Knowledge-based Support to Process Integration in ODE

Fabiano Borges Ruy, Gleidson Bertollo, Ricardo de Almeida Falbo

Federal University of Espírito Santo

Fernando Ferrari Avenue, Vitória - Espírito Santo, 29060-900, Brazil

+55 (27) 3335-2167

{fruy, gbertollo, falbo}@inf.ufes.br

ABSTRACT

Process integration in Software Engineering Environments (SEE) is very important to allow tool integration. In this paper, we present a knowledge-based approach to improve process integration in ODE, an ontology-based SEE.

Keywords

Software Engineering Environments, Software Process, Knowledge-based Support.

1. INTRODUCTION

Software Engineering Environment (SEEs) aims to combine techniques, methods and tools to support the software engineer in the task of building software products, considering all the software process activities, such as management, development and quality control [1].

Developing quality software products on schedule and considering planned costs has always been a challenge to software organizations. However, the quality of a software product depends heavily on the quality of the software process used to develop it [2]. In the context of SEEs, this fact leads to the concern of process integration, i.e. an explicit link between the SEE's tools and the software process [3], giving rise to Process Centered SEEs (PSEEs).

Due to the complexity involved not only in integrating software process, but also in many tasks supported by SEEs, some technologies have been adopted in the construction of such environments as a way of improving their functionalities. Among these technologies, it is the use of knowledge-based techniques. Thus, it is important that a SEE makes available an infrastructure to deal with knowledge, so that their tools can offer knowledge-based support to developers.

This paper presents how process integration is supported by an infrastructure for knowledge integration in ODE (Ontology-based software Development Environment) [4], a Process Centered SEE. This infrastructure allows to handle knowledge bases and to perform inferences using a Prolog engine. Process integration in ODE is based on an approach of explicitly defining the software process for a project and then controlling its execution.

The goal of this paper is to present an approach that combines both objects and logic in a knowledge integration infrastructure able to create knowledge bases and to manipulate them through the execution of inferences. In that way, it is possible to obtain a knowledge-based support in several environment tasks. The paper focuses on the use of this infrastructure for process integration in ODE.

The paper is organized as follows: section 2 discusses integration in SEEs. Section 3 presents ODE and its knowledge integration infrastructure. In section 4, the process integration in ODE is discussed, considering process definition and control. Related works and conclusions are discussed in sections 5 and 6, respectively.

2. INTEGRATION IN SOFTWARE ENGINEERING ENVIRONMENTS

Although benefits can be derived from individual CASE tools that address separate software process activities, the real power of CASE technology is achieved only through integration. Isolated tools are "point solutions" in the sense of being used to assist in particular software engineering activities [5]. They do not directly communicate with other tools, are not tied into a project database, and generally have their own concepts, data, representations, and ways of accomplishing the purposes to which they were built. However, when working together, the level of effectiveness becomes significantly higher.

Integration – of tools, processes, artifacts, and views – has been considered one of the most challenging issues on software engineering environment (SEE) research [6]. Integration demands consistent representations of software engineering information, standardized interfaces between tools, homogeneous means of communication between software engineers and tools, and an effective approach that enables SEE to move among various platforms [5].

The identification of the need for integrated support to software engineering activities throughout the software lifecycle represents the genesis of Software Engineering Environments (SEEs) [6]. Thus, SEEs can be defined as integrated collections of tools that facilitate software engineering activities across the software lifecycle [6]. A SEE provides a means of integrating people in a software development organization with the software process and with the supporting technology [7]. To do that, SEEs must provide an infrastructure for tool integration that should address [3] [1]:

- *Data Integration*: addresses the way tools share data. A SEE must provide data management services, allowing tools to share project and software engineering information.
- *Presentation Integration*: concerns commonality of user interface. The user interfaces in a SEE must be homogeneous and consistent, allowing developers to alternate between the tools without substantial style changes, and therefore, with less impact on productivity and learnability.
- *Control Integration*: relates to the ability of a tool to notify and initiate actions in another tool, controlling the events that might occur and sharing functionalities.
- *Process Integration*: concerns the linkage between the tools and the software development process. To integrate tools, it is necessary to have a strong focus on the software process management. The process must define the tools a developer can use, and when he/she will have access to these tools, according to the activities of the process he/she is performing.
- *Platform Integration*: relates to the independency of the platform on which the environment and their tools will run.
- *Knowledge Integration*: refers to the establishment of the semantics of the information exchanged between the various tools in a SEE. With the growing complexity of software processes, it becomes necessary to offer knowledge-based support to help software engineers in their tasks and to manage the knowledge captured during the software projects. In this way, knowledge, as well as data, has to be available to the environment in order to be shared among their tools.

The need for (semi-)automated support for the software process, in addition to tool support for artifact development, gave rise to a new class of SEEs, called Process-centered Software Engineering Environments (PSEEs), which integrate tool support for software artifact development with support for the modeling and execution of the software processes that produce those artifacts [6]. The explicit representation of software processes, their products, and their interactions, is the foundation over which modern integrated development environments are built. In providing more powerful ways of describing and implementing software engineering processes, PSEEs have also provided a powerful means of integrating processes and tools, and (partially) automating tasks [6]. In general, a PSEE should include mechanisms to [7]: (i) guide sequences of activities that compose the defined process; (ii) manage products that are being developed; (iii) invoke tools that developers require to do their tasks; (iv) perform automatic actions that do not need human intervention; (v) allow communication between the persons who are developing the project; (vi) gather metric data automatically; (vii) reduce human errors and (viii) provide project management with current accurate status information.

3. ODE: AN ONTOLOGY-BASED SEE

ODE (*Ontology-based software Development Environment*) [4] is a Process-centered SEE, developed using ontologies. ODE's design premise is based on the following argument: if the tools in a SEE are built based on ontologies, tool integration can be improved. The same ontology can be used for building different tools supporting correlated software engineering activities. Moreover, if the ontologies are integrated, integration of tools built upon them can be facilitated. This is the main feature that distinguishes ODE from other SEEs: its ontological basis. Especially in SEEs, ontologies can be used to reduce terminological and conceptual confusions, facilitating shared understanding and communication between people with different needs and points of view. Besides the standardization of concepts provided by it, an ontology allows a refined communication between the tools that compound the environment [4].

Among the ontologies on which ODE is based, one must be highlighted in the context of this work: the software process ontology [1]. This ontology defines the main concepts related to software process and is the basis for process integration in ODE.

ODE's architecture reflects its ontological basis. It has two levels: the Base Level and the Meta-Level, as shown in figure1. The base level defines the classes that control the processes defined in the environment (the *Control* package) and their tools. The meta-level (or knowledge level) defines classes that describe knowledge about objects in the base level. Because of that, these classes are named prefixed by the character "K", as will be seen in section 4.

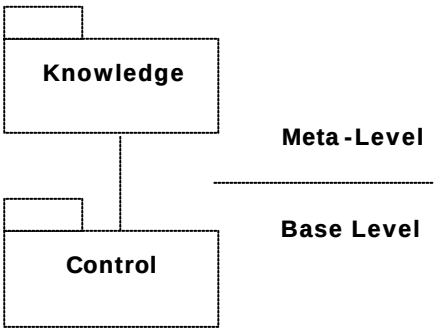

Figure 1 – ODE's Layered Architecture.

The classes in the meta-level are directly derived from the ontologies and its objects can be viewed as items of an ontology instantiation [4]. These instances constitute the environment's knowledge, which can be used by their tools. In the case of the software process ontology, instances of knowledge classes describe knowledge about software processes, its activities, artifacts produced and consumed, resources and procedures (methods, techniques and guidelines), among others.

However, an ontology does not intend to describe all the knowledge involved in a domain, but only that one that is essential to conceptualize the domain (minimal ontological commitment [8]). Therefore, new classes, associations, attributes and operations have to be defined to deal with specific design decisions made in the application level. In fact, the ontology is a general, common sense model, and then it does not contain all necessary modeling elements to treat applications' requirements.

The classes in the base level are also built based on the ontologies. The main classes and associations are derived from the ontology, preserving the same constraints as Knowledge's model. To deal with software processes, for example, the classes of the *Control* package were developed. They are used for defining and tracking software processes for projects developed in ODE. Some of these classes have a corresponding class at the meta-level. This way, the meta-level is used to describe base-level objects' characteristics in the *Control* package (see section 4).

3.1 ODE's Inference Layer

In general, conventional systems that aim to represent knowledge, including object-oriented systems, keep their knowledge stored in a data repository, from where it can be consulted, and the rules related with this knowledge are embedded into the application code, turning impossible to take advantage of the whole potential that the knowledge-based technology can provide.

A first step towards a more effective approach to deal with knowledge is to choose a more suitable form of representation. Among the several forms of representing knowledge, there is the logical paradigm approach, where facts are stored and accessed from a data repository, and the rules are isolated from the application. Keeping rules isolated of the application is a flexible way to maintain the knowledge without the complete restructuring of applications. The result of its use is a more robust system, with a flexible knowledge base that can grow and change as the business evolves [9]. Besides, the use of artificial intelligence techniques enables to improve the manipulation of these rules, allowing inferences.

If a SEE is able to deal with knowledge about projects, processes, activities, resources, methods and tools, it can offer intelligent support to managers and developers, by guiding, monitoring and aiding the use of tools during the software development. Therefore, in the ODE's context, an infrastructure was developed to support knowledge integration. This infrastructure has a layer responsible for combining objects and logic, the Inference Layer, endowing the SEE with the capability to represent knowledge through rules, and to perform inferences. Using the inference layer, knowledge can be captured from the environment's repository, represented in knowledge bases using a logical language (Prolog), and can be treated through an inference mechanism.

When introducing a logical language in an object-oriented environment, it is possible to take advantage of the benefits of a second paradigm. Situations as the ones that involve recursivity or working with many relationships between objects, demanding pattern matches or several accesses to objects' information, are likely to be treated by the logical paradigm. Those kinds of situations are quite common, especially in operations involving knowledge.

3.1.1 How the Inference Layer works

One of the goals of the Inference Layer is to make available in ODE a way of representing knowledge logically, using Knowledge Bases (KBs). The purpose is to express, through facts and rules, information that can be extracted from the environment, as, for example, data about past projects.

The logical representation of rules can bring advantages, because rules can be modified without the need to change or recompile the system. Taylor [10] says that keeping rules embedded in code works in small systems, but it does not scale up well and it requires reprogramming every time a rule changes. He also says that the harder task of rule manipulation is make rules easy for business people to understand and use. This can be reached through friendly interfaces. Systems that combine objects and rules with graphical user interfaces become more robust, more understandable, and easier to maintain [9].

The Inference Layer must provide the knowledge engineer with a simple way to create and to manipulate KBs. For such, an editor of Knowledge Bases Structures was developed. In this editor, predicators and rules can be defined. Those structures are used as a framework for instantiating KBs. In KB instantiation, facts are defined from predicators (defined in the knowledge base structure) and data extracted from the environment, and the rules are those defined in the knowledge base structure. Starting from an instantiated KB, it is possible to manipulate its knowledge through a Prolog inference engine [11] incorporated into the layer.

Figure 2 illustrates the general operational scheme of the Inference Layer, and figure 3 shows its internal structure, which are detailed follow.

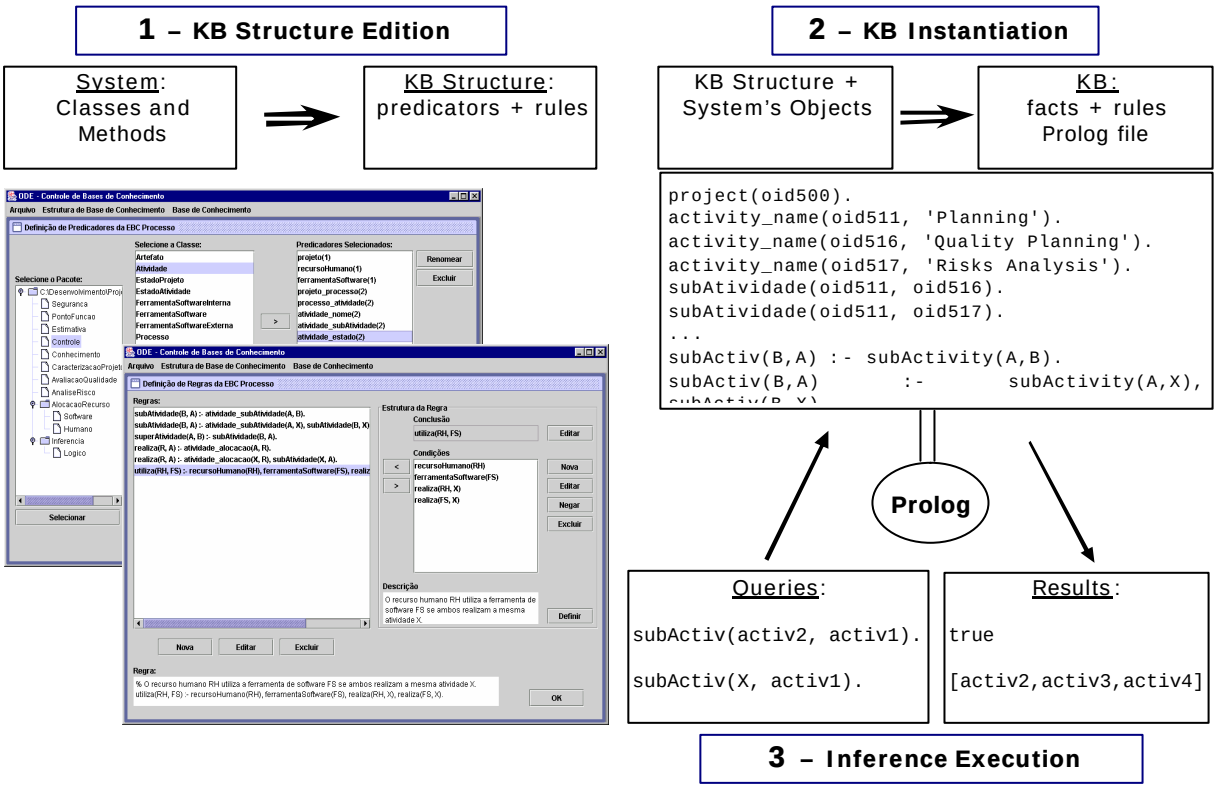

Figure 2. Inference Layer use Scheme.

3.1.2 Editing Knowledge Base Structures

The Knowledge Base Structures Editor allows to create and to maintain KB structures. Since a KB is formed by a collection of facts and rules, a KB structure has to have the necessary elements to create the facts and rules. The KB structures should not have facts, but just its structure. Therefore, predicators and rules compose them. *Predicators* are symbols that make statements using *terms*. Each predicator has an arity associated to it, which indicates the number of terms required for creating the statement. Through the predicators, the simplest statements are created: the atomic statements, called *predicates*. A predicate containing *variables* as terms is an *open predicate*. If all the terms are *constants*, the predicate is called a *closed predicate* or a *fact* [12]. *Rules* are modeled as a conjunction of conditions plus a conclusion. The conditions and conclusion of the rules are represented by predicates [12].

Figure 3 shows the *Logical Model* package that models the concepts of first order logic according to the OO paradigm. This is the internal representation of knowledge base structures and knowledge bases in ODE.

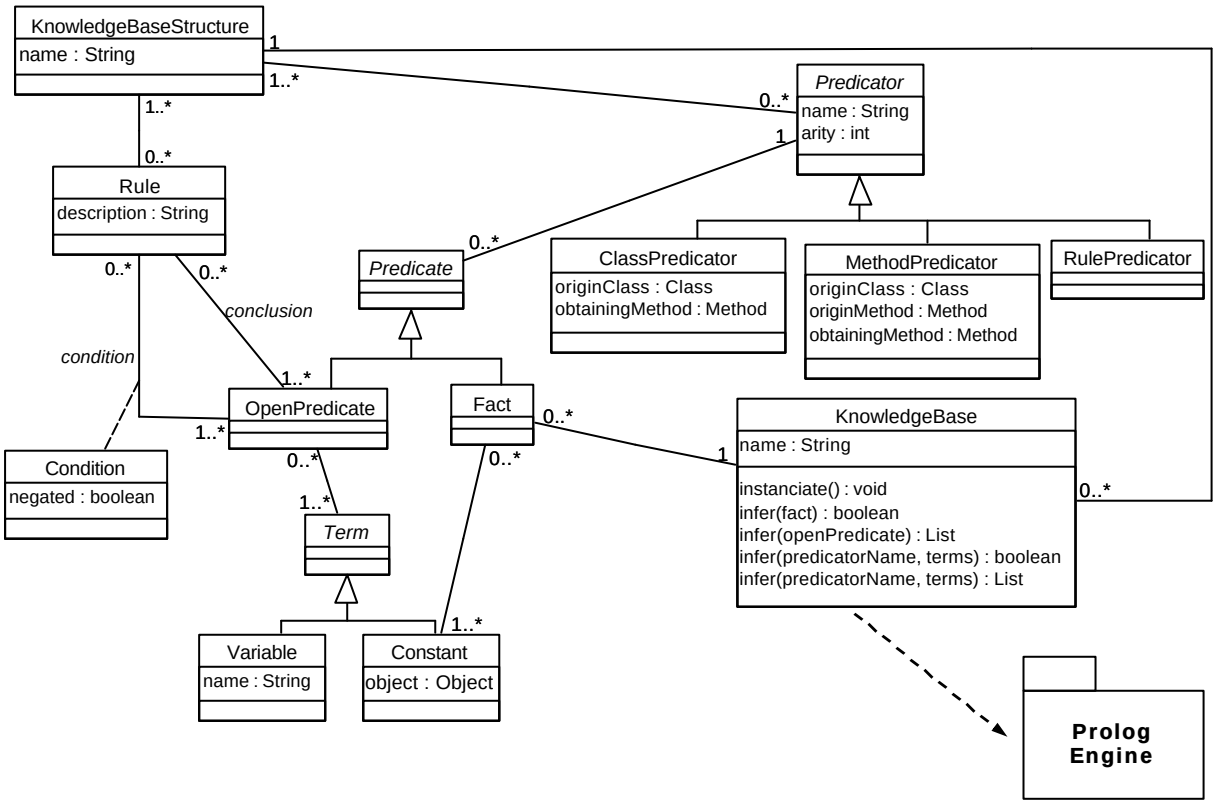

Figure 3. Internal Structure of the Inference Layer – Class Diagram of the Logical Model Package

The first step to create a KB structure is to define its predicators. Starting from the ODE's package hierarchy, this can be done by using computational reflection to obtain information about classes and methods of the various packages that compound the environment. Two types of predicators can be created: **`ClassPredicator`**, to represent classes, and **`MethodPredicator`**, to represent the attributes and associations of classes, defined through their corresponding accessing methods. Those predicators store, besides their names and arities, information about their origin (class or method and class) that is

useful to instantiate a KB. Figure 4 shows some predicators being defined starting from the ODE's package hierarchy.

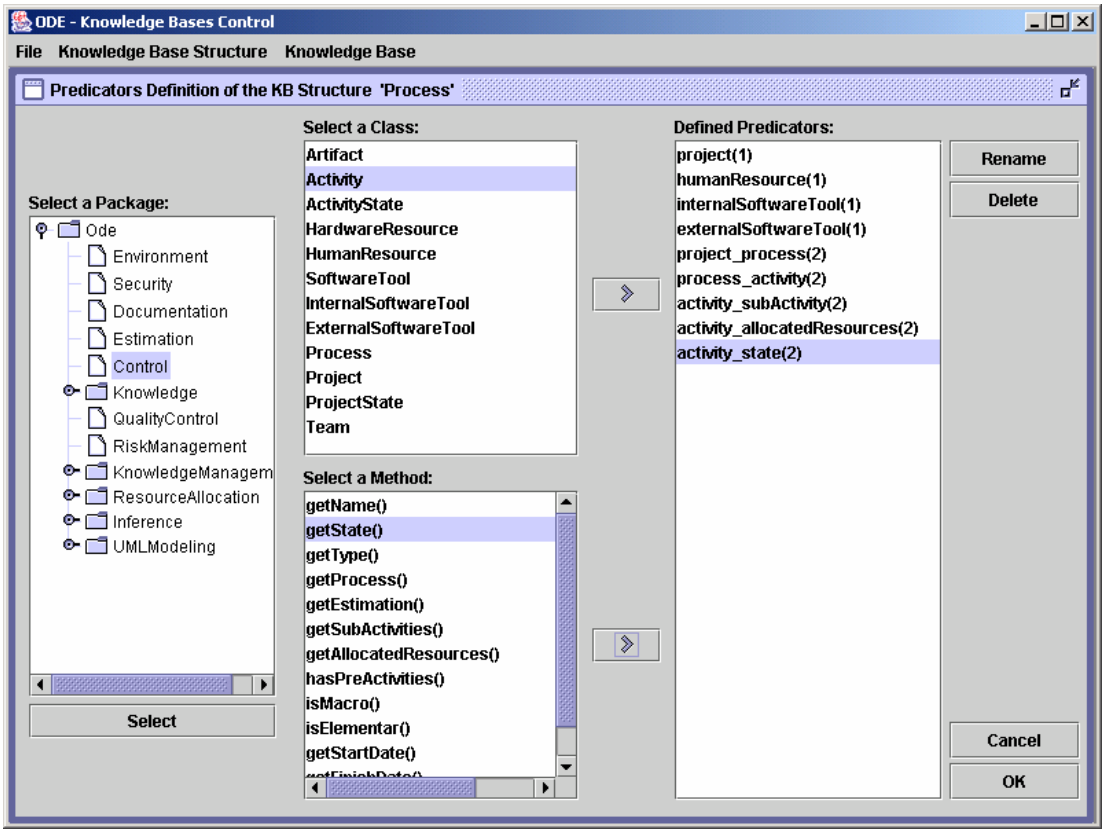

Figure 4. Predicators Definition

Once the predicators have been defined, rules can then be defined. A rule always owns one conclusion and, at least, one condition, represented by open predicates. Conditions are created from existing predicators and their associated terms (constants or variables), giving rise to predicates. A condition can be negated or not and its number of terms is determined by the arity of the predicator. A conclusion is created using a **`RulePredicator`** and associated terms, compounding an open predicate. Rule predicators are not created during predicator definition, but only when the rule heads (conclusions) are defined. Figure 5 illustrates the definition of rules using the previously defined predicators.

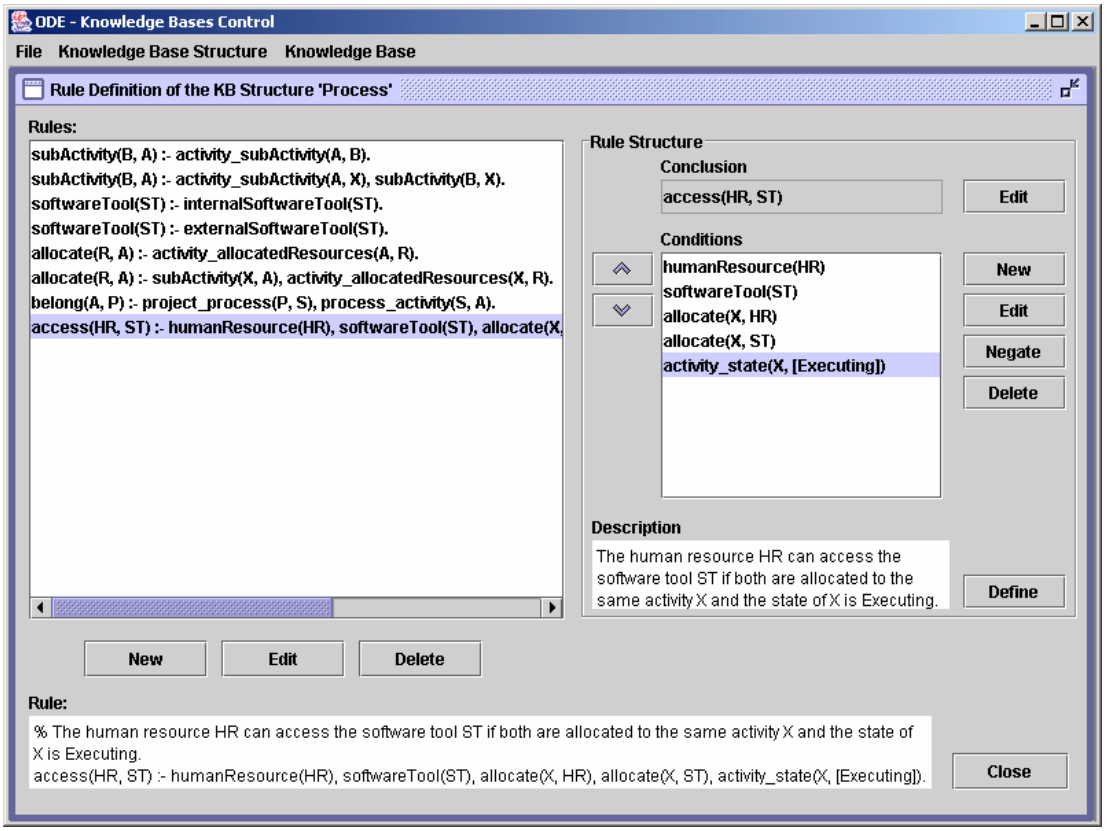

Figure 5. Rules Definition

3.1.3 Instantiating Knowledge Bases

During KB instantiation, a KB structure is used. Predicators originate facts, and rules are copied into the new KB. The most important step is the transformation of predicators in facts. For each predicator, a list of facts is created obtaining data from the environment. There are two ways of creating facts, according to the type of the predicator used:

- `ClassPredicator`: generates facts with arity 1 (one), which logically represent the instances of its origin class. Each fact generated has the name of the predicator and, as its single term, one instance (object) of the class. For example, be a class predicator with name `humanResource`, whose origin class is `HumanResource`. If `John` is an instance of this class, then a fact `humanResource([John])` is created.
- `MethodPredicator`: generates facts with arity 2 (two), which are logical representations of attributes and associations of the origin class. The facts are created in the form **`method_predicator(instance, method_return)`**. So, a method predicator named `activity_state`, that represents an association between objects of the classes `Activity` and `ActivityState`, originates facts like `activity_state([Requirements Analysis],[In Execution])`, indicating that the activity `Requirements Analysis` is in the state `In Execution`.

At the end of instantiation, the lists of facts and rules are used to produce a Prolog file, which is the logical representation of a KB. The file generated has a set of facts and rules as shown in figure 6. Notice that constants created from business objects are represented by their respective object identifiers (oid).

```
% File Control.pl
% Knowledge Base “Control”

% Facts Base:
project(oid50_0).
project_process(oid50_0, oid50_10).
process_activity(oid50_10, oid50_11).
. . .
process_activity(oid50_10, oid50_19).
activity_name(oid50_11, 'Planning').
activity_name(oid50_12, 'Requirements Analysis').
activity_name(oid50_13, 'Design').
activity_name(oid50_14, 'Implementation').
activity_name(oid50_15, 'Tests').
activity_name(oid50_16, 'Risk Analysis').
activity_name(oid50_17, 'Quality Planning').
activity_name(oid50_18, 'Risk Management').
activity_name(oid50_19, 'Risk Identification').
subActivity_(oid50_16, oid50_11).
subActivity_(oid50_17, oid50_11).
subActivity_(oid50_18, oid50_16).
subActivity_(oid50_19, oid50_16).
. . .

% Rules Base:
% B is subactivity of A if the activity A has B as its subactivity.
subActivity(A, B) :- subActivity_(A, B).

% B is subactivity of A if B is subactivity of X, that is subactivity of A
(transitivity).
subActivity(A, B) :- subActivity_(A, X), subActivity(X, B).
. . .
```

Figure 6. Example of a generated Prolog File.

3.1.4 Executing Inferences

Creating KB structures (i.e., defining their predicators and rules), and instantiating them in KBs are steps that must occur previously to the inference execution. When the Inference Layer has an instantiated KB, it is able to attend to its main purpose: to perform logical inferences on knowledge, using a Prolog engine. A query can be done in two distinct ways, depending on the type of the predicator used:

- **`Facts`**: since all terms in the query are constants, then the result indicates whether the fact is true or false;
- **`OpenPredicate`**: since it has a variable as a term, the query result is a list containing the variable possible values.

In both cases, what the Inference Layer does is to transform the objects in logical data and then perform the inference on the Prolog file generated during the KB instantiation, using a Prolog engine. The result of the inference is transformed into objects and returned to the application.

The inferences can be executed in two modes: via graphical interface and via code. The first is simpler and used through a graphical application, in which the user mounts the desired queries and obtains the inference results only visually. The main goal of this functionality is to allow testing the KB structures defined and the KBs created, performing inferences in order to check coherence of the defined predicates and rules. Thus, the Knowledge Engineer can check a KB before it is effectively used in a system.

Besides, this approach facilitates users to learn how the inference layer works, so they can use the second mode of inference (via code) better.

As shown in figure 7, the user defines the queries by choosing the corresponding elements needed to mount an open predicate or a fact. Initially, a predicator should be selected and its terms determined. After that, the inference is executed, and the results are presented in the frame on the right. It should be notice that business objects are shown as `Strings` between brackets.

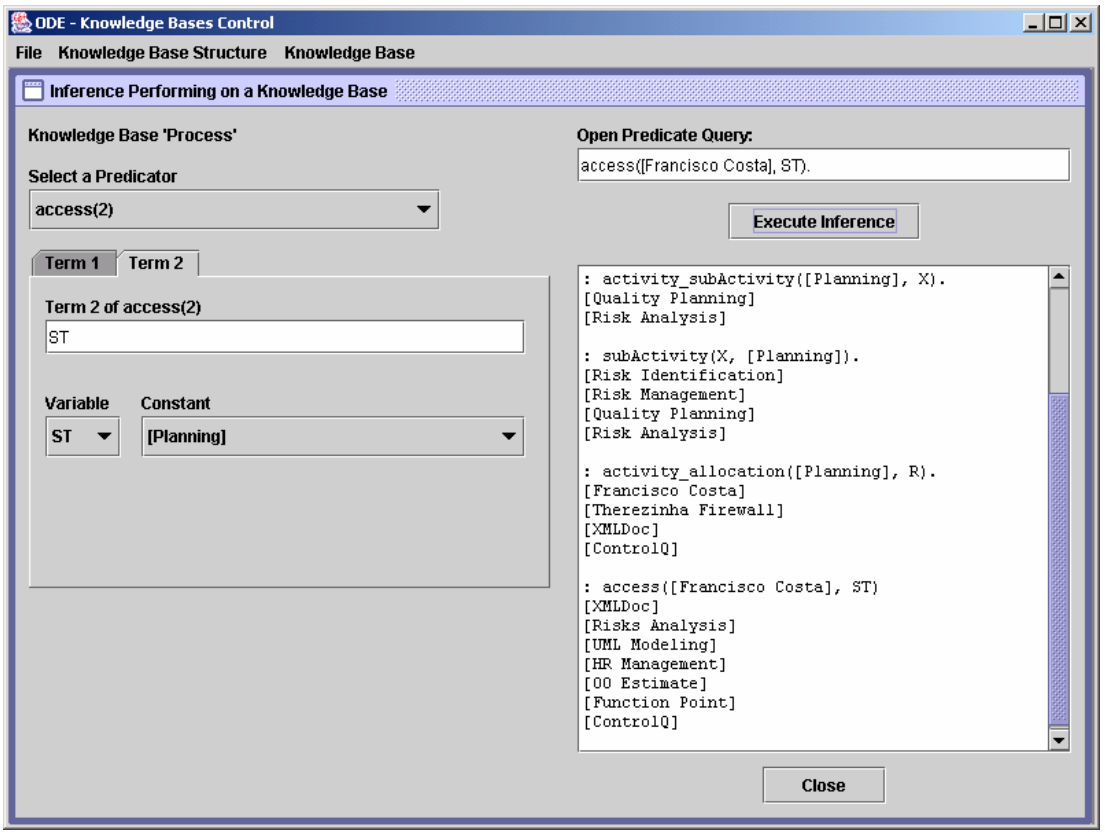

Figure 7. Inference Execution via Interface

The second mode to use inferences is accomplished via implemented code. Even keeping rules isolated, the systems need to access them to work. The intention is to make the environment obtain results through rules of a KB. Working like this, operations purely codified can be replaced by calls to a KB, with the advantage that the rules involved in those operations, because they are interpreted, can be easily changed, without need to recompile the system. It is important to emphasize that the system continues to operate with objects, obtained, now, through a different paradigm.

To perform inferences via code, the `KnowledgeBase` class provides two methods for query: one for facts and another for open predicates. For both methods, the name of the inference predicator and its terms should be passed as arguments. The constant terms are application objects and the variable terms are instances of the class `Variable`.

Figure 8 shows a fragment of code used to perform an operation via logical inference, which the result is a list of objects. As it can be observed, what should be done to perform the operations is to infer on a KB, passing the predicator name and an array of objects as terms. Internally, the KB searches for the

predicator and creates the terms using the objects passed as arguments. Then the predicate is mounted to perform the inference and the result is treated and returned.

```
//Get the subactivities of the passed activity
public List getSubActivities(Activity activity) {
  //Performing inference with OpenPredicate
  Variable var = new Variable("X");
  Object[] terms = new Object[] {var, activity};
  List lstSubActivities = knowledgeBase.infer("subActivity", terms);
  return lstSubActivities;
}
```

Figure 8. Inference Execution via Code

4. PROCESS INTEGRATION IN ODE

Since ODE is a process centered SEE, process integration is essential to it. ODE's process kernel was built using the software process ontology presented in [1] as its foundation. This ontology was designed to support software process definition, tracking and integration in a PSEE. Every tool compromised with this ontology shares a common vocabulary, facilitating the communication between tools and allowing reuse.

To deal with software processes, first the environment should support their definition. The process definition, in turn, is used to establish an explicit connection between the environment's tools and the defined processes. In the next subsections, important features of the process integration in ODE are discussed, including process definition and process control, establishing the connection with the tools that automate it.

4.1 Process Definition in ODE

Software process definition is a hard and knowledge intensive task. To do that, project managers must have a high level of experience and knowledge. Several aspects should be considered, such as the development team characteristics, project specific features, and organization's knowledge level in software engineering.

ODE uses a three-level approach to process definition [13], as shown in Figure 9. Each level has different features that influence the processes definition in it. At the higher level, the standard process of the organization is defined. The organizational process contains the process assets (activity, artifacts, resources and procedures) that should be part of the processes of any organization's project. At the intermediate level, the standard process can be specialized to consider development technologies, paradigms or specific application domains. During the specialization, process assets can be added or modified, in agreement with the context of the specialization (technology, paradigm or application domain). Finally, in the lowest level of the process definition model, it is the instantiation of a standard process or a specialized process for a specific project. The process instantiation consists in tailoring a standard or specialized process for a specific project. In this adaptation, particularities of the project and development team characteristics should be considered. At this moment, the life cycle model to be used during the development is defined, and new activities, as well as consumed and produced artifacts, used resources and adopted procedures, can be added to the process.

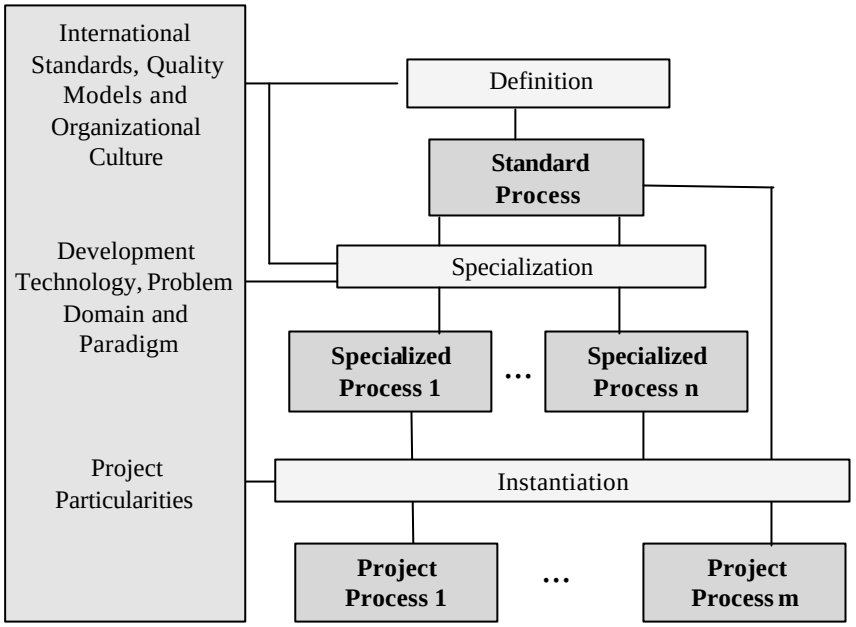

Figure 9. Process Definition Model.

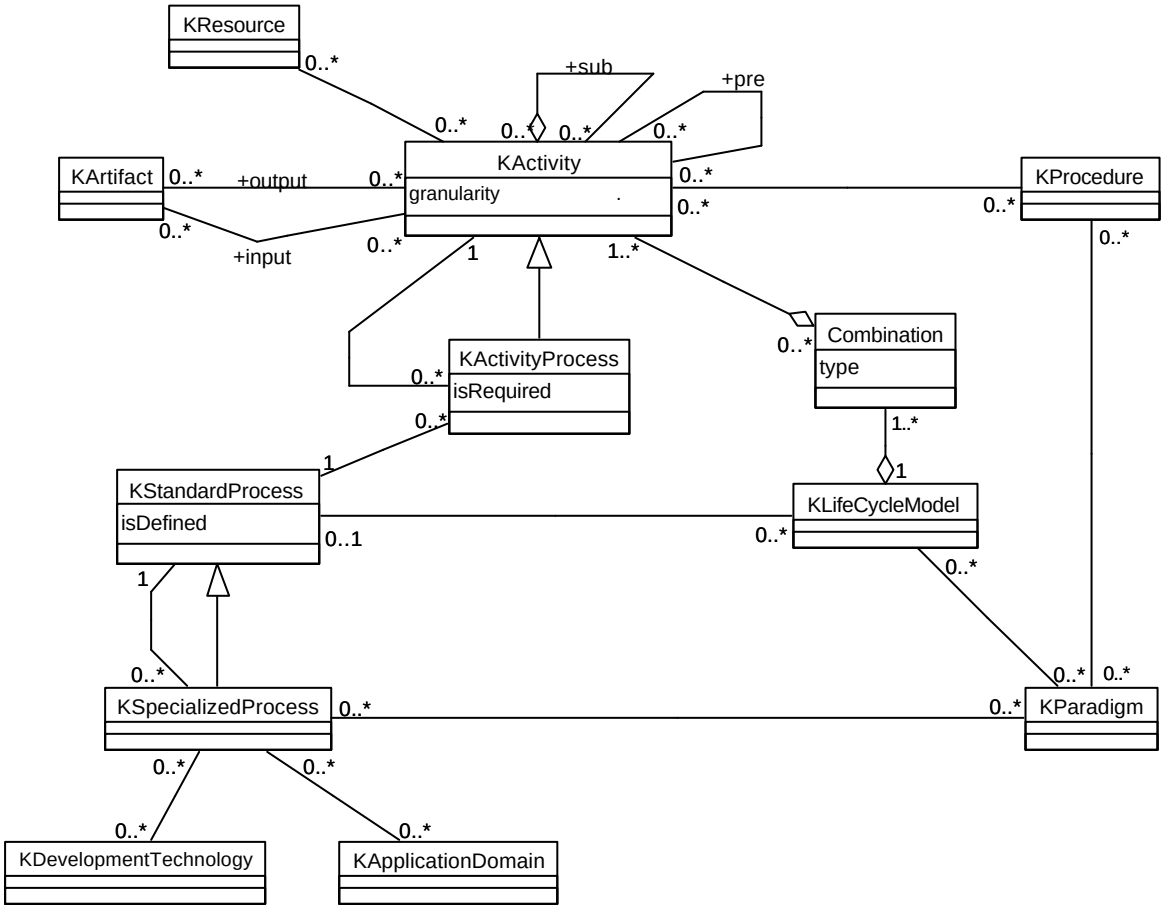

Figure 10. The Knowledge Package.

The different levels of process definition occur in the different levels of ODE's architecture (presented in the figure 1). The standard and specialized processes describe the activities (and the others process assets) that should belong to all the organization's project processes. In other words, those processes describe the knowledge regarding the processes of that organization. This way, according to ODE's architecture, standard and specialized processes are defined in the Meta-Level, i.e., in the *Knowledge* Package, shown in Figure 10.

Once the life cycle models for a standard or specialized process has been defined, these can be used in the project process instantiation. It should be emphasized that, unlike standard and specialized processes definition that occurs in the Meta-Level of ODE's architecture, the project process definition occurs in the base level, more specifically in the *Control* Package, shown partially in Figure 11. The class `Process` represents project processes, defined for specific software projects (represented by the class `Project`). A project process can be defined based on a standard or specialized process, or it can be defined starting from the knowledge repository of the environment. Therefore, a `Process` can have, or not, an association with `KStandardProcess`. When a project process is defined starting from a standard or specialized process, all their assets are replied for the new process, in agreement with the standard/specialized process and the life cycle model chosen. At that moment, the activities of the process are created as well as its associations with the corresponding artifacts (`KArtifact`), resources (`KResource`) and procedures (`KProcedure`).

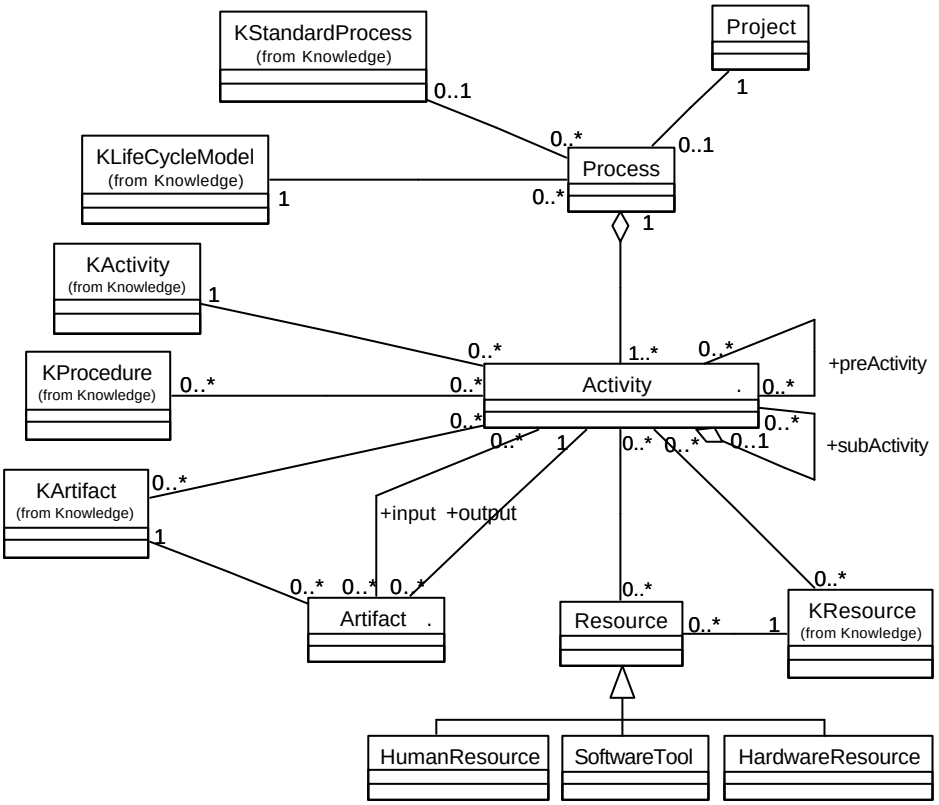

Figure 11. The Control Package (Partial Model).

The interface for standard, specialized or project process definition is similar and it is presented in Figure 12. The tree located on the left part of the window contains the process assets already defined for the process in definition (activities, artifacts, resources and procedures). On the right part, a panel makes possible to choose the process assets for the selected item in the tree.

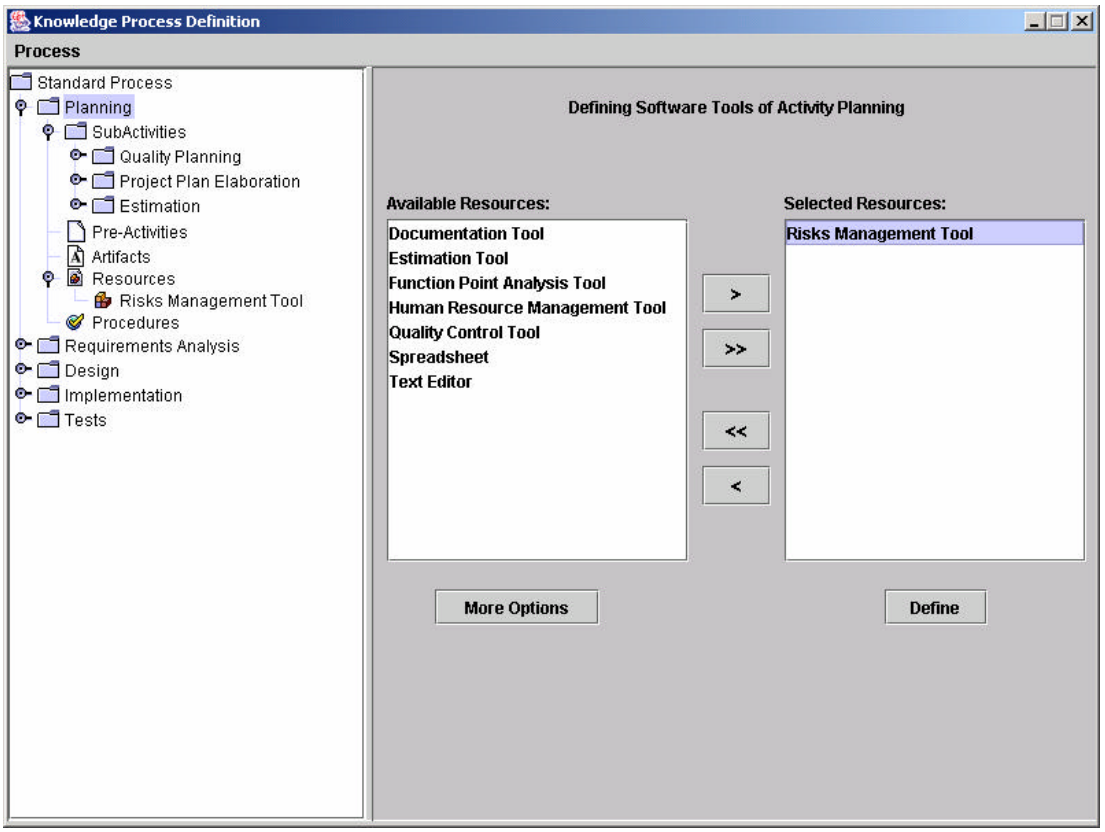

Figure 12. Standard, Specialized and Project Process Definition.

In this panel, the right list contains the elements already defined for the process. The left list contains the assets suggested by inferences performed in the knowledge base using the Inference Layer of the environment. The knowledge bases, previously defined by the Knowledge Manager based on the *Knowledge* package, are instantiated in the initialization of the tool. They are used through inferences to give suggestions for the project manager who is defining the process, offering knowledge based support to him/her.

Defining subactivities for a certain activity is an example of the use of inferences in the tool. The process definition tool should suggest, in a first moment, only the subactivities associated to an activity, as defined in the *Knowledge* package model. However, a more elaborated form of suggestion is given based on the *subactivities rule* shown in Figure 13. According to that rule, the subactivities of the subactivities are also exhibited, in agreement with the transitivity imposed by the rule, turning more effective the support to the process definition.

```
% Subactivity Rule (A is subactivity of B)
subActivity(A, B) :- subActivity_(A, B).
subActivity(A, B) :- subActivity_(A, X), subActivity(X, B).

% Resource Rule (Activity A uses Resource R)
use(A, R):- subActivity(X, A), use(X, R).
```

Figure 13. Some Used Rules

In a similar way, the *resource rule* (also shown in Figure 13) provides more effective suggestions for resource allocation. As shown in the code fragment of the Figure 14, the use of that rule allows an activity allocate resources that were defined for their subactivities. As shown in Figure 12, the Planning activity can allocate a risks management tool even if Planning doe not have Risk Analysis as one of its subactivities in the process that is being defined. Although in the Knowledge model the risks management tool is associated only to the Risk Analysis activity, that allocation becomes possible. This is a consequence of the performed inference using the *resource rule* (Figure 13) and of the fact that the Knowledge model defines Risk Analysis as a potential subactivity of Planning.

```
/** Gets the suggestion of resources for the argument activity. */
public List getSuggestedResources(Activity activity) {
  Variable var = new Variable("R");
  Object[] terms = new Object[] {var, activity.getKnowledge()};
  List lstKResources = knowledgeBase.infer("use",terms);
  return lstKResources;
}
```

Figure 14. Code Example using the Inference Layer.

4.2 Process Control in ODE

Once the project process is defined and the project initiated, the environment should support project controlling and tracking. This control includes various functionalities such as managing the states of the activities and controlling the access to the software tools in the environment, allowing the execution of the process activities.

During the definition of a project process, the types of necessary resource for accomplishing an activity are defined. For instance, a Software Engineer (instance of the class `KHumanResource`) can be assigned to execute the activity of Requirement Analysis. Only in a second moment, during resource allocation, a person is allocated, for instance, Maria (an instance of the class `HumanResource` of the *Control* package), that naturally should exercise Software Engineer's function. In a similar way, software tools are allocated to activities. For instance, once defined the need for a Modeling Tool to execute the activity of Requirements Analysis, there should be chosen which modeling tool available in the organization will really be used to support this activity.

Based on the process activities, and human resources and software tools allocated to them, the environment is able to configure itself according to the user logged on. In other words, starting from the identification of the human resource that is using the environment, the functionalities are made available in a personalized way, according to the user profile and the resource allocation to projects.

The availability of the tools of the environment is defined based on the project resource allocation and the states of the activities. In a project, a human resource should have access only to the necessary tools to perform his/her activities. Therefore, only those tools allocated to the activities to which the user is also allocated are shown in the menu. In addition, the activities should be in execution. Figure 15 shows this self configuration feature of ODE. It shows ODE's main interface when a Project Manager is logged on the environment. In the tool menu, there are links for tools that support allocation of tools, quality planning (ControlQ), risks analysis, documentation, etc. Those tools constitute the whole set of tools that the user can access, considering the active activities of the process (Planning) to which he/she is allocated.

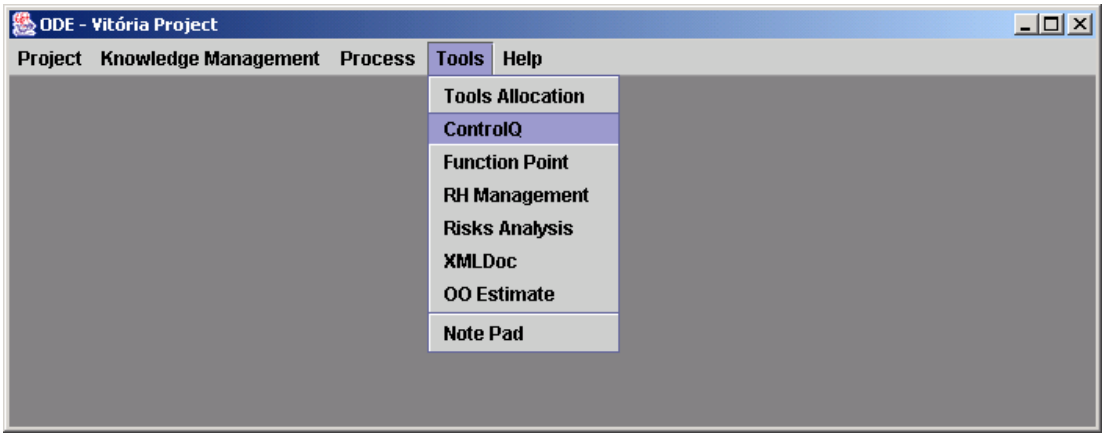

Figure 15. The Configured Environment.

The selection of tools that will be shown in ODE's tools menu is performed using the configuration knowledge base of the environment, which holds the *use rule*, shown in the figure 16, where ido1_131 represents the state "In Execution".

```
% The human resource HR can access the software tool ST if both are allocated
to the same activity X and the state of X is In Execution.
access(HR,SF) :- humanResource(HR), softwareTool(ST), allocate(X,HR),
                 allocate(X,ST), activity_state(X, oid1_131).
```

Figure 16. Rule of the Configuration Knowledge Base

In that way, when a user logs on the environment, the tool menu is dynamically configured as a result of inferences using the method shown in the figure 17. In this case, the Inference Layer is also supporting process integration in ODE, allowing the configuration of ODE's tool menu. This approach makes integration easier, more efficient and more flexible, since the used rule can be easily changed.

```
/** Gets the software tool available to the argument human resource. */
public List getAvailableTools(HumanResource humanResource) {
  Variable var = new Variable("ST");
  Object[] terms = new Object[] {humanResource, var};
  List lstTools = knowledgeBase.infer("use", terms);
  return lstTools;
}
```

Figure 17. Code Fragment of the Environment Configuration

Suppose that another approach for menu configuration is considered better. In this approach, the user might have access to all the tools allocated to activities to which he/she is allocated, independently of the state of the activity. Therefore, the user could access tools allocated to future activities and he/she can review his/her work using tools allocated to activities already finished. This change in the operation mode of the environment is fully supported by the Inference Layer, without rewriting ODE's code. In this case, the Knowledge Manager should only modify the use rule, removing its last condition and, later, instantiating the knowledge base again. In this way, it is not necessary to change or to recompile the code of the environment.

5. RELATED WORK

TABA Workstation [14] is a meta-environment that generates software engineering environments suitable to specific projects, based on the software processes defined. As ODE, TABA Workstation has tools that support process definition and management. Process integration in TABA is considered when, from a defined process, a SEE is instantiated according to the process assets. Like TABA, ODE supports process definition and management; however, the environment configures itself dynamically from human resource and tool allocation to the current project's process activities. It is worthwhile to point out that ODE is a configurable environment, while TABA is a meta-environment. Thus, ODE supports access to several projects in the same session, while the TABA's instantiated environments are project specific.

Prolog is also used in TABA to represent knowledge modules [1, 15]. It is used to formalize knowledge descriptions, transcribing ontology's axioms for a logical representation. TABA's and ODE's approaches are quite similar as to encapsulation and communication with Prolog engine. However, the concept of Knowledge Base Structures does not exist in TABA, neither supporting for rule edition, letting this task for the knowledge engineer do directly in Prolog. Furthermore, there is no functionality to support checking the edited knowledge bases.

Charon tool [15] provides support for software process modeling, simulation, execution and management. In this tool, facts are represented in knowledge bases and rules are incorporated into intelligent agents. An inference engine is used for the agents do queries and modifications in the knowledge bases. Thus, the facts are determined by the application and the modification of the rules consists in modification or inclusion of agents. There is no flexible way for maintaining facts or rules by the user.

Other process centered environments, such as EPOS [16] and Oz [17], use rules for modeling processes. However, the majority of these approaches use logic for software development specific purposes, like process modeling. They are not worried about providing an infrastructure to represent and manipulate knowledge bases in logic. In other words, such approaches do not provide automated support for editing predicates and rules, and generally, work with rules embedded in the application code.

6. CONCLUSIONS

This paper presented the knowledge-based approach for process integration in ODE, a process-centered environment. It presented ODE's tool supporting software process definition, as well as others knowledge-based functionalities for process control and environment configuration.

There is a strong connection between the tool of software process definition and the process ontology in which the environment is grounded. The use of inferences made possible to the tool to work in accordance with the ontology, respecting its relations and axioms. The situations discussed in the paper are part of a group of operations that approximate the tool to its conceptual model. Many times, performing those types of operations in the conventional way is extremely hard. However, the use of logical deductions made it possible to implement these operations in an easier way. This approach also focuses on the maintainability and extensibility of the environment.

The approach of leveled software process definition can lead to a continuous software process improvement based on the experience captured and projects' information that use the standard/specialized processes as basis. In this context, inferences can be used to support the retrieval and dissemination of historical data, obtained from previous projects, offering subsidies for a continuous process improvement.

7. ACKNOWLEDGMENTS

The authors acknowledge CAPES and CNPq for the financial support to this work.

8. REFERENCES

[1] Falbo, R.A., Menezes, C.S. and Rocha, A.R.; "Using Ontologies to Improve Knowledge Integration in Software Engineering Environments", Proceedings of the SCI'98/ISAS'98, Orlando, USA, 1998.

[2] Fuggetta, A. *Software Process: A Roadmap*, In: Proc. of The Future of Software Engineering, ICSE'2000, Limerick, Ireland, 2000.

[3] Pfleeger, S.L., *Software Engineering: Theory and Practice*, 2nd Edition, New Jersey: Prentice Hall, 2001.

[4] Falbo, R.A., Natali, A.C.C., Mian, P.G., Bertollo, G., Ruy, F.B. "ODE: Ontology-based software Development Environment", IX Congreso Argentino de Ciencias de la Computación, La Plata, Argentina, 2003, pp 931-940.

[5] Pressman, R.S., *Software Engineering: A Practitioner's Approach*, 5th edition, 2001.

[6] W. Harrison, H. Ossher, P. Tarr, *Software Engineering Tools and Environments: A Roadmap*, in Proc. of The Future of Software Engineering, ICSE'2000, Limerick, Ireland, 2000.

[7] Christie, A.M., *Software process automation: the technology and its adoption*. Pittsburgh, 1995.

[8] T. Gruber, *Towards principles for the design of ontologies used for knowledge sharing*, Int. J. Human-Computer Studies, 43(5/6), 1995.

[9] Rasmus, D.W. *Ruling Classes: The heart of knowledge-based systems*. Object Magazine, July 1995.

[10] Taylor, D. *Building intelligent objects.* Object Magazine, July/1995.

[11] DEIS (Dipartimento di Elettronica, Informatica e Sistemistica), Università di Bologna a Cesena, Italy. *tuProlog Developer's Guide*. Available in <http://lia.deis.unibo.it/research/tuprolog>. Accessed in March/2003.

[12] Araribóia, G. *Logic for Computation – An Introductory Study*. 1st. edition. Niterói: ILTC, 1989 (in Portuguese).

[13] Bertollo, G.; Falbo, R.A. *Automated Support for a Leveled Approach to Software Process Definition*. Proceedings of the II Brazilian Symposium on Software Quality, Fortaleza, Brazil, September, 2003 (in Portuguese).

[14] Falbo R.A., Menezes, C.S., Rocha, A.R.C. *Assist-Pro: a Knowledge-based Assistant to Support Software Process Definition*. Proceedings of the XIII Brazilian Symposium on Software Engineering, Florianopolis, Brazil, October 1999 (in Portuguese).

[15] Murta, L.G.P. *Charon: An Agent Based Extensible Process Engine*. Master Thesis, COPPE/UFRJ, Rio de Janeiro, March/2002 (in Portuguese).

[16] Jaccheri, M., Conradi, R., *Techniques for Process Model Evolution in EPOS*. IEEE Transactions on Software Engineering, v. 19, n. 12, December 1993.

[17] Ben-Shaul, I.Z., Kaiser, G.E., *Federating Process-Centered Environments: The Oz Experience. Automated Software* Engineering, vol. 5, n. 1, January 1998.