Towards an Integration of Software Product and Software Process Modeling

Noureddine Belkhatir
University of Grenoble
Laboratoire de Genie Informatique
BP 53
38041 Grenoble France
e-mail: belkhatir@imag.fr

Walcélio L. Melo
University of Brasilia
Departamento de Ciencia da Computacao
Campus Universitario, Asa Norte
70910-900 Brasilia, DF, Brasil
e-mail: melo@cic.unb.br

April 5, 1995

Abstract

This paper discusses the integration of software product and software process modeling and management in the context of "programming-in-the large" software engineering environments. We analyze the principal concepts developed to give a formal description of software products and how such products are manufactured in such environments.

Focusing on software products, we show the earliest attempts to give formal representations of the application of data modeling concepts with respect to software engineering requirements. This study has been carried out primarily in the context of software configuration management systems.

Focusing on the software processes, we highlight recent efforts in the development of programming languages to describe the software processes. We conclude by presenting the ADELE/TEMPO project, which is a software environment combining software product with software process modeling and configuration management with software process control.

Keywords: Programming-in-the-small, programming-in-the-large, software product modeling, process modeling, software engineering database, software engineering environment, process-oriented software engineering environment.

1 Introduction

The development and maintenance of "large" software systems is a complex task requiring good coordination among cooperative development activities. This requires efficient management of documents (e.g., specifications, programs, tests, etc.), of software performers (e.g., managers, programmers, designers, etc), of tools, and of software engineering techniques and methods.

The software crisis demonstrates that the improvement of software production process is a fundamental

condition to the improvement of software product quality. Works such as [Notkins, 1985] claim the necessity of software tools and integrated software development environments which can support, control and, in some cases, automate the large number of activities carried out by the software engineers during the software production process.

This paper presents a synthesis of the work on support systems in software engineering. We focus particularly in this paper on the integration of software product management with software process control in the framework of programming-in-the-large software engineering environments.

The paper is organized as follows:

1. We first highlight the main characteristics of the information manipulated in programming-in-the-large software development environments.
2. We discuss the two main conceptual components of a software engineering environment (SEE) for supporting programming-in-the-large information and activities, i.e., the the software product manager and the software process manager.
3. We then present the evolution of programming-in-the-large SEE towards an integration of product and process management, citing works in the field which present tangible results.
4. We conclude with the presentation of a software engineering environment (SEE) called Adele/Tempo, which is a process-oriented software engineering environment.

2 Information in programming-in-the-large

The software development process (hereafter called software process) is a set of activities performed in order to produce high-quality software systems. To support the software processes we need to be able to handle two main kinds of information: first, information about the involved resources and the produced software products, and second, knowledge about the description and the structuring of software activities. We discuss in the following sections the nature of these two types of information.

2.1 Resources and software product description

This portion of the information pool includes the modeling and management of software performers, as well as the software products and components (design, documentation, set of tests, programs, etc.), with their relationships, their use, and their evolution in versions. There is also more extensive information also concerning how software components can be organized in complex and composite structures, i.e., software systems. In section 4, we review the systems devoted to configuration management showing their evolution towards software engineering databases.

2.2 Activity description

This type of information (behavior model, activity control) includes descriptions of how to develop and maintain software products, how to share and synchronize the activities among users, how to integrate outside tools and control their use, and finally, how to describe and enforce policies (methods, procedures, conventions, etc.). Traditionally, this knowledge has been hard-coded in an environment which provides a fixed software process model. Recently, this issue has received attention from the software process community [Dowson, 1991; Osterweil, 1993]. The objective has been to allow software organizations to specify their own software process model by providing a software process description language. In section 5, we review the systems devoted to software process modeling.

3 Programming-in-the-large SEE

During the last decade, projects on SEE's (Software Engineering Environments) pointed out the importance of data integration and the important role of an object management system as a kernel of SEE's. Recently POSE's (Process-Oriented Software Engineering Environments) have been investigated as a new architecture of SEE's in which the software process is made explicit by a software process program. This program is used to drive the user interactions in the software development environment. A POSE is, in general, composed of two components: a software product manager and a software process manager. The software product manager is responsible for the management and control of all software products manipulated during the software process execution. The software process manager is the component supporting an explicit formalism to describe the software process. The software product manager uses an object-oriented database for controlling the allocations of software components and other software resources to the software processes.

The software process manager is in charge of the following problems:

- Large software systems are developed by several kind of software performers, each one has a different kind of competence (software engineers, software quality engineers, etc), carrying out cooperative activities at the same time. Thus a software process manager must provide ways to model their responsibilities and control their actions. A software process manager must also aid the scheduling of software activities that depend on human resources involved in the software processes.

- Empirical studies have shown that most of the effort of software performers is spent on coordination and communications tasks. An agent can use the competence and results of other agents during the cooperation process. The cooperation activity allows the sharing of information. Thus, we need to:

  - Describe cooperation policies as well as enforce these policies as automatically as possible.

  - Support activity communication.

  - Monitor the utilization of resources by the parallel activities;

- Software engineering activities are characterized as time consuming. A software process manager must provide working spaces in which such activities can be carried out in insolation during long periods of time, e.g. days, weeks, and even months. Each working space can be populated with different sets of software products depending on the task to be performed. The software activities performed inside a working space can change depending on the software development strategy provided for it.

- Each software organization has its own way of developing and maintaining their software systems. Thus a SEE must be driven by the software processes. To make this possible, the software processes must be formally described in a software process program . The software processes enaction must be controlled by the SEE in according to the description done in such program.

- Change is inherent to software processes. We must provide mechanisms for supporting the evolution of software processes.

On the other hand, the software product manager is in charge of the following problems:

- Programming-in-the-large is concerned with the development of large software systems. The size of such systems forces us to decompose them into more manageable units that we call "*modules*". The modular break-down of large software systems aids us in better understanding and maintaining them. A software product manager must provide concepts to reflect the hierarchical structure of large software system. It must also provide mechanisms making it possible to store and manipulate large-granularity software products or components.

- These kinds of software systems have a long life, in general 5 to 10 years or more. A software product manager must control their evolution through time and control their versions.

- These kinds of software systems are difficult to build because many different tools are necessary. The inter-relationship of software components makes the build process complex. So a software product manager has to provide automatic support for build tasks.

4 Resource and product management in software engineering environments

Two major phases have characterized the evolution of software product management in the framework of software engineering environments (SEE's) for these last twenty years. The first includes software configuration management systems built around file management systems. The second is SEE's built around databases.

4.1 Software configuration management systems

[Tichy, 1985] defines configuration management (CM) as the discipline of controlling the evolution of complex software systems, and software configuration management (SCM) as its specialization for computer programs and associated documents. Several software configuration management systems have been proposed in order to control modifications within sets of software components. However, each system has a specific representation and a specific functionality to be used for software process control.

Important challenges facing software development and maintenance include the problems of version control and tools control. Version control allows us to keep any software system consisting of many versions and configurations well organized. Construction of a complex software system requires that the application of tools is controlled. Tools have to be automatically activated in a precise order to avoid the production of ridiculous results or useless re-work. The application of a tool to a component of a software system can trigger the execution of other tools in two ways: (1) results produced by application of a tool must be used by other tools or (2) the modification of a software component can trigger the modification of other software components. This tool execution process can be propagated until the whole software system is built up. Version management, revision management and tools control mechanisms have been introduced by the development of configuration management systems such as RCS [Tichy, 1985], SCCS [Rochkind, 1974] or MAKE [Feldman, 1979].

SCCS and RCS are principal examples of version control systems. They provide good support for keeping track of the evolution of versions of software systems. SCCS introduces the revision concept and the delta mechanism to manage software component modifications. RCS has the same functionalities as does SCCS, but with some improvements. It manages a set of revisions of the source program, the documentation, and the test data, while providing selection mechanisms for composing configurations. It is based on a general model for describing multi-version/multi-configuration systems. DSEE [Leblang and Chase, 1985] is another system which consistently integrates version control and configuration management. It has a rich set of facilities for version management, and configurations are built using "configuration threads".

MAKE, developed in 1976, is the first system permitting tools control through automatic construction of a software configuration based on the analysis of the software components' dependencies. Construction is done by automatic tool activation. The construction process is described in a special file called "makefile", which is maintained by the user. However, these tools provide only marginal support both for understanding the dependencies of large software systems composed of several modules and also for keeping track of relations among documents, source code, and test cases. Some research efforts have attempted to integrate a version control system with a configuration management system. MAKE in combination with RCS has been proposed; the revisions selected by RCS are passed along to MAKE for configuration building.

Some highly specialized configuration management tools are built into SEE. They include NSE [Miller, 1989], GANDALF [Habermann and Notkins, 1986] and ADELE 1 [Estublier et al., 1984]. As in DSEE and JASMINE [Marzullo and Wiebe, 1986], GANDALF [Habermann and Notkins, 1986] uses the concept of the *system model*. This concept allows us to describe the relations among components, the information on versions, and the software construction rules with a Module Interconnection Language (MIL) [Perry, 1987]. Rather than using the system model to define the architecture of a large software system, the ADELE 1 configuration manager uses a module dependence graph and the expression of multi-version system composition by constraints to derive a configuration.

These systems use files as a repository and are managed by classical file systems management. Furthermore, the management of software activities remains at a low level (build tasks).

4.2 Software engineering databases

Software engineering databases are intended to support all the data management needs of software environments, including program compilations, software versions, software configurations, and nested and long transactions. Researchers have attempted to design an appropriate data model to support the entire software life cycle of information. The concepts developed above, like *system model*, are easily modeled by objects and relations. Classical databases provide these concepts, hence the interest in database technology. This interest has been shown primarily in the use of relational technology, which has been proven inappropriate, and which has been followed by the use of data models allowing more semantic expressions.

4.2.1 Relational model

In these models, unlike the *system model* approach, object types, relationship types and constraints are not fixed but are explicitly expressed. The experimentation conducted on relational database management systems has shown that they are not adapted to support SEE's [Penedo, 1986; Bernstein, 1987]. These systems were not satisfactory, for example, when managing files (long fields), revisions, system build activities, and data schema evolution. Furthermore, relational systems, which have been developed essentially for commercial applications, present the following drawbacks:

- It is difficult to model complex objects. Relational databases have an unstructured view of the objects.
- Lack of encapsulation. The operations applicable to an object are not identified; their semantics are completely left to the realm of applications.
- It is difficult to represent complex constraints and software structure.

- In general, there is a lack of mechanisms for dealing with dynamic aspects (events, constraints, triggers) allowing the management of complex applications.

4.2.2 Object-Relation Model = O.O + E-R data models

The database community has been interested in the application of research to domains such as software engineering, office information systems, and CAD/CAM; this interest has led to the development of semantic and object-oriented models which provide better expression for modeling software products and software systems.

Semantic models were developed to provide a higher level of abstraction for modeling data and to provide powerful mechanisms for representing structural aspects of software products. [Peckhan and Maryanski, 1988] summarize the motivations often cited in the literature in support of semantic data models over the traditional data models (especially the relational data model). The most frequently used semantic data model is the ER model (Entity-Relationship) [Chen, 1976] extended with the notion of object identity. This data model allows the description of conceptual aspects of entities and associations between them.

Object-oriented data models allow the description of behavioral aspects, which are extremely important in several domains including software engineering, multi-media databases, computer-aided design (CAD), etc. Object-orientation provides better paradigms for constructing reusable software components encouraging an incremental design (inheritance mechanism). The provided encapsulation concept allows us to minimize the interdependency among separate software components and thus facilitates software evolution and maintenance [Snyder, 1986]. Objects communicate through their interfaces, hiding their internal organization and thus providing data abstraction. The requirements of new application domains for databases, such as office information systems, computer-aided design, and software engineering, have resulted in a particular interest in object paradigms. Object-oriented databases become the ideal repository of the information that is shared by multiple users, multiple products, and multiple applications on different platforms.

The software engineering community has been interested in these results [Hudson and King, 1989; Dittrich et al., 1987] and has responded with a new generation of SEE's. The new developments include specialized databases which:

- Efficiently integrate "object-oriented" and "semantic" data models essentially the entity-relationship data model specially tailored for modeling software components and products. The first is valued for the dynamic aspects of its modeling power. The second is valued for its data structuring features. The "Object-relationship" model is the symbiosis of these two cited data models. The "Object-relationship" model makes it possible (1) to define explicitly the relationships at the conceptual level and (2) to include structuring properties and propagation properties for modification effects.

- Properly include management services, making it possible to control the evolution of software components in versions and their composition in configurations [Zdonik, 1987]. ADELE 2 [Belkhatir and Melo, 1994a] and PCTE+ interface [Boudier et al., 1988] follow this approach.

ADELE 2

Systems such as ADELE 2 [Belkhatir and Melo, 1994a] are essentially concerned with software product modeling. ADELE 2 provides functionality to support software component evolution and multiple version configurations. Different mechanisms are also provided to allow the user to specify component properties in terms of attributes, and to consistently control the software product structure. ADELE 2 is similar to GANDALF and DSEE, but also incorporates new concepts derived from database technology.

We list below the principal characteristics of ADELE 2:

- It allows the development of modular systems. A module in ADELE 2 is composed of two distinct parts: an interface and a realization.
- A family, which consists of the module concept extended to the versions, is equivalent to the modules of GANDALF but includes more interface versioning.
- ADELE 2 supports constraints on elaborated version compositions. Configuration composition is extended by the constraints on component characteristics (attributes).
- The description of configuration components is deduced from the dependency graph and is not predefined as in CEDAR, GANDALF, or DSEE.
- ADELE 2 uses predefined types to define the family-base objects [Belkhatir and Estublier, 1987]. These predefined types allow the software performers to invoke tools implicitly.
- ADELE 2 provides the concept of data schema making it possible to specialize the pre-defined ADELE 2 software object types with new ones. Thus, depending on the software system, new object types can be created and new relationship types can be defined to link objects belonging to such object types.

Pcte+

The use of recent database technologies is best illustrated by the object management system of PCTE+ [Oquendo et al., 1991]. The goal of this system is to provide a common repository for the development of the software engineering environment in Europe.

PCTE+ [Boudier et al., 1988] includes several noteworthy features:

- a transparent management of the object base,

- distribution over a set of work stations,
- integrity control of the object base,
- concurrency control of objects,
- version and configuration management.

PCTE+ uses a power data model which is derived from the Entity-Relationship model. The PCTE+ data model is organized in a set of SDS's (Schema Definition Sets), each one describing a sub-schema of the overall database schema. Users can create a *"Working Schema"* composed of a list of SDS in order to have their own database view. One important drawback of Pcte+ is the lack of an encapsulation mechanism allowing methods to be associated to objects.

4.2.3 Synthesis

PCTE+ and ADELE 2 are representative of actual investigations in the conception of an object database able to model software products as well as support the evolution of such products. PCTE+ is a European standard for which a prototype has been developed, and ADELE 2 is an industrial quality software product.

The integration mechanisms provided by these systems allows the inclusion of tools comprising software product life-cycle, including revision control tools, automatic building tools, configuration management tools, etc. However, the software process management aspects (e.g., tools execution model, users model, tasks model, process chaining, method management) are not taken into account.

Real SEE's must integrate software product and software process aspects. At the base of such environments we should find an active software engineering database. Such database should be composed with a software product manager (structural and functional of the product and its evolution) and a software process manager (grouping the information on the development environment, tools and methods). Systems as MARVEL [Kaiser et al., 1990] and ALF/PCTE [Derniame et al., 1992] highlight these aspects and make clear the advantages of explicitly modeling the software processes.

5 Software processes modelling

The set of all software engineering activities, encompassing the entire software life cycle from the customer's requirements to maintenance, is termed the *software process*. A software process programming language is the way to describe how the software processes must be undertaken according to a software process model. A software process programming language makes it possible to:

- describe different activities carried out during software process execution;

- control ordering, synchronization and concurrency among activities;
- model the software components produced and consumed by activities;
- integrate tools used to perform activities;
- define and control software performers involved in the activities.

Several process-oriented SEE's have been built to support these dynamic aspects. The efficiency of these systems differs at several points depending on the software process modeling paradigm used.

5.1 The rule-based paradigm

In this approach, knowledge about the activities of a generic software development process is explicitly modeled by rules. *"Artificial Intelligence"* techniques and active databases are used, for instance planning and blackboard techniques or trigger mechanisms. The tools are integrated into the environment through the use of pre- and postconditions over their inputs and outputs. The rules may differ depending on the implementation chosen by the system (backward and/or forward reasoning, static or dynamic planning, hierarchic and sequential/parallel planning, Event-Condition-Action rules). This approach is used primarily for high-level tasks and is employed by MARVEL [Kaiser et al., 1988b], MERLIN [Peuschel et al., 1992] and ADELE 2 [Belkhatir and Melo, 1994a].

MARVEL is one of the earliest Process-Oriented SEE's based on production rules to allow the modeling and control of software processes. A rule is composed of: (1) a precondition and a postcondition that are defined in terms of attributes of objects in the object base, and (2) an operator used to integrate tools. MARVEL manages rule execution by backward and forward reasoning. The application of these two mechanisms together is called *opportunistic processing*. The main innovation of MARVEL is to extend the mechanism of the Schema Definition Set (SDS) proposed by PCTE with the concept of strategy [Kaiser et al., 1988a]. A strategy is a set of definitions (rules, tools, object types, and relationship types) which can be imported or exported.

MERLIN provides two types of production rules to represent software process with different execution mechanisms. One rule type is used to model the declarative part of process definition. Rules of this type are executed by backward reasoning. Other rule types are used to model procedural knowledge; they describe tool activation ordering by post-conditions. These kinds of rules are executed by forward reasoning. Such rules reduce the inconsistency problems of Marvel's opportunistic processing.

ADELE 2 [Belkhatir and Melo, 1994a] uses the concept of Event-Condition-Action (ECA) rules when modelling the development activity. Whenever an event occurs, an action is executed if some condition is satisfied.

5.2 Procedural paradigm

This approach to software process modeling proposed in [Osterweil, 1987] is such that the complete software process is defined as a meta-program. It is described by means of a formal language, which is written by the environment administrator before the activation of the process. This description is considered to be a specification of how the software development processes are to be conducted. ARCADIA [Taylor and *et al.*, 1988] and TRIAD [Ramanathan and Sarkar, 1988] are examples of this approach. Both systems have extended the ADA language with new capabilities for supporting software processes.

The ARCADIA project aims at building a process programming environment based on a prototype ADA-like process programming description language, called APPL/A [Sutton et al., 1990], and supported by an object-base [Penedo, 1991]. The principal extensions of APPL/A over ADA are (1) relations among software components, (2) trigger upon relation operations, (3) integrity semantic constraints on relations, and (3) some transaction constructs. Software derivation tasks are embedded in relation definitions and are automatically executed after software change. Triggers are used to propagate updates on relations.

TRIAD is another research prototype system heavily influenced by ARCADIA ideas. Software development policies are described by an imperative language called CML (Conceptual Modeling Language), which is also based on ADA [Sarkar and Venugopal, 1991]. CML is composed of: 1) an object-oriented semantic data model which allows trigger definition on data types; 2) a tool model; 3) a model describing the different role types played by the human software performers; and 4) an activity model. The activity model is used to describe the activity hierarchy and how each activity may be performed. CML provides primitives to synchronize parallel activities.

ARCADIA has a more advanced data model than does TRIAD, and semantic constraints may be more easily described and handled. However, the activity model of TRIAD is more suitable for describing software development policies than is that of ARCADIA because Triad explicitly models and executes activity synchronization and decomposition.

5.3 The Graph/Net paradigm

This approach uses a graph/net notation for software processes. The semantic model describes the syntax and semantics of the graph/net. Techniques such as Petri-Nets have been adapted and extended to model software processes, in terms of control flow and particularly in the aspect of concurrency. MELMAC [Deiters and Gruhn, 1990] is one of the software process management environments which introduces the Petri-Net concepts. MELMAC uses the FUNSOFT nets as the basis for executing software process models and for analyzing them. FUNSOFT nets are a high-level type of Petri nets whose semantics are defined in terms of Predicate/Transition nets. FUNSOFT nets consist of a Petri

net structure (where places called channels represent a product, transitions called agencies represent tasks, and edges represent I/O relationships between tasks and products). Agencies, channels, and edges can have attributes used to manage the net. Software process models are validated by simulation [Gruhn, 1991]. However, simulation results are not valid for all software processes.

5.4 Synthesis

No consensus has been reached as to which paradigms should be used and no single approach can satisfy all the needs of process modeling. Production rules cannot fully support versioning and CM, and the rule-based approach does not allow definition of the software process at a high level of abstraction. The main drawback of the process programming approach is that no algorithm of a particular software process can be described completely in advance. The Graph/Nets approach best illustrates the aspect of concurrency and allows a validation of software processes by simulation. However, such an approach not efficiently executable, and process customization and evolution is difficult. Current research includes multi-paradigm approaches such as “SPECIMEN”. SPECIMEN merges FUNSOFT nets with the MERLIN process modeling language based on rules. Another example is the EPOS system [Jaccheri & Conradi, 1993]. EPOS combines planning techniques for the static rule-based reasoning, the Graph/Nets paradigm for dynamic triggering of task networks, and the process programming paradigm when elaborating task types. However, Process-Oriented SEE’s always remain centered around one basic paradigm.

6 Towards an integration of product and process modeling

The integration of the software product and process aspects considerably increases the level of automatic assistance provided by a software engineering environment. Current architectures make use of the federated database approach, where each database acts as a development space or an activity domain dedicated to a software development step. The problem is to maintain the consistency of these different databases.

In this approach, software development is carried out at different workstations, each software performer having his/her own environment view in the form of a sub-object base, which is generally mono-version and is called a working base. Each working base can be dedicated to one software life cycle step (conception, development, test, etc.). Current approaches have a weakness coupling, which uses a multi-schema approach to describe all information about software products and processes, and support an object exchange protocol between the different databases with what is essentially the check-in/check-out model [Feiler, 1991].

Some projects try to integrate the software product and process modeling efficiently — for example ALF/PCTE [Legait et al., 1989; Derniame et al., 1992], ADELE/TEMPO [Belkhatir and Melo, 1994b;

Melo, 1993] and MARVEL [Kaiser et al., 1990]. ADELE/TEMPO uses the data model of ADELE 2 [Belkhatir and Melo, 1994a] to support complex objects and to integrate activity management aspects. Particular attention is given to synchronization and control of software activities. A prototype of ADELE/TEMPO, which integrates these parts, has been developed and experiments have carried out on the modeling of development environments such as NSE [Courington, 1989], ANDROMEDE [Chauvet, 1990] and PALAS [Reynier, 1988] to validate our approach.

SEAMAN [Tombros et. al., 1995], ALF/PCTE+ and ADELE/TEMPO are representative of the current trends in repositories of software process. The strength of these systems is their power model, which efficiently integrates software product database and process modeling and management. However, some aspects of the efficiency of these prototype systems remain to be evaluated.

7 Our approach: the Adele/Tempo system

As figure 1 reveals, ADELE/TEMPO consists of two basic parts:

1. ADELE 2 – a software product manager. The Adele database is used as a persistent object-base for storing software components and for tracing software project's progress. This data base is driven by an entity-relation-attribute data model incorporating object-oriented concepts, such as multiple inheritance, late binding, and polymorphism.

2. TEMPO – a software process manager [Melo, 1993]. Event-condition-action rules (ECA) and a trigger mechanism are called by TEMPO, making it possible to control software process execution. TEMPO also offers concepts for activity structuring and mechanisms to support cooperative work environments.

7.1 Modeling of software products

The ADELE 2 [Belkhatir and Melo, 1994a] data model is derived from an entity-association model and integrates object-oriented concepts. The basic entities of the model are the object type and the relationship type. Each entity (object and relationship) possesses static (attributes) and dynamic (methods, event-condition-action temporal rules) properties.

The data model supports complex objects referred to as aggregates. An aggregate is an object linked to its components by relationships. For example, a PASCAL module can consist of an interface and an implementation. A PASCAL module can consequently be represented as an object linked to two other objects by two types of relationships, possesses-interface and possesses-implementation. Aggregate semantics are defined by the dynamic properties of the relationship linking the aggregate to its components. The semantics are defined by the user; any aggregate can thus be defined by the use of its own semantics and consistency constraints.

The atomics objects in ADELE 2 are the **elements**. An **element** is an object not decomposable and which is usually associated with a file. There are two kinds of elements: **source elements** and **derived elements**. The source elements are normally produced by human beings while the derived elements are produced by an automatic tool (compiler, linker, etc.) from the source elements, perhaps in combination with other elements.

The elements are always parts of more complex structures; they are thus termed dependent objects. There are two kinds of complex structures: **revision set objects** and **composite objects**.

A **revision set** models the evolution of a source element and its derived elements. It is a sequence of revisions where each revision contains an evolution of the source element and possibly its derived elements.

An object type is a definition in intention of a set of objects with common characteristics. An object type is described (user-defined) by (1) a name, (2) a domain which is a predicate that expresses the integrity constraint of the composition, (3) a set of attributes, (4) and consistency constraints and actions (methods).

We use the term of **composite object** to denote an object composed of different revision sets and other composite objects. Examples of composite objects are the **family**, the **interface** and the **realization** objects.

A **family** is composed of:

1. a set of **interfaces**,
2. a revision set called a *main* (default),
3. other revision sets.

```
TYPEOBJECT family IS Cobj ;
   STRUCT
      main : Rset; -- Revision set  specifying module function
      * : list_of Rset ;
      * : list_of interface ;-- Interface  variants of this family  (module)
END family ;
```

An **Interface** is a composite object dependent on a unique family and composed of:

1. a set of **realizations**,
2. a revision set called a *main* (default),
3. and others revision sets.

```
TYPEOBJECT interface IS Cobj ;
   STRUCT
      main : view ; -- view is a revision set containing part of the interface

      * : list_of Rset; -- other views of specifications, manuals, test sets, etc

      * : list_of realization ; -- Body variants that implement interface views

END interface ;
```

A **realization** is a composite object dependent on a unique interface and composed of:

1. a revision set called a *main* (default),
2. and other revision sets.

```
TYPEOBJECT realization IS Cobj ;
   STRUCT
      main : prog ; -- prog is a revisions set including a program body
      * : list_of Rset ;-- other revision sets: specifications, manuals, test sets...
END realization ;
```

Each revision must fit the characteristics of the variants to which it belongs; consequently, all revisions would share the same attributes and relations. We call an object's characteristics its attributes and its relationships. The root object **family** is linked with its interface objects by the **has_interface** relationship, and each interface object is linked with its realization objects by the **has_realization** relationship.

7.1.1 Relations

Any manipulated object can have relations with any other object. A relation type is defined by its domain (source-destination), its attributes, and some characteristics such as the cardinality or the DAG characteristic which indicates whether or not a relation must be acyclic. Some relations are predefined and others are built-in: for example, when an interface I of a family F is created, I is automatically bound to F by the relationship "`has_interface`". These relations are managed by the ADELE database while other relations must be set and managed by users.

7.2 Modeling software processes

The TEMPO process programming language is used to describe software processes. The TEMPO process manager is in charge of interpreting the software policies described in a software process program. Consequently TEMPO is in charge of controlling the execution of software processes. A software process

model of considerable size can thus be written by using various software process types. A software process type can aggregate other software process types.

For example, an activity to check a module design document comprises two sub-processes:

1. The modification activity that makes changes to the design document.
2. The revision activity that approves design document modifications that have been made.

```
MonitorDesign ISA PROCESS;
       CONTROL md;
                sub = ModifyDesign;
       CONTROL rd;
                sub = ReviewDesign;
END_OF MonitorDesign;
ModifyDesign ISA PROCESS;
       ATTRIBUTES
                begin_date = DATE := now();
                end_date = DATE;
                deadline = DATE;
        METHODS . . .
        RULES  . . .
END_OF ModifyDesign;

ReviewDesign ISA PROCESS; ...
```

The example above shows the software process type `MonitorDesign`, composed of the sub-processes `ModifyDesign` and `ReviewDesign`. The activity coordinating the module design document modification is represented by the `MonitorDesign` type. `ModifyDesign` is the type which describes the design document modification process, and `ReviewDesign` is for revising this modification. It is possible, for every process type, to define attributes, methods, and event-condition-action rules.

7.2.1 The temporal constraints

Constraints are described by temporal-event-condition-action (TECA) rules. TECA rules are similar to ALF [Derniame et al., 1992], DAMOKLES [Dittrich et al., 1987], and HIPAC [Mccarthy and Dayal, 1989] trigger rules.

A TECA rule that goes like this:

"WHEN *event* Do *Method*"

where:

event is a predicate expressing an event about the present or past state of the system or about the object base.

method is a method.

Example:

```
EVENT delete_sensible = (!cmd == remove AND
                        (!object\comp/state == released OR
                         !object@(status == validated)); PRIORITY 5
```

This line expresses that event "`delete_sensible`" will be true whenever there is an attempt to delete a component (`!cmd == remove`), which is either a component of a released configuration (`!object/comp/state == released`) or which has been in the past the status validated (`!object@(status == validated)`). The expression "`!object`" represent the name of the object receiving method "`!cmd`". Similarly all parameters of the called method can be checked, as well as previous values of attributes and object when changed by the methods.

We added the operator "@" in the expression defining the event in order to be able to lay conditions on the past. This operator is interpreted in relation to the log of object evolution. All updates performed on an object is stored in this log (changing of attributes and events). Temporal constraints are checked following a reverse scanning of the history from the triggering of the event to the satisfaction of the Temporal constraint. These constraints are expressed in relation to object properties (attributes and events stored in the objects log). If Temporal constraints are not checked at any time at all, then no operation will be executed.

In the example showned in figure 2, when event e4 occurs, the rule;

```
WHEN (e4 and  @(e1)) DO method-X;
```

is triggered. The object history on which event e4 occurred is scanned to check if the event e1 occurred previously (the "`@`" constructor). The `method-X` is executed if event `e1` has already been recorded in the object history. Even if the history holds other information that might change the execution context of the rule (late-binding of information), the scanning process of the history stops when the expression given in the "`@`" constructor is met. For example, with the previously defined rule, "`@`" constructor will only be satisfied when the scanning process of the history meets the last `e1` event (see figure 3)

For example:

```
ModifyDesign ISA PROCESS;
        ATTRIBUTES
                begin_date = DATE := now();
                end_date = DATE;
                deadline = DATE;
        METHODS
```

```
                continue_execution;
                . . .
         RULES
(1)             AFTER WHEN deadline_arrived
                      DO stop_execution;
(2)             PRE WHEN (continue_execution AND @(not deadline_changed))
                    DO ABORT;
END_OF ModifyDesign;
```

1. The rule described in line 1 specifies that the design document modification activity must stop when the date foreseen has been reached.

2. The rule in line 2 states that resumption of the activity if has not been completed yet first requires that the termination date be changed.

The methods

A method is program written in a simple imperative language similar to UNIX'S

```
   METHOD delete ;
      IF [%state == stable] THEN ABORT
      ELSE "rmobj %name ";
   END delete;
```

This method enables to suppress objects with unstable states. The late-binding mechanism is used during the execution of methods. In the above example, when the method "delete" is executed, the variable *"%name"* takes the identifier of the object being deleted as value.

TECA rules and short transactions

Triggers are similar to production rules since they define the dynamic behavior of all the objects of a given type: the encapsulation principle is respected.

The actions associated with triggers fall into one of the following categories:

Pre actions. Before the execution of an operation on an object of type T, an event occurs and the triggers defined in the type T as pre actions are executed (those for which "event" in "ON event DO Action" is true). This kind of action allows testing of preconditions and command extensions.

Post actions. After the execution of the operation but before its commit, triggers in post-action are executed. These triggers can analyze the consequences in the database and, since they are executed inside the transaction they can undo (rollback) the operation. They can also extend the command by performing other computations.

After actions. After an operation is committed, other triggers are executed. These actions make it possible to modify the database after the command (for instance asserting new states).

Abort action. If the operation fails or aborts, all actions including those performed by pre- and post-triggers are undone, then abort actions are executed. This mode allows execution of actions in response to abnormal behavior.

Pre triggers and post triggers must succeed for an activity instance to be allowed to start and commit.

7.2.2 Examples of utilization of TECA rules

Definition of TECA rules into data model

Figure 4 presents an example of use of TECA rules in the data model. TECA rules in the data model are used to describe constraints about the manipulation of software components and software products. Such rules are independent of the context where and when software components are handled. In order words, TECA rules when defined in the data model are useful for description of (1) integrity constraints about the relationships between software components, and (2) software policies which are context independent or invariants.

In this *body* type description we find in lines 1 the definition of attribute *lines* which represents the number of lines in the body. *Lines* is declared *COMP* which means the value provided at instantiation is not the attribute value but the program that, when executed, will return the real attribute value. In line 9 the value of line is the result of the execution by UNIX shell of *wc -l !filename* i.e the number of line in file *!filename*.

Line 2 is a pre-condition which specifies that if the event `delete_official` occurs, the command which triggered this event must be aborted. Event *delete_official* in defined line 12 occurs when the command `delete` is applied to an official body (i.e. an object body with attribute state equal to official). Line 3 expresses a post-condition on event `replace_body_c` defined in line 13. When the command `replace` is applied to a c program body (an object with the attribute language equal to c) this program must be compiled. If compilation is successful (line 9) the binary object is recorded with its source code (line 10) and the line numbers of the source object is computed and recorded (line 11).

The relation `comp` relates a configuration with its components. Before replacing a component of a configuration (line 5), the number of lines of the configuration (`!O` refers to the origin of the relation i.e. the configuration), is reduced by the number of line of the component (`!D` refers to the relation destination i.e. the replaced component, `!D%lines` is the value of attribute lines of the component); after the replace command (line 7), the actual number of line of the component is added to the number of lines of the configuration (line 8). That way, the number of line of all configurations is always up to date and recursively.

Description of TECA rules into process model

On the other hand, TECA rules when defined in the software process model they (1) describe fragments of software activities, (2) specify software policies which are context dependent, (3) define ordering of software activities, and (4) pre- and post-condition about the actions of user performers. For instance, figure 5 gives a fragment of software process definition, where:

1. The rule described in line 1 specifies that the design document modification activity must stop when the date foreseen has been reached.
2. The rule in line 2 states that resumption of the activity if has not been completed yet first requires that the termination date be changed.

8 Conclusion

We have addressed different evolution trends in SEE towards an integration of mechanisms for supporting software product and software process modeling and management. We believe that such an integration will represent a major step forward in the field through the production of Process-Oriented SEE (POSE).

We have shown that the current state of the technology is capable of providing either:

- Monolithic SEE's with embedded tools, services, and policies, like PACT [Thomas, 1989]; or
- General platforms supporting only data modeling, like PCTE [Boudier et.al, 1988].

We believe the next step will be to provide platforms integrating software product modeling with software process modeling, with an emphasis on tool integration. Different models are involved. Entity-Relationship and Object-Oriented models are required and need to be refined and adapted.

The ADELE/TEMPO project, on which we are working, is a contribution to the solution of these problems. On the one hand, we have developed a software engineering database to support product modeling, manipulation, and evolution aspects. Full-scale experiments have been conducted in industrial environments (Airbus, Hermes space shuttle, etc.). On the other hand, a software process modeling language has been implemented making it possible to describe software process activities. A prototype of a software process manager has been implemented above Adele database for supporting the execution of software process describe in such language. The execution of such activities is controlled by a trigger mechanism. Each process step occurrence aggregates a set of entities. Each entity is managed by the Adele database.

Future development and research goals include:

- Realization of an object type ("point and click"), user-friendly, graphic interface to enable users

to execute activities by means of graphic support.

- Management of software process evolution. Since software development has a long duration period, coordination and synchronization strategies can change during the course of execution. We thus need a mechanism by which these strategies can be changed without interrupting the execution of cooperating processes.

We believe that we offer a design context which helps to clarify the numerous complex coordination activities found within a SEE.

Acknowledgements

We would like to express our thanks to Carolyn Seaman, Barbara Swain, and to the referees for suggesting substantial and helpful revisions to the original text, to Rubby Cassalas for extending ADELE 2 data model with the inheritance mechanism, to Christophe Gadonna for maintaining almost alone ADELE 2, to Jean Chouanard for extending ADELE 2 with a client-sever architecture which facilitated the implementation of the ADELE/TEMPO prototype, and to Jacky Estublier for providing a work environment in which this work was carried out.

During this work W. L. Melo was supported by the Technological and Scientific Development National Council of Brazil (CNPq) under grant No. 204404/89-4.

References

Belkhatir, N. and Estublier, J. (1987). Experience with a database of programs. In *ACM SIGPLAN Notices*, 22(1):84-91.

Belkhatir, N. and Melo, W. L (1994a). "Supporting Software Development Processes in Adele 2." In *The Computer Journal*, 37(7):621-628.

Belkhatir, N. and Melo, W. L (1994b). "Collaborating Software Engineering Processes in Tempo. In C. B. Medeiros (Ed.), *Journal of the Brazilian Computer Society*, 1(1):24-35.

Bernstein, P. A. (1987). Database system support for software engineering: an extended abstract. In *Proc. of the 9th Int'l Conf. on Software Engineering*, Monterey, CA.

Boudier, G., Minot, R., and Thomas, I. M. (1988). An overview of PCTE and PCTE+. In Henderson, P., editor, *Proc. of the 3rd ACM Software Engineering Symposium on Practical Software Development Environments*, volume 24 of ACM SIGPLAN Notices, pages 107-109, Boston, MA.

Chauvet, J. L. . Andromede and the aerian traffic control (in French). In *Genie Logiciel & Systemes experts*. No 21. Dec 1990.

Chen, P. S. (1976). The entity-relationship model towards a unified view of data. *ACM Transaction on Database Systems*, 1(1):9-36.

Courington, W. (1989). *The Network Software Environment*. Sun Microsystems, Inc.

Dayal, A.; Hsu, M.; Landin, R. Organizing long-running activities with triggers and transactions. In *Proc. of the ACM SIGMOD*, pages 204–214,May 1990, Atlantic City.

Deiters, W. and Gruhn, V. (1990). Managing software processes in the environment MELMAC. In [Taylor, 1990].

Derniame, J.-C., Godart, C., Gruhn, V., and Lonchamp, J. (1992). Process-Centered IPSEs in ALF. In G. Forte, N. H. M. and Muller, H. A., editors, *Proc of the 5th Int'l Workshop on Computer-Aided Software Engineering (CASE'92)*, pages 179-190, Montreal, Quebec, Canada. IEEE Computer Society Press.

Dittrich, K., Gotthard, W., and Lockemann, P. C. (1987). Damokles - a database system for software engineering environments. In R. Conradi, T.M. Didriksen, D. W., editor, *IFIP Int'l Workshop on Advanced Programming Environments*, volume 244 of LNCS, pages 353-371. Spring-Verlag, Trondheim, Norway.

Dowson, M. (1991). *Proc. of the First Int'l Conf. on the Software Process*. October 21–22. Redondo Beach, CA. IEEE COMPSOC Press.

Estublier, J., Ghoul, S., and Krakowiak, S. (1984). Premilinary experience with a configuration control system for modular programs. *ACM SIGPLAN Notes*, 9(3):149-156.

Feiler, P. (1991). *Configuration management models in commercial environments*. Technical Report CMU/SEI-91-TR-7, Carnegie-Mellon University, Software Enginnering Institure.

Feldman, S. I. (1979). Make: a program for maintaining computer programs. *Software-Practice and Experience*, 9(4):255-265.

Gruhn, V. (1991). The software process management environment Melmac. In *First European Workshop on Software Process Modeling*, pages 191-201, Milan, Italy.

Habermann, A. and Notkins, D. (1986). Gandalf: Software development environment. IEEE Transaction on Software Engineering, SE-12(12):1117-1127.

Hudson, S. E. and King, R. (1989) Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. In *Trans. on Database Systems*, 14(3):291–321.

Jaccheri, M. L and Conradi, R. (1993). Techniques for process model evolution in Epos. *IEEE Trans. on Software Engineering*, 19(12):1145–1156.

Kaiser, G. E., Barghouti, N. S., Feiller, P. H., and Schwanke, R. W. (1988a). Database support for knowledge-based engineering. *IEEE Expert*, 3(2):18-32.

Kaiser, G. E., Feiller, P. H., and Popovich, S. (1988b). Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40-49.

Kaiser, G. E., Barghouti, N. S., and Sokolsky, M. H. (1990). Preliminary experience with process modeling in the Marvel software development environment kernel. In *Proc. of the 23th Annual Hawaii Int'l Conf. on System Sciences*, pages 131-140, Kona, HI.

Leblang, D. and Chase, R. P. (1985). Configuration management for large scale software development efforts. In *Proc. of the Workshop on Software Engineering Environment for Programming in Large*. Harwichport, Massachussets. June.

Legait, A., Menes, M., Oquendo, F., Griffiths, P., and Oldfield, D. (1989). ALF: its process model and its implementation on PCTE. In Bennett, K. H., editor, *Proc. of the 4th Conf. on Software Engineering Environments, Software Engineering Environments - Research and Practice*, pages 335-350. Ellis Horwood Books, Durham, UK.

Marzullo, K. and Wiebe, D. (1986). A software system modelling facility. *Proc. of the ACM Software Engineering Symposium on Practical Software Development Environments* December, pages 121–130.

McCarthy, D. R. and Dayal, U. (1989). The architecture of an active database management system. In *Proc. of ACM SIGMOD 89*, pages 215–224, Portland, OR, May 1989.

Melo, W. L. (1993). *Tempo: Un environnement de developpement Logiciel Centre Processus*. These de doctorat, Universite Joseph Fourier, Laboratoire de Genie Informatique, Grenoble, France. Le 22 Octobre 1993.

Miller, T. (1989). Configuration management with the NSE. In Long, F., editor, *Proc. of Int'l Workshop on Software Engineering Environments*, volume 467 of LNCS, pages 99-106. Springer-Verlag, Berlin, 1990, Chinon, France.

Notkins, D. (1985). The Gandalf project. *The Journal of Systems and Software*, Vol. 5, No 2, May 1985.

Oquendo, F.; Boudier, G.; Gallo, F.; Minot, R.; Thomas I. (1991). The PCTE+'OMS: A software engineering database system for supporting large-scale software developpement environments. In *Proc. of the 2nd Int'l Symp. on Database Systems for Advanced Applications*. Tokyo, Japan.

Osterweil, L. J. (1987). Software processes are software too. In *Proc. of the 9th Int'l Conf. on Software Engineering*, pages 2-13, Monterey, CA.

Osterweil, L. (1993). *Proc. of the 2nd Int'l Conf. on the Software Process*. Berlin, Germany. IEEE Press

Peckhan, J. and Maryanski, F. (1988). Semantic data models. *ACM Computing surveys*, 20(3):153-190.

Penedo, M. (1986). Prototyping a project master data base for software engineering environments. In *Proc. of the 2nd ACM Soft. Eng. Symposium on Soft. Practical Development Environments*, SIGPLAN Notices, Palo Alto, CA.

Penedo, M. (1991). Acquiring experiences with the modeling and implementation of the project life-cycle process. *IEE Software Engineering Journal*, 6(5):285-302.

Perry, D. (1987). Software interconnection models. In *Proc. of the 9th Int'l Conf. on Software Engineering*, pages 61-69, Monterey, CA.

Peuschel, B., Schafer, W., and Wolf, S. (1992). A knowledge-based software development environment supporting cooperative work. Int'l Journal of Software Engineering and Knowledge Engineering, 2(1):79-1-6.

Reynier, M. J. (1988). Using the Palas environment for implementing embedded real-time software. In *Proc. of the Int'l Workshop on Soft. Eng. and its Applications Toulouse*. Vol 1, pp 469-480, 5-9 decembre 1988. (In French).

Ramanathan, J. and Sarkar, S. (1988). Providing customized assistance for software lifecycle approaches. *IEEE Transactions on Software Engineering*, 14(6):749-757.

Rochkind, M. (1974). The source code control system (SCCS). *IEEE Transactions on Software Engineering*, 1:370-376.

Sarkar, S. and Venugopal, V. (1991). A language-based approach to building CSCW systems. In *Proc. of the 24th Annual Hawaii Int'l Conf. on System Sciences*, pages 553-567, Kona, HI. IEEE Computer Society, Software Track, v. II.

Snyder, A. (1986). Encapsulation and inheritance in object-oriented programming languages. *Comm. of the ACM*, pages 84-91.

Sutton, S. M., Heimbigner, D., and Osterweil, L. J. (1990). Language constructs for managing change in process-centered environments. In [Taylor, 1990], pages 206-217.

Taylor, R., editor (1990). *Proc. of the 4th ACM Software Engineering Symposium on Software Practical Development Environments*, volume 15 of ACM SIGSOFT Software Engineering Notes, Irvine, CA.

Taylor, R. N. and et al (1988). Foundations for the Arcadia environment architecture. In Henderson, P., editor, *Proc. of the 3rd ACM SIGSOFT Software Engineering Symposium on Software Practical Development Environments*, volume 24 of ACM SIGSOFT Software Engineering Notes, pages 1-13, Boston, MA.

Tichy, W. (1985). Rcs - a system for version control. *Software-Practice and Experience*, 15:637-654.

Thomas, I. (1989). Version and configuration management on a software engineering database. *ACM Software Engineering Notes*, 14(7):23–25, November, 1989.

D. Tombros, A. Geppert, K. R. Dittrich (1995). "SEAMAN: Implementing Process-Centered Software Development Environments on Top of an Active Database Management System". Univ. of Zurich, Dep. of Computer Science, Technical Report. IFI-95.02

Zdonik, S. B. (1987). Version management in an Object-Oriented Database. In *Proc. of the IFIP WG2.4 Int. Workshop on Advanced Prog. Env*, Trondheim Norway, 16-18 june 1986, Springer Verlag(ed.) LNCS 244, Feb. 1987.

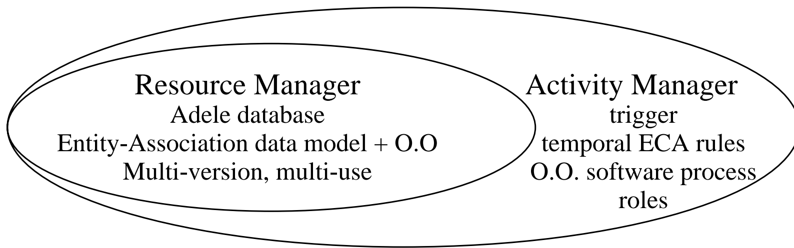

Figure 1: An overview of Tempo.

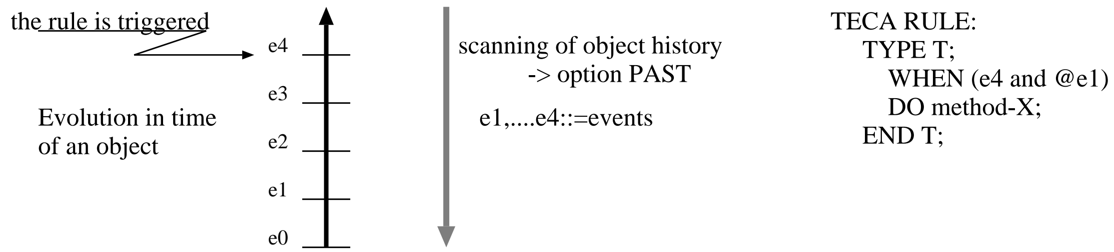

Figure 2: Log of events.

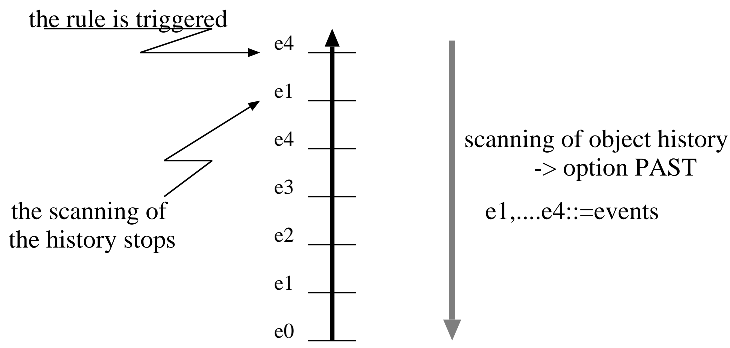

Figure 3: Analyzing temporal events

```
  TYPEOBJECT body ;
     DEFATTRIBUTE
1       lines COMP = INTEGER;
2    PRE WHEN delete_official DO ABORT;
3    POST WHEN replace_body_c DO
4         "store_binary %name" ;
  END body;

  TYPERELATION comp ;
5 PRE  WHEN DEST replace_body_c DO
6         "modify_attr !O -a line-conf = %line-conf - ~!D%lines" ;
7    POST WHEN DEST replace_body_c DO
8         "modify_attr !O -a line-conf = %line-conf + ~!D%lines" ;
  END comp ;

  DEFACTION store_binary;
9    IF "cc -c !filename" THEN
10     {"replace %name -do" ;
11      "modify_attr %name -a lines = \"wc -l !filename\"" } ;
  END store_binary;

  DEFEVENT
12   delete_official = [ !command=delete, state=official ];
13   replace_body_c =  [ !command=replace, language = c ];
  END
```

Figure 4: An example of the utilisation of TECA rules in the data model

```
ModifyDesign ISA PROCESS;
  ATTRIBUTES
    begin_date = DATE := now();
    end_date = DATE;
    deadline = DATE;
  METHODS
    continue_execution;
    . . .
  RULES
(1) AFTER WHEN deadline_arrived
      DO stop_execution;
(2) PRE WHEN (continue_execution AND
               @(not deadline_changed))
          DO ABORT;
END_OF ModifyDesign;
```

Figure 5: An example of the utilisation of TECA rules in the process model