`. In all the cases, the result is stored in the variable `extResult`, whose type has to be declared according to the expected result type.

# Chapter 4

# SPADE-1 2.0 architecture

This chapter presents the architectural design of the SPADE-1 2.0 environment [BBFL94]. In this presentation a top-down approach is adopted. First, a very high-level description is presented, then, each component is expanded in a detailed structure. The description of the SPADE-1 architecture is given by means of two classes of entities: *components* and *connectors* [GS93, AG94b, AG94a].

Components are computational modules or repositories of data. A component has a state and an interface, which is a set of connection points. The interface specifies the actions the component is able to perform and the way other entities access its status.

Connectors are link protocols that define two endpoint interfaces. Two components may be bound by a connector, if they are able to play the roles specified by the connector interfaces.

The description of SPADE-1 architecture in terms of its components and connectors is not meant to be a formal specification of protocols and matching interfaces. A more accurate formalization is far beyond the scope of this thesis. Instead, this organization framework has been chosen to provide an ordered presentation of the architectural issues. We use different shapes for boxes and lines to display different implementations and roles in the SPADE-1 2.0 architecture.

Section 4.1 sketches SPADE-1 logical parts. Sections 4.2 and 4.3 give the vocabulary and a straightforward graphic notation for components and connectors. This notation will be used in sections 4.4, 4.5, and 4.6, which present in details the architecture of SPADE-1 2.0.

## 4.1 SPADE-1 architecture overview

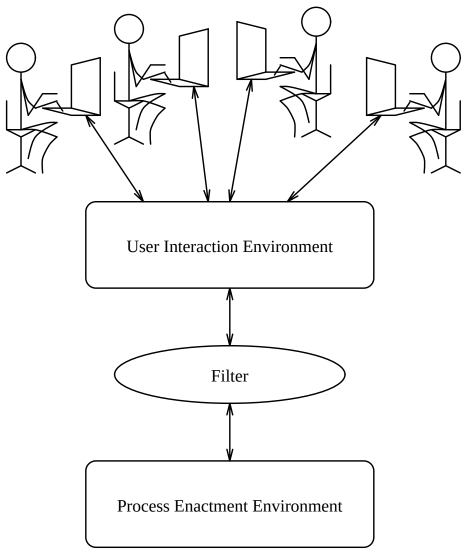


Figure 4.1: High-level view of SPADE-1 architecture.

SPADE-1 design is based on the idea that the paradigm used to model the process and support its enactment, and the paradigm used to guide the interaction with users can be reasonably kept distinct. This is useful to support distribution, evolution and improvement of the environments, and the integration of different types of paradigm in the same PSEE.

Thus, the SPADE-1 environment is logically divided into three main parts:

- the Process Enactment Environment;
- the User Interaction Environment;
- a filter connecting these two parts.

The Process Enactment Environment includes facilities to execute a SLANG specification, creating and modifying process artifacts. The User Interaction Environment manages the interaction between users and the process. The filter, called SCI (SPADE Communication Interface), provides communication between the Process Enactment Environment and the User Interaction Environment.

## 4.2 Basic architectural elements

Basic components and connectors are the building blocks of the SPADE-1 2.0 architecture. These components and connectors are well-known and commonly used elements. They are abstraction provided by modern programming languages [GJ82], as well as facilities and mechanisms supplied by existing operating systems, in particular Unix.

These elements are described from a *dynamic* viewpoint. I.e., we represent *instances* of components and connectors. We are not interested in describing static relationships (e.g., "uses" or "is-a" relationships) between modules.

### 4.2.1 Components

Basic components in the SPADE-1 2.0 architecture are general and well-known entities. We distinguish four basic components:

**File** : files in the SPADE-1 2.0 architecture are Unix files. The role played by a file is:

- *Repository.* A file plays the role of a *Repository* when other components access the data it contains.

**Process** : this component refers to the environment and the computational resources related to a Unix process. A process can play the following roles:

- *Executor.* A process executes some code.
- *Parent-Process* or *Child-Process.* These roles are intuitively related to the `fork` Unix system call. A process that somehow causes the creation of another process, plays the *Parent-Process* role. The created process plays the *Child-Process* role.

**Function** : functions in SPADE-1 2.0 are coded in $O_2$C, C or C++. A function can play the following roles:

- *Obj-Owner.* A function may have private variables which are objects or references to objects.
- *Code.* Functions are code executed by processes. Moreover, functions may call other functions or methods on objects.
- *Reader-Writer.* Functions play the *Reader-Writer* role when accessing files (i.e., they own the reference to a file, and one of their task is to access that file).
- *Client* or *Server.* These roles refer to a Unix socket connection. Functions are connected by a socket if they own, in their (private) variables, the socket endpoint descriptors. A *Server* function accepts connections, while a *Client* function requests a connection[1].

[1]Usually socket connections are represented as links between *processes*. We use a different approach, in order to locate precisely which function is responsible for the management of a socket connection endpoint.

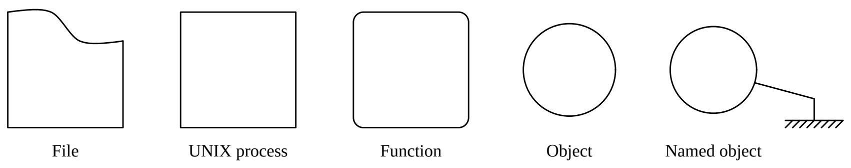


Figure 4.2: Basic components.

**Object** : an object is a data structure with associated methods. Objects belonging to the SPADE-1 2.0 architecture can be objects in the $O_2$ object-oriented data base or C++ objects. An object can play the following roles:

- *Obj-Owner* or *Obj-Instance*. These roles refer to objects referenced by other objects. Objects holding the reference in their data structure play the role of *Obj-Owner*. Referenced objects play the role of *Obj-Instance*.
- *Code*. An object defines methods. From this point of view, it can play the role of *Code*.
- *Server*, *Client*, *Reader-Writer*. These roles are played by an object as it was described for functions.

**Named object** : it is an object which act as a persistency root in the $O_2$ data base (see section 5.1). It can play the roles of an object.

Figure 4.2 shows the graphic appearance of basic components.

### 4.2.2 Connectors

Connectors link up components, provided they are able to play the roles defined by connectors endpoint interfaces. These are the connectors used in the SPADE-1 2.0 architecture, together with the roles accepted by each connector:

**Execute** : it connects an *Executor* to a *Code*. Code can be a function or a method of an object.

**Call** : it connects a *Code* to a *Code*. For example, a function can call another function, or a method on an object.

**Link** : it connects a *Client* to a *Server*. Links, in the SPADE-1 2.0 architecture, are represented by Unix sockets. The asymmetric appearance of the connector highlights the different roles of the connected components.

**Hold** : it connects an *Obj-Owner* to an *Obj-Instance*. For example, a function may have an object as a private variable or share the object with another object.

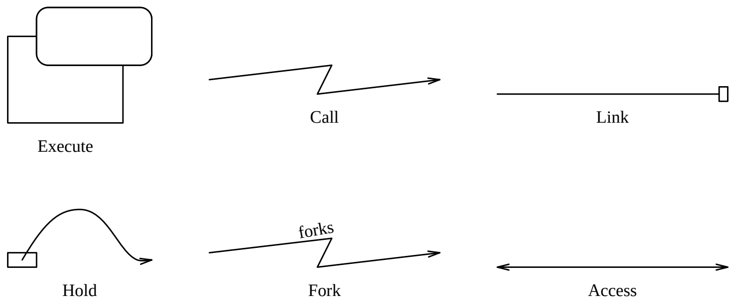


Figure 4.3: Basic connectors.

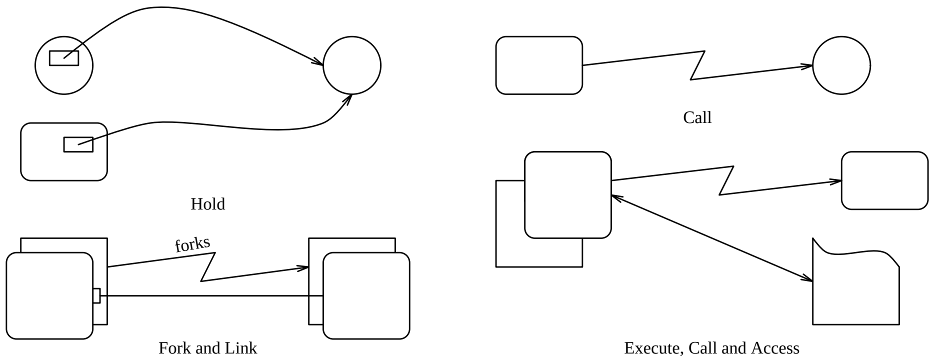


Figure 4.4: Example of components and connectors composition.

**Fork** : it connects a *Parent-Process* to a *Child-Process*. This connector represents, intuitively, the Unix spawning mechanism. In the SPADE-1 2.0 architecture, the fork connector may link up processes running on different hosts.

**Access** : it connects a *Reader-Writer* to a *Repository*. This connector represents a "uses" relation. Components access files, use their contents or use them as repository of data. At this level of description, there is no difference between reading from a file or writing in it.

Figure 4.3 shows the graphic appearance of basic connectors, while figure 4.4 shows some examples of basic components linked by basic connectors.

## 4.3 Composite architectural elements

The basic elements presented in the previous section can be combined to obtain composite elements. Composite elements provide more powerful functionalities than the basic ones. Moreover, they provide abstraction and expressiveness.

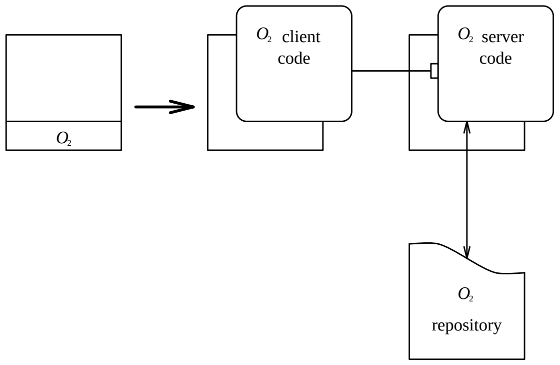


Figure 4.5: Composite components.

### 4.3.1 Composite components

Composite components are macros for the composition of basic components. Often, they inherit some of the roles of their constituents, in addition to the newly defined ones.

$O_2$ **client** : it is a Unix process connected to the $O_2$ server. The $O_2$ client can play all the roles played by a Unix process. In addition, the $O_2$ client is able to successfully execute code which accesses $O_2$ objects, stored in the $O_2$ object-oriented repository. This component is actually a macro definition for a Unix process executing a particular function (the $O_2$ client code), connected via socket to another Unix process executing the $O_2$ server code. The $O_2$ server manages the interaction with the $O_2$ data repository (see fig. 4.5). Nonetheless, it is not described as a SPADE-1 architectural component. This is why the $O_2$ server, as a stand alone code, does not make much sense. Even if it is a necessary component for the whole system, it does not connect to SPADE-1 components other than $O_2$ clients. Thus, the connection with the server is implicitly included in the clients.

### 4.3.2 Composite connectors

Composite connectors are obtained adding a communication protocol to a basic connector or composing connectors and components.

**SPADE communication protocol** : simple, unstructured communication between components in the SPADE-1 2.0 environment is ruled by the SPADE communication protocol. The SPADE communication protocol is based on the Unix socket facility. SPADE protocol communication primitives provide abstraction. This connector links a *Client* to a *Server*. The graphic appearance of the connector allows to identify the roles of the connected components (see fig. 4.6). The protocol consist in:

1. *Communication initialization*: the server initializes the communication access port, using the `openPort` primitive. It passes a port number as a parameter, receiving a port descriptor as a result. The server is then able to receive connection requests on its host at the specified port.
2. *Connection setup*: clients request the creation of a communication link to the server using the `openConnection` primitive. Clients provide, as parameters, the host name of the server and its communication access port number. They receive, as a result, the connection descriptor. The server accepts a connection using the `acceptConnection` primitive. It passes, as a parameter, the port descriptor, and it receives, as a result, the new connection descriptor.
3. *Data exchange*: clients and server exchange atomic data using the `readMsg` and `writeMsg` primitives. They take a string as argument. The string is transmitted on the communication link using a particular message format and managing possible errors.
4. *Connection closing*: the server and the clients close the connection using the `closeConnection` primitive, passing, as a parameter, the connection descriptor.
5. *Communication shutdown*: the server shuts down the communication access port using the `closePort` primitive, passing, as a parameter, the port descriptor.

The `acceptConnection` and `readMsg` primitives are blocking.

**SCI communication protocol** : the SCI communication protocol, presented in details in section 7.1, rules the communication between the SPADE Communication Interface and its clients. The SCI communication protocol uses the lower-level primitives defined by the SPADE communication protocol, providing information hiding and abstraction. This connector links a *Client* to a *Server*. The graphic appearance of a connectors ruled by the SCI protocol is shown in figure 4.6.

$O_2$ **Socket** : $O_2$ sockets are intended to be a simple and flexible mechanism to exchange complex data within the $O_2$ data base [PV93, CPV94]. The idea is to communicate via two *objects pipeline*, with a non-blocking *rendez vous* communication setup. This connector links a *Client* to a *Server*. The asymmetric appearance of the connector highlights the different roles of the connected modules. The communication protocol consists in:

1. *Communication initialization*: the communication is initialized creating a persistent object of class `Port`.

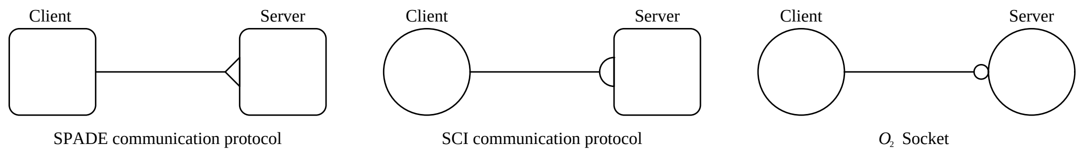


Figure 4.6: Composite connectors.

2. *Connection setup*: the connection is established through a non-blocking rendezvous. The client calls the `connect` method on the `Port` object. The method returns the reference to a `Connection` object. The server accepts the connection request calling on the `Port` object the `accept` method. It receives, as a result, a reference to a `Connection` object.

3. *Data exchange*: during data exchange, there is no difference between the two connected components. Only $O_2$ objects pass through $O_2$ sockets. If an atomic or structured value need to be communicated, then it must be enveloped in an object. Connected components exchange objects calling the `read` and `write` methods on their `Connection` object. The `write` method takes, as input parameter, the reference to an object to be delivered to the counterpart connection. The `read` method returns an object[2]. The object returned by a read operation is of the class `Object`. The recipient must know its actual class to use (cast) it properly.

4. *Connection closing*: $O_2$ sockets do not support any explicit shutdown procedure. Communication channels are not bound to any particular persistency root. The $O_2$ socket is lost as soon as both clients dereference their connections.

5. *Communication shutdown*: the communication access port can be shut down, dereferencing the corresponding `Port` object.

## 4.4 Process Enactment Environment

Process Enactment Environment task is the execution of a kernel SLANG process model specification.

The components of the enactment environment are:

- The $O_2$ OODBMS.

[2]Due to a problem concerning $O_2$ persistency management, objects coming out of a connection are still persistent. Actually, they would not survive an abort or a quit operation, but they are accessible only in transaction mode.

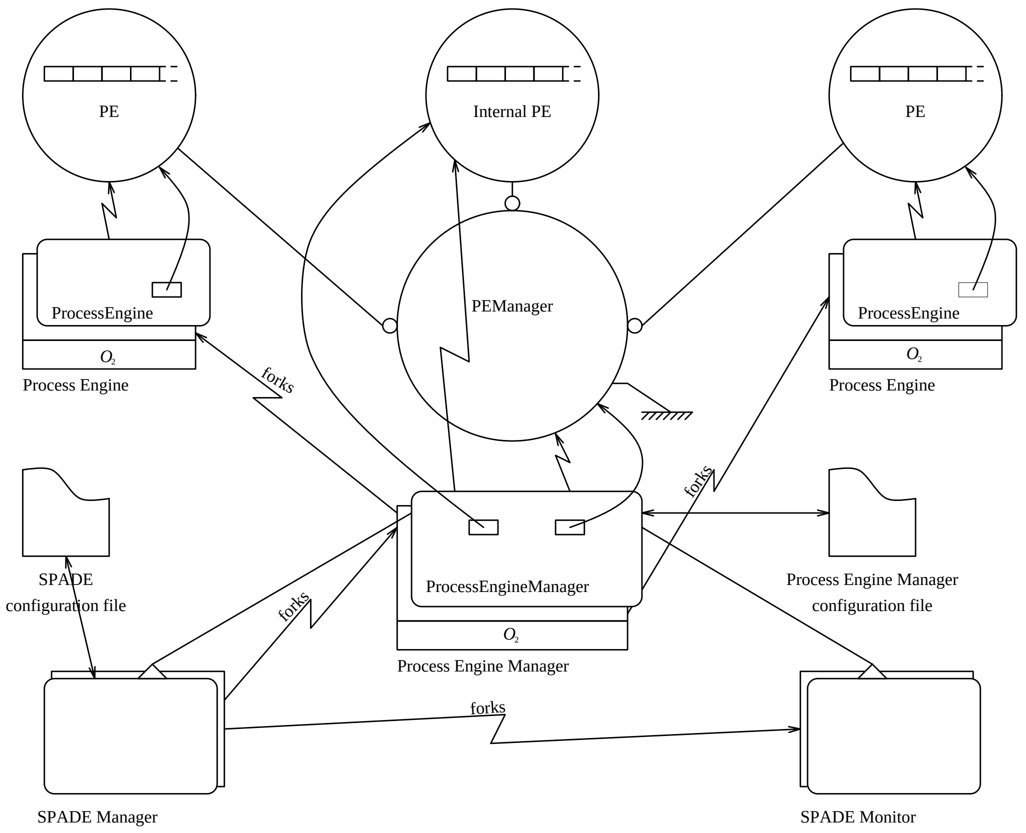


Figure 4.7: Process Enactment Environment Unix processes.

- The SLANG Interpreters.
- The Process Engines and the Process Engine Manager.
- The SPADE Manager.
- The SPADE Monitor.

### 4.4.1 $O_2$ OODBMS

The $O_2$ data base, described in section 5.1, is the backbone of the SPADE-1 2.0 Process Enactment Environment. It supports execution, data management and persistency. $O_2$ has a client-server architecture, in which clients are charged for computations (methods execution), while the server coordinates the access to persistent data. For the purpose of describing SPADE-1 architecture, only the $O_2$ client element is outlined. The $O_2$ database is considered an “implicit playground” containing the objects and functions forming the SPADE-1 2.0 architecture.

### 4.4.2 SLANG Interpreters

SLANG Interpreters are $O_2$ objects of class `SI`. They call the methods that implement the SLANG language interpretation algorithm on active copies. Active copies are $O_2$ objects of class `ActiveCopy`. Every active copy is referenced by the named object `ACPool`, which acts as a root of persistency. Active copies communicate with the Process Engine Manager by means of an $O_2$ socket connection. They are also connected to the SCI by a SCI protocol connection (see fig. 4.8).

### 4.4.3 Process Engines

Process Engines are responsible for the execution of multiple SLANG Interpreter instances. They are $O_2$ clients. They execute the $O_2$ function `ProcessEngine`, which holds and manages an $O_2$ object of class `PE` (see fig. 4.7). The `PE` object contains an $O_2$ socket connection to the Process Engine Manager, from which the Process Engine receives active copies to execute. In addition, the `PE` object contains references to the managed SLANG Interpreters. Every `PE` object is referenced by the named $O_2$ object `PEPool`, which acts as a root of persistency (see fig. 4.8).

### 4.4.4 Process Engine Manager

The Process Engine Manager coordinates the Process Engines, and manages the communication in the Process Enactment Environment. It is an $O_2$ client. It executes the `ProcessEngineManager` $O_2$ function which manages the `PEManager` named object. If requested, the `ProcessEngineManager` function is able to execute also the duties of a Process Engine, managing a `PE` object (see fig. 4.7).

Process Engine Manager behavior, regarding active copy allocation policy and Process Engine invocation, can be customized using the Process Engine Manager configuration file.

### 4.4.5 SPADE Manager

The SPADE Manager is the interface towards the SPADE-1 system administrator. It is a Unix process. Its task is the initialization of the SPADE-1 environment. At startup time, the SPADE Manager reads from the SPADE configuration file information about the system architecture customization. Then, it forks the Process Engine Manager, the SPADE Monitor and the SCI. The SPADE Manager is connected to the Process Engine Manager with a SPADE protocol connection, through which the system administrator can send messages, influencing SPADE-1 behavior.

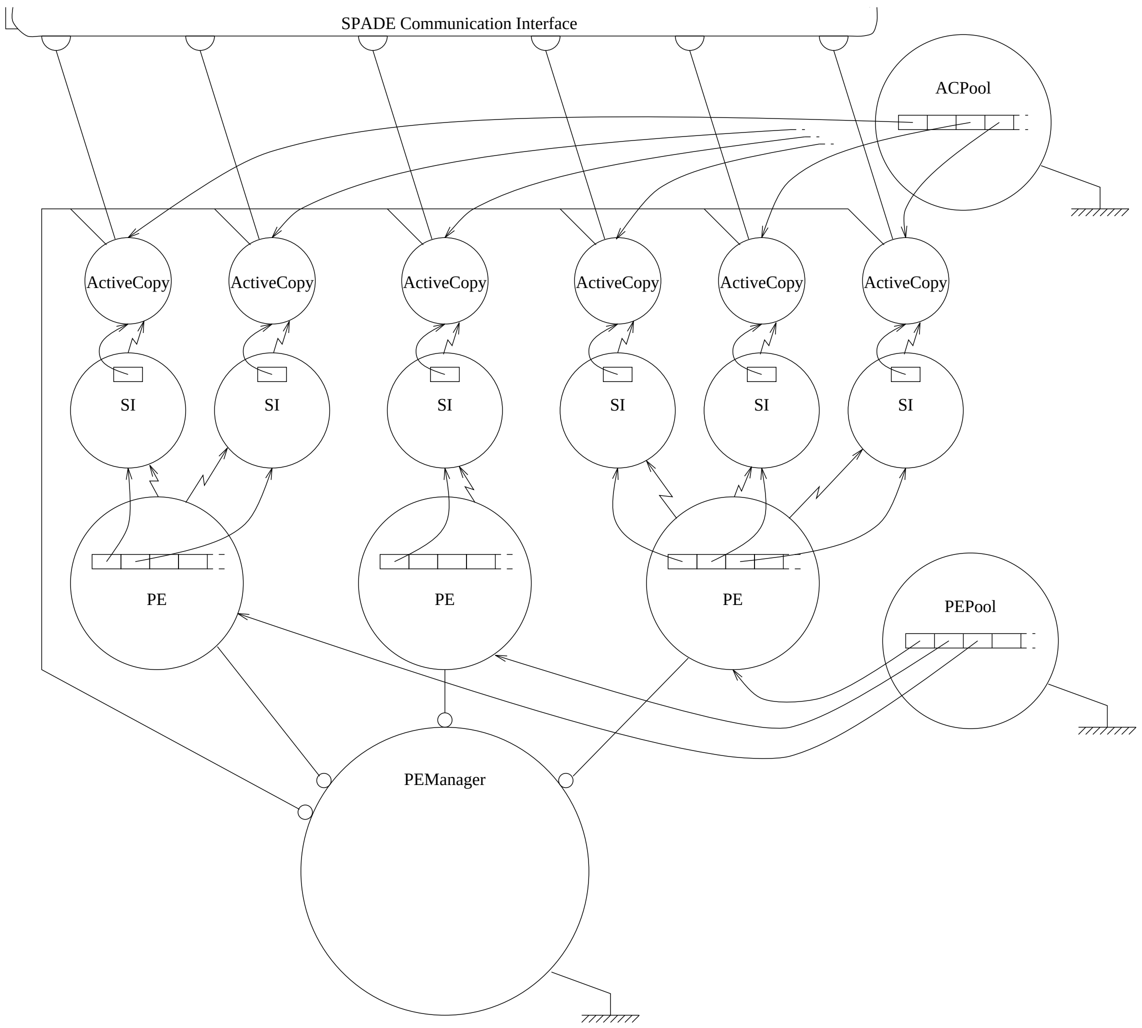


Figure 4.8: Process Enactment Environment $O_2$ objects.

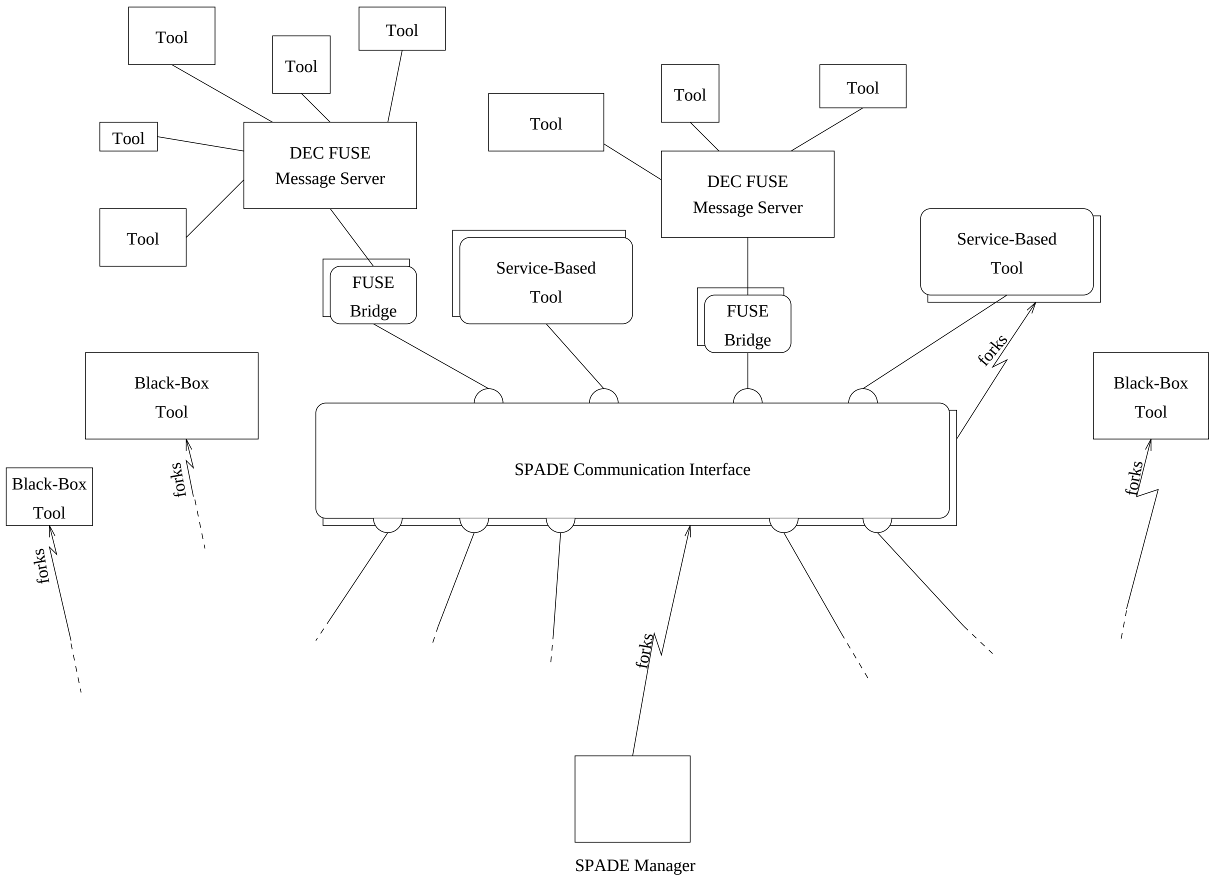


Figure 4.9: The SCI and the User Interaction Environment.

### 4.4.6 SPADE Monitor

The SPADE Monitor displays graphically information about process enactment. It is a Unix process, connected to the Process Engine Manager by a SPADE protocol connection, through which it receives audit information.

## 4.5 SPADE Communication Interface

The SPADE Communication Interface is a filter which allows communication between the PEE and the UIE. The communication is based on the message passing paradigm, and follows a precise protocol. The SCI is a Unix process. It is connected with each SLANG Interpreter and each service-based tool or tool integration environment. The SCI allows the SLANG Interpreters to reach the services exported from the tools in the UIE, and dispatches notifications of relevant events, coming from the UIE, to the SLANG Interpreters.

## 4.6 User Interaction Environment

The User Interaction Environment task is the interaction with the software process agents. Users coordination and interaction is achieved through integrated tools. Tools in the UIE can be *black-box tools* or *service-based tools* depending on the level of control integration. Black-box tools are forked by SLANG Interpreters as a consequence of black transition firings. Service-based tools provide services as follow-ups to SLANG Interpreters requests, and notify the PEE of relevant events. Tools can be Unix processes or $O_2$ clients, depending on the level of data integration. Service-based tools can be directly connected to the SCI or can belong to integrated software development environments (see fig. 4.9).

Chapter 5

# Process Enactment Environment

The Process Enactment Environment is responsible for the execution of the process model. During execution, process activities, designed by the process modeler using the SLANG language, are invoked and concurrently interpreted. Interpretation of active copies produces effects in the User Interaction Environment and reacts to actions performed by human actors, providing support to software development.

The SPADE-1 2.0 Process Enactment Environment is composed of:

- SPADE Manager;
- Process Engines;
- Process Engine Manager;
- SLANG Interpreters;
- SPADE Monitor;
- $O_2$ OODBMS.

Process model enactment is accomplished through the application of the SLANG Interpreter algorithm to active copies, which are instances of activity definitions [Car94]. Multiple SLANG Interpreter instances are concurrently executed within each Process Engine. The Process Engine Manager coordinates the Process Engines and audits the environment [VZ93]. The SPADE Manager is the front-end towards the SPADE-1 system administrator. The SPADE Monitor displays graphically the audit information it receives from the Process Engine Manager.

In the following sections details on these components will be provided. Section 5.1 presents the $O_2$ OODBMS, which represents the backbone of the entire Process Enactment

Environment. Section 5.2 presents the SPADE Manager. Sections 5.3 and 5.4 describe the Process Engine Manager and the Process Engines, respectively. Section 5.5 outlines the SLANG Interpreter. Section 5.6 shows the PEE architecture at work. Section 5.7 describes auditing messages. Section 5.8 is about PEE configuration issues.

## 5.1 $O_2$ OODBMS

$O_2$ [O292], [Deu91] is a distributed Object-Oriented Data Base Management System, based on a client-server architecture.

The logical structure of an $O_2$ data base is bound to a *schema*, i.e., a collection of names and definitions of classes, types, applications, objects and values. There may exist any number of logically separate schemas at a time.

A *base* is a collection of data whose structure conforms to the structural definition in a schema. Several different bases might be associated with each schema.

Data manipulation is achieved using the $O_2$C language, an extension of ANSI-standard C, as well as the $O_2$SQL, an *ad hoc* database object-oriented query language, whose syntax is styled on IBM SQL standard, and which is likely to become the SQL standard for OODBMSs. $O_2$ also provides an interface towards standard programming languages, namely C and C++.

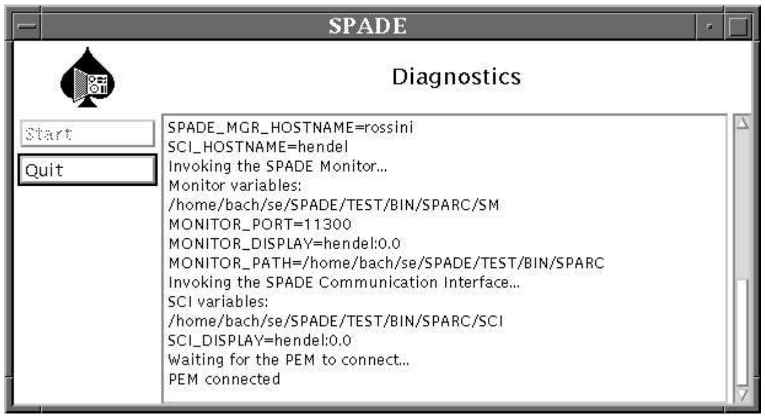
Data are represented by *values* and *objects*. A value is an instance of a given *type*. A type is a generic description of a data structure in terms of atomic types (integers, characters, and so on) and structured types (tuples, sets, and lists). An object is an instance of a given *class* and encapsulates a value and the behavior of that value. The behavior of an object is fully described by the set of *methods* attached to it.

An object or a value may be given an identifier, i.e., a *name*, by which $O_2$ commands, methods, and application programs may refer to it quickly and specifically. Such name is global within the schema.

Objects and values in the system can be either *persistent* or not. An object is persistent if it remains in the database after the successful termination of the transaction which created it. Persistence is granted as follows: *Every named object in the database is persistent, and every component of a persistent object is persistent. No other objects are persistent.* The same rule applies to values.

Thus, named objects and named values are the roots of persistence. That is, they are used as handles from which every persistent object or value can be reached.

Every update to persistent data must be performed within a transaction. If two clients access the same object or value in transaction mode, the locks obtained by the first client force the second to wait. Thus, critical sections corresponding to updates should be limited



```
SPADE_MGR_HOSTNAME=rossini
SCI_HOSTNAME=hendel
Invoking the SPADE Monitor...
Monitor variables:
/home/bach/se/SPADE/TEST/BIN/SPARC/SM
MONITOR_PORT=11300
MONITOR_DISPLAY=hendel:0.0
MONITOR_PATH=/home/bach/se/SPADE/TEST/BIN/SPARC
Invoking the SPADE Communication Interface...
SCI variables:
/home/bach/se/SPADE/TEST/BIN/SPARC/SCI
SCI_DISPLAY=hendel:0.0
Waiting for the PEM to connect...
PEM connected
```

Figure 5.1: SPADE Manager graphic interface.

in time, and should not involve several objects, in order to improve performance and avoid deadlocks[1].

Objects and values not bound to a persistency root are automatically garbage-collected at the end of a transaction.

## 5.2 SPADE Manager

The SPADE Manager is the Unix process which actually enters the SPADE-1 environment. The SPADE Manager presents to the SPADE-1 system administrator a Motif [Hel91] graphic interface, from which it is possible to start and stop process enactment (see fig. 5.1). A window presents diagnostic messages about system startup.

SPADE Manager behavior can be customized using a configuration file (see 5.8). When enactment starts, the SPADE Manager reads its configuration file and then forks the Process Engine Manager, the SCI and the SPADE Monitor processes. After this, the SPADE Manager waits for the request, from the Process Engine Manager, of the creation of a communication link, through which further messages can influence system behavior.

Whenever needed, it is possible to abort process enactment from the SPADE Manager. In that case, the SPADE Manager sends a quit message to the Process Engine Manager. The Process Engine Manager broadcasts it to every Process Engine, causing their termination. Before terminating, the Process Engines "freeze" the execution of the process, allowing post-mortem analysis of the status of the process.

[1]In the current implementation of $O_2$, locks are associated with pages rather than with objects. This may lead to the odd situation in which clients working on completely different objects within the same base can experience deadlocks. $O_2$Technology is steering towards replacing page locking with object locking.

## 5.3 Process Engine Manager

The Process Engine Manager is the "pivot" of the Process Enactment Environment architecture. It collects and manages data about the PEE status and coordinates the execution of the Process Engines.

Process Engine Manager tasks are:

- Process Engines management: the Process Engine Manager invokes Process Engines processes when needed, allocates invoked active copies to Process Engines, trying to balance the Process Engines load, and manages the communication among different Process Engines.
- SLANG Interpreters management: the Process Engine Manager is connected to each SLANG Interpreter and manages information coming from active copies execution, including active copy invocations, transition firings and terminations.
- Monitoring management: the Process Engine Manager receives event notifications from the Process Engines and the SLANG Interpreters, and translates them into messages, forwarded to the SPADE Monitor.
- System management: the Process Engine Manager receives messages from the SPADE Manager and acts accordingly.
- Configuration management: the Process Engine Manager, thanks to its central role, is able to store updated information about PEE status and configuration.

Communication among the Process Engine Manager, the Process Engines and the SLANG Interpreters is based on message exchange. The Process Engine Manager uses the $O_2$ socket primitives to open and close communication channels, and to exchange data. Each message is an $O_2$ object, containing:

**reference number** : an integer, representing the message identifier.

**name** : a string, representing the type of the message.

**parameter** : a string, containing message parameters.

**data** : a list of $O_2$ objects.

## 5.4 Process Engines

Process Engines are responsible for the management and execution of multiple SLANG Interpreter instances. They are connected to the Process Engine Manager by an $O_2$ socket, through which they receive active copies to be executed. Upon receiving active copies, they create SLANG Interpreter instances and allocate active copies to them. They implement logical concurrent execution of SLANG Interpreters, calling the `execute` method on each SLANG Interpreter they are managing, in a round robin loop.

## 5.5 SLANG Interpreters

The SLANG Interpreter task is the execution of an active copy. In the description of the PEE, the SLANG Interpreter has been outlined as the executor of the SLANG Interpreter algorithm. As it will be explained in chapter 6, the real SLANG language interpretation algorithm is implemented in the methods attached to the active copy $O_2$ object. In addition, SLANG Interpreter data structures and references to the connections with the Process Engine Manager and the SCI are in the active copy object. Actually, the SLANG Interpreter $O_2$ object (of class `SI`), managed by the corresponding Process Engine, is just an envelope for the active copy. It provides an abstract interface towards the Process Engine, recording the status of the active copy execution, and hiding the different operations to be performed on the active copy, depending on its status. The SLANG Interpreter $O_2$ object exports to the Process Engine the single method `execute`, which executes a step of the SLANG Interpreter algorithm, calling the appropriate methods on the active copy object it owns.

For the sake of clearness, in the following description we use the term "SLANG Interpreter" to denote the `SI` object together with its active copy. We stress that the the SLANG Interpreter object is an architectural abstraction, more than a computational module.

## 5.6 Process Enactment Environment at work

### 5.6.1 Process enactment startup

The Process Engine Manager is forked by the SPADE Manager when enactment starts. At startup time, the Process Engine Manager performs the following steps:

- Initializes the $O_2$ database, resetting the persistency roots.

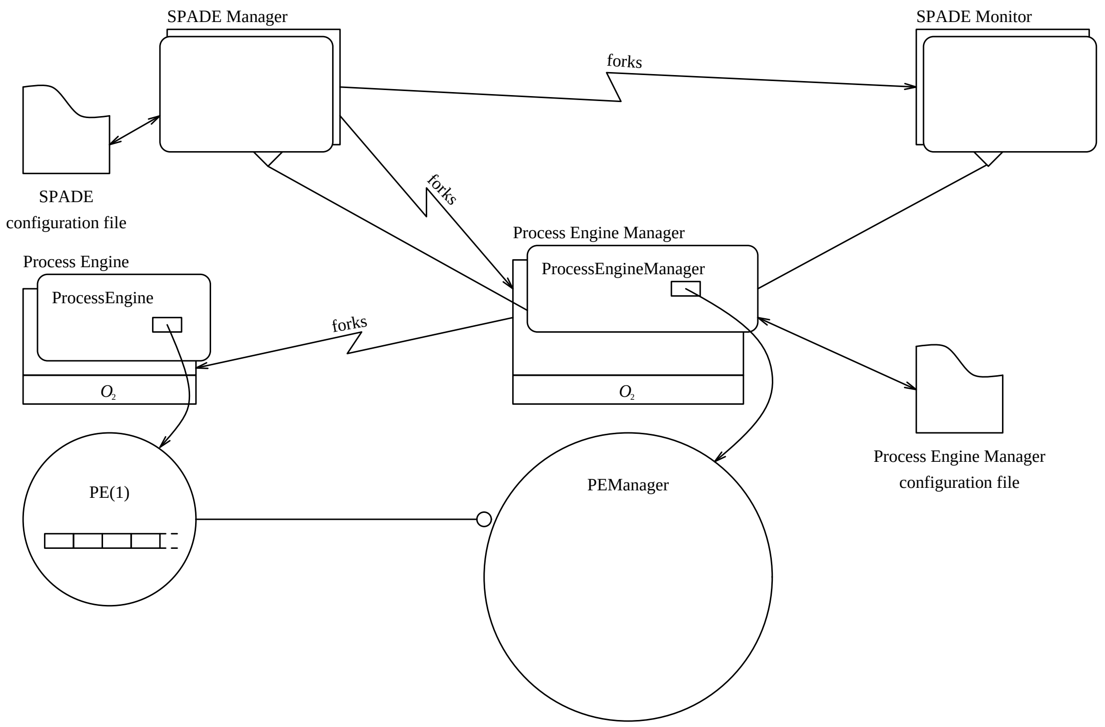


Figure 5.2: Process Enactment Environment startup.

- Connects to the SPADE Manager and the SPADE Monitor (if monitoring is enabled), using the SPADE protocol primitives.
- Retrieves from a configuration file information about SLANG Interpreters and Process Engines static and dynamic creation and allocation policies, as well as Process Engines distribution in the LAN (see section 5.8.2).
- Forks the Process Engine startup instances and creates a communication link with each invoked Process Engine using the $O_2$ socket primitives. Figure 5.2 shows a snapshot of the PEE at this point, assuming that a single Process Engine has been forked at startup time.
- Retrieves the root active copy from the $O_2$ database and sends a message to an active Process Engine, containing:
  - a null reference number;
  - the `ExecuteAC` name;
  - a null parameter;
  - the root active copy, as data.

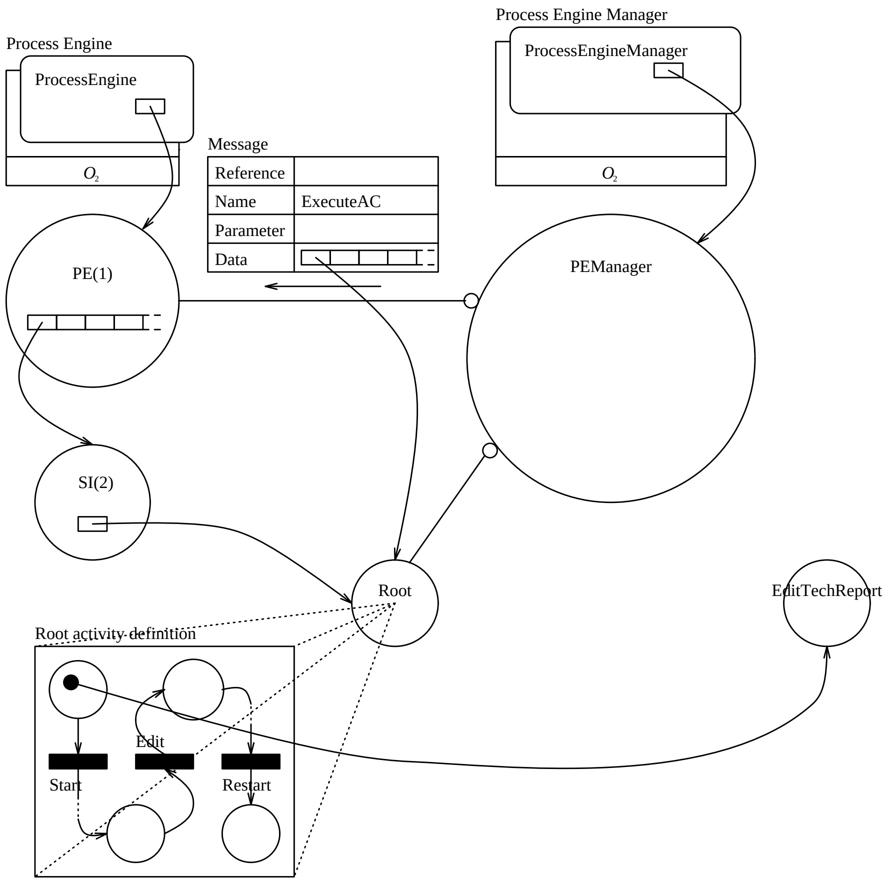


Figure 5.3: Root active copy invocation.

The recipient Process Engine retrieves the active copy from the message, creates a SLANG Interpreter instance and allocates the root active copy to it. Figure 5.3 shows a snapshot of the situation at this point. The figure shows also an example of root activity definition.

After completing setup, the Process Engine Manager enters a loop, in which it polls the connections with the Process Engines, the SLANG Interpreters, and the SPADE Manager, looking for incoming messages. The Process Engine Manager parses received messages and acts accordingly.

The Process Engine, which received the root active copy, begins the execution of the corresponding SLANG Interpreter, this way actually starting enactment.

## 5.6.2 Active copies management

Let us assume the root active copy invokes another active copy, e.g., named `EditTech-Report`, which is a token in the root active copy net (see fig. 5.3). The black transition actually invoking the active copy[2] executes the action `SLANGInterpreter`. Consequently, the SLANG Interpreter, which is interpreting the root active copy, removes the token from the net and sends a message to the Process Engine Manager, containing:

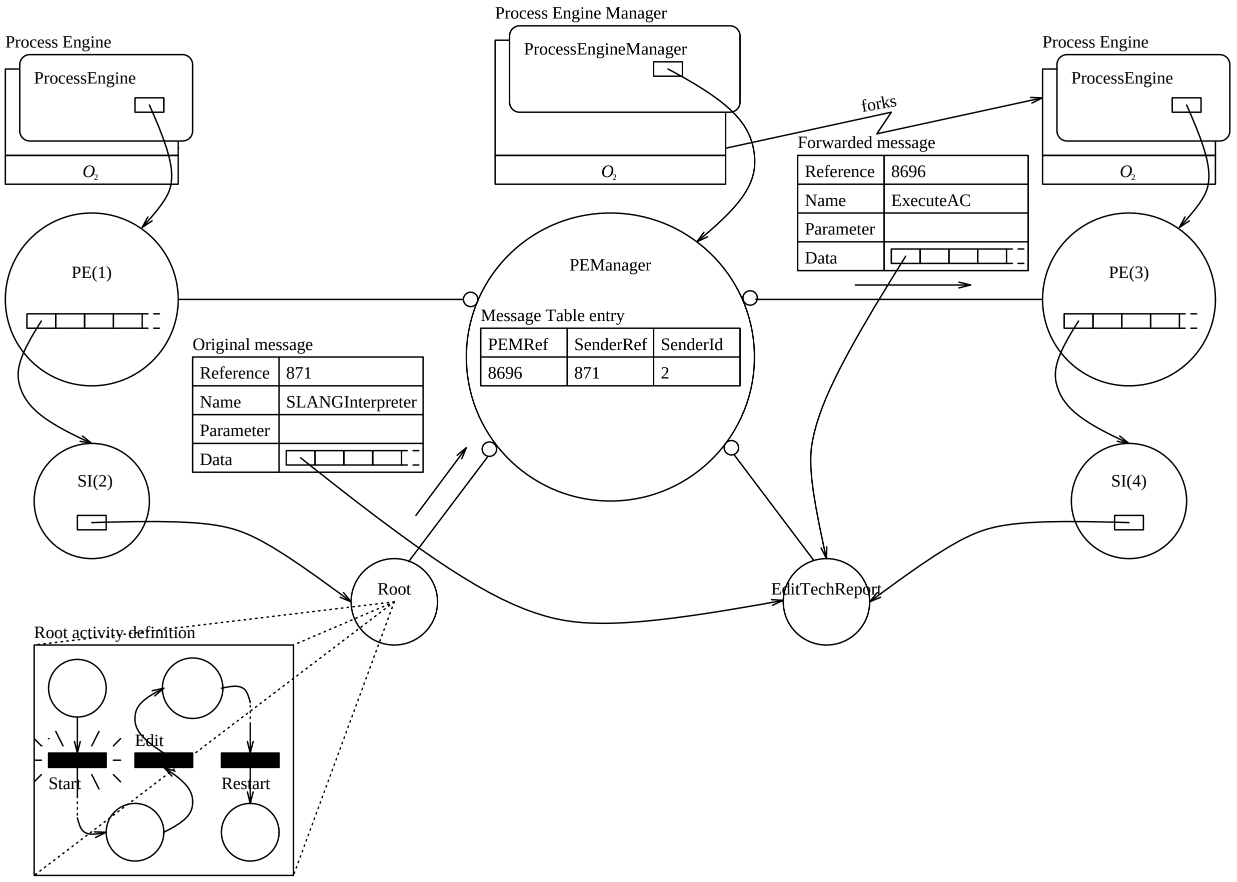
- a message reference number;
- the name `SLANGInterpreter`;
- the invoker SLANG Interpreter identifier as parameter;
- the invoked active copy as data.

When the Process Engine Manager, during its connection polling loop, finds the message, it performs the following steps:

1. It generates a unique message reference number and adds an entry to a *message table*. The entry contains the newly generated reference number, together with the original reference number and the sender SLANG Interpreter identifier.

2. The Process Engine Manager uses the information about the PEE status to decide to which Process Engine deliver the active copy. Suppose that a single Process Engine has been invoked at startup time and that the configuration parameters have been set so that a new Process Engine must be invoked, in order to receive the newly

[2]The activity invocation construct is a full-SLANG feature. The construct is expanded into kernel-SLANG, before being interpreted. The actual operation of removing the token, representing the invoked active copy, from the invoker active copy net, is performed by a black transition.



Forwarded message

| | |
|---|---|
| Reference | 8696 |
| Name | ExecuteAC |
| Parameter | |
| Data | |

Message Table entry

| PEMRef | SenderRef | SenderId |
|---|---|---|
| 8696 | 871 | 2 |

Original message

| | |
|---|---|
| Reference | 871 |
| Name | SLANGInterpreter |
| Parameter | |
| Data | |

Figure 5.4: Active copy invocation example.

invoked active copy. The Process Engine Manager forks the Process Engine process and retrieves the Process Engine connection, using the $O_2$ socket primitives.

3. The Process Engine Manager sends a message to the Process Engine containing:

   - the new message reference number;
   - the `ExecuteAC` message name;
   - a null parameter;
   - the active copy to be executed as data.

When the Process Engine finds the message, it creates a SLANG Interpreter instance associated with the received active copy, and begins its execution. The situation, at this point is depicted in figure 5.4.

Suppose the active copy `EditTechReport` has fired a transition marked as *suspending.* The Process Engine, which is executing the SLANG Interpreter associated to the

`EditTechReport` active copy, retrieves the active copy from the SLANG Interpreter. Them, it creates, in the active copy, a reference to the SLANG Interpreter object and it sends a message to the Process Engine Manager, containing:

- the reference number of the message which started the active copy;
- the name `SuspendedAC`;
- a null parameter;
- the suspended active copy, `EditTechReport`, as data.

Upon receiving the message, the Process Engine Manager, using the message reference number as a key, retrieves from the message table the invoker SLANG Interpreter identifier, as well as the original message reference number. Then, it forwards a message to the invoker SLANG Interpreter containing:

- the original invocation message reference number;
- the name `TerminatedAC`;
- a null parameter;
- the suspended active copy as data.

The recipient SLANG Interpreter uses the information contained in the message to terminate the firing of the black transition which previously invoked the active copy.

Now, the root active copy owns the `EditTechReport` active copy as a token. It can modify its contents or even its definition (hopefully, improving it). The situation, at this point, is shown in figure 5.5.

Suppose that, after having modified the `EditTechReport` active copy, the root active copy decides to resume its execution. At this moment, the `EditTechReport` active copy token is in the place which represents the postset of transition `Edit`. Again, a black transition in the net executes the `SLANGInterpreter` action. Consequently, the root active copy SLANG Interpreter removes the token from the net and sends a message to the Process Engine Manager containing:

- a message reference number;
- the `SLANGInterpreter` message name;
- a null parameter;
- the restarted active copy as data.

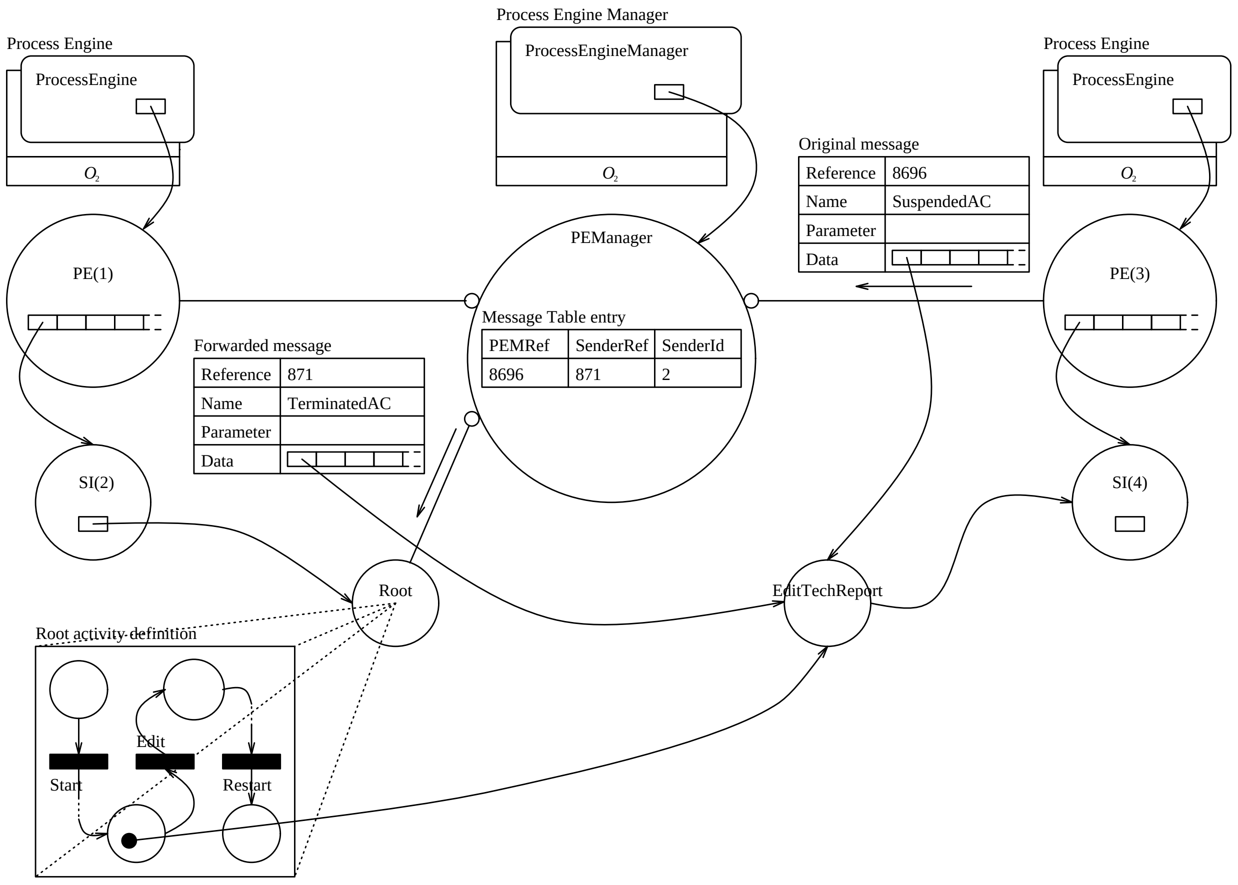


Figure 5.5: Active copy suspension example.

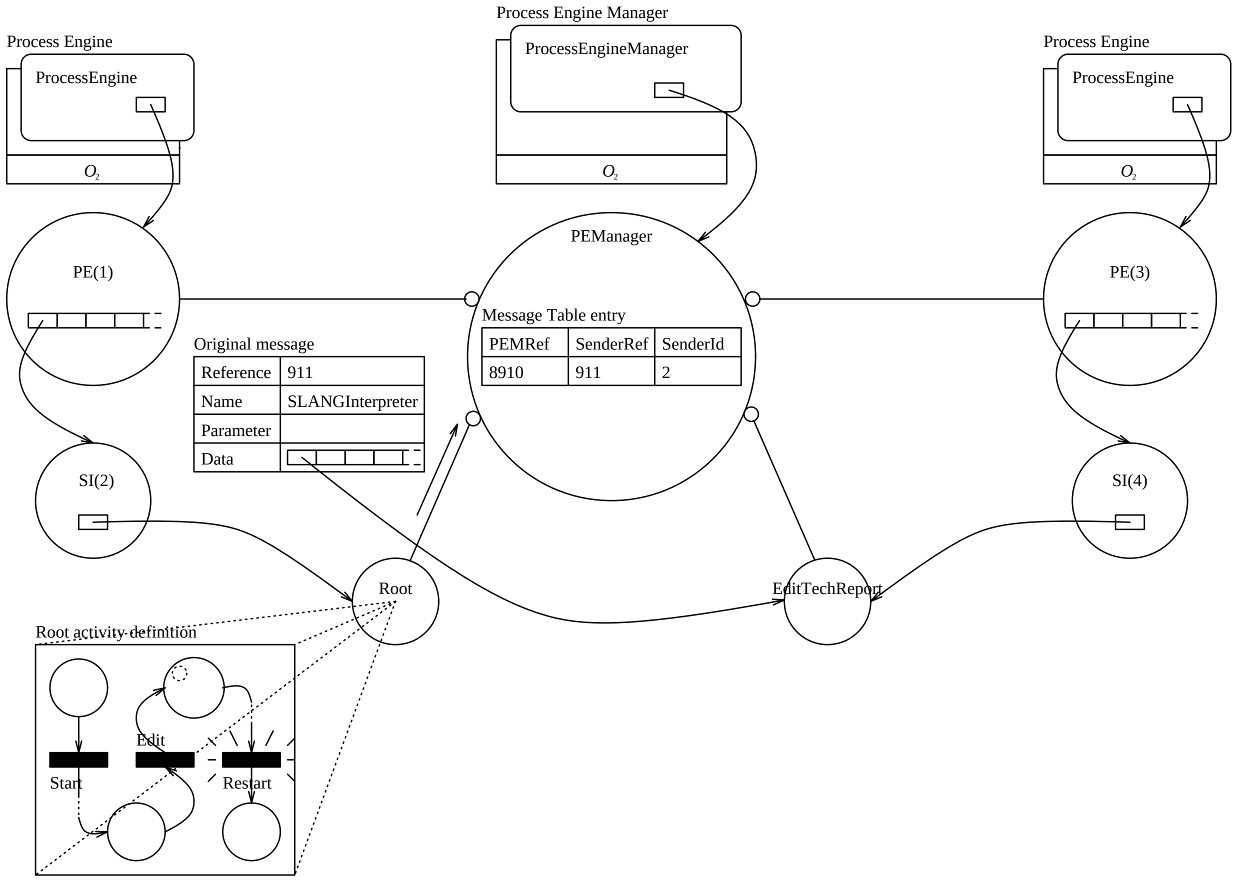


Figure 5.6: Active copy restart example.

The Process Engine Manager receives the message, and performs the following actions:

1. It generates a unique reference number and adds an entry to the message table. The entry contains the newly generated reference number, together with the original reference number and the root SLANG Interpreter identifier.

2. The Process Engine Manager recognizes the active copy as a previously suspended one. Then, using the reference to the `SI` object contained in the active copy, *directly* allocates the active copy to the SLANG Interpreter that was interpreting it before suspension.

The SLANG Interpreter, realizes that the suspended active copy has come back and resumes its execution. This active copy restart example is presented in figure 5.6.

Let us suppose the active copy `EditTechReport` fires a transition marked as *ending*, i.e., it terminates. The Process Engine, which is executing the SLANG Interpreter instance corresponding to the `EditTechReport` active copy, retrieves the terminated active copy from the SLANG Interpreter and sends a message to the Process Engine Manager containing:

- the reference number of the message which (re)started the active copy;
- the name `TerminatedAC`;
- a null parameter;
- the terminated active copy `EditTechReport` as data.

After this, the Process Engine destroys the SLANG Interpreter instance.

When the Process Engine Manager finds the message, it performs the following steps:

1. Using the message reference number as a key, it retrieves the invoker SLANG Interpreter identifier and the original message reference number from the message table.
2. Sends to the invoker SLANG Interpreter a message containing:
   - the original invocation message reference number;
   - the name `TerminatedAC`;
   - a null parameter;
   - the terminated active copy, `EditTechReport`, as data.

Upon receiving the message, the recipient SLANG Interpreter uses the attached information to complete the firing of the black transition which restarted the active copy.

The PEE status at this point is shown in figure 5.7.

When the root active copy terminates, the whole enactment has to stop. The Process Engine Manager kills the active Process Engines, closes connection with both the SPADE Manager and the SPADE Monitor, and saves the root active copy in a persistent variable of the $O_2$ database. This way, useful information about the terminated process can be retrieved using the query features of the $O_2$ database.

## 5.7 System monitoring

The Process Engine Manager is responsible for the notification of relevant events to the SPADE Monitor. As the previous example points out, every active copy invocation, termination, suspension and restart, involves the Process Engine Manager. This is the reason why only the Process Engine Manager is connected to the SPADE Monitor: all relevant information about the process enactment development passes through the Process Engine Manager which notifies events to the SPADE Monitor.

The Process Engine Manager notifies to the SPADE Monitor the following events:

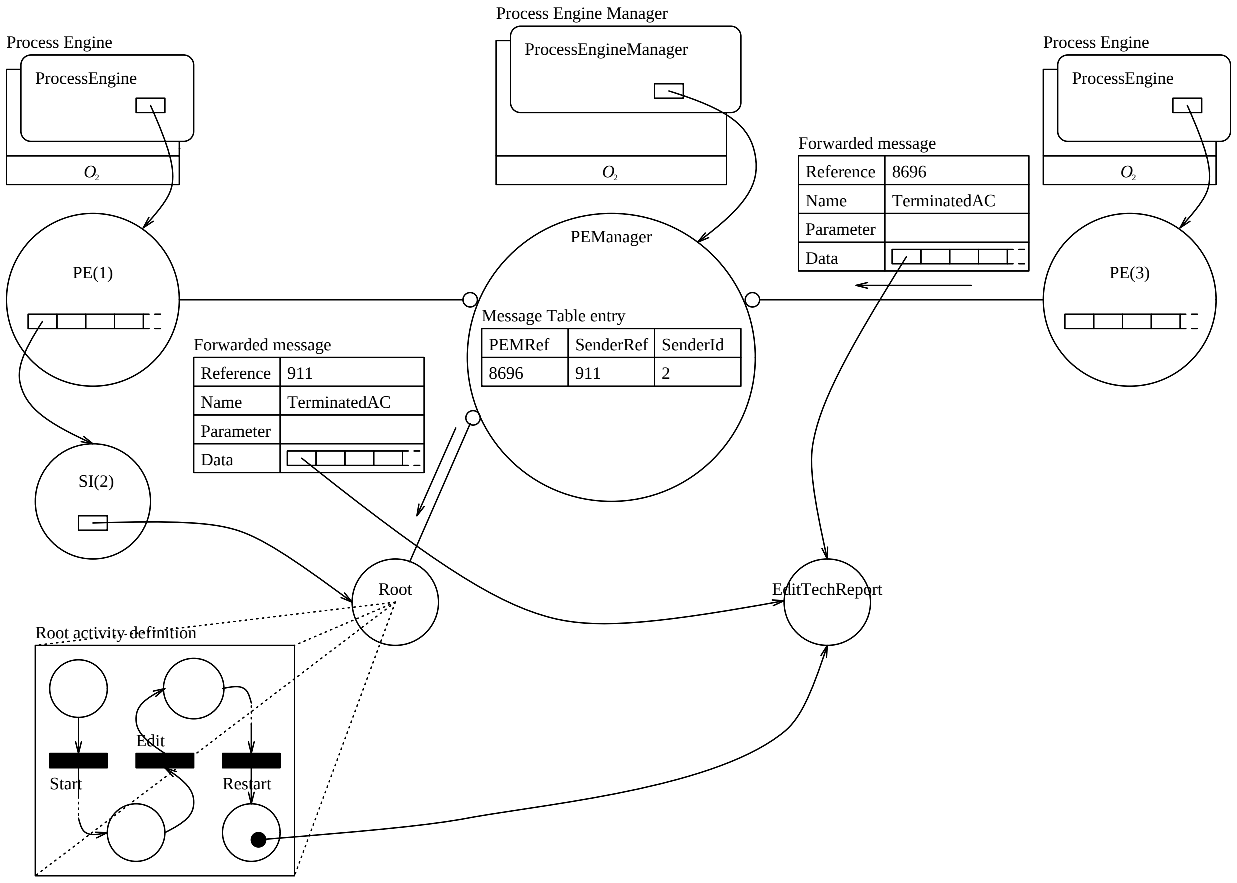


Figure 5.7: Active copy termination example.

**Process Engine invocation** : the Process Engine Manager sends a message to the SPADE Monitor containing the Process Engine identifier and the host name on which the Process Engine runs.

**Process Engine termination** : the Process Engine Manager notifies the SPADE Monitor of the termination of a Process Engine, passing its identifier in the message.

**Active copy invocation** : the Process Engine Manager notifies the SPADE Monitor of active copy invocations, passing the following information:

- the invoked active copy name;
- the invoked active copy identifier;
- the invoker active copy identifier;
- the Process Engine to which the invoked active copy has been allocated.

**Active copy firing** : the Process Engine Manager receives from the SLANG Interpreters the notifications of transition firing. The firing notification messages contain:

- a null reference number;
- the `TransitionNotify` name;
- the active copy identifier and the fired transition name as parameters;
- null data.

A similar message is forwarded to the SPADE Monitor by the Process Engine Manager.

**Active copy suspension** : the Process Engine Manager notifies the SPADE Monitor of the identifier of the suspended active copy.

**Active copy restart** : the Process Engine Manager notifies the SPADE Monitor of the identifier of the restarted active copy.

**Active copy termination** : the Process Engine Manager notifies the SPADE Monitor of the terminated active copy identifier.

## 5.8 PEE configuration

The SPADE-1 environment can be configured using two configuration files. General configuration variables are specified in the SPADE configuration file, while variables concerning Process Engine Manager specific policies are specified in the Process Engine Manager configuration file.

### 5.8.1 General configuration

The SPADE configuration file contains entries in the form:

```
variable value
```

Lines beginning with a `#` are considered to be comments. The SPADE system administrator can select the name of the configuration file setting the environment variable `SPADE_CONF_FILE` before invoking the SPADE Manager. If no configuration file has been chosen, the SPADE Manager looks for the default configuration file `.SPADE` in the current working directory.

A description of the meaning of each configuration variable follows:

`O2_SERVER_NAME` : specifies the $O_2$ server host[3]. It is mandatory.

`O2_SYSTEM_DIR` : specifies the $O_2$ system directory. It is mandatory.

`SPADE_SYSTEM_NAME` : specifies the SPADE-1 $O_2$ system. It is mandatory.

`SPADE_BASE_NAME` : specifies the SPADE-1 $O_2$ base. It is mandatory.

`SPADE_ROOT` : specifies the SPADE-1 file system root directory. It is mandatory.

`PE_MGR_HOSTNAME` : specifies the host on which the Process Engine Manager runs. It is mandatory.

`PE_MGR_ARCH` : specifies the Process Engine Manager program executable format. Currently, it can take only the value `SPARC`[4]. It is mandatory.

`PE_MGR_CONFIG_FILE` : specifies the Process Engine Manager configuration file name. The default value is the file `.PEM.defaults`, in the `CONF` directory of the SPADE-1 file system.

`PE_MGR_LOG_FILE` : the Process Engine Manager log file. It is optional. If this variable is not set, notification and error messages are simply discarded.

`PE_ARCH` : the Process Engine program executable format. Currently, it can take only the value `SPARC`. It is mandatory.

[3]All hosts in the configuration files must be specified using their host name. IP addresses are not allowed.

[4]Since the $O_2$ OODBMS version used by the SPADE-1 environment is compiled for the Sun SPARC architecture, all $O_2$ clients can be compiled for the SPARC architecture only.

`PE_LOGGING` : enables Process Engines logging. It can take the values `yes` or `no`. The default value is `no`. If set to `yes`, log files with a unique name (composed by the `.ProcessEngine` prefix, followed by the Process Engine process id) are created under the `/tmp` directory.

`SCI_ARCH` : the SPADE Communication Interface program executable format. It can take the values `SPARC` or `MIPS`. It is mandatory.

`SCI_HOSTNAME` : specifies the SPADE Communication Interface host. It is mandatory.

`SCI_DISPLAY` : specifies the SPADE Communication Interface display. It is mandatory.

`SCI_LOG_FILE` : the SPADE Communication Interface log file. It is optional. By default, no log file is created.

`MONITOR` : enables/disables monitoring. It can take the values `yes` or `no`. This variable must be set to `no` if the SPADE Monitor executable is not present, or if no monitoring is required. It is mandatory.

`MONITOR_ARCH` : the SPADE Monitor program executable format. Currently, it can take the values `SPARC` or `MIPS`. It is mandatory, if `MONITOR` is set to `yes`.

`MONITOR_HOSTNAME` : the host on which the SPADE Monitor runs. It is mandatory, if `MONITOR` is set to `yes`.

`MONITOR_DISPLAY` : specifies the SPADE Monitor display. It is mandatory, if `MONITOR` is set to `yes`.

Here follows a sample configuration file:

```
#
# SPADE SAMPLE CONFIGURATION FILE
#

# O2 PARAMETERS
O2_SYSTEM_DIR /home/rossini/O2
O2_SERVER_NAME rossini
SPADE_SYSTEM_NAME SPADE
SPADE_BASE_NAME SPADEBase

# SPADE file system root dir
```

```
SPADE_ROOT /usr/local/SPADE

# PE MANAGER PARAMETERS
PE_MGR_HOSTNAME rossini.cefriel.it
PE_MGR_ARCH SPARC
PE_MGR_CONFIG_FILE /usr/local/SPADE/CONF/.PEM.test
PE_MGR_LOG_FILE /tmp/.ProcessEngineManager

# PE PARAMETERS
PE_ARCH SPARC
PE_LOGGING yes

# SCI PARAMETERS
SCI_ARCH SPARC
SCI_HOSTNAME bellini
SCI_DISPLAY bellini:0.0
SCI_LOG_FILE /tmp/.SCI

# MONITOR PARAMETERS
MONITOR yes
MONITOR_ARCH MIPS
MONITOR_HOSTNAME bach
MONITOR_DISPLAY bellini:0.0
```

### 5.8.2 Process Engine Manager configuration

The behavior of the Process Engine Manager, concerning the allocation of active copies (and therefore SLANG Interpreter instances) to the Process Engines, can be customized. The Process Engine Manager configuration file can be specified using the `PE_MGR_CONFIG_FILE` variable in the SPADE configuration file. The Process Engine Manager configuration file contains entries in the form of:

```
   variable value
```

and it is divided in two main parts: the first one specifies some general parameters for the Process Engine Manager (see below), the second one specifies the host name for each Process Engine. The Process Engine-related part starts with a line beginning with an asterisk. The number of Process Engine host names must be equal to the Process Engine Manager configuration variable `MAX_PE_INST`.

The variables and their meanings are:

`MAX_PE_INST` : it takes an integer value. It specifies the maximum number of Process Engine processes ($O_2$ clients) that will be available during enactment. The variable value does not include a possible Process Engine inside the Process Engine Manager process.

`STARTUP_PE_INST` : it takes an integer value. It specifies the number of Process Engine processes that will be invoked at startup time. Since it takes a relatively long time for a Process Engine to start, when a timely response from the Process Engines is needed, it is recommended to invoke all the Process Engine instances at startup time.

`INTERPR_INST_THRESHOLD` : it takes an integer value. When the Process Engine Manager has to determine the recipient Process Engine of a newly invoked active copy, it looks for the less loaded Process Engine. If the minimum load is bigger than the specified threshold a new Process Engine process is invoked, if possible.

`PE_MGR_IS_A_PE` : it takes the values `yes` or `no`. If set to `yes`, it will cause the Process Engine Manager to have also a "PE behavior", i.e., the Process Engine Manager will behave as both a Process Engine Manager and a Process Engine. This is useful when there is only one $O_2$ client available for the SPADE-1 environment. It should be clear that performance is improved if no "PE behavior" is requested to the Process Engine Manager.

`FIXED_PE_MGR_INTERPR_INST` : it takes an integer value. This variable must be set when "PE behavior" for the Process Engine Manager is requested. If set to a non-zero value it will limit the number of SLANG Interpreter instances assigned to the Process Engine inside the Process Engine Manager process.

`HOST` : it specifies the name of a host in the LAN that will execute a Process Engine process.

Lines beginning with a `#` are considered to be comments.

Here is an example:

```
#
# PEM SAMPLE CONFIGURATION FILE
#

MAX_PE_INST 4
```

```
# This means that no more than four PEs will be running.

STARTUP_PE_INST 0
# No PEs will be invoked at startup time:
# their invocation will simply follow
# the needs of process enactment.

INTERPR_INST_THRESHOLD 5
# During enactment, if every running PE has five
# or more running SLANG Interpreter instances
# and a new active copy must be interpreted,
# another PE is forked by the PEM, if possible.

PE_MGR_IS_A_PE no
# No "PE behavior" is requested inside the PEM process.

FIXED_PE_MGR_INTERPR_INST 0
# Since no "PE behavior" is set,
# this variable has no meaning.

************************************

# Here begins the PE-related part

HOST bellini
# A PE instance will be executed
# on the machine named "bellini".

HOST hendel.cefriel.it
# A PE instance will be executed on the machine whose
# name is "hendel.cefriel.it".

HOST rossini
# A PE instance will be executed
# on the machine named "rossini".
```

```
HOST bellini
# Another PE instance will be executed
# on host "bellini"
```

We stress that the Process Engine host names have to be as many as specified by the `MAX_PE_INST` variable.

Another example follows. This time, we will describe a Process Engine Manager configuration file that allows the use of a single $O_2$ client for process enactment. This configuration is mandatory whenever only one $O_2$ client license is available. The Process Engine Manager process behaves as a Process Engine Manager and a Process Engine.

```
#
# PEM SAMPLE CONFIGURATION FILE
# (single O2 client)

MAX_PE_INST 0
# This means that no PE processes will be running.

STARTUP_PE_INST 0
# Since there are no PE processes,
# no PEs will be invoked at startup time.

INTERPR_INST_THRESHOLD 0
# Zero threshold must be specified
# since no PE instances will be forked at enactment time.

PE_MGR_IS_A_PE yes
# "PE behavior" is requested inside the PEM process.

FIXED_PE_MGR_INTERPR_INST 0
# No fixed number of SLANG Interpreter instances
# for the PEM's PE must be specified,
# since all the SLANG Interpreter instances
# will be executed by the PE inside the PEM.

**********************************
# No PE host names are needed
```

# Chapter 6

# SLANG Interpreter

The interpreter is the heart that makes a SLANG process model work. It enacts active copies evaluating guards, firing transitions and interacting with the user environment through the SCI.

## 6.1 Requirements for the SLANG interpreter

Recalling the principles in chapter 2, we present below the basic requirements related to the SLANG interpreter.

- *SLANG definition*: the definition of the SPADE modeling language must be used. The interpreter must implement kernel SLANG syntax and semantics. Full SLANG primitives are accomplished by kernel SLANG net macros.
- *Process model and data definition independency*: the interpreter code must not depend either on one particular activity definition or on one particular type definition. Therefore, it accept an activity and a set of types as parameters.
- *Time constraints*: though SPADE-1 is not a hard real-time system, time is a variable of the enactment. It is necessary to express time constraints like any other guard condition. The interpreter must also take into consideration the particular nature of such guards because their value may not depend only on input places.
- *Efficiency*: this roughly means a fast execution using available resources. Much effort has been made in order to pursue good performances. This requirement has been the most effective guideline for the development of each part of the interpreter. To some extent, efficiency is the only requirement, while other functional requirements are to be considered constraints.

Other requirements have been set by the global architectural design:

- *Step-by-step execution*: the multiple execution of active copies, in the Process Engine, is a "round-robin" execution of a single step of the interpreter, one for each active copy. Thus the interpreter algorithm must be shaped to export a step-by-step execution method.
- *Global data access and concurrency*: the interpreter must be able to access places belonging to other running active copies (shared and global places), and must be responsible for concurrency control over such operations.

## 6.2 Interpreter schema

The term "SLANG interpreter" is often overloaded. Actually there is no function nor method implementing what is meant to be the SLANG interpreter algorithm. This algorithm is a virtual single function, accepting an active copy with an initial marking, and yielding the same active copy evolved to a final state. Due to the *step-by-step* requirement this logical function is unfolded and distributed along several executions of one method.

Another reason why it is difficult to locate the interpreter code, is that some parts of it are not even present in the SPADE-1 schema. These parts, concerning the evaluation of guards and the execution of actions, are tightly related to the model definition. During the enaction the *static* interpreter code, the one from the SPADE-1 $O_2$ schema, uses some *dynamic* code, the one implementing guards and actions, previously generated according to the model definition.

In this perspective the interpreter can be analyzed at three levels of abstraction: a *logical* level where we take into consideration one entire execution, focusing on the overall behavior of the interpreter. The *execute* level, where the objective is the analysis of one single step. And last the *implementation* level that describes the single methods in the *static* code and the functions of the dynamic code. This conceptual schema is shown in figure 6.1.

For the purpose of giving a precise context to each component of the interpreter, we present first the *logical* level, then how the logical components are mapped on the objects and methods of the *implementation* $O_2$ schema.

## 6.3 Logical level: the interpreter algorithm

At this level of abstraction the interpreter is a *function* taking an active copy, i.e., an activity definition with a *state*. The activity definition is a SLANG net, the state is the

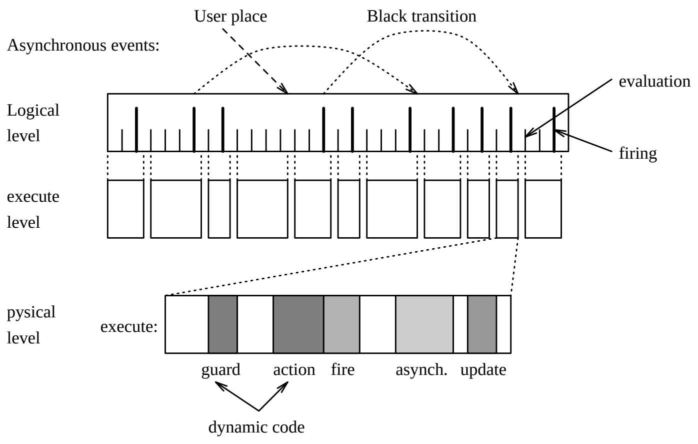


Figure 6.1: Interpreter conceptuals levels.

initial marking. This function processes the active copy causing the evolution of the active copy state. The active copy is returned as the result of the function as soon as one ending transition is fired.

**The algorithm** The algorithm driving the net execution is straightforward:

```
1 - initialize data structures

2 -   choose a transition
3 -   evaluate the guard
4 -   IF enabled THEN fire the transition
5 -   look for asynchronous events
6 -   update data structures
7 -   repeat from (2) until an ending transition is fired

8 - STOP
```

The algorithm repeatedly searches for an enabled transition, the first one that is enabled is immediately fired. After the firing or after the unsuccessful evaluation of the transition, the interpreter checks the connections (see section 6.6) for an asynchronous event occurrence, in case it executes the corresponding actions.

Steps 2 to 7 correspond to the *single step*. The semantics of the language concerning guards and actions is implemented in step 3 and 4 by some dynamically generated functions (see section 6.8).

Much of the algorithm logic is hidden inside the auxiliary data structures mentioned in step 1 and 6 (see section 6.7). They implement the policy for choosing the guards to be evaluated (step 2). In the logical level the evaluation sequence follows some precise criteria.

- *priority*: lower priority transitions are taken into consideration only after all the higher priority ones have been evaluated with a negative result.
- *efficiency*: the interpreter first evaluates the transitions that are much likely to be enabled. Transitions whose preset has not changed since the last negative evaluation are not evaluated again.

Step 5 controls the mechanism of interaction with the "outside" world. Namely, catches incoming messages, i.e.,black transition terminations and notifications directed to some user place. Step 4 performs some output operations related to the firing of black transitions. Thus the I/O mechanisms are implemented by step 5 and 4 respectively.

Note that transition evaluations are, by far, the most frequent events in the net evolution, so an asynchronous events look-up for every guard evaluation seems to be a waste of time. Our little experience in "using" SPADE-1 has emphasized the importance of a short reaction time to user actions. This has to be a strong point for the development of the algorithm, so we preferred to give the highest priority to user messages.

## 6.4 Execute level

The starting point for the analysis of the algorithm is the `execute` method of the class `ActiveCopy`. It implements the sequence 2..6 in the logical view given in section 6.3 on page 70.

At this level the execution is a sequence of separate steps. The steps are independent from each other. The Process Engine that executes the interpreter, can not distinguish between two execution steps. As we can see in figure 6.1, one single step can yield either one or no one transition firing (i.e. when all the transitions are evaluated with negative result).

In order to pursue the correct overall behavior, the interpreter has to save the context of each step in temporary data structures. As explained in the following section, these data structures are part of the `ActiveCopy` type.

The `execute` method returns `TERMINATE` when an ending transition is fired, `SUSPEND` when a suspending transition is fired, otherwise it returns `SUCCESS`.

## 6.5 Implementation level

The central component of the interpreter is one active copy object, thus the structure of the interpreter corresponds to the type of the class `ActiveCopy`. An architectural schema of this class is sketched in figure 6.2. The most important data held by one `ActiveCopy` object concerns:

- activity definition
- activity state
- additional data structures
- links to other architectural components

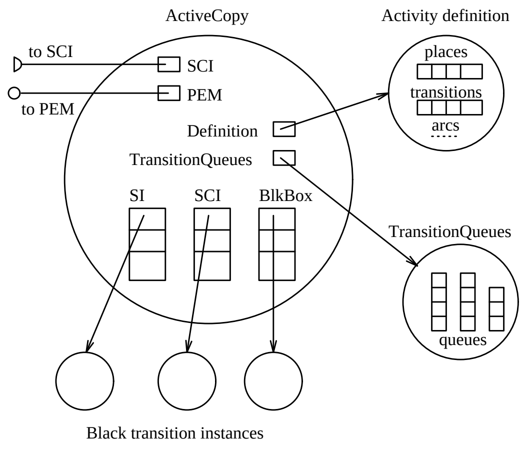


Figure 6.2: Active Copy architecture.

Other minor variables are present. One is an integer counting the black transition instances, that is used to give them a unique identifier. Another two are not used at all by the interpreter, but serve the suspension mechanism (see section 5.6.2), they are the SI holding the active copy, and a flag relating the state of the active copy (running/suspended). Finally the active copy identifier, an integer, used by the interpreter when notifying transition firings and activity invocations.

### 6.5.1 Active copy definition and types

The dynamicity of the SPADE process modeling approach supports activity and types evolution during active copy execution. This means that one active copy may be edited. Its definition may change while the global definition, the one contained in the `Activities` global place, does not. This apply to types as well, so for example, one active copy using the class `Document`, may add some new features to that class without affecting the global definition of `Document` stored in the place `Types`. For such reason an active copy must be a self-contained object, and must be equipped with copies of both activity and types definition.

Activity definition holds the data specifying the SLANG net topology and all the information attached to arcs, places and transitions (weight of the arcs, places types, transitions guards and actions, transition priority etc.). The attribute `definition`, an object of the class `Activity` does this job.

Type definitions are listed in a set of `MetaType` objects. Each one provides the $O_2$C code that defines attributes and methods of the class, and other information related to the type management system (see appendix C.4).

The interpreter actually uses a little part of these definitions, in particular only the net topology is consulted run-time. All the other relevant aspects of the activity and types are implicitly accessed through compiled functions implementing guards and actions (see section 6.8).

### 6.5.2 Active copy state

The state of the active copy is the collection of all the model data associated with that particular activity instance. In a SLANG model, data are tokens, thus the state is the marking of the net. Each place defines a token repository, namely an object of the class `Container`, the set of all the containers holds the state of the active copy.

There is no particular attribute encapsulating the state in an `ActiveCopy` object. Instead, each token container is directly accessible through its `Place` object in the activity definition (see figure 6.3).

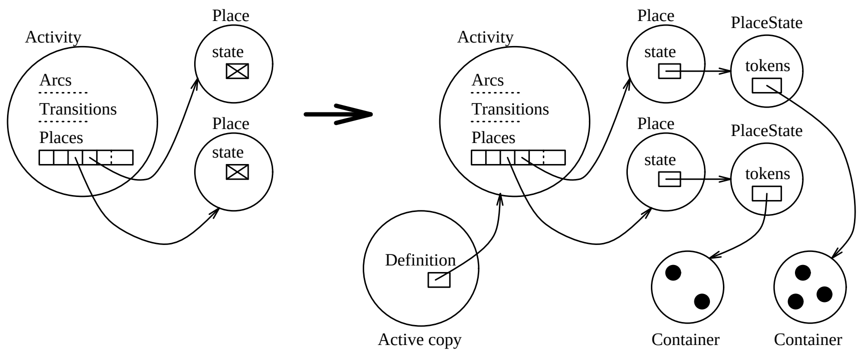


Figure 6.3: Activity and Active copy with state.

**Local places and shared places** The `state` attribute of the class `Place` is the link that dynamically binds the `Container` object associated to a place. As far as place state is concerned we must distinguish two kinds of places. Local places, which are private in the activity, and shared places, that are accessible by two or more active copies. A local place has a private state, its container is not accessible by any other active copy.

Shared places can be further partitioned. A shared place that is in the activity implementation, is said to be an *actual* place and has a direct reference to the container. Conversely, when the active copy shares the place through a link, i.e., when the place belongs to the activity interface, the shared place is called *formal* place, and it has a symbolic reference to the actual container.

Given these preliminary definitions, we can examine container references in details. `state` is an object of class `PlaceState`, this class has two sub-classes: `LocalPlace` and `RemotePlace`. There are only instances of these two classes. The first case apply whenever the place is normal or an actual shared place. The second class represents a symbolic link to a shared formal place.

Both `LocalPlace` and `RemotePlace` have a private data structure containing the pointer to the actual container. `LocalPlace` objects are initialized with a brand new container, while the initial value for `RemotePlace` objects is `nil`. A `RemotePlace` also holds a couple of other items that are the symbolic link to the actual container:

`owner` : it is the integer identifier of the active copy holding the actual place. Comparing activity invocation with procedure call in a structured programming language, it is just like the address of the activation record containing data.

`remoteName` : it is the string identifier of the place containing the actual container, in the owner activity definition (the actual name). Again in usual programming languages it is the *offset* of the variable in the activation record.

**Access to place contents** In both `LocalPlace` and `RemotePlace` the actual container is returned by the method `contents`. In class `LocalPlace` it simply returns the pointer to the container; in `RemotePlace` the private pointer to the actual container is initially `nil`; the first call to `contents` solves the symbolic reference searching `ACPool` for the *owner* active copy and then its definition for the right place; every following call will simply return the previously computed pointer.

### 6.5.3 Locks

The interpreter must control concurrent access to shared places. The critical section is between step 3 and 4 of the algorithm (page 70). In order to show the potential inconsistency, we ought to expand these two steps a little bit. The operation sequence is:

```
3 -
    a. collect input places container: build input tuple
    b. call the guard function passing input tuple
    c. get the enabling tuple
4 -
    a. IF the enabling tuple is not empty (enabled):
    b. THEN extract the tokens belonging to the enabling tuple
    c.      Call the action function passing the enabling tuple
    d. ELSE drop the enabling tuple
```

Suppose there is one shared place in the preset, what happens if the shared container is modified by another running active copy between step 3.a and 4.b?

**The lock mechanism** In order to avoid these inconsistencies, the collected containers must be previously locked. For this purpose, a boolean private attribute called `locked` is used in class `Container` to implement a simple semaphore. The primitives that set and release the lock are the two public methods `testAndSet` and `unlock`. If the value of `locked` is `true`, `testAndSet` returns `false`, else is sets `locked` to `true` and returns `true`. `unlock` always set `locked` to `false`.
The above sequence is slightly modified by the additional lock mechanism, here is the result:

```
3 -
    a. try and lock input shared places container
    b. IF one lock fails:
       THEN unlock the successfully locked places
            append the transition to its queue
            go to asynchronous events (5)
    c. ELSE (if all input places are locked successfully)
    d.      call guard function passing input tuple
    e.      get the enabling tuple

4 -
    a. IF the enabling tuple is not empty (enabled):
    b. THEN extract the tokens belonging to the enabling tuple
    c.      unlock all the locked input places
```

```
d.      call the action function passing the enabling tuple
e. ELSE (not enabled)
f.      unlock all the locked input places
```

Step 3.a correspond to applying method `testAndSet` to all the shared places in the transition preset. On a successful lock, the interpreter evaluates the guard, if the transition is enabled, prior to the transition firing, it pops the tokens of the enabling tuple out of their containers and unlock the input tuple. If the lock operation fails, due to the fact that one place is already locked by some other active copy, all the other previously locked places are released, then the transition is enqueued, so that it will be evaluated again.

## 6.6 Active copy connections

One active copy has four classes of connections:

- one $O_2$ socket connecting the PEM;
- one SCI protocol socket connecting the SCI;
- a set of $O_2$ socket connection to $O_2$ tools;
- a set of sockets connections for black box tools.

The first two are stable links, they are set up by method `prologue` of the class `ActiveCopy`, and live as long as the active copy is executed by the interpreter. The other ones are created when necessary (i.e., when a black box tool is invoked and when it is necessary to pass $O_2$ objects to user tools). Each one of these temporary connections, serves one single tool invocation or one single service. When the firing of the transition corresponding to the service is completed, the $O_2$ connection is dropped and implicitly closed.

**Connection with PEM** This connection serves both some notifications and other active copy exchange. Notifications regard transition firing or activity invocation event occurred during the enactment process. The mechanism of the activity invocation underlying the SPADE-1 enactment environment is accessible through this connection. In this implementation, only outgoing notifications are used, however the mechanism may be used also to control some services exported by the interpreter. For example the interpreter could provide a graphical local monitoring facility that could be switched on and off by the PEM.

**Connection with the SCI** This is the access port to the user environment. It is based on the SCI protocol. Service requests and notifications are multiplexed on this connection.

## 6.7 Additional data structures

`execute` method computes one single step of enactment. In order to save the "context" of each step, the `execute` code relies on some additional data structures. Two kinds of information need to be persistent during the whole execution (logical view):

- some transition queues, recording the transitions that are likely to be enabled. Used in step 2.
- black firing instances, each one recording the data associated with one firing of a black transition. An asynchronous event, communicating the end of an external action, matches one of these record and uses its information to complete the firing.

### 6.7.1 Priority

Before we explain the behavior of the transition queues, it is necessary to define in a precise way another property of transitions.

An early approach to the problem of satisfying time constraints in a process model introduced priorities among the features of SLANG. The original idea was to label some transitions of an activity *As Soon As Possible* (ASAP). The meaning of such distinction is that *ASAP* transitions are to be considered *urgent* actions, thus the interpreter should fire ASAP transition first. Or in other words:
*non-ASAP transition can be fired only after all the ASAP transitions have been evaluated and are not enabled.*

This primitive partitioning schema has been extended to an ordinal scale of priorities. The priority value for a transition is an integer, the semantics is:
*a transition labeled with priority $p$ can be fired if all the transitions with priority $q$ such that $q > p$ have been evaluated and are not enabled.*

In the original definition of SLANG, priorities are never mentioned. They definitely do not add computational power to the ER nets formalism, and more, they are far from being a suitable solution to real time process modeling. However, they may be useful in order to tune the behavior of the interpreter, some full SLANG functional properties of a net, costing a complex expansion, can be easily achieved by adjusting priority levels.

### 6.7.2 Transition queues and evaluation policy

The transition queues are the image of the partitioning of the transition set, defined by priority values. Transition queues are encapsulated in an object of the class `TransQueue`. Each priority level defines a queue. The queues are updated in such a way that, at a given time, they contain only the transitions that are likely to be enabled.

Two methods access the queues for writing: `append(t)` appends the transition `t` to the queue corresponding to its priority, while `insert(t)` puts `t` at the beginning of the corresponding queue. Method `popFirst` returns the first transition, extracted from the queue of the highest not empty priority level.

When the execution begins, the `TransQueue` object is filled in with all the transitions of the activity. This is done in step 1 of the logical view (page 70). Then, during usual execution steps, the evaluation policy is very simple: in step 2, the interpreter takes the first transition of the queues calling method `popFirst`.

The evaluation policy actually depends on the way the interpreter updates transition queues. When a transition is evaluated, it is extracted from its queue, if the evaluation fails (i.e., the transition is not enabled) the transition is simply dropped. On a successful evaluation, the transition is appended to its corresponding queue (this is because there might have been more than one enabling tuple, the transition is potentially enabled by other tuples), and then fired. Transition queues are updated (step 6) whenever one of the following events takes place:

- the preset locking fails (thus the transition can not be evaluated). In this case the transition is *appended* so that the evaluation is delayed. Anyway, the transition must be evaluated before it is extracted from the queue.
- one transition is evaluated and is not enabled. The transition is simply extracted from the queue.
- one transition is evaluated and is fired. The transition is *appended* because the firing takes one tuple, other enabling tuples, not computed during the first evaluation may be present. The transitions, whose preset has been touched by the firing, are *inserted.*
- one token is inserted into a user place. All the transition having the user place in their preset, are *inserted.*
- a black transition completes its firing. The transitions, whose preset has been touched by the firing, are *inserted.*

Summing up, these are the updates to the `TransQueues` structures for each execution step:

- extract evaluated transition;
- append fired transition (if any);
- insert transitions affected by the firing;
- insert transitions affected by user places events;
- insert transitions affected by black transition events;
- append transitions with shared places in preset.

If a transition is already present in the queue, an `insert` moves the transition to the front of the queue, while an `append` moves the transition to the end of the queue.

### 6.7.3 Black transition firing instances

Black transitions firings are not atomic events (see section 6.10.2), their execution may be active over several execution steps. Moreover, one black transition may fire many times so that different external actions originated by the same transition can be active at the same time, and the order in which they terminate is not predictable. Thus, the interpreter must store the informations bound to each black transition firing. These are:

- the transition;
- the enabling tuple;
- some other information concerning the external action and the external result.

The first two items are common to every firing instance class, they do not need any comment. The third one depends on the semantics of the black transition firing. As explained further in section 6.10.2, a black transition has three different semantics: black box, SCI service request and SLANG interpreter. Each one is represented by a different object, the class hierarchy is shown in figure 6.4.

The `BlkBBO2` and `BlkSCIO2` subclasses correspond to external action that require $O_2$ data connections.

The interpreter has three lists containing `BlkBB`, `BlkSCI` and `BlkSI` objects, named `BBHeap`, `SCIHeap` and `SIHeap` respectively. New objects are created and filled in by method `fire` (section 6.10), then they are inserted in the corresponding list. The interpreter consults the black instances lists when asynchronous events are examined by method `readAsynchEvent` (see sections 6.11).

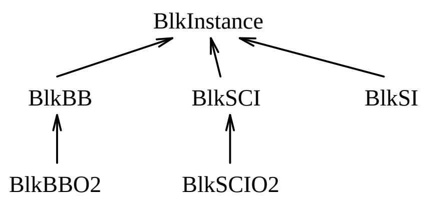


Figure 6.4: Class hierarchy for black transition firing instances.

## 6.8 Guard and action functions

The interpreter basically evaluates guards and executes actions. As already pointed out, the following constraints apply to guards evaluation and action execution:

- *SLANG semantics*: guards and actions are pieces of SLANG code, thus their implementation strongly influences the semantics of the language.
- *Dynamicity*: guards and actions are process-dependent. They may use all the methods and definitions of the user abstract data types. There can be no limitations for user guards and actions other than the SLANG syntax and semantics (e.g., limited fan-in and fan-out for transitions).
- *Independency of the interpreter code*: the interpreter uses informations taken from the activity definition and from the abstract data type schema. Such data are parameters for the interpreter, i.e. they can not be anyhow "hard wired" in the interpreter code.
- *Performances*: a fast action execution and an efficient guard evaluation should be achieved. Note that while these actions play a small part in the whole SPADE-1 environment architecture, they are executed very often. Hence, their code is critical with respect to computational overhead problems.

**More about dynamicity: evolution** The dynamicity requirement, as defined above, is only a part of the problem. The other part concerns the evolution of the process. In SPADE-1, process evolution is achieved by means of language reflectivity.

SPADE supports two main classes of change. In SLANG, an activity definition is a token. Hence it can be modified by the process. When that activity definition (the token) is used in the activity invocation (i.e. when an active copy is created), the SLANG interpreter must bind the latest definition. This discussion is exactly the same for type definitions.

The other class of change concerns active copies. It is also possible to modify active copies by suspending them and editing them in the form of a token. When the execution is resumed, the interpreter uses the new version of the active copy.

In order to have a truly dynamic environment, the SLANG interpreter should really act as an *interpreter*. Thus it should perform time consuming interpretation and dynamic type checking. Just to give an example, it should look up the type definition each time it creates a new object or each time it executes a method.

**SPADE-1 solution** There is clearly a trade off between dynamicity of the model and fast execution. The SPADE-1 solution is a compromise between dynamicity and performance needs. In fact the SLANG interpreter of SPADE-1 can be logically divided in two distinct parts. The *static* code is the part coming with the SPADE-1 schema. This part is completely independent with respect to the process model definition. It implements the evaluation policy, the asynchronous events and the access to common data.

The *dynamic* (or we better say *semi-dynamic*) part concerns the model dependent information. In particular it implements guards evaluation and actions execution. This part consists of some functions attached to transitions. These functions are generated by the interpreter using all the information from types and activity definition. The functions related to an activity are generated and compiled when the activity (the token) is created or modified.

Note that each time a type or an activity definition is modified, other types and activities may require modifications. For example we may modify a class `C` that is used by some activities (e.g., in places type or local variables) and in other types (e.g., in methods code or type definition). When `C` is actually modified, a new version of the class is compiled. As a follow up of the modification of `C`, the activities and the types that *use* `C` must be updated. Again the updated types may be used by other types and activities. Hence, this process must be recursively repeated in order to cover the *transitive closure* of the *uses* relationship. The mechanisms for the management of types and activities are presented in appendix C.

Summing up:

- all the model-dependent information are assigned and available from the activity and types definitions;
- these information seldom change;
- the changing does not affect running active copies;

- interpreter code must not be affected by model-dependent information. Thus the code concerning these sections may be *dynamically* generated and compiled according to the particular type and activity definition;
- the interpreter must provide the modeler with some means to manage types and activity *dynamic* compilation.

#### Dynamic functions

Let us concentrate on the functions implementing guards and actions. Functions are generated when the activity definition is completed. Dynamicity is granted because variable declarations, assignments and return statements of these functions are built according to the activity definition. Thanks to the meta-schema feature provided by $O_2$, new classes and function can be added to the $O_2$ schema at run-time. The new functions are then compiled through a

```
Meta_schema->command()
```

They become part of the schema and are managed just like every user-defined class or type.

The names of the generated and compiled functions are stored in some attributes of the corresponding transition. Since the $O_2$C language doesn't allow pointers to functions, the only way of calling a dynamically determined function is an $O_2$ query, for example the query for a guard evaluation is:

```
o2query(result, (self->guardName+"($1)"), param);
```

where `self` is the transition being evaluated and `param` is a list of containers representing the actual transition preset. This query is part of the interpreter code, so the signature of the functions is standard.

## 6.9 Guards

### 6.9.1 Guards compilation

The guard expression is evaluated by a function called by the interpreter. The function signature for a transition called "Trans" is:

```
function _G_Trans(param:list(Container)):Tuple;
```

Parameter `param` correspond to the input tuple, the result of the function is the enabling tuple. If the transition is not enabled, the function returns `nil`. The outline of the compiled function for a guard is:

1. variable declarations:
   (a) result tuple (returned by the function);
   (b) input places variables and sets;
   (c) auxiliary boolean variables;
   (d) user-defined local variables;
2. assignments of input places sets;
3. `for` cycles;
4. translated guard;
5. return statements;

The evaluation is compiled in a set of nested `for` cycles that correspond to the implicit existential quantifiers (see appendix A.1). User guard is completed with all the necessary declarations, input places are assigned with the appropriate cast to the corresponding container; then quantifiers are expanded in other cycles (see next section) and, eventually, the code for the return statement is added. All these information are retrieved from the preset definition.

### Compilation of guard boolean expression

This section refers to point 4 of previous section. The power of the SLANG language is due to guards expressive semantics. As the $O_2$C language does not compute quantified logical expressions, this ability must be achieved by translating such predicates into some $O_2$C code. The idea is simple: the guard quantifier expressions have the form:
`forall` $varId$ `in` $setVar$ : $expr(varId)$
or
`exists` $varId$ `in` $setVar$ : $expr(varId)$
which, in usual notation, means:
$\forall varId : varId \in setVar \Rightarrow expr(varId)$
and
$\exists varId : varId \in setVar \wedge expr(varId)$
In the first case the value of the expression is initially `true`, then, for every element $varId$

of the set *setVar*, the function computes the rest of the expression, a `false` value is returned as soon as the expression is not satisfied by that value of *varId*. The second expression is just the opposite.

The translation grammar for quantified expression is presented. The BNF syntax for user guards is:

$user\_guards \rightarrow quant\_expr$

$quant\_expr \rightarrow simple\_expr$

$quant\_expr_0 \rightarrow$ `forall` $varId_1$ `in` $varId_2$ `:` $quant\_expr_1$

$quant\_expr_0 \rightarrow$ `exists` $varId_1$ `in` $varId_2$ `:` $quant\_expr_1$

$simple\_expr \rightarrow \ldots$

(other usual rules define the plain boolean expression of the $O_2$C language)

The translation (expansion) schema for the last two rules is:

**forall**: translation($quant\_expr_0$) is

```
_tmp0 = true;
for(varId1 in varId2 )
{
translation(quant_expr1) /* the result of expr1 is assigned to _tmp1 */

 if (_tmp1 == false)
 {
   _tmp0 = false;
   break;
 }
}
```

**exists**: translation($quant\_expr_0$) is

```
_tmp0 = false;
for(varId1 in varId2 )
{
translation(quant_expr1) /* the result of expr1 is assigned to _tmp1 */

 if (_tmp1 == true)
 {
   _tmp0 = true;
   break;
 }
}
```

For a complete example of a guard function expansion see appendix A.2.

## 6.10 Actions

An action body is the SLANG ($O_2$C) code which processes tokens from the enabling tuple and produces new tokens for output places. The term "transition firing" refers to action execution, that corresponds to step 4 in the logical view. As explained in section 6.8, the interpreter has a static part and a dynamic part. The dynamic one is mostly concerned with guards and action execution. The action execution static part is jointly implemented by method `fire` of the class `ActiveCopy` and method `exe` of the classes `WhiteBody` and `BlackBody`. The following paragraph describes the details about the static code. Other paragraphs focus on the dynamic part classifying the transition according to their color.

#### `fire` method

The execution is implemented by the `fire` method on the ActiveCopy object. This method takes the enabling tuple and the transition to be fired. Each transition has an `action` attribute which can be either a `BlackBody` or a `WhiteBody` object. The `fire` method dynamically binds the transition action and calls the `exe` method on it. On `WhiteBody` actions, `exe` calls the fuction corresponding to the whole action body and updates output places with the resulting tokens. On `BlackBody` objects, it calls the function imlementing the *pre-action* and returns the external action command and its parameters.

The result of the `exe` call is analyzed by `fire` method, whose primary functionality is to implement the different semantics of transition firings, in particular four cases apply:

- white transition;
- black box tool;
- service request;
- SLANG interpreter.

### 6.10.1 *White* actions

*White* actions are associated with white transitions, their execution covers the whole action body in one step and immediately flushes the results into output places. Input variables are defined and automatically bound to the enabling tuple. Outpot variables are initialized with new tokens or empty sets, in this way the modeler can directly refer to objects. Local variables declaration is simply added to the function code, so no initial values are provided other then the default ones. The scope of every variables include all the action code. Input and output variables can be used like any other local variable. The `return` statement can not be used. The tokens actually inserted in the output places correspond to the very last assignments to output variables.

**Compilation schema** a white transition action is associated with one $O_2$ function, the signature for a transition called "White" is:

```
function _A_White(param:Tuple):Tuple;
```

This function takes the enabling tuple and computes the output tuple. The outline of the compiled function for an action body is:

1. variable declarations:
   (a) result tuple (returned by the function);
   (b) input and output places variables;
   (c) user-defined local variables;
2. assignments of input places;
3. creation of new tokens or empty sets for output variables;
4. user action code;
5. return statement.

### 6.10.2 *Black* actions

Black transitions actions are made of two distinct bodies called *pre-action* and *post-action*. Each one generates an $O_2$ function. The pre-action is executed by the `fire` method. The post-action is executed by the `completeFiring` method (see section 6.11).

**Pre-action** The pre-action code does all the computations that are necessary in order to determine the external action and its parameters. Output tokens are not declared in pre-action bodies. Local variables are declared as usual. The signature of a pre-action function for a transition called "Black" is:

```
function _Pre_Black(param: Tuple): tuple(extA: string,
                                         parL: list(Object));
```

This function takes the enabling tuple and returns the values of `extAction` and `parList`. The outline of the compiled function for a pre-action body is:

1. variable declarations:
    (a) result tuple (returned by the function);
    (b) input place variables;
    (c) user-defined local variables;
    (d) `o2 string extAction`;
    (e) `o2 list(Object) parList`;
2. assignments of input places;
3. creation of new tokens or empty sets for output variables;
4. user action code;
5. return statement;

**Post-action** The post-action code retrieves the result of the external action. The result is an object. User may declare the result type, in this case, the proper variable is declared and assigned. The result is given to the user by means of the `extResult` variable. The input tuple is visible and corresponds to the tuple that enabled the firing. Local variables are declared as usual, but their value is initialized. thus the assignments of the pre-action get lost. Output tokens are declared and initialized. The signature of a post-action function for a transition called "Black" is:

```
_Post_Black(param: Tuple,
            _extR: Object): Tuple;
```

This function takes the enabling tuple and returns the values of `extAction` and `parList`. The outline of the compiled function for a pre-action body is:

1. variable declarations:
    (a) result tuple (returned by the function);
    (b) input place variables;
    (c) user-defined local variables;
    (d) `extResult`
2. assignments of input places;
3. creation of new tokens or empty sets for output variables;

4. assignment and cast for `extResult`;
5. user action code;
6. return statement;

**`extAction` and `parList`** These two variable define the external action and the optional $O_2$ parameters for that operation. The value of `extAction` in particular, determines the semantics of the black transition firing and, as a consequence, the behavior of the interpreter. The meaning and the use of `extAction` is widely explained in chapter 3. Now we focus on the reaction of the interpreter given each class of command. Three classes of actions are recognized by the `fire` method:

**SLANGInterpreter** : the value of `extAction` is `"SLANGInterpreter"`, `parList` contains the active copy that must be executed. It corresponds to the invocation of another SLANG Interpreter, this is presented to the kernel SLANG user like any other black box $O_2$ tool. Its actual implementation is the focus of the whole enactment environment. These are the action performed by the interpreter:

- a unique identifier is generated.
- a `BlkSI` object is created and filled in with the unique identifier, with the enabling tuple and with the transition.
- the `BlkSI` object is inserted in the temporary list for SLANG interpreter instances (`SIHeap`).
- an $O_2$ message is posted to the PEM connection with the unique identifier and the active copy.

**ServiceRequest** : the value of `extAction` is a string beginning with `"ServiceRequest"`, `parList` may contain $O_2$ objects. It is the invocation of a service. From the SLANG interpreter point of view it does not metter whether the service is provided by the SCI or by an integrated tool. The interpreter then checks the presence of the `"-O2"` option at the end of the `extAction` string. If the option is present:

- a unique identifyer is created;
- the $O_2$ connection is set up;
- the objects of the `parList` are sent over the $O_2$ connection;
- a `blkSCIO2` object is created and filled in with the proper data concerning the identifier and the $O_2$ connection as well as the transition and the enabling tuple;

- the `"ServiceRequest"` prefix is stripped off, and the number of the $O_2$ connection is appended to the message string.
- the message is posted to the SCI.

If the request does not involve $O_2$ data, i.e. the `"-O2"` is not present, a `blkSCI` object is created and filled in as usual, while `parList` value is ignored. In any case the black transition instance tag is appended to the `SCIHeap` list.

**BlackBox** : if the value of `extAction` does not match the two conditions above, it is interpreted as a UNIX black box command. Again the interpreter distinguishes $O_2$ black box tools thanks to the `"-O2"` option terminating `extAction` string. In case, the $O_2$ connection is created, the objects given in `parList` are sent over and a `BlkBBO2` instance is recorded. The mechanism used to treat black box action is slightly different from other message-based control given by the stable connections with the SCI or with the PEM. The black box action is actually invoked by a special function called `StartUnixTool`. The function forks a process connected by a pipe to the interpreter, the interpreter records in the `BlkBB` or `BlkBBO2` instance, the file descriptor of the pipe. This third party process executes the action through the `system` system call, which suspends the process. After the termination of the external action and consequently of the `system` function, the third party process sends out a control message on the pipe and then terminates.

Summing up, these are the commands executed when there is no $O_2$ link:

- a `BlkBB` object is created;
- the tuple and the transition are attached;
- the `StartUnixTool` function is called with `extAction`;
- the integer returned by the function (the file descriptor) is recorded in the `BlkBB` objects.

Some other steps are added when the $O_2$ connection is needed.

## 6.11 Asynchronous events

From the model point of view, the interaction between user environment and process model is achieved through two primitive concepts:

- user places, and
- black transitions.

Both user places and black transition are managed through a set of control messages. Sometimes the control messages (either "user place" or "black transition") are multiplexed on one channel. In other circumstances, they are associated with one private channel. The objective of this section is mapping of these primitives on the connections between an ActiveCopy object and the "outside" world. In particular here we deal with the incoming messages and the related action performed by the interpreter.

Referring to the `readAsynchEvent` method, the interpreter selects in sequence its input channels:

**SCI** : examined by `readSCIEvent` method. The SCI connection can deliver user notifications or black transition termination corresponding to service requests.

**PEM** : the method `readO2Event` reads the Process Engine Manager connection. The PEM connection returns executed active copies corresponding to SLANG interpreter black transition terminations.

**UNIX pipes** : the terminations of black box tools are notified by these connections. Note that while PEM and SCI messages are multiplexed on a single connection and are tagged with a unique identifyer, UNIX pipes are one-to-one with black box tools firing instances, thus only one ending message can arrive from one pipe. The interpreter scans the `SIHeap` list that contains `BlkBB` objects, each one containing the pipe file descriptor. When one message is received, the corresponding black transition is terminated and the pipe is closed.

For each kind of connection and for each kind of messages we describe the reaction of the interpreter.

### 6.11.1 User place messages

User place messages are very simple: one message corresponds to one token. The interpreter always creates one object of the class `Message`, then, it parses the incoming string and writes the message fields in the newly created token. The message specifies the name of the user place in which the token must be inserted. If the place exists in that activity definition, then the token is inserted in the corresponding container.

### 6.11.2 Black transition messages

Black transition messages (remember we are talking about incoming messages) have to match the black instance that generated the external command. On a SCI or a PEM message, the interpreter searches `SCIHeap` or `PEMHeap` lists for the instance matching the identifier contained in the message. Black box instances do not need any further search, because the necessary information are bound to the channel identifier. Once the right instance has been picked up from the temporary lists, the transition firing is completed by the method `completeFiring`.

One more problem concerns the optional values returned by the firing through the `extResult` variable. Its value, that is an object, must be passed to the `completeFiring` method. In order to get this object, the SLANG interpreter always calls method `result` on the black instance objects. Then, thanks to the dynamic binding, the proper method is executed according to the actual type of that instance. $O_2$ instances, black box or service requests, return the object extracted by the $O_2$ connection. Other service requests return an object of the class `Message` filled in with the information given by the message.

# Chapter 7

# SPADE Communication Interface

The SPADE Communication Interface is responsible for the communication between the User Interaction Environment and the Process Enactment Environment. If the process model being executed in the PEE could not interact with the human environment, it would be quite useless. On the other side, if the UIE could not receive control information, it would be without guidance.

The SCI is a Unix process. Communication between the SCI and entities belonging to the User Interaction Environment and to the Process Enactment Environment is based on the message passing paradigm and follows the SCI communication protocol. The SCI and the communicating entities are organized as a client/server architecture.

SCI clients can be divided into:

- *PEE clients*: they are PEE entities which want to communicate and interact with the "outside world", i.e., the User Interaction Environment. Actually, they are SLANG Interpreters[1].
- *UIE clients*: they are entities in the UIE which provide services and notifications of relevant events to the PEE. They are represented by integrated service-based tools as well as *bridges*, i.e., interfaces to integrated service-based tool sets.

PEE clients request services to UIE clients or to the SCI itself. UIE clients and the SCI reply to service requests. Moreover, UIE clients notify the PEE of relevant events. The SCI tasks are the dispatching of service requests and replies between SCI clients, and the multicasting of notifications, coming from UIE clients, to PEE clients which have advertised their interest in receiving them.

[1]At this level of description, the term "SLANG Interpreter" refers to the `SI` object together with its `ActiveCopy` object.

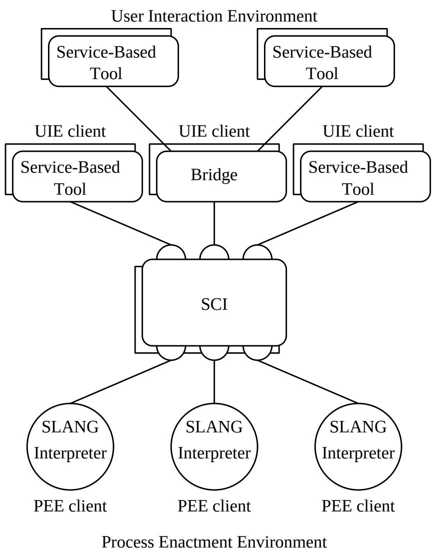


Figure 7.1: High-level view of SCI architecture.

Section 7.1 describes the SCI communication protocol. Sections 7.2 and 7.3 define the semantics of the exchanged message. Section 7.4 outlines the inner structure of the SCI.

## 7.1 SCI communication protocol

The SCI communication protocol defines rules to exchange messages through connections between the SCI and its clients. The SCI communication protocol is based on the SPADE communication protocol and provides abstraction and information hiding. The protocol consists in:

1. *Communication initialization*: the SCI creates the communication access port, using the `openSCIPort` primitive.

2. *Connection setup*: SCI clients connect to the SCI using the `connectToSCI` primitive, passing, as input parameters, the SCI host name, their own client type (i.e., "PEE" or "UIE"), and a proposed identifier (which is non-zero if they have been invoked by the SCI, see section 7.2.2). If the connection setup is successful, they receive, as output parameters, their final identifier and the connection descriptor.

   The SCI checks-in clients connection requests using the `checkInSCIClient` primitive. The primitive retrieves the client kind, the proposed client identifier, and the client connection descriptor. If the SCI interprets the connection request as illegal, it rejects the connection using the `checkOutSCIClient` primitive, passing, as a parameter, the connection descriptor. Otherwise, the SCI accepts the connection request using the `acceptSCIClient` primitive, passing, as a parameter, the final client identifier.

3. *Message exchange*: the SCI and its clients exchange messages using the `readSCIMsg` and `writeSCIMsg` primitives. Messages are strings with the following format:

   <*modifier*> <*reference*> <*address*> <*name*> <*data*>

   A description of each field follows:

   - *modifier*: a string, which contains information to interpret the rest of the message. It may be null, if the message meaning is uniquely identified otherwise.
   - *reference number*: an integer, representing the message identifier.
   - *address*: a dot-separated list of integers, which may represents either the address of the message recipient or the address of the message sender.
   - *name*: a string, representing the type of message.
   - *data*: a string, containing the message data.

   The SCI and its clients own a unique identifier, represented by an integer number. The SCI identifier is zero. SCI clients identifiers are assigned by the SCI either during connection setup or implicit invocation. UIE clients may be interfaces to sets of tools, called *bridges*. Thus, addresses are structured in a hierarchical way, using an IP style, to allow SLANG Interpreters to easily and flexibly specify the recipients of their service requests. A composite address is a list of dot-separated identifiers. The leftmost identifier is the bridge one, while the rest of the address uniquely identifies entities managed by the bridge (e.g., address "11.3" specifies the service-based tool whose identifier is 3, managed by the bridge identified by 11). The SCI is concerned only with the first field of composite addresses. Interpretation of complete addresses is left to bridges.

4. *Disconnection*: SCI clients request to disconnect from the SCI using the `disconnect-FromSCI` primitive, passing, as input parameter, the SCI connection descriptor. The SCI receives the disconnection request message and checks-out the client using the `checkOutSCIClient` primitive.

5. *Communication shutdown*: the SCI closes the communication access port, using the `closeSCIPort` primitive.

## 7.2 Service requests

PEE clients can request services to tools in the UIE or to the SCI itself. PEE clients request services sending a service request message to the SCI.

The service request message contains:

- a client-defined message reference number;
- the address of the service provider;
- the service name;
- the service data.

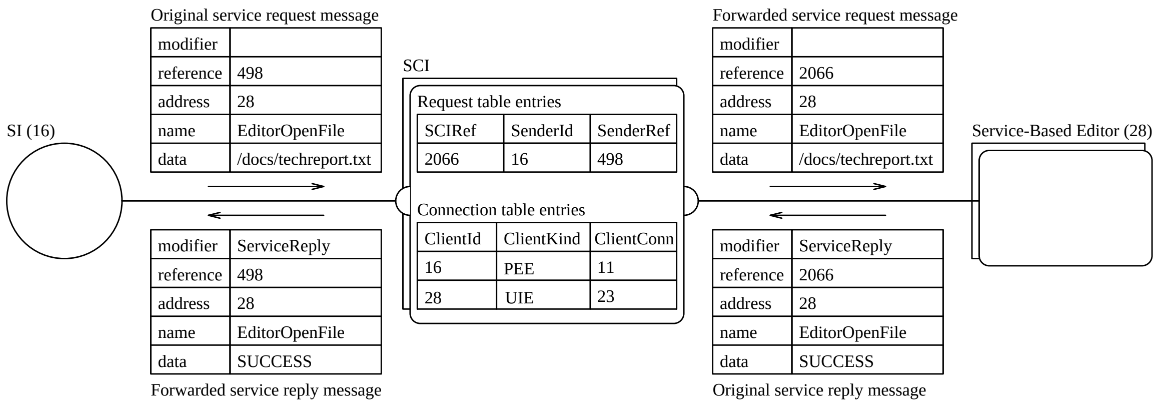
Since a message coming from a PEE client can be only a service request, the message modifier is missing.

### 7.2.1 Tool services

When the SCI receives a service request message from a SLANG Interpreter, it checks the leftmost identifier in the recipient address. If the identifier is zero, the message represents a service request to the SCI itself. Otherwise, the message recipient is a service-based tool in the UIE (i.e., a UIE client or a tool managed by a UIE client). In the latter case, the SCI extracts from the message address the recipient SCI client identifier. Then, using the identifier as a key, the SCI retrieves the corresponding connection from the *connection table*, in which the identifier of each SCI client is stored, together with its kind and connection. Successively, the SCI generates a unique message reference number and stores it in the *request table*, together with the sender message reference number (which is not unique) and the sender identifier. Finally, the SCI forwards the message, with the new reference number, to the recipient UIE client. If the UIE client is a bridge, the message is forwarded to the recipient tool or to another (subordinate) bridge.

The recipient tool processes the message, executes (if possible) the requested service, and produces a service reply message, containing:

- a service reply message modifier (i.e., `ServiceReply`);
- the reference number of the corresponding service request;
- its own address;
- the reply name;
- the reply data.



Original service request message

| | |
|---|---|
| modifier | |
| reference | 498 |
| address | 28 |
| name | EditorOpenFile |
| data | /docs/techreport.txt |

Forwarded service reply message

| | |
|---|---|
| modifier | ServiceReply |
| reference | 498 |
| address | 28 |
| name | EditorOpenFile |
| data | SUCCESS |

Request table entries

| SCIRef | SenderId | SenderRef |
|---|---|---|
| 2066 | 16 | 498 |

Connection table entries

| ClientId | ClientKind | ClientConn |
|---|---|---|
| 16 | PEE | 11 |
| 28 | UIE | 23 |

Forwarded service request message

| | |
|---|---|
| modifier | |
| reference | 2066 |
| address | 28 |
| name | EditorOpenFile |
| data | /docs/techreport.txt |

Original service reply message

| | |
|---|---|
| modifier | ServiceReply |
| reference | 2066 |
| address | 28 |
| name | EditorOpenFile |
| data | SUCCESS |

Figure 7.2: A service request example.

The message follows backward the path of the service request message and reaches the SCI.

Upon receiving a service reply message, the SCI uses the message reference number as a key to retrieve the service requester identifier, as well as the original reference number, from the request table. Then, using the requester identifier as a key, the SCI retrieves the requester connection from the connection table. Finally, it forwards the message to the requester with the original reference number.

For example, let us consider a SLANG Interpreter requesting a service to a service-based editor, as depicted in fig. 7.2. The SLANG Interpreter identifier is 16 and its connection identifier is 11. The Editor identifier is 28 and its connection identifier is 23. The SLANG Interpreter requests the service `EditorOpenFile` on the `/docs/techreport.txt` file to the editor, sending a message to the SCI. The message is marked with its own reference number, i.e., 498. The SCI receives the message and recognizes it as a service request. Then, the SCI retrieves the editor identifier from the message, and uses it as a key to retrieve the editor connection identifier from the connection table. Successively, the SCI generates a unique message reference number, i.e., 2066, and stores it, together with the SLANG Interpreter identifier and the original message reference, in the request table. Finally, the SCI forwards the message to the editor with the new reference number. The editor receives the service request message and successfully opens the file, displaying its contents to the user which owns the editor. As a consequence, the editor produces a reply message containing the string `SUCCESS` as data, with the reference number of the corresponding service request, i.e., 2066. The SCI receives the message and recognizes it as a service reply. Therefore, the SCI retrieves the identifier of the SLANG Interpreter which requested the service and the original message reference from the request table, using the reply reference number as a key. Then, the request table entry is removed.

The SCI retrieves the SLANG Interpreter connection identifier from the connection table, using the SLANG Interpreter identifier as a key. Finally, the message is forwarded to the SLANG Interpreter with the original reference number, i.e., 498.

### 7.2.2 SCI services

PEE clients can request services to the SCI, sending service request messages with an address of zero. The SCI provides two kind of services to PEE clients:

- Configuration.
- Integrated service-based tool invocation.

**Configuration** SLANG Interpreters can ask the SCI to receive notification messages, matching a particular pattern, in a place of the activity they are executing. This is achieved by requesting to the SCI the configuration service. The service request message name is `ConfigSCI` and the message data contains a string in the following format:

<*operator*> <*address pattern*> <*name pattern*> <*place name*>
where:

- *operator* takes the values "`+`" or "`-`", stating that the configuration data must be respectively added or removed from the SCI database.
- *address pattern* may be:
  - a constant address (e.g., `23` or `26.6`), stating that only notification messages coming from a particular tool must be matched;
  - an address terminated by an asterisk (e.g., `19.71.*` or `29*`), stating that all messages, whose address begins with the same pattern, must be matched;
  - a single asterisk, stating that all addresses must be matched.
- *name pattern* is either a constant name (e.g., `EditorFileClosed`), stating that only those particular messages must be matched, or a name terminated by an asterisk (e.g., `Editor*`), stating that all messages whose name begins with the same constant pattern must be matched.
- *place name* is the name of a user place in the activity definition of the active copy being executed by the requester SLANG Interpreter.

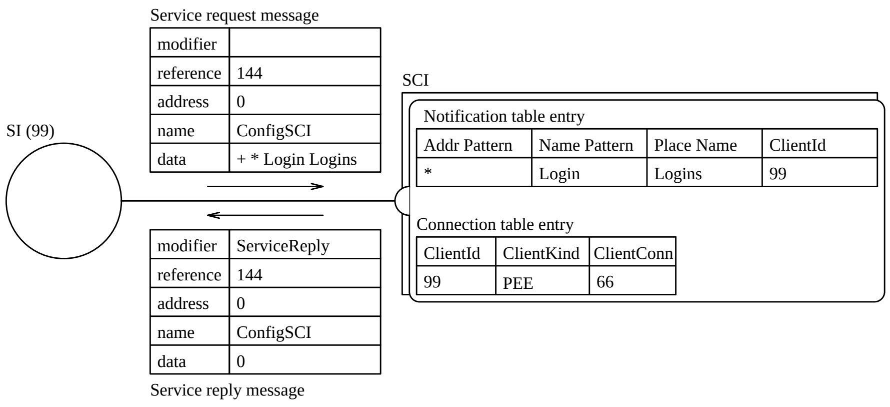


Figure 7.3: A configuration request example.

The SCI, upon receiving a configuration message from a SLANG Interpreter, updates the *notification table*, adding or removing an entry containing the specified address pattern, name pattern, and place name, together with the SLANG Interpreter identifier. Then, the SCI replies with a message containing, as data, the integer `0`, if the operation could be performed, the integer `-1`, otherwise.

For example, suppose that the SLANG Interpreter, whose identifier is 99, wants to receive all the notification messages, whose name is `Login`, in the user place named `Logins`. The messages produced and the notification table entry added are showed in fig 7.3.

**Tool invocation** PEE clients can request to the SCI the invocation of an integrated service-based tool. The service name is `StartTool`. The data part of the service request message contains the host name on which the tool must be executed, followed by the tool command.

When the SCI receives an invocation request message from a SLANG Interpreter, it generates a unique client identifier, and stores it in the request table, together with the SLANG Interpreter identifier and the message reference number. Since client identifiers and message reference numbers are generated using the same counter, consistency is maintained. Then, the SCI forks, on the specified host, the tool, passing, as process environment variables, the tool identifier and the SCI host name. The tool retrieves its own identifier and the SCI host name from the environment and passes them, as parameters, to the primitive `connectToSCI`, when requesting a connection to the SCI. The SCI checks-in the connection request and finds the proposed identifier. Consequently, it checks if there is a corresponding entry in the request table. If it is so, the SCI accepts the tool connection request and it sends a reply message to the SLANG Interpreter containing, as the message data, the identifier of the invoked tool. If the SCI does not find any entries associated to

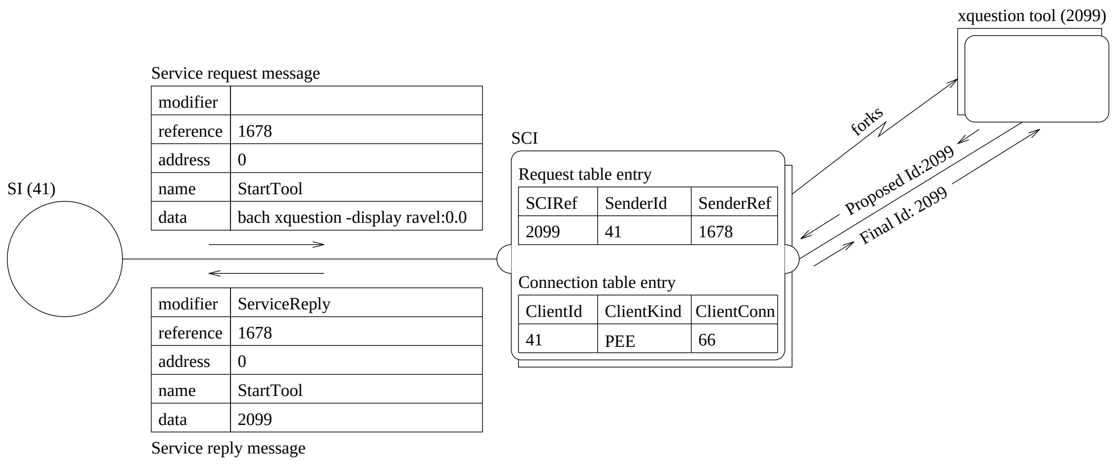


Figure 7.4: An integrated service-based tool invocation request example.

the tool identifier, the connection is refused. Moreover, if the SCI fails forking the tool, it sends a reply message with a negative tool identifier.

For example, consider fig. 7.4, in which the SLANG Interpreter, whose identifier is 41, wants to invoke the integrated service-based tool `xquestion` on host `bach`.

## 7.3 Notifications

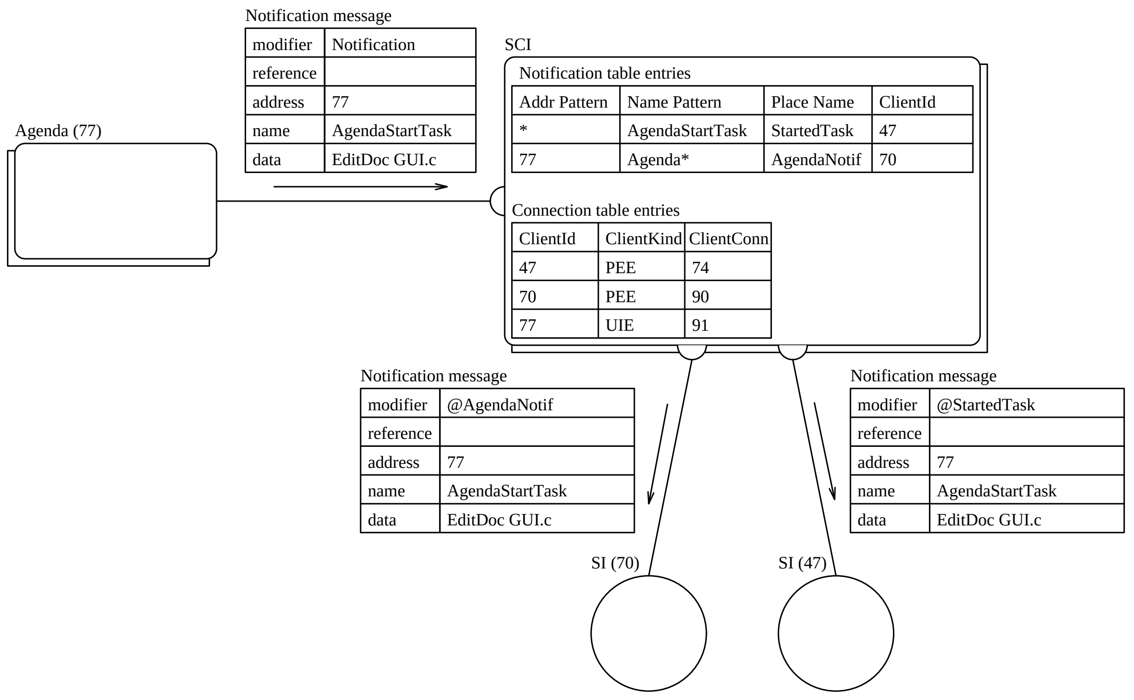
Tools in the UIE can notify the PEE of relevant events, sending a notification message to the SCI. The SCI multicasts notifications to SLANG Interpreters which previously registered themselves for that message.

Notification messages contain:

- the notification message modifier (i.e., `Notification`);
- the address of the notifying tool;
- the notification name;
- the notification data.

Since notification messages are not bound to any request coming from the PEE, the message reference number has no meaning.

Upon receiving a notification message, the SCI matches the message name and address against every entry in the notification table. For each matching entry, the SCI retrieves the SLANG Interpreter identifier and the place name in which the message has to be delivered (as a token). Then, the SCI forwards a message containing:



Notification message

| | |
|---|---|
| modifier | Notification |
| reference | |
| address | 77 |
| name | AgendaStartTask |
| data | EditDoc GUI.c |

Notification table entries

| Addr Pattern | Name Pattern | Place Name | ClientId |
|---|---|---|---|
| * | AgendaStartTask | StartedTask | 47 |
| 77 | Agenda* | AgendaNotif | 70 |

Connection table entries

| ClientId | ClientKind | ClientConn |
|---|---|---|
| 47 | PEE | 74 |
| 70 | PEE | 90 |
| 77 | UIE | 91 |

Notification message

| | |
|---|---|
| modifier | @AgendaNotif |
| reference | |
| address | 77 |
| name | AgendaStartTask |
| data | EditDoc GUI.c |

Notification message

| | |
|---|---|
| modifier | @StartedTask |
| reference | |
| address | 77 |
| name | AgendaStartTask |
| data | EditDoc GUI.c |

Figure 7.5: A notification example.

- a message modifier, consisting of the symbol "`@`", followed by the place name;
- the address of the notifying tool;
- the notification name;
- the notification data.

Upon receiving notification messages, SLANG Interpreters process it, and put the corresponding token in the proper user place of the activity definition corresponding to the active copy they are executing.

As an example, consider the management of a service-based agenda tool. The agenda presents to the user a set of tasks to be performed. The user chooses a task and begins his/her job. Two activities have been defined in the process model. The activity `AgendaManager`, whose task is the management of a particular instance of the agenda tool, and the activity `GlobalReport`, which produces a daily report about all the tasks performed by software developers. Consider now an instance of the agenda tool, whose identifier is 77. The tool is managed by an active copy, instance of the `AgendaManager` activity, executed by a SLANG Interpreter, whose identifier is 70. There is also an active copy, instance of the `GlobalReport` activity, executed by a SLANG Interpreter, whose identifier is 47. The `AgendaManager` active copy has asked the SCI to receive all the notification messages, coming from tool 77, in the place named `AgendaNotif`. The `GlobalReport` activity instance has asked the SCI to receive all `AgendaStartTask` messages, coming from all the agenda tools, in the place named `StartedTask`. Suppose now that the agenda tool presents to the user two possible tasks to perform:

1. write the documentation regarding the software module `GUI.c`;
2. prepare the cost estimation of the project EDAPS.

The user, hating to think about money, chooses the first task. Therefore, the agenda tool notifies the PEE of the user's choice, sending a message to the SCI. The message contains:

- the notification message modifier;
- the agenda address (i.e., `77`);
- the notification name, `AgendaStartTask`;
- the notification data, i.e., `EditDoc GUI.c`.

Upon receiving the message, the SCI recognizes it as a notification. Then, the SCI matches it against the notification table, obtaining two matching entries. The first one corresponds to the SLANG Interpreter 47, and contains the place name `StartedTask`. Thus, the message is forwarded to the SLANG Interpreter with `@StartedTask` as the message modifier. The second matching entry corresponds to SLANG Interpreter 70, and contains the place name `AgendaNotif`. Then, the message is forwarded to the SLANG Interpreter with `@AgendaNotif` as the message modifier.

The SLANG Interpreters, upon receiving the message, produce the corresponding tokens in the named user places, in the activity definition of the active copy they are executing.

## 7.4 SCI modules

The SCI, in order to successfully perform its tasks, maintains an updated database regarding the connected clients and the messages they exchange. Moreover, the SCI manages the SCI protocol when communicating with its clients.

The SCI relies on two modules in doing its job: the Port Manager and the Message Manager (see fig. 7.6).

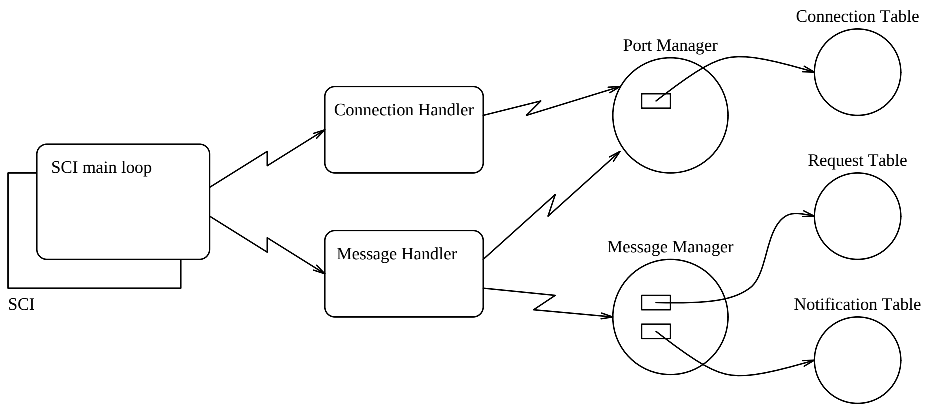


Figure 7.6: SCI internal modules.

The Port Manager provides high-level primitives to manage the SCI communication protocol, and it maintains in the connection table the information about SCI clients identifiers, kinds and connections, allowing simple queries and easy updating. Moreover, the Port Manager handles exceptions like SCI clients hangups and Unix signals.

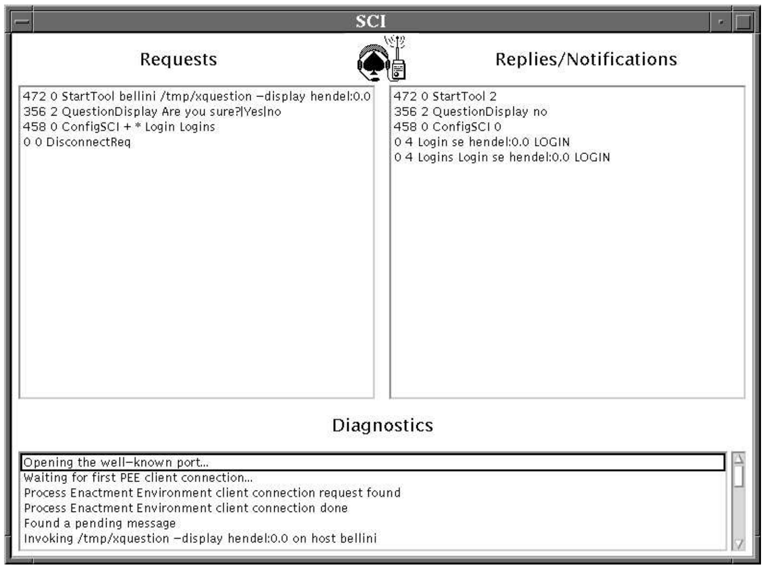
The Message Manager module provides high-level primitives to retrieve and parse incoming messages, to handle errors in message fields values, and to forward request, reply and notification messages to SCI clients. Moreover, the Message Manager maintains the request and notification tables, providing primitives for their updating and querying.

SCI execution is event driven. The SCI main loop function continuously polls the communication access port and the connections with the SCI clients, waiting for an event to occur.

When a connection request from a SCI client appears on the SCI port, the connection handler function is called. The connection handler checks-in the connection request, manage the connection protocol, and updates the connection table using the primitives provided by the Port Manager module.

When a message appears on one of the SCI clients connections, the message handler function is called. The message handler function uses the primitives provided by the Message Manager to retrieve the message and interpret its meaning. Then, the message handler reacts to the message, forwarding request, reply or notification messages, or providing requested services, as needed. The connection handler relies on the Port Manager methods, for the necessary queries on the connection table.

The SCI Motif graphical interface presents, to the SPADE-1 system manager, a graphic display of the exchanged messages (see fig. 7.7). The SCI window displays two message lists: the request service message list and the reply/notification message list. The SPADE-1 system manager can monitor the message traffic and diagnose misbehaviors.



```
472 0 StartTool bellini /tmp/xquestion −display hendel:0.0
356 2 QuestionDisplay Are you sure?|Yes|no
458 0 ConfigSCI + * Login Logins
0 0 DisconnectReq
```

```
472 0 StartTool 2
356 2 QuestionDisplay no
458 0 ConfigSCI 0
0 4 Login se hendel:0.0 LOGIN
0 4 Logins Login se hendel:0.0 LOGIN
```

```
Opening the well−known port...
Waiting for first PEE client connection...
Process Enactment Environment client connection request found
Process Enactment Environment client connection done
Found a pending message
Invoking /tmp/xquestion −display hendel:0.0 on host bellini
```

Figure 7.7: SCI graphical interface.

# Chapter 8

# User Interaction Environment

The User Interaction Environment is the environment in which interaction with process agents takes place. Human actors in the software process cooperate among each other using tools. Editors, compilers, e-mail managers, are examples of widely used tools. Software productivity is improved if such tools are able to interact among each other, in order to provide an integrated environment [SvdB93]. Controlling and coordinating tool interactions requires an approach to tool integration that is both flexible and adaptable to suit different user needs, as well as simple and efficient, in order to meet the requirements of tool developers.

Tools integrated in the SPADE-1 2.0 environment belong to two big groups, according to their integration granularity: *black-box tools* and *service-based tools*. Black-box tools are viewed by the process model as an atomic transformation performed on some input, that may produce some output or have some effects in the User Interaction Environment. No interaction between the process and the black-box tool occurs during tool execution. Moreover, from the process perspective, it is not relevant whether the function is performed automatically by the tool or with user intervention. Service-based tools, instead, provide a programmatic interface that can be used to integrate the functionalities offered by the tool in the process.

This chapter approaches the problem of tool integration in the User Interaction Environment.

## 8.1 Control integration

The *control integration* approach to tool integration is based on viewing a Software Development Environment (SDE) as a collection of services provided by different tools. Actions carried out by a tool are announced to other tools via control signals, represented, for example, by messages. The tools receiving such signals can decide if the other tool actions require that they take any actions themselves.

Following the control integration approach, the SPADE-1 User Interaction Environment includes a collection of service-based tools, each one exporting some services [gro93]. In the SPADE-1 environment, service-based tools do not directly exchange messages with each other. They are completely submitted to the process model commands. In this perspective, the process model is a privileged service client and the only tool controller. Having the ability of controlling one single tool, as well as the interaction between two or more tools, it is the process model that regulates the actions and the cooperative work of the user environment agents.

The effective control over SPADE-1 tools is accomplished through two simple mechanisms, based on message passing: *service request* and *event notification*. These two mechanisms are mapped on the SLANG constructs of black transitions and user places.

A service request is the command, coming from the Process Enactment Environment, that invokes a service. The service may accept some parameters and return some structured or atomic results. Tools in the user environment must always answer to a service request with a reply message which states that the service is terminated. The SPADE Communication Interface delivers the service replies to the SLANG Interpreter that made the request.

Notifications acknowledge the process model for relevant events that occurred in the User Interaction Environment. Notifications are one-way messages. They are originated by service-based tools and are delivered to the SPADE Communication Interface. The SCI, in turn, broadcasts notification messages to all the SLANG Interpreters that registered for that kind of message. Then, it is up to the process model to react to the event notification in the proper way.

## 8.2 Data Integration

SPADE-1 process models manipulate complex data, described and stored using the $O_2$ data base. The problem of data integration concerns the possibility for the process model exchange $O_2$ objects with user tools, in order to have them processed, and to collect other $O_2$ objects that are the results of the service performed by the tool. Some tools of the User Interaction Environment may have access to the $O_2$ repository. These tools, either black-box or service-based, may use $O_2$ objects and return them back to the process like other atomic parameters attached to control messages.

SPADE-1 tackles this problem by setting up dedicated *connections* for transmitting $O_2$ parameters and the $O_2$ result, using the $O_2$ sockets primitives. These connections are established when the service (or the tool) is invoked and are implicitly closed when the service terminates.

Service-based tools retrieve their connection using an identifier attached to the service request message. Black-box tools retrieve the connection identifier from the process arguments. Tools are then responsible for matching the $O_2$ connection, for retrieving parameters and for posting the result before the service (or tool) termination.

This solution is orthogonal with respect to the service granularity and to the control mechanism. It can be used with any combination of parameters and results, i.e., one service may require $O_2$ parameters and return a atomic reply, another may return an $O_2$ object needless for $O_2$ parameters, and, of course, one can take $O_2$ parameters and have an $O_2$ result.

## 8.3 Black-box tools

Black-box tools are viewed by the process model as functions performed on some input and producing some output. A black-box tool is invoked by the SLANG interpreter as a follow-up of the firing of a black transition. The actual invocation and the termination of the tool are managed by a program called `StartUnixTool`. Its only function is to fork the tool and then notify the interpreter for the tool termination.

Black-box usually are Unix programs, like `make`, `cc`, or `emacs`. They take atomic data as input and produce some output.

Black-box tools can also manipulate $O_2$ objects. These tools must be $O_2$ clients and must be structured as follows:

1. Connection to the $O_2$ data base.
2. Connection to the SPADE-1 environment.
3. Retrieval of the $O_2$ parameters (if any).
4. Execution of the tool body.
5. Deliver of the $O_2$ result (if any).
6. Disconnection from the SPADE-1 environment.
7. Disconnection from $O_2$ and termination.

## 8.4 Service-Based Tools

Service-based tools provide services and notify relevant events to the Process Enactment Environment. They can be invoked by the SPADE Communication Interface as a follow-up of a request coming from the PEE, as well as by a user who wants to interact with the process enactment.

Service-based tools implementation follows these steps:

1. Connection to the SCI, using the SCI protocol primitives.
2. Main loop:
   (a) service request retrieval;
   (b) service request reply;
   (c) events notification.
3. Disconnection from the SCI.

Upon receiving a service request, they perform, if possible, the requested service and then they send a reply message containing the result or simply an acknowledge of the performed action.

Service-based tools can receive $O_2$ objects as service requests parameters. Management of structured message data is trasparent to the user.

## 8.5 Integration of SDEs based on message passing

It is possible to integrate Software Development Environments (SDE) based on message passing into SPADE-1 2.0.

Possible candidates to integration in the SPADE-1 2.0 environment are:

- Steven Reiss' FIELD environment [Rei90].
- Hewlett-Packard SoftBench [Ger90].
- Sun Microsystems ToolTalk [Sun91].
- DEC FUSE [Dig91b].

The above listed SDEs are based on the *message passing approach*, i.e., tools in the SDE communicate by passing messages informing other tools of their actions, and requesting services from other tools. Each SDE owns a *message server*, which plays the central role of

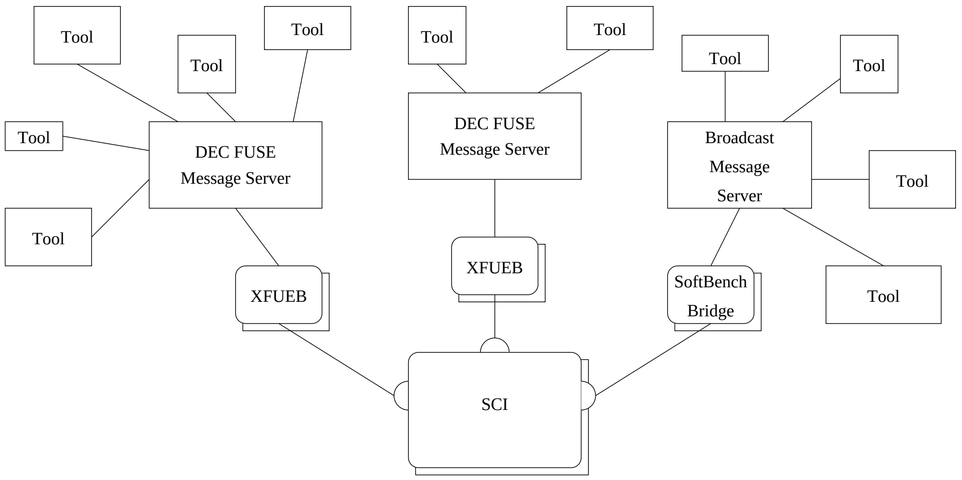


Figure 8.1: Integration of different SDEs in the SPADE-1 environment.

dispatcher of all messages between tools. The presented SDEs are single-user, i.e., message servers cannot be shared between users. There are no or little mechanisms to integrate and coordinate SDE intances owned by different users. Moreover, there are no standard protocols that allow the communication between different kinds of SDEs.

SPADE-1 provides such control and integration in a flexible way. The integration of SDEs in the SPADE-1 environment is achieved building an *ad hoc* filter, called *user environment bridge*, which map the SCI protocol to the target integrated system protocol (see fig. 8.1). Bridges play the double role of SCI clients and SDE tools. Messages coming from the PEE are delivered to bridges, which interpret their meaning, and forward them to tools in the SDE they are connected to. Tool instances in a particular SDE can be selected by service requesters thanks to the hierarchical structure of SCI protocol addresses. Viceversa, messages produced by tools (replies to service requests or notifications) are "captured" by the bridge, sharing the same SDE instance. The bridge translates them in the proper format, and, finally, it forwards them to the SCI.

Currently, a bridge for the DEC FUSE software development environment has been developed. The FUSE bridge, called XFUEB (X FUSE User Environment Bridge), allows the PEE to interact directly with tools integrated in FUSE [Dig91a]. Integration of new tools in FUSE requires little effort and message dispatching is completly trasparent to the tool developer. Integration in the SPADE-1 environment allows interaction between tools belonging to different instances of FUSE.

For example, suppose user Antonio has to supervise an important document before Giovanni reads it. The FUSE service-based editor in the Antonio's FUSE instance, has been opened on the document by a service request coming from the PEE. When Antonio finishes and closes the file, a notification message is generated. The XFUEB catches the message and forwards it to the SCI. The PEE receives the message and, following the execution of the supevisoring activity, a service request message is sent to Giovanni's instance of DEC FUSE. The corresponding XFUEB receives it and translates it into a DEC FUSE message requiring the invocation of an editor on the specified file.

## 8.6 Tool developer interfaces

SPADE-1 2.0 provides to tool developers sets of primitives to easily build tools integrated in the SPADE-1 environment.

### 8.6.1 $O_2$ black-box tools

Black-box tools that do not need to exchange structured data ($O_2$ objects) do not require any particular integration primitive.

SPADE-1 2.0 offers to the tool developer a set of high-level primitives to build $O_2$ black-box tools. These primitives allow the tool to retrieve of $O_2$ parameters and to deliver the $O_2$ result. They are built on top of the C interface to the $O_2$ OODBMS. A description follows.

`connectToSPADEparam` : it connects the tool to the SPADE-1 environment. It takes the process arguments as parameters, namely `argc` and `argv`. It returns a negative integer value if an error occurs.

`readO2parList` : it retrieves the tool $O_2$ parameters. It takes, as argument, the address of a `void` pointer. The pointer represents the handler of an $O_2$ object list. It returns a negative integer value if an error occurs.

`writeO2result` : it writes an $O_2$ object result. It takes, as argument, a `void` pointer, representing the handler of an $O_2$ object. It returns a negative integer value if an error occurs.

`disconnectFromSPADEparam` : it disconnects the tool to the SPADE-1 environment. It returns a negative integer value if an error occurs.

For example, consider the tool `showobj`. Its task is to display the contents of an $O_2$ object. Users can modify its contents using the O2 graphic interface. The outline of the tool implementation follows.

```
#include <o2_c.h>
```

```
#include <stdio.h>
#include <spadeo2tool.h>

int main(argc, argv)
  int argc;
  char *argv[];
{
  static O2_sinit o2sinit; /* O2 init structure */
  int o2LookMode = 0;      /* uses O2 Look */
  void *objList;           /* object list reference */
  void *queryResult;       /* dummy query result */
  void *obj;

  /* initializes the O2_sinit structure */
  o2sinit = ...

  /* connects to O2 */
  o2_link_init(argc, argv, &o2sinit, o2LookMode);
  ...

  /* connects to SPADE in order to receive
     its O2 parameter list */
  connectToSPADEparam(argc, argv);

  /* retrieves the O2 parameters */
  readO2parList(&objList);

  /* extracts the first element of the list */
  o2query(&obj, "$1[0]", o2_handle(objList));

  /* displays the object */
  o2query(&queryResult, "$1->display", o2_handle(obj));

  /* sends back the (modified) object */
  writeO2result(obj);
  /* disconnects from SPADE */
  disconnectFromSPADEparam();

  /* disconnects from O2 */
  o2_link_end();

  /* terminates */
  return;
}
```

### 8.6.2 Service-based tools

Service-based tools that only need to exchange atomic data can be built using high-level primitives written in the C language, and archived in the `spadetool` library. These primitives allow a tool to connect to the SPADE-1 environment (actually to the SCI), to receive service requests messages, to send replies and notification, and to disconnect from the environment. Received and sent messages are of type `Message`, and contain:

**name** : a string, representing the message name.

**par** : a string, representing the message data.

The "tool kit" is composed of the following set of functions:

`int connectToSPADE()` : it links the tool to the SPADE Communication Interface, this way allowing message exchange with the Process Enactment Environment. If the tool is not invoked by the SCI, the environment variable `SCI_HOSTNAME` must be explicitly set (to the name of the SCI host) before calling the `connectToSPADE` method. `connectToSPADE` returns an integer, representing the SCI socket file descriptor on success, or a negative value on failure. Usually, the tool developer has not to bother with this descriptor. Nonetheless, the socket file descriptor may be useful in event driven programming (i.e., it can be used in a call to the `XtAddInput` Xt Toolkit function).

`int recvRequest(Message* msg)` : it fills the `Message` structure argument with an incoming request message. It returns zero on success or a negative value if an error occurs. `recvRequest` fails if there are pending service requests. The tool must reply to a received service request before fetching the next one[1].

[1] It must be noted that the request-reply sequentiality constraint is a limit of the particular interface implementation, not of the service request protocol.

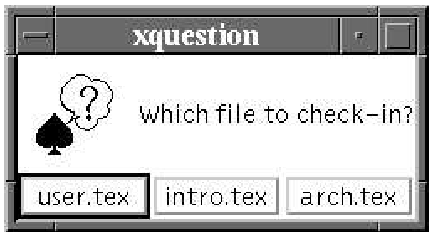


Figure 8.2: The `xquestion` service-based tool.

`int sendReply(Message* msg)` : it sends the reply message passed as the function argument. It returns zero on success, a negative value if an error occurs. The call to `sendReply` fails if there is not a service request to reply to.

`int sendNotification(Message* msg)` : it sends a notification message to the process enactment environment. The notification message is passed as an argument. It returns zero on success, a negative value if an error occurs.

`int disconnectFromSPADE()` : disconnects the tool from the SPADE-1 environment. It returns zero on success, a negative value if an error occurs.

Consider, for example, a graphic integrated service-based tool, called `xquestion`. It presents a question and a set of answers to the user, displaying, on the user screen, a window with a set of buttons. The user can answer clicking on the corresponding button (see fig. 8.2).

The tool offers two services: `QuestionDisplay` which creates the window on the user screen, and `QuestionKill` which requests tool termination. During startup, the tool advertises the PEE of the availability of its services, using the `QuestionAvailable` notification message, passing, as the message parameter, the display it is connected to.

The (incomplete) code of the `xquestion` tool follows.

```
#include <spadetool.h>
...

int main(int argc, char** argv)
{
  int connection;
  Message *startMsg;
  ...

  /* connects to the SPADE environment */
  connection = connectToSPADE();

  /* notifies service availability */
  startMsg = (Message *) malloc(sizeof(Message));
  startMsg->name = "QuestionAvailable";
  startMsg->par = getenv("DISPLAY");
  sendNotification(startMsg);

  ... /* X initialization */

  /* adds the SPADE connection file descriptor to the sources
     of events to be managed, associating the toolMsgHandler
     function to service requests */
  XtAppAddInput(appContext, conn, XtInputReadMask, toolMsgHandler, NULL);

  /* silently enters the X main loop */
  XtAppMainLoop(appContext);
}

void toolMsgHandler()
{
  Message* msg;
  char* answer;
  int result;

  /* initializes the message */
  msg = (Message *) malloc(sizeof(Message));

  /* reads the request */
  result = recvRequest(msg);

  /* if message check-in fails, returns */
  if (result < 0) return;

  /* parses the message */
  if (strcmp(msg->name, "QuestionDisplay")) == 0) {
    /* displays the windows and returns a string
       containing user's choice */
    answer = displayWindow(msg->par);
    msg->par = answer;
    sendReply(msg);
  }

  if (strcmp(msg->name, "QuestionKill") == 0) {
    /* disconnects from SPADE */
    disconnectFromSPADE();
  }

  return;
}
```

### 8.6.3 $O_2$ service-based tools

Tools that need to exchange $O_2$ objects to accomplish their services, can use an high-level interface to the SPADE-1 environment, written in the $O_2$C language. The interface is represented by an object of class `SPADEInterface`. The interface provides methods to connect to the SPADE-1 environment, receive requests, send replies and notifications, and disconnect from the SPADE-1 environment. Received and sent messages are of class `ToolMsg`. Each message contains:

**name** : an $O_2$ string, representing the message name.

**par** : an $O_2$ string, containing atomic data.

**o2par** : a list of $O_2$ objects, containing the structured objects involved in service requests.

The `SPADEInterface` class offers the following methods:

`connectToSPADE` : it links the tool to the SPADE Communication Interface, this way permitting, message exchange with the Process Enactment Environment. If the tool is not invoked by the SCI, the environment variable `SCI_HOSTNAME` must be explicitly set (to the name of the SCI host) before calling the `connectToSPADE` method. `connectToSPADE` returns an integer, representing the SCI socket file descriptor on success, or a negative value on failure. Usually, the tool developer has not to bother with this descriptor, as well as with the internals of the `SPADEInterface` class.

Nonetheless, the socket file descriptor may be useful in event driven programming (i.e., it can be used in a call to the `XtAddInput` Xt Toolkit function).

`recvRequest` : it returns an incoming request message[2], or a null reference if no requests are present. The call to `recvRequest` fails also if there are pending service requests. The tool must reply to a received service request before fetching the next one[3].

`sendReply` : sends a reply message. It takes the reply message (of type `ToolMsg`) as argument. The call to this method fails if there are no service request to reply to.

`sendNotification` : sends a notification message to the Process Enactment Environment. The notification message is passed as an argument and cannot contain an $O_2$ parameter. If present, the `O2par` field is simply ignored.

`disconnectFromSPADE` : disconnects the tool from the SPADE-1 environment.

An example follows:

```
function Tool;
function body Tool
{
  SPADEInterface spade;
  ToolMsg msg;

  o2 myClass myObject;
  o2 string mystring;

  spade = new SPADEInterface;

  /* connects to the SPADE environment. We suppose
     that SCI_HOSTNAME env var has been set to the proper value */
  spade->connectToSPADE();

  while (true) {
    /* waits for a service request */
    do
```

[2]WARNING! Due to an $O_2$ bug the message received is **persistent** (despite it has been removed from any persistency root) and must be modified while in transaction mode.

[3]It must be noted that the request-reply sequentiality constraint is a limit of the particular interface implementation, not of the service request protocol.

```
  msg = spade->recvRequest();
while (msg == nil);

/* parses the message and acts accordingly */
if (msg->name == "Service1") {

  myString = ...;
  myObject = ...;

  /* creates a new message and replies */
  msg = new ToolMsg;
  msg->name = "Service1";
  msg->par = myString;
  msg->O2par = list(myObject);
  spade->sendReply(msg);
}

if (msg->name == "Service2") {
  ...

  if (...) {
    /* sends a notification */
    msg = new ToolMsg;
    msg->name = "Event";
    msg->par = ...;
    spade->sendNotification(msg);
  }

  /* creates a new message and replies */
  msg = new ToolMsg;
  msg->name = "Service2";
  msg->par = ...;
  msg->O2par = list(...);
  spade->sendReply(msg);
}
```

```
    else {
      /* unknown service */
      msg = new ToolMsg;
      msg->name = "Error";
      msg->par = "UnknownService";
      spade->sendReply(msg);
    }
  }
}
```

This example describes the function `Tool`, implementing a service-based tool able to provide two services: `Service1` and `Service2`. The tool connects to the SPADE-1 environment and then waits for a service request. When the tool receives a service request, it matches the message name against the names of the services it is able to provide. If match fails, a reply, containing an error message is sent to the service requester. If the message name matches `Service1`, the tool processes the message and produces a reply. If the name matches `Service2`, the tool processes the message. If some condition holds true, the tool sends a notification message. In every case, a reply message is sent back to the service requester.

### 8.6.4 DEC FUSE service-based tools

Tool developers can develop tools integrated in the DEC FUSE and controlled by the SPADE-1 process model. DEC FUSE tools must adhere to the SPADE-1 “philosophy” and verify these constraints:

- they must not bypass the process;
- they cannot send request messages to the PEE;
- they must register their messages to the SPADE-1 environment.

Message registration is achieved including the `til` file of the tool in the `SPADE.til` file, and the prototypes of the messages in the `SPADE.proto` file. These files will be used in generating FUSE bridge module.

#### DEC FUSE-$O_2$ service-based tools

Tools integrated in the DEC FUSE environment can exchange only atomic data. SPADE-1 provides a set of primitives to overtake this limit, allowing to exchange $O_2$ objects.

The primitives are based on the C interface to the $O_2$ OODBMS. Tools that want to exchange $O_2$ objects have to declare for each service request, an additional parameter named `SPADE` of type `char *`. This system parameter must be passed as an argument to the `getO2par` function, which retrieves the corresponding list of $O_2$ objects from the database. In addition, every reply to a service request must be preceeded by a call to `sendO2par`, passing, as a parameter, the handle of a list of $O_2$ objects.

As an example, consider a service-based object editor integrated in the DEC FUSE environment. One of its exported services is the editing of an $O_2$ object. An outline of function which is called when the `EditObject` message is received, follows.

```
void FUSE_RECV_EditObject(char* command,
                          char* SPADE,
                          int call_id)
{
  void* o2par;
  char* par;

  /* retrieves the O2 object list */
  o2par = getO2par(SPADE);

  ...  /* uses the object list */

  /* sends the object list */
  sendO2par(SPADE, o2par);

  /* replies to the service request */
  FUSE_reply(call_id, par);
}
```

# Chapter 9

# Conclusions

Our work concerned the development of SPADE-1 2.0. In this thesis we presented the basic features and the design and implementation issues of the SPADE-1 2.0 environment. By re-designing the SPADE-1 architecture and by implementing the whole system from scratch, we have been able to provide:

- Improvement and extension of the SLANG modeling language including dynamicity, powerful semantics and straightforward tool integration primitives (process modeler point of view).
- Concurrent interpretation of SLANG activities, based on a distributed configurable system.
- Powerful mechanisms for tool integration (tool developer point of view), control integration at different granularity levels, and data integration.

SPADE-1 2.0 is not meant to be a commercial package, nevertheless much efforts have been made to achieve good performance and easy-to-use powerful features. In this perspective we claim that our work yielded a usable and complete system.

**Future work**

In order to prove the effectiveness of the SPADE principles, SPADE-1 2.0 must be used. Thus, one task for the future is modeling and enacting real software processes. These models should be able to use (and test) all the features of SPADE-1 2.0.

There are other future plans that are related to the validation of SPADE-1 2.0. In particular the system supporting dynamicity must be tested. Accordingly, a model that implements the editing of activities and types (meta-process) should be provided. A totally dynamic mechanism with garbage collection facilities should be studied, considering performance trade-offs. Some facilities for the SLANG process modeler should be implemented, e.g., a SLANG debugger or a syntax-driven SLANG editor. These SLANG programming tools should come with integration facilities in order to be integrated in the meta-process.

Commercial Software Development Environment, like Sun Tooltalk or HP SoftBench, should be integrated in the SPADE-1 2.0 environment, as it is for DEC FUSE.

The architecture of SPADE-1 2.0 could be analyzed in a more formal way. Components could be given precise interfaces and connectors could be defined in terms of formal protocols. This activity could help us in understanding the problems related to architectural description languages, and could highlight inconsistencies or redundancies in our framework.

### SPADE-1 2.0 numbers

Just to give the flavor of the implementation issues of SPADE-1 2.0 we report some numbers.

SPADE-1 2.0 has been developed at CEFRIEL. The complete process took about ten months. We delivered more than 16000 lines of code, 6000 lines of $O_2$C code corresponding to over 100 $O_2$ classes, and more than 10000 lines of C and C++ code. We produced about 300 pages of documentation.

# Bibliography

[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 Computer Society Press.

[AG94a] Robert Allen and David Garlan. Beyond definition/use: Architectural interconnection. In *Proceedings of the Workshop on Interface Definition Languages*, Portland (OR), 1994.

[AG94b] Robert Allen and David Garlan. Formalizing Architectural Connection. In *Proceedings of the 16th International Conference on Software Engineering*, Sorrento (Italy), May 1994.

[BBFL93] S. Bandinelli, L. Baresi, A. Fuggetta, and L. Lavazza. Requirements and Early Experiences in the Implementation of the SPADE Repository. In *8th International Workshop on Software Processes*, Berlin, 1993.

[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.

[BdPS93] P. Battiston, G. Galli de' Paratesi, and M. Signori. L'architettura di SPADE-1, un Ambiente di Supporto al Processo di Sviluppo del Software. Technical report, CEFRIEL, June 1993.

[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, (USA), May 1993. IEEE.

[BFG93a] S. Bandinelli, A. Fuggetta, and S. Grigolli. Process Modeling-in-the-large with SLANG. In *Proceedings of the Second International Conference on the Software Process*, Berlin (Germany), February 1993.

[BFG93b] Sergio Bandinelli, Alfonso Fuggetta, and Carlo Ghezzi. Process Model Evolution in the SPADE Environment. *IEEE Transactions on Software Engineering*, 19(12):1128–1144, December 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.

[Car94] Antonio Carzaniga. The SLANG Interpreter. Technical report, CEFRIEL, 1994.

[CPV94] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing and Implementing Inter-Client Communication in the $O_2$ Object Oriented Database Management System. In *Proceedings of the AICA ISOOMS*, September 1994. (to appear).

[Deu91] O. Deux. The $O_2$ System. *Communications of the ACM*, 34(10), October 1991.

[Dig91a] Digital Equipment Corporation, Maynard, Massachusetts. *DEC FUSE EnCASE Manual*, December 1991. Version 1.1.

[Dig91b] Digital Equipment Corporation, Maynard, Massachusetts. *DEC FUSE Handbook*, December 1991. Version 1.1.

[Fer92] Fabrizio Ferrandina. Uso di Basi di Dati ad Oggetti come Supporto ad un Ambiente per la Modellizzazione dei Processi di Produzione del Software, 1992. Politecnico di Milano.

[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.

[Ger90] C. Gerety. HP SoftBench: a new generation of Software Development Tools. *HP journal*, June 1990.

[GJ82] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts.* John Wiley & Sons, 1982.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, Englewood Cliffs (NJ), 1991.

[GMMP91] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzé. A Unified High-level Petri Net Formalism for Time-critical Systems. *IEEE Transactions on Software Engineering*, February 1991.

[gro93] The CEFRIEL group. Functional Requirements for the SPADE-1 Basic Toolset. Technical Report RT930XX, CEFRIEL, Via Emanueli, 15 - 20126 Milano (Italy), December 1993.

[Gru91] V. Gruhn. *Validation and Verification of Software Process Models.* PhD thesis, University of Dortmund, 1991.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing Company, 1993.

[GZ94] Giorgio Girelli and Edoardo Ziliani. Un insieme di strumenti avanzati integrati in un ambiente centrato sul processo: il caso di SPADE-1. Tesi di laurea, Politecnico di Milano, Dipartimento di Elettronica ed Informazione, July 1994.

[Hel91] Dan Heller. *Motif Programming Manual.* O'Reilly & Associates, Inc, 1991.

[HSO90] D. Heimbigner, S. 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.

[ISO91] ISO 9000. *Quality Management and Quality Assurance standards.* ISO–International Organization for Standardization, first edition, 1991.

[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.

[Lip93] Patrizia Lippi. Definizione del linguaggio SLANG per la specifica dei processi software. Tesi di laurea, Politecnico di Milano, Dipartimento di Elettronica ed Informazione, July 1993.

[Mas93] Beppe De Mastro. A SLANG Model for a Software Product Review Process. Technical report, CEFRIEL, 1993.

[O292] O2. *The O2 User Manual.* O2 Technology, 1992.

[Par94] Alessandro Parimbelli. L'integrazione di strumenti in un ambiente centrato sul processo: l'esperienza di SPADE. Tesi di laurea, Politecnico di Milano, Dipartimento di Elettronica ed Informazione, February 1994.

[Pic93] G. Picco. Modeling a real software process with slang. Internal Report RI93059, CEFRIEL, Via Emanueli, 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.

[PV93] Gianpietro Picco and Giovanni Vigna. The SPADE Way to Inter-Client Communications in $< O_2$. Technical report, CEFRIEL, December 1993.

[Rei90] S. Reiss. Connecting Tools using Message Passing in the FIELD Program Development Environment. *IEEE Software*, pages 57–67, July 1990.

[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.

[Sun91] Sun MicroSystems, Inc. *Solaris Open Windows: The ToolTalk Service*, 1991.

[SvdB93] D. Schefström and G. van den Broek. *TOOL INTEGRATION.* John Wiley & Sons, 1993.

[tea93] GoodStep team. The SLANG 1.1 Process Modeling Language Reference Manual. Project deliverable, GOODSTEP, October 1993.

[VZ93] Giovanni Vigna and Edoardo Ziliani. The SPADE-1 Process Enactment Environment Architecture. Technical report, CEFRIEL, 1993.

# Appendix A

# SLANG language

## A.1 Guards formal semantics

**Notation:** Let $AC$ be active copy whose activity definition contains a place $P$; then:
$C_P^{AC}$ is the set of all the tokens currently in place $P$ of active copy $AC$.
Let $S$ be a set:
$S_1$ is the set $S$ itself
$S_*$ is the power set of $S$
$S_n$ where $n = 2, 3, 4, ...$ is the subset of $S_*$ made of all the elements of $S_*$ having cardinality $n$.

**Example:** place $P$ of active copy $X$ contains tokens $a$, $b$, and $c$:
$C_{P,1}^X = \{a, b, c\}$
$C_{P,2}^X = \{\{a, b\}, \{a, c\}, \{c, b\}\}$
$C_{P,3}^X = \{\{a, b, c\}\}$
$C_{P,*}^X = \{\{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{c, b\}, \{a, b, c\}\}$
$C_{P,4}^X = \emptyset$

**Guard semantics** : each free variable, corresponding to a place in the enabling tuple, is implicitly under the scope of an existential quantifier, thus a transition with two places A and B with user guard:
$p(A, B)$
has the actual guard:
$\exists A : \exists B : A \in C_{A,w_A} \wedge B \in C_{B,w_B} \wedge p(A, B)$

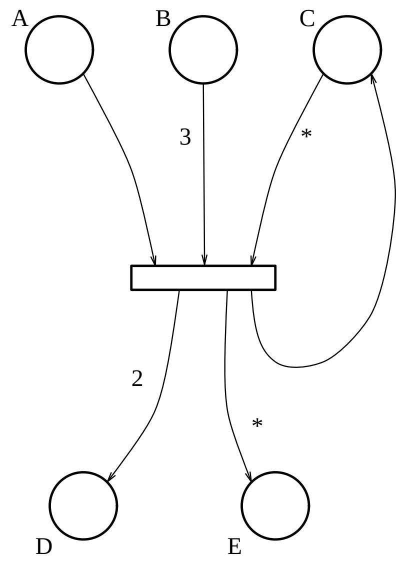


Figure A.1: Example of a Transition.

where $w_A$ and $w_B$ are the weights of the arcs connecting places $A$ and $B$ to T.

## A.2 Example of guard and action expansion

**Example of variables in a transition** We have the transition shown in figure A.1, being `typA`, `typB`, `typC`, `typD` and `typE` the types associated with places `A`, `B`, `C`, `D` and `E`. The following local variable are defined:

```
o2 typB b;
o2 typC c;
o2 integer i;
```

The topological definition together with arc weight defines the following variables visible to the guard:

```
/*                         input variables */
o2 typA A;
o2 unique set(typB) B;
o2 unique set(typC) C;
/*                         local variables */
o2 typB b;
o2 typC c;
```

```
o2 integer i;
```

and these are visible to the action:

```
/*                       input variables */
o2 typA A;
o2 unique set(typB) B;
o2 unique set(typC) in_C;
/*                       output variables */
o2 unique set(typD) D;
o2 unique set(typE) E;
o2 typC out_C;
/*                       user local variables */
o2 typB b;
o2 typC c;
o2 integer i;
```

The following definition are given for guard and action:
Guard:

```
forall c in C: exists b in B: (A->cod == b->xxx) && (b->yyy == c->zzz)
```

Action:

```
 .. .... .... ..
 ... .. . ... .. .
```

Here it is not important the actual action code, it is preferable to insert here a meaningless code in order to focus on the compilation expansion.
Local variables:

```
o2 typB b;
o2 typC c;
```

They follow the standard $O_2C$ syntax.
Places A, B, C, D and E contain tokens of types `typA`, `typB`, `typC`, `typD` and `typE` respectively. It is necessary to have all the types used in these sections as part of the schema; the expansion and compilation methods assume all the necessary classes and types are compiled in the schema.
The function associated with the guard defines the following variables:

```
 A: typA;
 B: unique set(typB);
 C: unique set(typC);
```

corresponding to input places, other temporary variables are declared by the quantifiers expansion:

```
o2 boolean _tmp0;
o2 boolean _tmp1;
o2 boolean _tmp2;
```

The complete code for this function is:

```
/*   GUARD   */
function G_T(param:list(Container)):Tuple;
function body G_T(param:list(Container)):Tuple
{
o2 Tuple result;
o2 unique set(typA) A_set;
o2 typA A;
o2 unique set(unique set(typB)) B_set;
o2 unique set(typB) B;
o2 unique set(unique set(typC)) C_set;
o2 unique set(typC) C;
o2 boolean _tmp0;
o2 boolean _tmp1;
o2 boolean _tmp2;
o2 typB b;
o2 typC c;
A_set = (o2 unique set(typA))(param[0]->contents);
B_set = (o2 unique set(unique set(typB)))(param[1]->powerN(3));
C_set = (o2 unique set(unique set(typC)))(param[2]->power);
for(A in A_set)
 for(B in B_set)
  for(C in C_set)
  { _tmp0=true;
    for( c in C)
    { _tmp1=false;
      for( b in B)
      { _tmp2=(A->cod == b->xxx) && (b->yyy == c->zzz);
        if(_tmp2)
        { _tmp1=true; break; }
      }
      if(!_tmp1)
        { _tmp0=false; break; }
    }
  if (_tmp0)
  {
    result = new Tuple;
    result->elements = list();
    result->elements += list(unique set(A));
    result->elements += list(B);
    result->elements += list(C);
    return result;
  }
 }
return (o2 Tuple)nil;
};
```

Note that the return statements included by the expansion assigns the elements of the resulting tuple according to their type.

Input variable declared and assigned with the appropriate cast like in the guard function. Output variable are declared and initialized:

```
o2 unique set(typD) D;
o2 unique set(typE) E;
o2 typC out_C;
```

The only ambiguous name is `C` so input and output variable are called `in_C` and `out_C` respectively;

The function associated to the action is:

```
/*   ACTION   */
function A_T(param:Tuple):Tuple;
function body A_T(param:Tuple):Tuple
{
o2 Tuple result;
o2 typA A;
o2 unique set(typB) B;
o2 unique set(typC) in_C;
o2 unique set(typD) D;
o2 unique set(typE) E;
o2 typC out_C;
/* user local variables */
o2 typB b;
o2 typC c;
/* parameters substitution and init */
 A= (o2 typA)element(param->elements[0]);
 B = (o2 unique set(typB))param->elements[1];
 in_C = (o2 unique set(typC))param->elements[2];
 D= unique set();
 E= unique set();
 out_C= new typC;
/* user action code */
 .. .... .... ..
 ... .. . ... .. .
/* return values */
result = new Tuple;
result->elements = list();
result->elements += list( D);
result->elements += list( E);
result->elements += list(unique set( out_C));
return result;
};
```

# Appendix B

# Example of tool integration

This example uses the following tools:

```
Black box tools:
     - emacs
     - latex

Service-based tools:
     - SPADELogin
     - Xquestion

The interface of SPADELogin is
     Notifications:
     Services:     -

The interface of Xquestion is:
     Notifications:
     Services:
```

The SLANG ADTs used in the process model corresponding to the example are:

```
class Element inherit ModelType public type
    tuple(name:        string,
          responsible: string,
          fileName:    string,
          compOK:      boolean)
    method public myTypeIs: string
```

```
end;

method body myTypeIs: string in class Element {
  o2 string tmp;
  if ((self->name)[0:2] == "Doc")
        tmp = "Document";
  else
        tmp = "Chapter";

  return tmp;
};

class Document inherit Element public type
    tuple(chapters:    list(string))
end;

class Chapter inherit Element public type
    tuple(ownerDoc:    string)

end;

class User inherit ModelType public type
    tuple(name:        string,
          role:        string,
          usrDisplay:  string,
          xQuestionId: string)
end;

class Answer inherit ModelType public type
    tuple(objName:     string,
          value:       string)
end;

class UnixFile inherit ModelType public type
    tuple(fileName:    string,
          owner:       string)
```

```
end;

class Counter inherit ModelType public type
    tuple(value:    integer)
end;
```

The root activity is shown in Figure B.1 and the corresponding transition guards and actions are the following:

```
ACTIVITY Root

TRANSITION RejectLogin
guard:
Answers1->value == "No" && Answers1->objName == AskingUsers->name

action:

TRANSITION ConfigureSystem
guard:
true

prologue:
extAction = "ServiceRequest 0 ConfigSCI + * Login* LoginMessages";

epilogue:


TRANSITION EvaluateLogin
guard:
true

local:
o2 User usr;

action:
Answers1->objName = LoginMessages->parameter(0);
for (usr in RegistredUsers) {
    if (usr->name == LoginMessages->parameter(0)) {
```

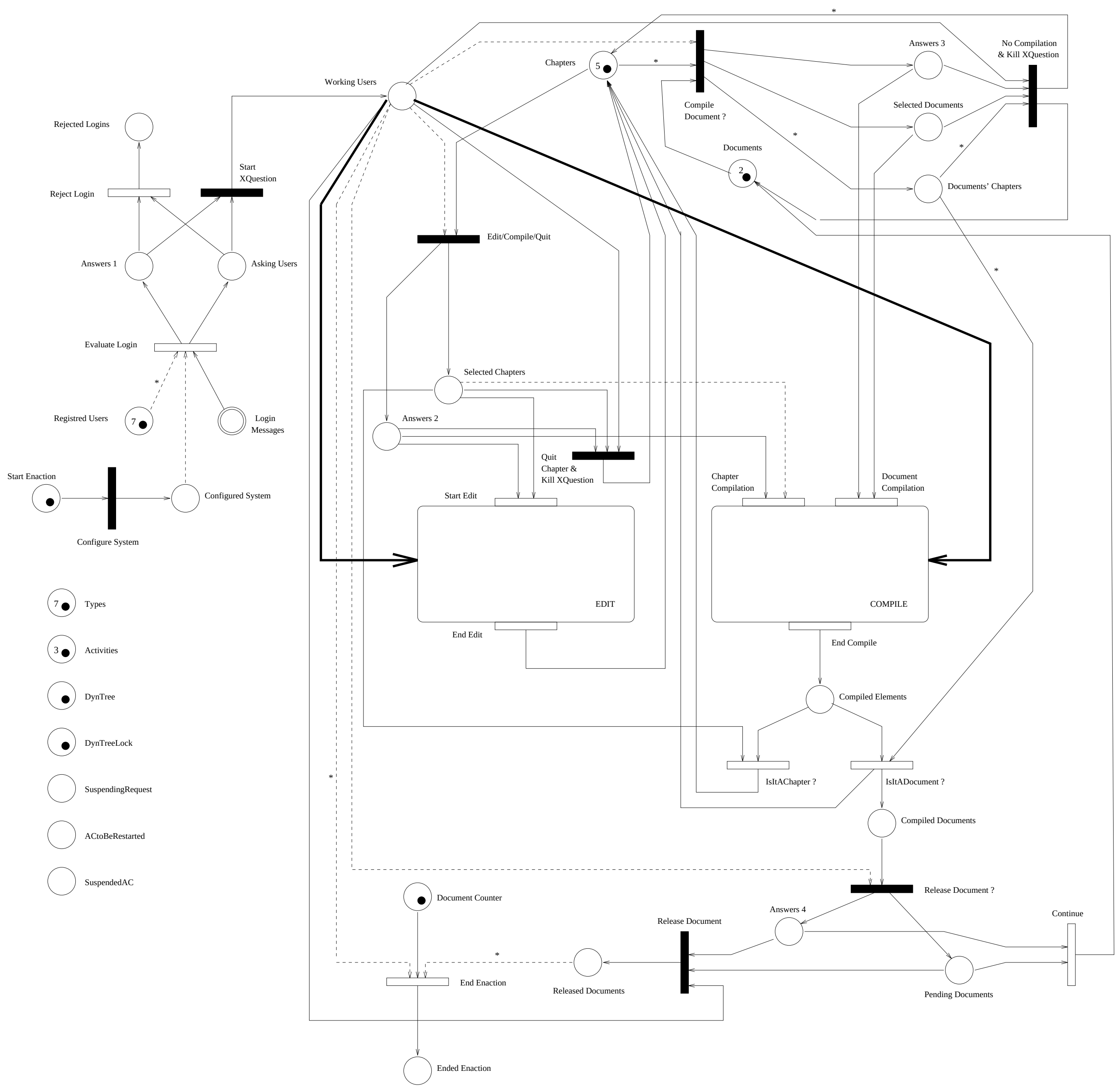


Figure B.1: Activity Root (and initial state for the root active copy).

```
        AskingUsers->name = usr->name;
        AskingUsers->role = usr->role;
        AskingUsers->usrDisplay = LoginMessages->parameter(1);
        Answers1->value = "Yes";
        break;
    }
}
if (Answers1->value != "Yes") {
    Answers1->value = "No";
    AskingUsers->name = LoginMessages->parameter(0);
}


TRANSITION StartXQuestion

guard:
Answers1->value == "Yes" && Answers1->objName == AskingUsers->name

prologue:
extAction = "ServiceRequest 0 StartTool rossini
             /home/bach/se/SPADE/TEST/BIN/SPARC/xquestion -display "
             + AskingUsers->usrDisplay;

epilogue:
WorkingUsers = AskingUsers;
WorkingUsers->xQuestionId = extResult->parameter(0);


TRANSITION DocumentComp
guard:
forall s in Documents->chapters: exists ch in Chapters:
ch->name == s && Documents->responsible == WorkingUsers->name
&& WorkingUsers->role == "DR"

local:
o2 string s;
```

```
o2 Chapter ch;

prologue:
extAction = "ServiceRequest " + WorkingUsers->xQuestionId +
            " QuestionDisplay Do you want to Compile your Document ?\
              |Yes|No|";

epilogue:
Answers3->objName = Documents->name;
Answers3->value = extResult->parameter(0);
SelectedDocuments = Documents;
for (ch in Chapters) {
   DocumentsChapters += unique set(ch);
}


TRANSITION NoCompilation
guard:
forall ch in DocumentsChapters:
ch->ownerDoc == SelectedDocuments->name &&
WorkingUsers->name == SelectedDocuments->responsible &&
WorkingUsers->role == "DR" && Answers3->value == "No"

local:
o2 Chapter ch;

prologue:
extAction = "ServiceRequest " +
  WorkingUsers->xQuestionId + " QuestionKill";

epilogue:
Documents = SelectedDocuments;
    for (ch in DocumentsChapters) {
        Chapters += unique set(ch);
}
```

```
TRANSITION IsItADocument
guard:
forall ch in DocumentsChapters:
ch->ownerDoc == CompiledElements->name &&
CompiledElements->myTypeIs == "Document"

local:
o2 Chapter ch;

action:
CompiledDocuments->name = CompiledElements->name;
CompiledDocuments->responsible = CompiledElements->responsible;
CompiledDocuments->fileName = CompiledElements->fileName;
CompiledDocuments->compOK = CompiledElements->compOK;
for (ch in DocumentsChapters) {
    CompiledDocuments->chapters += list(ch->name);
}


TRANSITION IsItAChapter
guard:
CompiledElements->myTypeIs == "Chapter" &&
CompiledElements->name == SelectedChapters->name

action:
Chapters->name = CompiledElements->name;
Chapters->responsible = CompiledElements->responsible;
Chapters->fileName = CompiledElements->fileName;
Chapters->compOK = CompiledElements->compOK;
Chapters->ownerDoc = SelectedChapters->ownerDoc;


TRANSITION EditCompileQuit
guard:
Chapters->responsible == WorkingUsers->name &&
  WorkingUsers->role == "CR"

prologue:
extAction = "ServiceRequest " + WorkingUsers->xQuestionId +
            " QuestionDisplay Choose an action to perform on
            your Chapter |Edit|Compile|Quit|";

epilogue:
SelectedChapters = Chapters;
Answers2->objName = Chapters->name;
Answers2->value = extResult->parameter(0);


TRANSITION QuitChapter
guard:
Answers2->value == "Quit" &&
Answers2->objName == SelectedChapters->name &&
WorkingUsers->name == SelectedChapters->responsible

prologue:
extAction = "ServiceRequest " +
  WorkingUsers->xQuestionId + " QuestionKill";

epilogue:
Chapters = SelectedChapters;


TRANSITION ReleaseDocument
guard:
CompiledDocuments->responsible == WorkingUsers->name &&
WorkingUsers->role == "DR"

prologue:
extAction = "ServiceRequest " + WorkingUsers->xQuestionId +
            "QuestionDisplay Do you want to release your
            Document ?  |Yes|No|";
```

```
epilogue:
Answers4->objName = CompiledDocuments->name;
Answers4->value = extResult->parameter(0);
PendingDocuments = CompiledDocuments;


TRANSITION Continue
guard:
Answers4->value == "No" &&
Answers4->objName == PendingDocuments->name

action:
Documents = PendingDocuments;


TRANSITION RelDocAndQuit
guard:
Answers4->value == "Yes" &&
Answers4->objName == PendingDocuments->name &&
WorkingUsers->role == "DR"

prologue:
extAction = "ServiceRequest " +
WorkingUsers->xQuestionId + " QuestionKill";

epilogue:
ReleasedDocuments = PendingDocuments;


TRANSITION EndEnaction
guard:
forall d in ReleasedDocuments: exists usr in WorkingUsers:
count(ReleasedDocuments) == DocumentCounter->value &&
d->responsible == usr->name && usr->role == "DR"
```

```
local:
o2 User usr;
o2 Document d;

action:

INVOCATION Compile

INTRANSITION CompileDocument
guard:
Answers3->value == "Yes" &&
Answers3->objName == SelectedDocuments->name

INTRANSITION CompileChapter
guard:
Answers2->value == "Compile" &&
Answers2->objName == SelectedChapters->name


INVOCATION:

ACTIVITY Edit

INTRANSITION StartEdit
guard:
Answers2->value == "Edit" &&
Answers2->objName == SelectedChapters->name
```

Activity Compile is shown in Figure B.2 and the corresponding transition guards and actions are the following:

```
ACTIVITY Compile


TRANSITION EndCompile
guard:
true
```

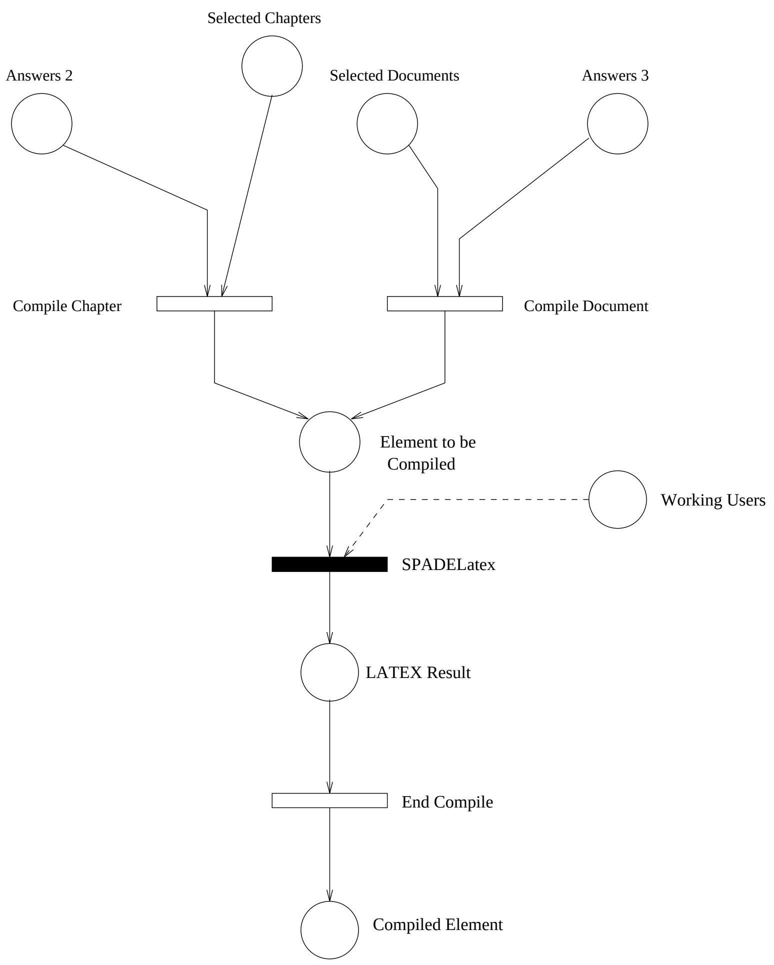


Figure B.2: Activity Compile.

```
action:
CompiledElements = LatexResult;



TRANSITION SPADELatex
guard:
ElementToBeCompiled->responsible == WorkingUsers->name

local:
o2 SpadeText tmpTxt;

prologue:
extAction = "/usr/local/X11R5/bin/xterm -display " +
            WorkingUsers->usrDisplay +
            " -e $SPADE_ROOT/DEMO/LATEXDOC/SPADELatex " +
            ElementToBeCompiled->fileName +
            ".tex; /bin/grep 'No pages' " +
            ElementToBeCompiled->fileName + ".log > " +
            ElementToBeCompiled->fileName + ".grp";

epilogue:
LatexResult = ElementToBeCompiled;
tmpTxt = new SpadeText;
tmpTxt->read_file(ElementToBeCompiled->fileName + ".grp", "w");
tmpTxt = new SpadeText;
if (tmpTxt->txtToStr == "\n")
    LatexResult->compOK = true;
else
    LatexResult->compOK = false;



TRANSITION CompileChapter
guard:
Answers2->value == "Compile" &&
Answers2->objName == SelectedChapters->name
```

```
action:
ElementToBeCompiled = SelectedChapters;



TRANSITION CompileDocument
guard:
Answers3->value == "Yes" &&
Answers3->objName == SelectedDocuments->name

action:
ElementToBeCompiled = SelectedDocuments;
```

Activity Edit is shown in Figure B.3 and the guards and actions of its transitions are:

```
ACTIVITY EDIT

TRANSITION ReEdit
guard:
WorkingChapter->compOK == false

action:
ChapterToBeEdited = WorkingChapter;
LogToBeViewed->fileName = WorkingChapter->fileName;
LogToBeViewed->owner = WorkingChapter->responsible;



TRANSITION Edit
guard:
ChapterToBeEdited->responsible == WorkingUsers->name

prologue:
extAction = "/usr/local/bin/emacs -display " +
  WorkingUsers->usrDisplay + " " +
  ChapterToBeEdited->fileName + ".chp";

epilogue:
ModifiedChapter = ChapterToBeEdited;
```

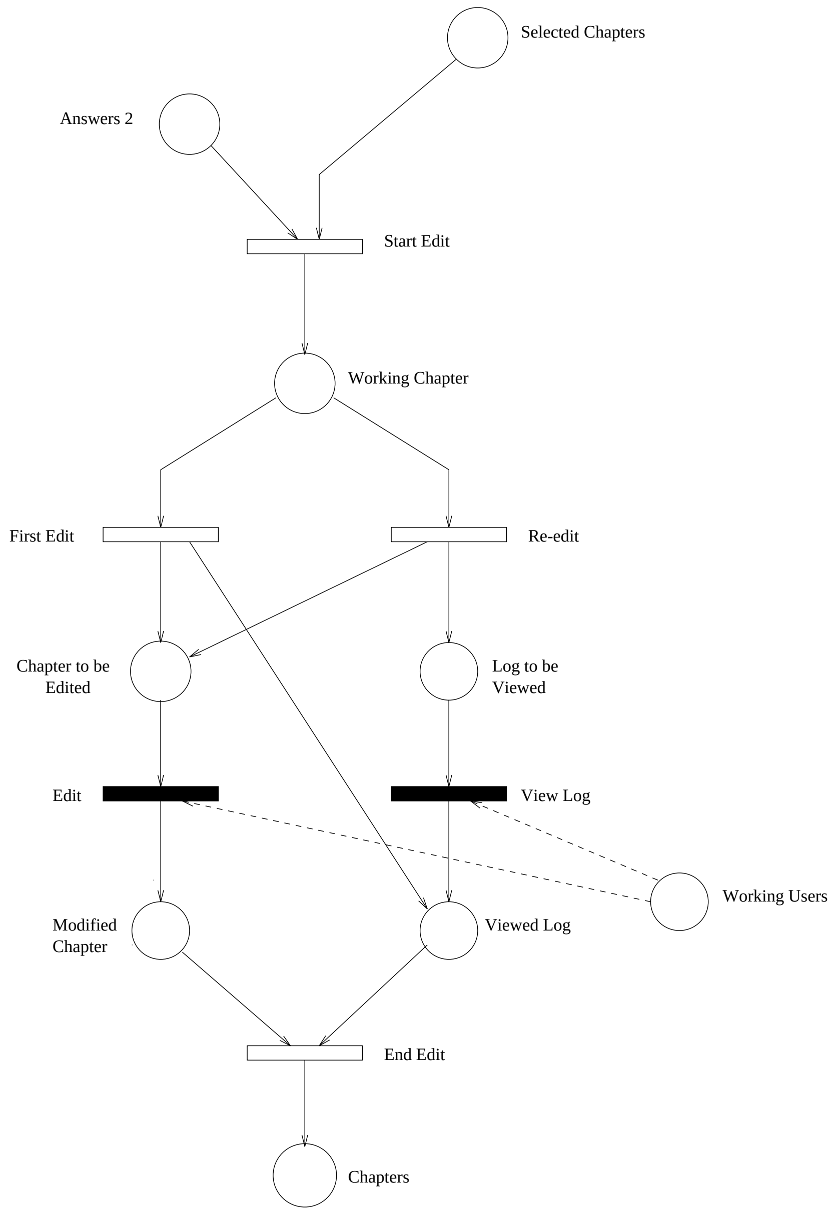


Figure B.3: Activity Edit.

```
TRANSITION StartEdit
guard:
Answers2->value == "Edit" &&
Answers2->objName == SelectedChapters->name

action:
WorkingChapter = SelectedChapters;


TRANSITION FirstEdit
guard:
WorkingChapter->compOK == true

action:
ChapterToBeEdited = WorkingChapter;
ViewedLog->fileName = WorkingChapter->fileName;
ViewedLog->owner = WorkingChapter->responsible;


TRANSITION ViewLog
guard:
LogToBeViewed->owner == WorkingUsers->name

prologue:
extAction = "/usr/local/bin/emacs -display " +
   WorkingUsers->usrDisplay + " " +
   LogToBeViewed->fileName + ".log " +
   " -f toggle-read-only ";

epilogue:
ViewedLog = LogToBeViewed;


TRANSITION EndEdit
guard:
```

```
ModifiedChapter->fileName == ViewedLog->fileName

action:
Chapters = ModifiedChapter;
```

Finally, the Root active copy initial state is given by the following assignments:

```
/* START ENACTION */

tk = new Token;

/* CHAPTERS place */

ch1 = new Chapter;

ch1->name = "TrialCh11";
ch1->responsible = "se6";
ch1->fileName = "$SPADE_ROOT/DEMO/LATEXDOC/chapter11";
ch1->compOK = true;
ch1->ownerDoc = "DocTrial1";

ch2 = new Chapter;

ch2->name = "TrialCh12";
ch2->responsible = "se7";
ch2->fileName = "$SPADE_ROOT/DEMO/LATEXDOC/chapter12";
ch2->compOK = true;
ch2->ownerDoc = "DocTrial1";

ch3 = new Chapter;

ch3->name = "TrialCh13";
ch3->responsible = "se9";
ch3->fileName = "$SPADE_ROOT/DEMO/LATEXDOC/chapter13";
ch3->compOK = true;
ch3->ownerDoc = "DocTrial1";
```

```
ch4 = new Chapter;

ch4->name = "TrialCh21";
ch4->responsible = "se5";
ch4->fileName = "$SPADE_ROOT/DEMO/LATEXDOC/chapter21";
ch4->compOK = true;
ch4->ownerDoc = "DocTrial2";

ch5 = new Chapter;

ch5->name = "TrialCh22";
ch5->responsible = "picco";
ch5->fileName = "$SPADE_ROOT/DEMO/LATEXDOC/chapter22";
ch5->compOK = true;
ch5->ownerDoc = "DocTrial2";

/* USERS place */

usr1 = new User;

usr1->name = "baresi";
usr1->role = "DR";

usr2 = new User;

usr2->name = "bandinel";
usr2->role = "DR";

usr3 = new User;

usr3->name = "picco";
usr3->role = "CR";

usr4 = new User;
```

```
usr4->name = "se5";
usr4->role = "CR";

usr5 = new User;

usr5->name = "se6";
usr5->role = "CR";

usr6 = new User;

usr6->name = "se7";
usr6->role = "CR";

usr7 = new User;

usr7->name = "se9";
usr7->role = "CR";

/* DOCUMENTS place */

dc1 = new Document;

dc1->name = "DocTrial1";
dc1->responsible = "bandinel";
dc1->fileName = "$SPADE_ROOT/DEMO/LATEXDOC/document1";
dc1->compOK = false;
dc1->chapters = list("TrialCh11", "TrialCh12", "TrialCh13");

dc2 = new Document;

dc2->name = "DocTrial2";
dc2->responsible = "baresi";
dc2->fileName = "$SPADE_ROOT/DEMO/LATEXDOC/document2";
dc2->compOK = false;
dc2->chapters = list("TrialCh21", "TrialCh22");
```

```
/* DOCUMENT COUNTER place */

cnt = new Counter;

cnt->value = 2;

/* TYPES place */

mt1 = new MetaType;

mt1->name = "UnixFile";
mt1->version = "1.0";

txt = new SpadeText;
txt->addStr("class UnixFile inherit ModelType public type");
txt->addStr("tuple(fileName: string,");
txt->addStr("      owner: string)");
txt->addStr("end;");

mt1->definition = txt;

mt2 = new MetaType;

mt2->name = "User";
mt2->version = "1.0";

txt = new SpadeText;
txt->addStr("class User inherit ModelType public type");
txt->addStr("tuple(name: string,");
txt->addStr("      role: string,");
txt->addStr("      usrDisplay: string)");
txt->addStr("      xQuestionId: string)");
txt->addStr("end;");

mt2->definition = txt;
```

```
mt3 = new MetaType;

mt3->name = "Document";
mt3->version = "1.0";

txt = new SpadeText;
txt->addStr("class Document inherit Element public type");
txt->addStr("tuple(chapters: list(string))");
txt->addStr("end;");

mt3->definition = txt;

mt4 = new MetaType;

mt4->name = "Chapter";
mt4->version = "1.0";

txt = new SpadeText;
txt->addStr("class Chapter inherit Element public type");
txt->addStr("tuple(ownerDoc: string)");
txt->addStr("end;");

mt4->definition = txt;

mt5 = new MetaType;

mt5->name = "Counter";
mt5->version = "1.0";

txt = new SpadeText;
txt->addStr("class Counter inherit ModelType public type");
txt->addStr("tuple(value: integer)");
txt->addStr("end;");

mt5->definition = txt;
```

```
mt6 = new MetaType;

mt6->name = "Answer";
mt6->version = "1.0";

txt = new SpadeText;
txt->addStr("class Answer inherit ModelType public type");
txt->addStr("tuple(objName: string,");
txt->addStr("      value: string)");
txt->addStr("end;");

mt6->definition = txt;

mt7 = new MetaType;

mt7->name = "Element";
mt7->version = "1.0";

txt = new SpadeText;
txt->addStr("class Element inherit ModelType public type");
txt->addStr("tuple(name: string,");
txt->addStr("      responsible: string,");
txt->addStr("      fileName: string)");
txt->addStr("      compOK: boolean)");
txt->addStr("method public myTypeIs: string");
txt->addStr("end;");
txt->addStr("method body myTypeIs: string in class Element {");
txt->addStr(" ");
txt->addStr("o2 string tmp;");
txt->addStr(" ");
txt->addStr("if ((self->name)[0:2] == \"Doc\"");
txt->addStr("   tmp = \"Document\";");
txt->addStr("else");
txt->addStr("   tmp = \"Chapter\";");
txt->addStr("return tmp;");
txt->addStr("};");
```

```
mt7->definition = txt;

/* ACTIVITIES place */

/* Here there are copy of Activity Root, Edit, Compile */

/* DYNAMIC TREE place */

dt = new DynamicTree;

dt->activeCopyId = 0;

tmpTreeNode = new TreeNode("root", 0);

dt->root = tmpTreeNode;

/* DYNAMIC TREE LOCK place */

tk = new Token;

/* PATH place */

ptN = new PathNode(0);

ptL = new PathList;

ptL->rootNode = ptN;
ptL->lastNode = ptN;

/* COUNTER place */

cntr = new CounterAC;

cntr->invokedActiveCopy = 0;
```

```
pl->addToken(cntr);

/* PLACECORRESP place */

acTab = new DynamicTable;

for (p in act2->places where p->corresp != unique set ()) {
    for (ref in p->corresp) {
        acTab->insertLocRef(p->name, (o2 KernelRef) ref);
    };
};
```

# Appendix C

# Dynamicity of types and activities

In SPADE-1, a model definition (activities and types) is implemented by means of some elements in the $O_2$ schema. In particular guards and actions are associated with $O_2$ functions. While types definition correspond to $O_2$ classes. The changing of the model causes the creation of new $O_2$ classes and functions.

The objective of the *schema management system* is to manage the dynamicity of the model. This means using the latest versions of types and activities. While keeping the older versions for the active copies that refer to them.

## C.1 Requirements

The schema management system should make sure that:

- every activity invocation binds the latest version of the activity definition and of all the types and classes used.
- every running active copy refers to the versions of types and activity definition taken at the time it has been invoked.

## C.2 Principles

Schema management system is based on the following ideas:

- The interpreter ignores every schema manipulation.
- The interpreter assumes that the received active copy is READY to be executed. That is: the functions corresponding to guards and actions are compiled and every class used by these functions is in the schema.

- Everything concerning schema is managed at Activity/Types definition time.
- As the editing of Activities and Types is part of the process, the schema management procedures are part of the meta-process. Thus it is the meta-process that is responsible for schema maintenance and consistency.

## C.3 Symbolic schema and $O_2$ schema

When an activity is invoked, the definitions contained in places `Types` and `Activities` apply. The newly created active copy uses these definitions. Suppose during the execution of this active copy, a type is modified and a new activity is invoked. Now the new definitions bind the last active copy types. So we have two running active copies using two different definitions of one symbolic type. The schema management system must manage different "instances" or versions of a single "symbolic" class. The "symbolic" class is the definition written by the process modeler. The $O_2$ schema, on the other hand, contains the *actual* "instances" of this class.

## C.4 The `SpadeClass` SPADE class (!)

Schema management is carried out through some additional data structures encapsulating types. A sort of type descriptor is provided by the class `SpadeClass`:

```
class SpadeClass inherit Token public type
    tuple(public timeStamp: SpadeDate,
          upToDate: boolean,
          o2class: string,
          name: string,
          dependencies: list(string),
          definition: SpadeText)
end;
```

Here is a brief description of each attribute:

**name** : is the symbolic class identifier defined by user. It appears in other classes type, places type and local variable declarations.

**o2class** : a class of the $O_2$ schema implementing the symbolic definition, this identifier is substituted to the class symbolic name in every actual class and function of the schema.

**upToDate** : is a boolean. It is `true` when the class is compiled in a consistent state. It is set to `false` when the class is modified or when its state is not consistent because of the modification of another class.

**dependencies** : is a list of all the classes related to this class. The consistency of the class depends directly on the consistency of all the classes of the dependency list. In other words, it is the list of the names of the classes "used" by this class. The names of the classes in this list are the symbolic ones.

**definition** : it is the textual definition of the class, it includes its type, methods signatures and methods bodies. As this text is edited by the process modeler, every variable declaration and every signature refers to symbolic names for classes.

**Dependency** : The dependency (or "use") relationship is the core of all the type management system. The meaning of this relation is straightforward, it is anyway useful to mention the different occurrences of a used class. We say a class $X$ uses class $Y$ iff:

- $X$ inherit from $Y$
- $Y$ is used in the type of $X$
- $Y$ is the return type of a method of $X$
- $Y$ is the type of a parameter of a method of $X$
- $Y$ is the type of a local variable of a method of $X$

Clearly the dependency relation is transitive, however the *dependencies* attribute of a `SpadeClass` stores the direct dependencies only.

The `Types` global place is a `SpadeClass` token container. Each ModelType class is represented by a `SpadeClass` token.

## C.5 The `TypeSet` SPADE class

Due to the dependency relations, the editing of one class often involves a large number of classes. The SPADE class `TypeSet` has been defined to group a set of related classes. The definition is simple:

```
class TypeSet
 public type
    tuple(classes: list(SpadeClass));
end;
```

This class implements the core procedures of the schema management system. Three methods are defined for this purpose:

```
method public closure(name:string) in class TypeSet;
method public substitute(c:SpadeClass):string in class TypeSet;
method public compile in class TypeSet;
```

Section C.6 gives the details about these methods, further an example of the use of such features in a *meta-process* is presented.

The functionalities of a `TypeSet` object assure consistency only if its class list is closed with respect to the dependency relation. In turns, for each SpadeClass $C$ of a TypeSet $T$, all the classes that are "used" by $C$ must be included in $T$. The forementioned example also shows a guard expression that binds a closed set of types.

Each active copy has its TypeSet. The types mentioned for an activity TypeSet are related to:

- places (typeName)
- local variables declarations

For example consider classes `A`, `B` and `C` defined by user as follows:

```
class A inherit ModelType
public type
tuple( num:integer,
       x: C )
end;

class B inherit A
public type
tuple( name:string )
end;

class C inherit ModelType
public type
tuple( stack: list(B) )
end;
```

The corresponding entries in a TypeSet are:

| *name* | *$O_2$class* | *upToDate* | *dependencies* | *definition* |
|---|---|---|---|---|
| A | _A1 | true | C | ... |
| B | _B3 | true | A | ... |
| C | _C1 | true | B | ... |

Class A depends on ModelType because of the *inheritance* relationship and it depends on C because of a *use* relationship. Note that, since dependency is a transitive relationship, actually class C depends on class A (and vice versa). However only direct dependencies appear in the table.

Each activity has a reduced TypeSet. The types stored in an activity TypeSet are related to:

- places (typeName)
- local variables declarations

when the activity is a token of the place Activities, the table shows the latest binding between actual and symbolic classes.

Every active copy holds in its activity definition its private table which keeps the values given at creation time, taken from the Activities place. These data are used whenever the active copy is suspended, for activity and type modification.

## C.6 Schema modifications

The goal of the schema management system (see section C.1) is well described by the following invariant statement:
*Every TypeSet must be "upToDate", if the TypeSet is bound to an Activity, then the Activity must be compiled with that TypeSet.* In this section the details of TypeSets maintenance and consistency procedures are investigated.

Every modification of a typeSet is originated by a type modification. So suppose the following TypeSet represent place Types:

| *name* | *$O_2$class* | *upToDate* | *dependencies* | *definition* |
|---|---|---|---|---|
| A | _A1 | true | B, D | ... |
| B | _B3 | true | - | ... |
| C | _C1 | true | B | ... |
| D | _X7 | true | C | ... |

while place Activities contains activity AC1 using types A and B. The activity is compiled with the binding A→_A1 and B→_B3, so the invariant statement is clearly satisfied.

Now suppose class `C` is modified and saved. This user action triggers some meta-process procedures:

1. compute the obsolete classes set (transitive closure).

   (a) set *upToDate* to `false` for class `C`.

   (b) for each class, check the classes in the `dependency` list. If one of the class of the list is *not upToDate*, then set that class *not upToDate*.

   (c) repeat step 1b until no more classes become obsolete.

   In our example the result would be:

| *name* | $O_2$*class* | *upToDate* | *dependencies* | *definition* |
|---|---|---|---|---|
| A | _A1 | false | B, D | ... |
| B | _B3 | true | - | ... |
| C | _C1 | false | B | ... |
| D | _X7 | false | C | ... |

2. for each obsolete class choose a new class name.

3. compile new classes prototypes:

```
class _C2
end;

class _X8
end;
...
```

   now the TypeSet is filled in with the new names:

| *name* | $O_2$*class* | *upToDate* | *dependencies* | *definition* |
|---|---|---|---|---|
| A | _A2 | false | B, D | ... |
| B | _B3 | true | - | ... |
| C | _C2 | false | B | ... |
| D | _X8 | false | C | ... |

4. for each obsolete class, every occurrence of any obsolete symbolic class name is replaced with the new actual name.

5. for each obsolete class, the new full description is compiled.

6. the Activities affected by table updates are compiled: every transition guard and action function is compiled.

Again, since all these procedures access the global places `Activity` and `Types`, they must be included in the meta-process activity.

## C.7 Activity invocation

The kernel-SLANG expansion of an activity invocation should provide a deep copy of all the type definition used in the activity in addition to the private copy of the above table. The definitions copied are taken from the place Types

## C.8 Schema garbage collection

This schema management system produces lots of classes and functions. Many of these classes become soon obsolete because of activity or types modifications. The schema management system should cope with the growing number of obsolete classes.

First of all let us define clearly what does *obsolete* mean. An actual class `_X` class becomes obsolete when:

- it is not yet the latest actual class of any symbolic class, in the activity definitions.

- no active copies (running or "token") use `_X` in their tables.

Note that the class is obsolete only when both these conditions apply. The first one is very easy to verify. An actual class is no longer the latest version of a *global* type definition, as soon as this definition is modified.

The second condition poses some serious problems. It is very difficult to verify because active copies and activities can be tokens, and the condition must be valid for the whole process model. Thus a complete check would involve all the active copies and their state. In this perspective it is not clear whether this procedure is part of the model or it must be performed "off line" by suspending the whole process.

Here are two ideas:

- No run-time garbage collection is provided. The number of the classes of the schema grows monotonically. Actual classes (under `ModelType`) are removed from the schema when the root active copy terminates.

+ : very simple and fast solution.

– : a long lasting model would probably be stuck because of schema saturation.

- A periodic "overnight" house keeping procedure could suspend the execution allowing a complete check of the model data. All the functions and classes that are not referenced by any definition can be deleted.

+ : an effective solution.

– : this procedure must be manually activated. It require global suspension mechanisms. This procedure should navigate throughout the active copies, visiting their state and their temporary edata structures.

A usable mechanism for schema garbage collection is not yet implemented.