TEMPO:
a support for the modeling of objects with dynamic behavior

W. L. Melo
University of Maryland
UMIACS,
College Park, MD, 20742 USA
e-mail: melo@umiacs.umd.edu

N. Belkhatir
LGI
BP 53
38041 Grenoble France
e-mail: Noureddine.Belkhatir@imag.fr

In A. Verbraeck, H.G. Sol, P. W.G. Bots (Eds.).
Dynamic Modelling and Information Systems.
North-Holland, Elsevier Science Publishers, 1994.
To appear.

Abstract

Recent developments have shown the need for an integrated view of the large scale software development environment that takes account of the artifacts and the way they are produced. To fulfill these requirements, the activities as well as software objects in development and maintenance must be managed. We describe in this paper the work that we are carrying out in order to support a software process modeling language where the dynamic behavior of software objects can be modeled and supported.

We shall discuss an original approach to process modeling known as an object-oriented software process modeling language. Special attention is paid on how object-oriented concepts, the *role* concept and triggers rules make it possible to describe in an integrated framework software process models.

1 Introduction

There is broad agreement that software engineering environments (SEE's) should provide explicit support for capturing and controlling software processes. Software processes should be explicitly represented and, in part, automatically executed by a SEE [8, 18, 22]. To satisfy this requirement, several research programs have explored data integration and centralized control using integrating platforms [30, 32]. These platforms provide support for product structuring, versioning, software configuration, and other engineering processes [27]. On the basis of this experience, the software engineering community agrees that concepts given by the entity-relationship-attribute data model (ERA) extended with Object-Oriented (O.O.) concepts are useful and realistic when used as a framework for process definition. But, although the basic O.O. concepts are a good starting point, as highlighted in different works, e.g.,[16], [23],[28], [6] and [3], they alone are not sufficient for capturing all the complexity of software engineering processes and their evolution. We claim that major capabilities such as *multiple object behavior* and *modeling of activities with long duration*, including long events and the time concept, need to be added to such a model.

We have attacked these problems in our work in the framework of the Adele project at LGI. Adele, which was initially a configuration management system [13], has been extended with user defined entities and relation types for supporting the definition and control of large software systems. To make it possible to support software process, the Adele language has been extended with event-condition-rules (ECA) and the Adele kernel has been extended with a trigger mechanism. This new system is called Adele V2 [3]. Adele V2 is now a commercial software product and is used for automatic software configuration activites [31] in various European companies, such as Matra-Space, and ESPRIT projects, such as REBOOT. On the basis of our experience working with Adele V2 we have concluded that the O.O. data model, even when integrated with ECA rules, is not sufficient for defining software process models. Once ECA rules are fragmented among data and relation types, it becomes difficult to control process enacting and manage changes in processes. To

overcome such drawbacks, we have launched the TEMPO project [4] in order to support software development processes. Thanks to TEMPO, an accurate set of software activities can be aggregated in process types and the static and behavioral description of objects, manipulated by such activities, can be re-defined according to their **roles** in a process step. As time plays an important role in any SEE, where we deal with long transactions, we have also studied how temporal information can be represented and how it can be exploited in the software process evolution. As we shall show later, we have decided to extend our ECA formalism with temporal logical operators. We have modified the TEMPO process engine to make it possible to interpret temporal ECA rules. This paper aims to present these extensions and examples of how they are used for modeling, evolving and enacting software processes.

1.1 Rationale

TEMPO is a Process-Oriented Software Development Environment (POSE) which focus on the following capabilities:

- management of resources shared by a team for enforcing cooperative work, by providing objects with roles;
- activity coordination and traceability of activity execution, by providing activity management;

1.1.1 Object roles

Using object-oriented technology, we can model software process steps (or sub-processes) using complex active objects [28], and as software objects associated with sub-processes also provide operation (methods), the combination of these two kinds of facilities could be used to describe statically (process model) and dynamically (process enactment) the software processes of a particular environment or company.

However, as the only structuring concept supplied by the classical O.O. paradigm is the classification concept, this model supports only one object behavior description, which can be refined by specialization using the class structure. All applications are supposed to agree in that structure, and consequently in that behavior. This mono-behavioral belief, which reigns in the O.O. world, impacts directly on the difficulties involved in process management. We claim that if we allow an object to behave differently depending on where, when and how is it used, and enable it to be seen through different prisms, it will be possible to manage process changes. Using such an approach, we could modify software processes by creating new ways to see and manipulate already instantiated objects in perfect harmony with old definitions.

1.1.2 Activity management

In an O.O. approach, objects interact by executing and exchanging messages via methods which support the active part. A number of mechanisms have been suggested for controlling and synchronizing interaction between methods. The most influential is the trigger mechanism based on Event-Condition-Action rules [17]. ECA rules have been proposed to manage communication by extracting actions specific to method driving from method definition. A similar approach has also been used in Darwin [20], which controls the exchange of messages between objects by providing *laws* to govern such exchanges.

The trigger mechanism is most useful when integrated with short transactions. However, in the framework of process models, activity has a long duration (long transaction). Thus, to coordinate method execution and control activity evolution, we need mechanisms that aid us to keep track of activity chaining and trace its execution. Therefore, we must introduce concepts and mechanisms for dealing with *time*. This leads to:

1. a log database which contains capabilities for tracing object states;
2. capabilities to reason with temporal events.

1.2 Outline

We are fulfilling the requirements presented in the rationale section in TEMPO model by enhancing 1) the Adele platform with history management and versioning services for any object and 2) extending the Adele trigger mechanism to support temporal ECA rules. In order to show how these requirements have been accomplished, this paper is organized as follows. Section 2 presents an overview of the TEMPO software process modeling language. More information about this model can be obtained in the cited references

[5]. We draw a preliminary comparison between the functionalities provided by TEMPO for supporting process evolution, through the **role** concept, with regard to the specialization concept provided by the O.O. paradigm. We also highlight the importance of time constraint management in software process evolution and how this requirement is taken into account by TEMPO and we present our conclusions in section 6.

2 An overview of TEMPO

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

- A resource manager using Adele as a persistent object base for storing objects and activities and for tracing the project's progress. Adele supports an entity-relationship data model which is extended with object-oriented concepts like inheritance, methods and encapsulation.
- An activity manager. The temporal event-condition-action rules (TECA) and the trigger mechanisms are called by the activity manager, which also offers definition concepts, activity structuring using process and role concepts within a process, and work environment support.

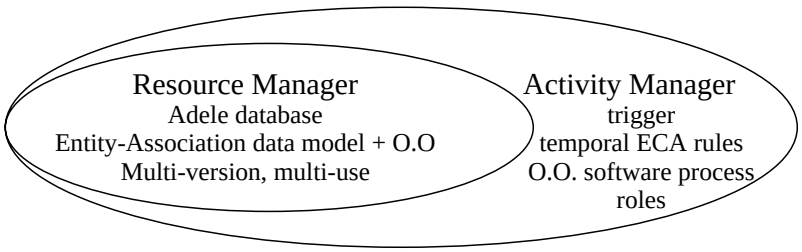

Figure 1: An overview of TEMPO.

2.1 The Adele data model

The Adele data model is derived from an entity-association model and integrates object-oriented concepts [3]. The basic entities of the model are object type and relationship type. Each entity (object and relationship) possesses static (attributes) and dynamic (methods, event-condition-action temporal rules) properties. Relationships are binary.

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. The Pascal module object can consequently be represented as an object linked to two other objects by two types of relationship, 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, using its own semantics and consistency constraints.

In Adele a type is defined by an interface part and an implementation part which describe type instance properties.

The notions of interface and implementation are similar to those used in programs written using languages such as ADA and MODULA, or indeed certain object oriented languages.

The interface part contains the type properties which are visible and are exported. The implementation part contains private properties and the implementation of visible methods.

2.2 The Tempo software process modeling language

TEMPO [5] is an executable formalism for describing and enacting software process models. It is an object oriented approach extended by the addition of a multi-behavioral facility. The multi-behavioral facility is a major problem currently being researched in a wide range of fields. The problem arises when developing large, complex systems characterized by the presence of several agents, working on shared resources and using multiple representations and multiple development strategies. In this context we need a way of expressing relationships between multiple points of view. This requires the expression of relationships between various representations and various development activities.

There are three sides to our approach:

- modeling of software activities by software process types;
- Analysis of the various points of view and software component life cycle states using the role concept;
- describing software temporal constraints by temporal-event-condition action rules (triggers rules). ECA rules are extended, using a temporal modality, in order to support long transactions (long duration activities). The temporal modality is applied to events and allows reasoning in relation to past activities.

2.3 Software process types: modelling software process models

TEMPO describes and executes software processes. A software process model of considerable size may thus be written by a group of various software process types. A software process type has a recurrent definition. It is a mixture of several software process types. The concepts of specialization/generalization and composition/decomposition, defined in the data modelling portion, are also used to model the software processes.

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

1. A sub-process which models the modification activity allowing modifications to the design document.
2. A sub-process which models the revision activity allowing approval of any design document modifications which have been made.

```
MonitorDesign ISA PROCESS;
       CONTROL md;
                sub = ModifyDesign;
                card = 1;
       CONTROL rd;
                sub = ReviewDesign;
                card =1;
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 in the TEMPO formalisme by the MonitorDesign type. This is composed of two sub-processes: ModifyDesign and ReviewDesign. 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 temporal constraints by using the event-condition-action rules.

2.4 The temporal contraints

Due to the long life duration of software processes we also need management mechanisms for time constraint and traceability. As for supporting process evolution we need to be able to reason about execution sequences, we have introduced temporal reasoning in TEMPO to plan and schedule activities in software process. Thus, temporal properties can be used either for scheduling those activities that are carried out in isolation by a WE or for aiding synchronization and cooperation with other WE's. In order to integrate temporal constraints or specification in TEMPO, we are incorporating event-condition-action formalism (ECA) with temporal features and extending Adele's trigger in order to make time reasoning possible. This new

formalism is a nice combination of standard ECA rules and temporal logic. In this way, we are adding a new dimension to TEMPO for managing process evolution using temporal knowledge in the field of software process management.

2.4.1 Temporal event-condition-action rules

Temporal contraints are described in TEMPO by temporal event-condition-action rules (TECA). TECA rules in TEMPO are similair to Alf [9], Damokles [10] and HiPAC [19] rules.Interpretation and execution of these rules are based on Adele's triggers and its object management system [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
                      IFPAST not deadline_changed
                        FROM last(deadline_arrived) UNTIL now()
                          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 (it hasn't been completed yet) first requires that the termination date be changed.

2.4.2 TECA rules execution module

TECA rules are defined in the data model (not shown in this article) and in the software process module. They are inherent in the hierarchy of object types and software processes. In the data model, the TECA rules describe integrity limitations which are independent of the object's usage context. On the other hand, these rules are used to express the software development strategy used in the software process model: order of activity execution, activity synchronization and software resource usage limitations.

A TECA rule is expressed in the following manner: "WHEN temporal-event DO Method", where "temporal-event" is the temporal predicate expressing :

1. an event in the present environmental state or

2. a state in the present or past object management system.

Method is an instruction sequence.

```
DEFEVENT delete_obj = [ !cmd = rmobj] ;
```

The delete_obj event is defined in this example as being the event which survives whenever the current command (!cmd) is an object removal command (rmobj).

A method is a program written in simple, direct language similar to the Unix shell.

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

This method allows for object removal in an unstable state.

A TECA rule, defined in a type, is executed by Adele's triggers whenever the related event is true for an instance of this type. There are four modes of trigger execution for each type:

```
PRE             {liste de triggers}
POST            {liste de triggers}
AFTER           {liste de triggers}
ERROR           {liste de triggers}
```

Some triggers act as pre-conditions (before the main action), while others act as post-conditions (after the main action). Any incoherence detected during trigger execution rejects the action performed on the database. Thus, for every action the following block executes:

```
PRE {liste de triggers}
    methode
POST {liste de triggers}
```

The entire block is considered to be an atomic, short transaction even if the related rules or corresponding action triggers other actions. A simple "ABORT" encountered in this block allows for complete cancellation of all operations performed in the block.

If the transaction is committed, the rules associated with the "AFTER" block are executed; otherwise, once the transaction is finished, the rules associated with the "ERROR" block are executed.

A relationship's TECA rules are executed each time an action is performed on a relationship (create, destroy, etc.). We added rules to control actions performed on objects linked by a relationship. Thus each object may have a behaviour determined by its relationship to other objects; this is how aggregates are controlled, for example.

The clause "event" in the TECA rules is interpreted with respect to the historical log of the Adele object base. Temporal limitations are verified by an inverted route of the object's historical log from the moment the event starts until the temporal restriction is met. If the temporal restriction is not verified, no action will occur.

2.5 Object with roles

The problem with multiple perspectives or multiple viewpoints often occurs during a software product life cycle. This is due to the fact that several users treat objects concurrently, using different views of the objects with limited, controlled actions specific to their activity. These users, controlled by multiple development strategies, handle different models of the same product. A SEE should provide a work environment which can describe and control these various aspects.

Thanks to the role concept, TEMPO allows each software process occurrence to have local contraints and properties for each object treated [4].

Roles are of a defined type. A role type may reference different types of objects. This allows for the integration of various types of behaviour and properties, coming from different types of objects, within a unique perspective. By using this concept, TEMPO unifies the treatment of a heterogeneous set of objects. The advantage of this approach is that a set of object types with different static and dynamic characteristics may, using the role concept, be viewed during a specific software process execution step in a coherent, homogeneous fashion. This coherence is maintained by using the multiple heritage rules used in the object oriented models. The principal difference is based on the extent of the roles. At the definition level, a role type is viewed as the specialisation of the types it contains (see diagram above). However, at the instance level:

1. Objects created from a role are not included in the role's specialised type extensions.

2. A subset of objects pertaining to these types may belong to the role.

A software process type may have several role types; a software process becomes a list of roles whereby each object type may have different roles. Consequently, two objects of the same type may be controlled differently within the same software process. At the same time, an object can play roles within different software processes. For example:

```
ReviewDesign ISA PROCESS;
ROLE under_review;
        derived_from = specification_document;
        . . .
ROLE requested_change;
        derived_from = cc_request;
        . . .
END_OF ReviewDesign;
```

3 Role discussion

One may claim that this kind of contextual behavior can be achieved by standart object-oriented techiniques. Roles and type lok similar; this raises the question: can roles be implemented in terms of typing and subtyping? Is the concept of role needed at all? We claim the role concept has the following properties:

1. Prevent type explosion.
   A role, as well as a type, is a template applied to a set of instances sharing the same definition (static and behavioral). A given object instance can be simultaneously a member of different roles (classes). Both roles and types can be seen as a viewing mechanism since a given object instance has a different description depending on the role (class) from which it is managed. One would need to create a subtype for all the possible combinations of roles for a single type, and to change instance type dynamically each time a new role is applied to it. However, there is a fundamental difference:

   > The association between an instance and its type is statically defined at instantiation time, while an instance can be dynamically bound to an arbitrary role at any time.

   Furthermore, since the instance may shared and play diffrent roles simultaneously, dynamic typing cannot be used. We introduce the possibility to changing type dynamically. In an O.O. system the type definition is created first, and then the instances of the types. In TEMPO on the other hand, the instances are usually created first, and are dynamically associated, for a while, to a (set of) role(s).

2. Identity is not altered.
   Since a given object can be simultaneously a member of different roles there are compatibility rules between the roles allowed for shared objects. In TEMPO, objects can change behavior depending on the context without changing identity.

3. Schema evolution-version.
   Schema evolution support is an important facility for a software engineering environnment, because we need to make it possible to envolve the characteristics of software objects manipuled during the software processes. The role concept naturally integrates a type evolution facility, since role types are similar to object types in O.O. languages. The role concept offers two kinds of evolution:

   (a) role definition can change generating role version;

   (b) objects can change its roles dynamically.

4 Evolving software process definitions

Evolutionary features are an integral part of TEMPO. We believe that an object oriented software process langage, such as TEMPO, which supports object with roles, can easily integrate evolution. This is facilitated, on the one hand, by O.O. paradigm concepts, such as multiple inheritance, late binding, polymorphism and, on the other hand, because the actual effects of any modifications are very localised [1].

Evolution in Tempo occurs at two levels of granularity, which are complementary (see figure 2):

1. In the object life cycle model; evolution is achieved by dynamic, incremental adding of roles. As TEMPO has an object-oriented approach, the semantics of a software process instance are defined by the behaviour of the objects in the various roles they are required to play in the instance. Consequently, any change in object behaviour, obtained by updating the corresponding roles, results in a change in the semantics of the instance.

2. In the process hierarchy; evolution by sub-process hierarchy development. On account of the long life cycle of software development activities, it is necessary to adjust the process model to take account of new development strategies, additional knowledge, etc. This results in a process decomposing into several sub-processes, several sub-processes unifying to become a single process, dynamic changes in process types and so on.

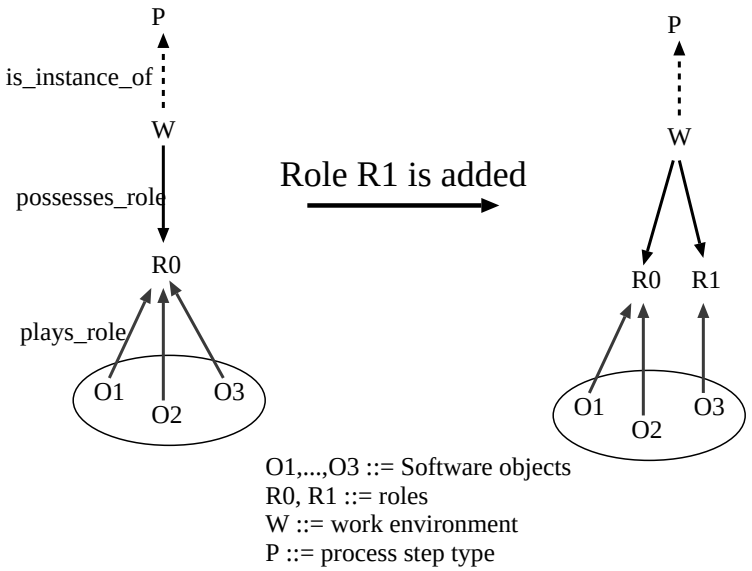

Figure 2: Evolving software process by role adding.

In the following pages we shall analyse each of these evolutions.

4.1 In the object life cycle model: adding roles

Role types are used to model the life cycle of a software object. This raises the problem of how to integrate new role type which correspond to unforeseen situations (exceptional situations). This approach may also be used to allow for incremental development of an object life cycle, gradually defining roles. An object participating in an activity has a structure and behaviour which complies with its role. Roles provide a way of tracking the various facets of object evolution.

For example:

> A software object representing a configuration evolves in compliance with its life cycle: specification, construction, test and qualification. For each of these stages, we can define a corresponding role: in the specification role, configuration properties are defined; in the construction role, there is a description of how the configuration should be generated; in the test role, procedures for adjusting the configuration are defined (compilation, linking and testing); and finally, in the qualification role, the configuration is approved prior to being made available to users.

Using this approach, the role type changing of an object represent stages in its life cycle. Transitions from one stage to the next are governed by temporal rules which are explicitly defined by the environment administrator (for example, the administrator will only add the qualification role to the configuration installation process once the configuration has been tested and approved).

Let W be an instance of software process type P currently being executed (see figure 2. The objects manipulated in this instance are visible and behave in compliance with the roles played in W.

Adding role type R1 to P triggers a change in object organisation in W, as certain objects which were playing role R0 in W will now be playing role R1.

The role change has two effects on process evolution:

1. An existing object O3, playing a given role R0 in W will have a new point of view, once it has changed its role to R1.

2. The behaviour of object O3 in the new role R1 changes, for new methods and rules are described in R1.

For an object, the switch from one role to another is dynamic; the switch causes the software process to evolve. On the one hand, a new type of role is dynamically added to the software process instance and, on the other hand, as the object changes roles it also changes behaviour. In order to control these changes, temporal event-condition-action rules are defined in the software process types.

For example:

When a configuration, playing the specification role, is in the "specified" state the following operations are executed to make the process model evolve:

1. Create the construction (modelling) role type.
2. Instantiate the role type in the configuration installation (instantiation) process, currently being executed.
3. Transfer the configuration to the new construction role (operation and implementation).
4. Continue execution of the configuration installation process (enacting).

We use this schema (modelling, enacting and software process performing) first proposed in [12] (but without execution support), to modify the description of software processes and execute them taking account of their evolution (see figure 3). Software development policies and the way they evolve are defined, in a uniform manner, by operations and rules expressed in role types / sub-process types belonging to the same description model. This links up with recent work in the field of process management which attempt to provide a common framework for describing software processes and their evolution, using a single formalism [2, 21].

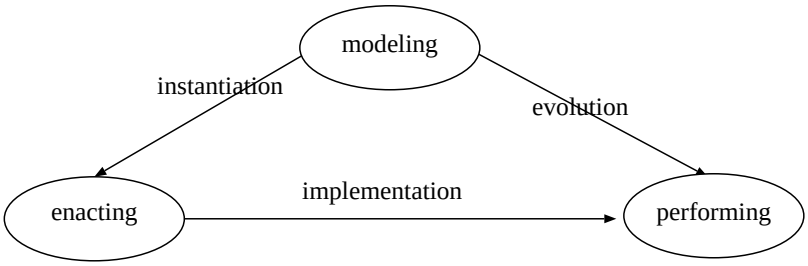

Figure 3: Evolution cycle.

4.2 In the software process: adding sub-processes

In a similar way as with roles, process instances are dynamic entities which are created and destroyed in line with a set of rules. In our approach, a software object and a process instance have common characteristics (attributes, methods and ECA rules). However, unlike objects, process instantiation requires a collection of roles controlled by the constraints associated with this instance. In this way the evolution of a software process instance can be compared with the evolution of an aggregate object in a collection of roles. The evolution process can thus be applied, recurrently, for each aggregate role.

5 Related works

Among the several kinds of process language that the software process community has been using to model the software processes into process-oriented SDEs (POSE), the rule-based, the procedural and the event-condition-action languages are the most representative.

5.1 Rule-based POSE

Rule-based POSEs are advantageous because the software process can be described by using logical declarations allowing to specify what the user wants rather than a detailed specification of how the results are to be obtained [33]. Using this behavioral approach, various prototypes of rule-based POSEs have been built, e.g., Marvel [15], Merlin [24], and Epos [7].

Although, our approach is not completely declarative, we consider that rule-based facilities are important when executing software processes. Therefore, we have used rules in order to envelop method execution as well as to allow TEMPO to take initiatives when possible, based on the rule conditions.

5.2 Programming-based POSE

Osterweil [22] has proposed the procedural approach whose key idea is a complete algorithmic description of software process by means of a formal language. This description is considered as a specification of how a software process is to be managed in the SDE by users and tools. Several on-going projects have been influenced by this idea, resulting in the construction of some experimental POSEs as, for example, Triad-CML [25], Arcadia-Appl/A [29]. Both these POSEs have extended the Ada language with new capabilities for supporting software processes. The main drawback with this approach is that no algorithm of a particular software process can be completely pre-described in advance. Another technical problem with these systems is the need to modify the Ada compiler.

Of course, we also have been influenced by this idea — the process type is described, in part, by a procedural formalism — however our solution is much more flexible than the systems we have quoted. Our language is interpreted and provides late-binding facilities. With these characteristics, it is easier to adapte the changes in the environment, without changing the process description.

Other POSEs, although much influenced by the procedural approach, have experimented other kinds of programming paradigms, as for example PSS-PML [6]. Although, in a different context, our formalism is also inspired from object-oriented languages and systems. The software processes are described using an OOER [1] formalism. We have broadly used type inheritance, methods and triggers. Like PSS-PML, we use roles in order to control software activities, but unlike PSS, which uses role only to model user activities, we have extended this notion to capture all resources manipulated by the activities.

5.3 Trigger-based POSE

In the trigger approach, software processes are modeled by a set of event-condition-action rules which are interpreted by a trigger mechanism tightly connected with a software database. Adele is one of the very few practical systems that has experimented this approach. Other examples are: Arcadia/Appl-A, Alf/Masp [9] and AP5. Appl/A has extended Ada language with programmable trigger upon relations. The automation of the software process is done by these triggers.

AP5 [14] is an active, in-core, relational database extension to Common Lisp. AP5 users can register triggers with the database. A trigger consists of a condition, written in first-order logic with temporal extensions, and a body, written in Lisp. Triggers can guarantee data base integrity by modifying or rejecting database transactions, and can invoke non-database activities in response to transactions. An AP5 trigger condition defines a database event to be announced, while a trigger body represents the code executed when an event is announced. AP5 events are announced after transactions are submitted but before they commit, allowing transactions to be modified or aborted. We believe that TEMPO could be implemented also above AP5, since AP5 provides O.O. facilities for supporting product structuring and trigger for supporting dynamic aspects of a SEE.

In PCTE+/Alf [9], triggers are used to control communication among parallel tasks by capturing changes on database objects, however, these tasks are modeled as in the Marvel 2.0 — i.e., pre- and post-conditions enveloping foreign tools — and managed by a specialized expert system shell connected to PCTE+.

Unlike Arcadia/Appl-A and Alf/Masp, the Adele trigger can be attached to both entity and relationships to envelop methods. Four types of trigger coupling can be used (pre, post, after and error) thus providing greater flexibility.

[1] Object Oriented Entity-Relationship-Attribute

6 Conclusion

The TEMPO model is designed for describing software processes and in particular for enforcing cooperative work.

The main capabilities of TEMPO are:

- The model used to describe processes is an object-oriented model. Each process step occurrence aggregates a set of entities. Each entity is managed by Adele-DB. When an entity is manipulated it is considered by TEMPO as process resource. A resource "plays a role" in a software process step. The role concept makes it possible to customize the characteristics and behavior of a resource. Resource attributes can be forbidden, modified, created and overloaded in order to satisfy the requirements of a process step. In the same way, resource behavior can be tuned, and specific communication and synchronization operations can be described in order to take account of events generated inside/outside a software process step.

- TEMPO provides new facilities for working with object log and process evolution. The trigger mechanism is under extension in order to make it possible to support temporal event interpretation, i.e, in event-condition-actions rules can be specified with temporal logic predicates. In this way, long transactions can be more successfully controlled; rules can verify object manipulation and evolution by analyzing operations performed on it during the software process life;

We believe that the unification of these features contributes to improved cooperative work in a team of developers and proper sharing of resources among a set of processes. The management of software process evolution is accomplished: by making it possible for an object, already instantiated, to change its behavior and static properties dynamically using the **role** concept; following object evolution in the time by a log database; and controlling such evolution by temporal ECA rules.

References

[1] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system Manifesto. In *Deductive and Object Oriented Databases*, Kyoto, Japao, 1989.

[2] R. Balzer. Generic process support. In Schafer [26].

[3] N. Belkhatir, J. Estublier, and W. L. Melo. Adele 2: a support to large software development process. In Dowson [11], pages 159–170.

[4] N. Belkhatir, J. Estublier, and W. L. Melo. Software process model and work space control in the Adele/Tempo system. In L. Osterweil, editor, *Proc. of the 2nd Int'l Conf. on the Software Process*, pages 2–11, Berlin, Germany, February 1993. IEEE CS Press.

[5] N. Belkhatir and W. L. Melo. Tempo: a software process model based on object context behavior. In *Proc. of the 5th Int'l Conf. on Software Engineering & its Applications*, pages 733–742, Toulouse, France, December 7–11 1992.

[6] R.F. Bruynooghe, J.M. Parker, and J.S. Rowles. PSS: a system for process enactment. In Dowson [11], pages 128–141.

[7] R. Conradi, E. Osjord, P.H. Westby, and C. Liu. Initial software process management in Epos. *IEE Software Engineering Journal*, 6(5):275–284, September 1991.

[8] B. Curtis, M. I. Kellner, and J. Over. Process modeling. *Communications of the ACM*, 35(9):75–90, September 1992.

[9] J.-C. Derniame, C. Godart, V. Gruhn, and J. Lonchamp. Process-Centered IPSEs in ALF. In N. H. Madhavji G. Forte and H. A. Muller, editors, *Proc of the 5th Int'l Workshop on Computer-Aided Software Engineering (CASE'92)*, pages 179–190, Montréal, Québec, Canada, July 6–10 1992. IEEE CS Press.

[10] K.R. Dittrich. The Damokles database system for design applications: its past, its present, and its future. In K. H. Bennett, editor, *Software Engineering Environments: Research and Practice*, pages 151–171. Ellis Horwood Books, Durhan, UK, 1989.

[11] M. Dowson, editor. *Proc. of the First Int'l Conf. on the Software Process*, Redondo Beach, CA, October 21–22 1991. IEEE CS Press.

[12] M. Dowson. Process variables and process change. In Schafer [26].

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

[14] N. Goldman and K. Narayanaswamy. Software evolution through interative prototyping. In T. Montgomery, editor, *Proc. of the 14th Int'l Conf. on Software Engineering*, Melbourne, Australia, May 1992. IEEE CS Press.

[15] G. E. Kaiser, N. S. Barghouti, and M. H. Sokolsky. 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, January 1990.

[16] J. Kimball. The EIS execution engine and the AAA process engine. In M. H. Penedo, editor, *Process Sensitive SEE Archicture Workshop*, Boulder, CO, September 1992.

[17] G.M. Lohman, B. Lindsay, H. Pirahesh, and K.B. Schiefer. Extensions to Starburst: objects, types, functions, and rules. *Communications of the ACM*, 34(10):94–109, October 1991.

[18] N. H. Madhavji. The process cycle. *IEE Software Engineering Journal*, 6(5):234–242, September 1991.

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

[20] N. H. Minsky. Law-governed systems. *IEE Software Engineering Journal*, 6(5):285–302, September 1991.

[21] K. Narayanaswamy. Enactment in a process-centered softwre engineering environment. In Schafer [26].

[22] L. J. Osterweil. Software processes are software too. In *Proc. of the 9th Int'l Conf. on Software Engineering*, pages 2–13, Monterey, CA, March 30-April 2 1987.

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

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

[25] S. Sarkar and V. Venugopal. 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, 1991. IEEE CS Press, Software Track, v. II.

[26] W. Schafer, editor. *Proc. of the 8th Int'l Software Process Workshop*, Germany, 1993. IEEE CS Press.

[27] I. Simmonds. Configuration management in the PACT software enginnering environment. *ACM Software Engineering Notes*, 14(7):118–121, November 1989.

[28] Y. Sugiyama and E. Horowitz. Building your own software development environment. *IEE Software Engineering Journal*, 6(5):317–331, September 1991.

[29] S. M. Sutton, D. Heimbigner, and L. J. Osterweil. Language constructs for managing change in process-centered environments. In R. Taylor, editor, *Proc. of the 4th ACM Soft. Eng. Symposium on Soft. Practical Development Environments*, volume 15 of *ACM SIGSOFT Soft. Eng. Notes*, pages 206–217, Irvine, CA, 1990.

[30] I. Thomas and B. Nejmeh. Definitions of tool integration in software engineering environments. *IEEE software*, 8(2):29–35, March 1992.

[31] W.F. Tichy. Rcs — a system for version control. *Software—Practice and Experience*, 15:637–654, 1985.

[32] A. I. Wasserman. Tool integration in software engineering environments. In F. Long, editor, *Proc. of Int'l Workshop on Software Engineering Environments*, volume 467 of *LNCS*, Chinon, France, September 18–20 1989. Springer-Verlag, Berlin, 1990.

[33] L.C. Williams. Software process modeling: a behavioral approach. In *Proc. of the 10th Int'l Conf. on Software Engineering*, pages 174–186. IEEE CS Press, 1988.