Collaborating Software Engineering Processes in Tempo

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

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

Abstract

We will show in this article how Tempo, a process-centered software engineering environment (SEE), assists in cooperative work by means of an approach based on a communication model. We will describe the executable formalism used to define software engineering activities, and we will show how constraints related to the use of objects in these activities are expressed using the role concept. We will then present our communication model. In this model, strategies governing the cooperation between various software processes are specified by the concept of active, programmable connections. A connection is a communication channel that links two roles. Message exchange is controlled using TECA rules (Temporal-Event-Condition-Action rules), executed by a trigger mechanism. These allow for programming of synchronization strategies between processes, propagating the effects of an executed action on one or more connection points. The Temporary modes of TECA rules allow for transactions of long duration, because these can be used to reason on past activities. Coherence control of objects handled by activities of long duration is performed by the work environments. The union between connections and work environments makes it possible to support cooperating processes and object sharing between these processes.

Keywords: Cooperative work, software engineering process, software engineering environments, contextual behavior of objects, trigger rules, support for communication and synchronization.

1 Introduction

The problems of developing large-scale software systems can be classified as programming-in-the-small, programming-in-the-large [21] and programming-in-the-many [12]. By programming-in-the-small we mean those activities associated with someone who develops a module or program alone. When a small group of software performers is in charge of developing or maintaining a software product composed of different kinds of software artifacts each of which can evolve into different versions, different kinds of problems are encountered — programming-in-the-large problems. Programming-in-the-many refers to software those activities involving the coordination and the control of activities carried out by large and/or distributed teams of software performers. Programming environments have been built to deal with programming-in-the-small problem. Version and configuration management systems have been constructed to face programming-in-the-large problems. With the more recent development of process-centered software engineering environments, programming-in-the-many has been identified as a major field where concepts, mechanisms and tools need to be provided [9]. Experimental studies show that most of the effort of software performers is spent on managing communication and collaboration between development team members working on the same project. Cooperation support mechanisms should be integrated into a software engineering environment (SEE) to offer a conceptual framework where activities involving resource sharing, coordination, collaboration and synchronization can be described and controlled by the environment. By using such an approach, the resource sharing strategies, communication and coordination within the development team, and synchronization of software engineering activities may be explicitly described using an executable formalism and then enforced in the software organization by a process-centered SEE.

Based on our experience in the development of Adele [4], an environment for programming-in-the-large which supports software product structuring and versions of software objects, we started the Tempo [5] project in order to take into account the production and evolution strategies of software systems. Tempo is

a SEE driven by an executable formalism which allows description of software process models, object views, and elaboration on the strategies of cooperation and communication [17]. In this paper, we will stress those aspects of Tempo which relate to cooperation support, with particular emphasis on the two following aspects:

1. Resource coordination. This is the problem of object sharing among team members. We will show how Tempo supports activities of long duration. Many software artifacts can be manipulated by activities which execute concurrently (cooperating processes). An activity takes place within a context called a work environment. A work environment can be linked to other work environments by a particular level of communication.

2. Cooperation between the agents who share the model of a common software process. We will introduce and develop the concept of active, programmable connections as a means of expressing the cooperation and synchronization strategies.

In order to achieve this objective this paper is organized as follows. Section 2 presents the architecture of the kernel of the Tempo environment. Section 3 treats the aspects related to the description and the control of software engineering activities by presenting suggested concepts for modeling the software processes. Next will be discussed the problem related to the description and management of multiple viewpoints and show how our approach, via the role concept, offers a contribution on this issue. Section 4 will deal with the communication model used to describe communication policies between software process occurrences. Section 5 will show how cooperative work is supported by work environments. Conclusions are given in section 6, with indications of further work.

2 The architecture of Tempo

As shown in figure 1, Tempo consists of the following components:

- A resource manager. Tempo's resource manager uses the Adele database as a persistent object base for storing objects and activities, and for tracing the project's progress [4]. It supports an entity-relationship data model which is extended with object-oriented concepts like inheritance, methods and encapsulation. Simple and composite objects with attributes and relationships can be described and managed. This component of the Tempo architecture is responsible for the data integration according to the conceptual model proposed by [26].

- An activity manager which is responsible for the control integration. This activity manager is driven by temporal-event-condition-action rules (TECA) and supported, in part, by Adele's trigger mechanism [5]. We enhanced Adele's trigger mechanism with the ability to manipulate temporal expressions [17].

- A process manager which offers the concepts of process type and role. Process enactment is supported by work environments (WE) wherein software activities are performed. The process manager, based on the activity manager, manages communication and synchronization between teams, and between agents involved in the same project. It also controls the consistency of complex objects used simultaneously in different work environments by different agents [7]. This component represents the conceptual component responsible for process integration in the Tempo architecture.

Adele 2 plays the role of resource manager and activity manager in the current version of Tempo [6]. Adele 2 [4] is a commercial product which is the result of the union of two long term projects in the framework of the *Laboratoire de Génie Informatique de Grenoble*. Adele 2 integrates the results produced by the Adele 1 and Nomade projects [3]. Adele 1 [3] was a version management system hard-coded with a configuration builder quite similar to the one of Rcs [25]. Nomade was a prototype of an active software engineering database. This database was driven by an object-oriented data model. The active part of this database was supported by a trigger mechanism, which was driven by event-condition-action rules. Nomade incorporated the version management system of Adele 1 for dealing with the evolution of software artifacts in versions. Adele 1's configuration manager was also included in the nucleus of Nomade. Tempo [17] is the successor of Nomade. Tempo is able to deal with user defined software process models, multi-points of view of software artifacts, temporal events, long-time duration activities, and it provides support for communication of software activities. The concepts and mechanisms proposed by Tempo are going to be incorporated into Adele 3, which is will be the new commercial version of Adele.

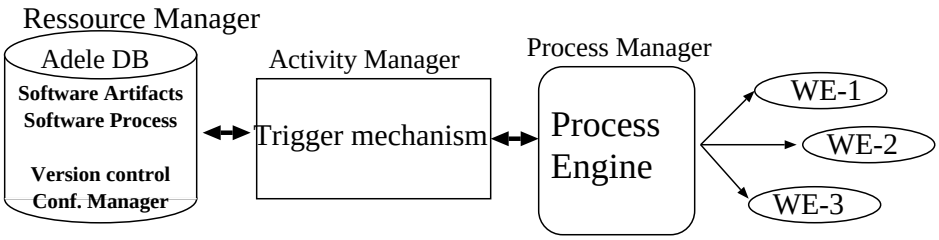

Figure 1: Conceptual architecture of Tempo.

3 The Tempo software process modeling language

The Tempo language is an executable formalism for describing and enacting software processes. 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 development activities.

There are four sides to our approach:

- The modeling of software activities by software process types.
- The analysis of the various points of view and software product life cycle using the role concept.
- The description of software temporal constraints by temporal-event-condition action rules. TECA 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.
- Strategies governing the synchronization and cooperation between different concurrent process occurrences are specified by connections referred to as active and programmable. The communication description strategy is made by rules defining specific synchronization strategies between roles, and propagating their effects when an action executes on one of the two connection points.

3.1 Data modeling

3.1.1 Objects and relationships

At the basis of the Tempo environment we find Adele's database which is used as a resource manager (objects, tools and activities) [4]. Software objects are represented using Adele's data model. This data model is based on the entity-relationship data model and integrates object-oriented concepts. Base entities of the model are object type and relation type. Each object (object like relation) has static properties (attributes) and dynamic properties (methods, temporal-event-condition-action rules).

The different software object types, such as Pascal programs, C programs, binary code, texts written in Latex or with any other text editor can be modeled by a set of object types. For each object type, a set of attributes can be defined to characterize those objects which belong to this type. A special attribute is also provided for storing the content of the software object, for example, the source code of a program. All objects can evolve by generating revisions. Abstract objects can also evolve in versions.

Another characteristic of the Adele data model is the framework it provides for supporting relations between software objects. Semantic relationships between software objects can be modeled by relation types. The Adele database's data model supports only binary relations. A relationship always links a source object to a destination object and its existence is linked to that of the objects it connects.

3.1.2 Versions

The Adele data model is based on the branch concept. A branch models the evolution of a simple object. A branch object is a sequence of revisions. Each revision contains a snapshot of the object attributes, such as in Nse [18] and Rcs [25].

Different kinds of derivation graphs can be defined, establishing explicit relationships between branches.

Version groups can be defined, establishing explicit relationships between branches. In this way, versions groups can thus be easily defined. The Adele database supports the generic concept of object (a branch is also an object) and a mechanism for shared attributes. Arbitrary versions and composite objects are created and managed using explicit relationships between the different components.

As we shall see later in this paper, Tempo's cooperative work support mechanisms use Adele's version management system. This system provides a simple, but efficient, way to support long-time duration tasks and parallel manipulation of software objects. Generally during the execution of the software processes, software tasks can spend many hours, days, or even moths to be accomplished. During that time a software object, O, can be allocated for long time to a task, T, in which the object O will be modified. However other tasks may also need of the object O. If a version management system is not used these later tasks would be suspended until the task T has produced a new release of the object O — cooperative work is absent is this scenario. With the support of a version management system, a version of the object O, say O', can be created and allocated to the task T whereas other tasks can continue to execute using the object O. This mode of work has been popularized in the literature as being the *check-in/check-out* model. Depending on the version management system used, access to the object O can be restricted to only those tasks that will use the object O as an input resource, i.e. the object O can be used but it cannot be modified whereas the task T has not released it. Other more powerful systems, like Nse [18], make it possible the modification in parallel of a same object. In this way, a version of the object O will be created to all tasks that request the manipulation of the object O. More elaborate merge mechanisms must be provided to support this high-demanding *check-in/check-out* model. Nowadays many version management systems provide some kind of support for merging versions of software objects. However, as far we know, only text objects are actually supported. Merge of structured objects continues to be a research topic.

3.2 Process modeling

3.2.1 The software process types

Software development policies of a specific software organization are formally described by a set of software process types. A software process type has a recursive definition, where a software process type can be composed of several others software process types. The concepts of specialization/generalization and composition/decomposition, defined in the data modelling portion, are also used to model the software process.

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 formalism 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.

For every process type it is possible to define attributes, methods and temporal constraints by using the temporal event-condition-action rules.

3.2.2 Temporal constraints

The flow of the software production process is controlled by temporal constraints. We need to provide

a conceptual framework allowing the tracing and persistency of anterior (past) states of software objects. On the other hand we need to provide a formalism in which constraints about software development activities can be described. Mechanisms which make it possible to enforced constraints in a software development environment must be also provided. The definition of constraints involves the specification of conditions about the current state of the software development environment as well about its past states.

Temporal constraints are described in the Tempo soft-
ware process language by Temporal-Event-Condition-Action rules (TECA) and supported by an enhanced version of Adele's trigger mechanism [7]. TECA rules are defined both in the data model and in the activity model. They are inherited in the hierarchy of types. In the data model, the TECA rules describe integrity constraints independently of the context of utilization. In the activity model, these rules are used to express software development policies: the execution order of activities, their synchronization, and constraints above the use of software resources.

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 program written in a simple imperative language similar to Unix's shell language.

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.

As shown in figure 2, when event e4 occurs, the rule

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

is triggered. The object history in which event e4 occurred is scanned to check if event e1 occurred previously (the "`@`" constructor allows expressing conditions in the past). "`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

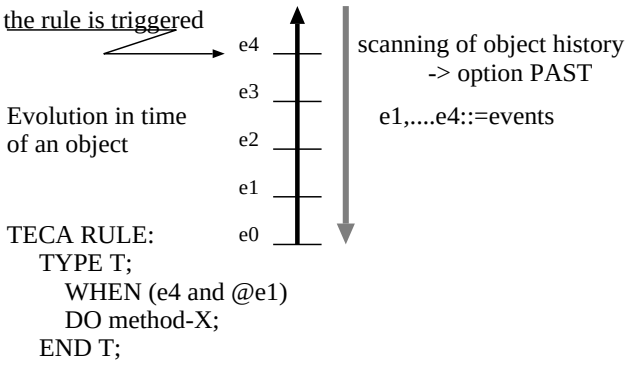

Figure 2: TECA rules execution.

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).

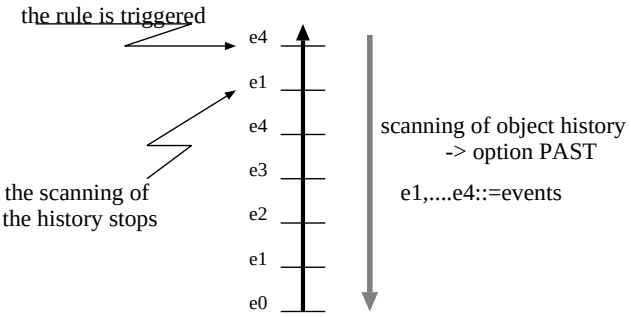

Figure 3: TECA rules execution and past events.

TECA Rules and short transactions

A TECA rule, defined in a type, is executed by Adele's trigger mechanisms [4] when the associated event is true for an instance of this type. The execution of TECA rules is related to the transaction mechanism in Adele.

Some rules can be defined as preconditions (before the main action), others as post-conditions (after the main action). Any incoherence detected during the execution of the rules leads to the rejection of the operation performed on the database. Thus, for every operation, the following block is executed:

```
PRE (list of triggers)
    methods
POST (list of triggers)
```

The whole block is considered a unique action even if the related rules or corresponding actions trigger

other operations. A primitive residing in this chunk, called "EXCEPTION", enables annulment of all the operations carried out in the chunk, causing the transaction to abort. If the transaction is validated, the rules associated with the block "AFTER" are executed; otherwise, the rules associated with the block "EXCEPTION" will be executed once the transaction is undone.

Some related work

Other SEE's, such as AP5 [19], Alf [20], Triad [22] and Appl/A [24], use ECA rules. Alf and Tempo provide four TECA rules execution modes (PRE,POST,AFTER and EXCEPTION) whereas AP5, Marvel, Triad and Appl/A support only one mode of execution, the mode POST. All these systems do not provide concepts for handling temporary constraints.

Tempo's event-condition-action rules are similar to those offered in Alf. However, Alf does not have the concept of method. Similarly to Marvel, the actions in Alf must be defined by the operators (production rules according to the MASP formalism). The execution of an operator can trigger a forward chaining process in the user's private space (ASP in the Alf terminology). ECA rules are defined elsewhere and executed by another mechanism called "trigger". In Tempo, we adopted only one concept to define both the constraints on the execution of methods and the constraints on the utilization of objects, i.e, the TECA rules.

Some other systems like AP5 [19], ODE [14] and SAMOS [13] also, in a way, provide ECA rules concerning time. However, these systems do not allow the specification of conditions about past actions. They limit themselves to specifying that for example, some actions (mainly methods) must be executed at an absolute/particular time in the future (every day at 9 a.m or tomorrow at 6 p.m.), etc.

3.2.3 Object with roles

Motivation

The problem of multiple perspectives or multiple viewpoints often occurs in the lifetime of a software. In this situation, users handle objects simultaneously, use different viewpoints of these objects, and carry out actions limited and directed by the constraints of their own activities. These users, directed by multiple development strategies, handle different models of the same product.

A SEE must therefore provide a framework permitting the description and control of these aspects in the environment. Tempo offers concepts allowing the description and structuring of multiple viewpoints. Based on the rule concept and for each object handled, every occurrence of software process can have constraints (TECA rules), local operations (methods) and local properties (attributes). For example, a module belonging to the Pascal object type, M1, has properties and constraints inherited from this type. Via the role concept, a module M1 can have new properties, new methods and new temporal constraints based on its role in an activity. For example:

```
TYPEOBJECT C_body ISA body;
 METHOD
  compilation;
  # With debug option
  link;
END C_body;

test ISA PROCESS;
 ROLE under_test;
  derived_from := C_body;
  METHOD
   compilation;
   # without debug option
END_OF test;

integration ISA PROCESS;
 ROLE under_integration;
  derived_from := C_body;
 METHOD
  compilation;
  #  without debug option, but
  #  with optimization option
END_OF integration;
```

The above example shows that the objects of the C_body type can be handled differently depending on the role they play. Objects of C_body type acting as under-integration in an integration process will be compiled differently from the one described in the C_body type. Likewise , when these objects act as under-test in a test process, they will be compiled differently.

Roles are defined by types. A role type can refer to different types of objects. This allows the integration of many behaviors and properties, coming from different types of objects, in a unique view. By using this concept, Tempo unifies the processing of a heterogeneous set of objects. The advantage of this strategy is that, using the role concept, a set of objects having different static and dynamic characteristics can be perceived in a homogeneous manner during the execu-

tion of a particular software process phase. This homogeneity is maintained by multiple inheritance rules used in the object-oriented models.

For example:

```
test ISA PROCESS;
 ROLE under_test;
  derived_from := C_body;
 METHOD
  compilation;
  # without debug option
 ROLE interfaces;
  derived_from := C_interface,
                  CPP_interface;
 METHOD
  list;
END_OF test;
```

The `C_interface` types (C programming interfaces) and `CPP_interface` (C++ programming interfaces) are specialized in the test process via the "*interfaces*" role. Objects of the "`C_interface`" or "`CPP_interface`" type playing this role will be handled by the methods described in the role. Therefore, the list method can be applied both to the C interfaces and to the C++ interfaces. Many roles can be described by a software process which then becomes a list of roles where every type of object can play different types of roles. As a result, two objects of the same type can be managed in different ways in a software process. Parallel to this, the same object can play different roles in different software processes.

Role discussion

In Tempo, objects can change the way they behavior depending on the context without changing their identity. A role is a template applied to a set of instances sharing the same definition (static and behavioral). A given object can simultaneously be a member of several roles. In other words, an object can simultaneously be shared and play different roles. The role mechanism corresponds thus to the grain size with respect to collaboration support.

The role concept is, in part, supported by Adele's version management system. In order to support concurrent manipulation of software objects, an objet, when playing a role, is duplicated and managed like an object branch. Such duplication is supported by Adele's version management system. In this way, the role concept provides a conceptual framework for version management and the branch mechanism provides the support mechanism of this framework.

By the role concept, it is possible to specify user-defined version management strategies. In some systems, like Nse [18], software performers, when trying to manipulate in parallel a same object in order to improve their productivity, have to follow the strategy provided by the underling version management system. In order words, cooperative work strategies are hard-coded in the core of those systems. In some tools, like Rcs [25], software performers can count only on very basic mechanisms for supporting cooperative work — *ad hod* strategies. Although, in some way cooperative work is supported by such tools, because a same object can be manipulated in parallel by several software performers, cooperative work strategies cannot be described neither automatically enforced in the environment. We belive that the role concept coupled to Adele's version management system provides a solution to these problems. The flexibility of version management tools, like Rcs, is used, i.e., cooperative work strategies are not hard-coded. Moreover, user-defined cooperative work policies can be both described in a model and automatically enforced by Tempo.

4 Communications protocol

Software engineering activities are characterized by a heavy demand for coordination, collaboration and synchronization, since software objects are shared by multiple users. One problem in such a situation is found at the level concerning the control of shared objects. For example, questions such as those listed below must be answered by the SEE:

- When, why and by whom was an object changed?
- How and when must these changes be given to the users who share that object?
- What are the effects of this change?
- In which cases must the modifications be accepted or refused?

These problems have been the object of numerous studies in various fields of research, especially in the database field. To solve them, various mechanisms have been proposed. In the sections below we will show how these problems guided our research, and Tempo's solutions for solving them.

4.1 Cooperation

In order to permit data exchange between users, mechanisms which aid and stimulate collaboration between them must be furnished. The environment must

furnish a communications protocol so that users may be advised of activity status within the environment. Owing to such notifications, users can know when and with whom they must exchange data, or in other words, when and under what conditions they must collaborate with each other. This is only possible when each user can be notified of the status of software processes being used by his colleagues in the environment.

During the life of the software processes, there may be several software process occurrences executing concurrently. Each user must therefore select those environments for which he wants to be notified. The SEE must allow each user to specify the important events needed to complete his activities, according to what is supposed to be accomplished. After taking the notification into account, the user can enter into the data exchange process and thus collaborate. The collaboration process is therefore composed of three steps: notification, decision and data exchange. Each of these steps must be supported by the SEE.

4.2 Synchronization

To ensure that communication between SEE users is controlled, they must be able to synchronize with each other while they develop their activities. Without a synchronizing control mechanism, a SEE cannot ensure that the results exchanged between users is correct. In a programming-in-the-large context, activities have a long duration; it is therefore necessary that users be synchronized as they develop their activities so that results obtained may be integrated. If the SEE does not control concurrent activity synchronization, results may develop such a degree of divergence that they then become impossible to integrate.

Suppose, for example, that two activities of long duration, A1 and A2, execute concurrently in the environment. Suppose also that these two activities simultaneously modify the same object, O. If the two activities are not synchronized during execution, there is the danger that modifications to object O will be impossible to integrate. The SEE must therefore support synchronization between activities of long duration as well as control that synchronization.

4.3 Our proposal for communications support

We have found a lot of work which concerns communication, collaboration, coordination and synchronization with a SEE. We concentrated our attention on the coordination, collaboration and synchronization strategies between software activities. To achieve this, we furnished a concept by which communication strategies between software activities can be described — connection concept.

Connections are used to allow two software process occurrences to become synchronized during execution. The connections are thus a communication channel between two occurrences. Figure 4 gives an example in which two software process occurrences, WE-1 and WE-2, are linked by a connection. By using this connection, the two occurrences can exchange messages during their execution.

By using connections, a software process occurrence can synchronize the sharing of its results with another occurrence which is neither a "*child*" nor a "*parent*" of that software process occurrence. This means that connections allow a software process occurrence to be informed of the status of other software process occurrences, and thus authorize an occurrence to react to those events caused by other occurrences. For example, an update of object O2 in the software process occurrence WE-2 can trigger operations in occurrence WE-1 because these occurrences are connected. Since the TECA rules may be used to reply to these events, the connections can thus be used to support collaboration between two or more software process occurrences.

To furnish a model in which connections and message exchange strategy can be described, we provide connection types. A connection type has the following style:

```
designing ISA CONNECTION;
 DOMAIN
  ModifyDesign:UnderDesign ->
  ReviewDesign:UnderRevision;

 PLUG-ON-RULES . . .
 ACTIVE-RULES . . .
 PLUG-OFF-RULES . . .
END_OF designing;
```

The connection type's domain is provided by the DOMAIN clause. Connections are always binary, meaning that they exist to connect one software process occurrence with another. A connection's granularity level is its role. This means that one connection type describes the connection strategy between one software process type and another, in a role. Connection instances are thus established between the roles of one occurrence and the roles of another occurrence. In the example above, the software process occurrences "`ModifyDesign`" and "`ReviewDesign`" can

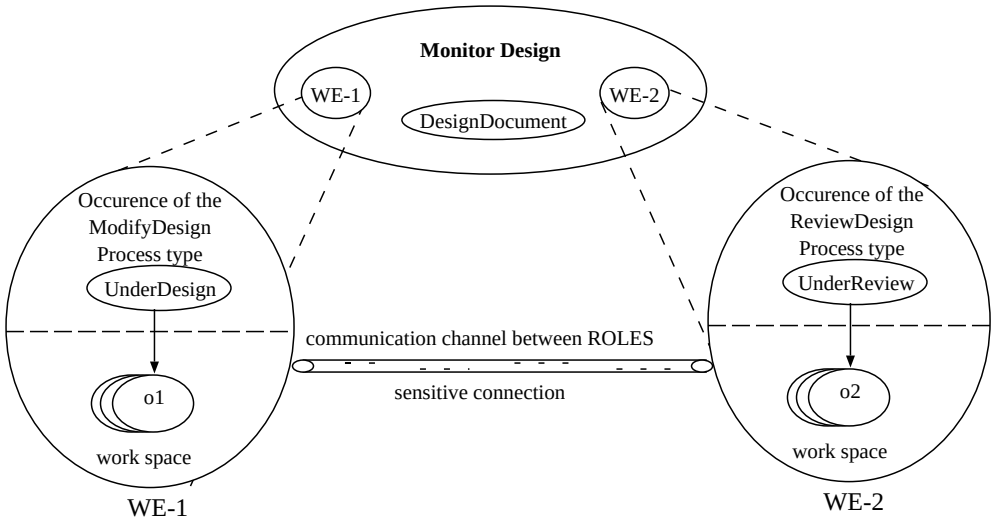

Figure 4: Example of a connection between two software process occurrences.

synchronize themselves and exchange data by means of the `designing` connection. This connection will be established between the roles "`Underdesign`" and "`UnderReview`", respectively.

4.3.1 Connection plug-on rules

The conditions under which two occurrences must be automatically connected are described in the PLUG-ON clause. For example:

```
designing ISA CONNECTION;
 DOMAIN
  ModifyDesign:UnderDesign ->
  ReviewDesign:UnderRevision;
 PLUG-ON-RULES
(1) WHEN createprocess
     UPON (SOURCE OR DEST);
(2) WHEN allocate_ressouces
     UPON (SOURCE OR DEST);
(3) WHEN continue_execution
     UPON (SOURCE OR DEST)
 COLLABORATION-RULES . . .
 PLUG-OFF-RULES . . .
END_OF designing;
```

In the example shown above, a connection of the designing type will automatically be established for the following events:

1. Whenever an occurrence of the software process type "`ReviewDesign`" or "`ModifyDesign`" is created.
2. Whenever new resources are allocated by the roles "`UnderDesign`" or "`UnderReview''`.
3. Finally, whenever the roles "`UnderDesign`" or "`UnderReview`" receive a message allowing them to continue execution.

4.3.2 Connection plug-off rules

In a manner similar to connection plug-on rules, we can describe for each connection type those conditions in which a connection must be broken. These conditions are described in the PLUG-OFF clause. For example:

```
designing ISA CONNECTION;
 DOMAIN
  ModifyDesign:UnderDesign ->
  ReviewDesign:UnderRevision;

 PLUG-ON-RULES
  WHEN createprocess
   UPON (SOURCE OR DEST);

  WHEN allocate_ressouces
   UPON (SOURCE OR DEST);

  WHEN continue_execution
   UPON (SOURCE OR DEST)

 COLLABORATION-RULES . . .

 PLUG-OFF-RULES
```

```
1) WHEN stop_execution
    UPON (SOURCE OR DEST);
2) WHEN finish_execution
    UPON (SOURCE OR DEST);
END_OF designing;
```

This example describes the following plug-off rules:

1. Whenever a message confirms validation of a halt in activity of one of the two connected software occurrences, then the connection between these two occurrences is broken.

2. Similarly, if one of the two cooperating processes terminates its activities, the connection between them is broken.

4.3.3 Collaboration rules

For every connection type we can describe a set of temporal event-condition-action rules which control the data exchange between two software process occurrences. To make this possible, collaboration rules must have access to objects handled for the two occurrences linked by the connection. The connection must also be capable of following operations performed on these objects. This means that an update on objects handled by the two software process occurrences A and B, which are linked by connection C, must trigger events not only in the context of occurrences A and B but also in the context of connection C. The TECA rules defined for this connection therefore deal with these events. For example:

```
designing ISA CONNECTION;
 DOMAIN
  ModifyDesign:UnderDesign ->
   ReviewDesign:UnderRevision;

 PLUG-ON-RULES
  WHEN createprocess
   UPON (SOURCE OR DEST);

  WHEN allocate_resources
   UPON (SOURCE OR DEST);

  WHEN continue_execution
   UPON (SOURCE OR DEST)

 COLLABORATION-RULES
1) WHEN design_completed
    UPON SOURCE
2) DO promote(%source);
3)    allocate(%source,occurenc_of(%dest));
4) WHEN design_reviewed
    UPON DEST
5) DO promote(%dest);
6)    IF (%dest.no_of_changes >= 0) THEN
7)    allocate(%dest,occure_of(%source));

 PLUG-OFF-RULES
  WHEN stop_execution
   UPON (SOURCE OR DEST);

  WHEN finish_execution
   UPON (SOURCE OR DEST);
END_OF designing;
```

The rules described in the COLLABORATION-RULES clause state that:

1. When the modification activity of the design document is completed by the responsible software process occurrence (line 1), the "`design_completed`" event is taken into account.

2. The modifications performed must then be propagated (line 2).

3. This document must be allocated to the software process occurrence undertaking the revision (line 3).

4. Once the revision activity of the design document is completed, the "`design_reviewed`" event is taken into account and processed by this rule (line 4).

5. The results obtained by this revision must be promoted. The promote operation is given for this purpose (line 5).

6. After promoting the revision activity results, verification of corrections is performed on the design document (line 6).

7. If corrections have been made, the design document is automatically allocated to the software process occurrence responsible for its modification (line 7).

The keywords "`ON SOURCE event/ON DEST event`" serve to inform that the operation which started the event event was performed on either the connection's source role or destination role, respectively.

5 Resource coordination: the work environments

In traditional database management systems object coherence must always be ensured by the system. In the software engineering context, where activities are of long duration, it is difficult to require that these objects stay coherent during software process execution [1]. For one thing, such incoherence comes from the integration of different views within a single description. On the other hand, this incoherence stems from the fact that different activities may share the same object over a long period of time. Nonetheless, a SEE must manage this incoherence so as to ensure cooperative, parallel processing during all stages of the software's life cycle [23].

To manage the coherence (or incoherence!) of shared objects, it is necessary to provide mechanisms to coordinate the people who manipulate concurrently those objects. With relational databases, coherence is assured by the concept of transactional atomicity, and coordination is taken into account by the serialization of these transactions. Although this type of mechanism is also necessary in software engineering, it does not provide an adequate solution since in a software engineering environment several concurrent activities share objects over a long period of time. In such a context, the transactional mechanism must be modified and/or extended to meet this new requirement.

5.1 The check-in/check-out model

A lot of work has been done in the field of SEE's to furnish a framework which supports coordination by building mechanisms to manage long transactions [2]. Generally, such work results in models for long transactions similar to the check-in/check-out model [10, 11, 15]. In this model, shared objects are taken from the central database and made available to users in their respective workspaces. Generally a workspace is implemented in the form of a file management system directory [25]. Once in the workspace, the user can modify the shared object with no conflict from other users in the environment who can continue to consult the version available in the central database.

5.2 Our approach

For every software process occurrence, Tempo provides a work environment in which activities are executed, and objects are modified by the use of automated (such as compilers) or interactive (such as text editors) tools, etc.

An object shared by multiple work environments may be modified within each work environment where that object is used. We start from the notion that we can create one or many versions of the same software object (Adele's object database allows this). Once a shared object becomes a target for modification, a new version of this object is created and made available to the user in the work environment where that modification was requested. The modification is made to the new object version, and not to the source object. This new version has a life span limited to that of the work environment in which it is located.

In order to control coherence between long transactions, we require that these transactions be performed in a hierarchical manner, like the one described in [11, 15, 18]. Thus, whenever two work environments wish to share the same object concurrently and modify it, these two environments must use the same root object.

5.3 Example of object sharing

Figure 5 shows an example of sharing a software object. The object O is shared between the software processes WE-1 and WE-2. After placement in the work environments of these two occurrences, object O may undergo updating. The updates are not propagated. That is to say, object O in occurrence WE-1 may be modified without affecting the activities happening in occurrence WE-2, and vice-versa. To render this possible, an alternative to object O is automatically created and made available for every occurrence whenever an update is made to this object. The created alternative is reserved for the work environment which corresponds to the software process occurrence. Figure 5 shows two, alternatives O1 and O2 of object O which are respectively reserved ¿from occurrences WE-1 and WE-2. Once under the control of the WE-1 (WE-2) work environment, the attribute values and the contents of the O1 (O2) alternative from object O can be changed by making revisions when object O1 (O2)'s contents are updated.

When an alternative is created and made available to an occurrence (work environment), it acquires all of the source object's characteristics. The alternative's attributes and contents are therefore identical to those of the source object. Once located within the software process's occurrence workspace, the alternative may be modified by revision controls. The attributes may also be updated locally.

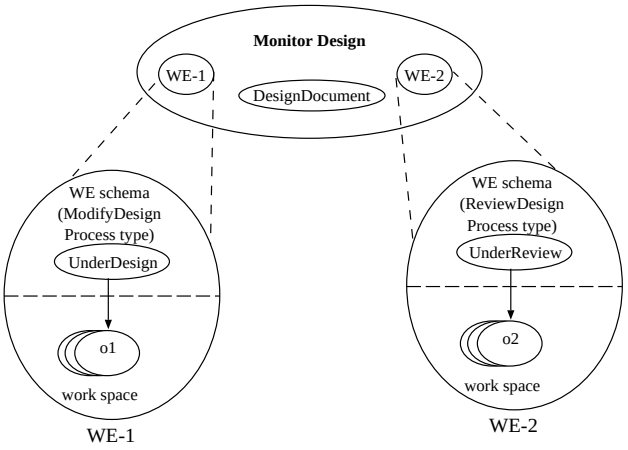

Figure 5: Example of object sharing between work environments.

6 Conclusion

In this paper we have shown how cooperative work is supported in a process-centered software engineering environment. It is based primarily on two components:

1. An object management system for controlling the objects shared by a unique data model which unifies descriptive data and relationships. Such sharing is based on the management of a hierarchy of component versions.

2. An activity manager controlled by an executable formalism which allows software process model descriptions. This model structures activities into basic units known as process types, which become work environments at execution time. The software production process is controlled by temporal event-condition-action rules.

Strategies governing the synchronization and cooperation between different concurrent process occurrences are specified by connections referred to as active and programmable. The communication description strategy is made by rules defining specific synchronization strategies between roles, propagating their effects when an action executes on one of the two connection points.

The temporary modes of TECA rules allow for transactions of long duration, because these can be used to reason on past activities. Coherence control of objects handled by activities of long duration is performed by the work environments. The union between connections and work environments allows for support of the cooperating processes and object sharing between these processes.

Future development and research includes:

1. Realization of an object type ("point and click"), user-friendly, graphic interface for Tempo to enable users to execute activities by means of graphic support.

2. Management of software process evolution. Since software engineering 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 stopping the execution of cooperating processes. Research is this way is going on [8].

We believe that we offer a design context which contributes to clarifying the numerous complex coordination activities found within a SEE. We feel that this will be the challenge for the next years in process-centered SEE's.

Acknowledgements

We would like to express our thanks to Yong-Mi Kim for suggesting substantial and helpful revisions to the original text. The anonymous reviewers and the editor, Prof. Claudia Medeiros, provided very valuable comments and suggestions that helped us improve the paper.

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

References

[1] R. Balzer. Tolerating inconsistency. In *Proc. of the 13th Int'l Conf. on Software Engineering*, pages 158–165, Austin, TX, May 1991. IEEE CS Press.

[2] N. S. Barghouti and G. E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, 1991.

[3] N. Belkhatir. *Nomade : un noyau d'environnement pour la programmation globale*. Thèse de doctorat, INPG, Grenoble, France, 1988.

[4] N. Belkhatir, J. Estublier, and W. L. Melo. Adele 2: a support to large software development process. In M. Dowson, editor, *Proc. of the First Int'l Conf. on the Software Process*, pages 159–170, Redondo Beach, CA, October 21–22 1991. IEEE CS Press.

[5] 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.

[6] 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.

[7] N. Belkhatir and W. L. Melo. Supporting software maintenace processes in Tempo. In *Proc. of the Conf. on Software Maintenance*, pages 21–30, Montreal, Canada, September 1993. IEEE CS Press.

[8] N. Belkhatir and W. L. Melo. Evolving software processes by tailoring the behavior of software objects. In *Proc. of the Conference on Software Maintenance*, Victoria, Canada, September 1994. IEEE CS Press.

[9] N. Belkhatir, W. L. Melo, J. Estublier, and A.-M. Nacer. Supporting software maintenance evolution processes in the Adele system. In C. M Pancake and D. S. Reeves, editors, *Proc. of the 30th Annual ACM Southeast Conf.*, pages 165–172, Raleigh, NC, April 8-10 1992.

[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] P.H. Feiler. Configuration management models in commercial environments. Technical Report CMU/SEI-91-TR-7, Carnegie-Mellon University, Software Enginnering Institure, March 1991.

[12] C. Floyd, F.M. Reisin, and G. Schmidt. STEPS to software development with users. In *Proc. of the 2nd European Software Engineering Conference*, Warwick, England, Septembre 1989.

[13] S. Gatziu, A. Geppert, and K. R. Dittrich. Integrating active concepts into an object-oriented database systems. In *Proc. of the 3rd Int'l Workshop on Database Programming Languages: Bulk Types & Persistent Data*, pages 399–415, Naflplion, 1991. Morgan Kaufmann.

[14] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In M. Stonebraker, editor, *Proc. of the ACM SIGMOD 92*, volume 21, no. 2 of *ACM SIGMOD Record*, pages 81–90. ACM Press, June 1992.

[15] G. E. Kaiser. A flexible transaction model for software engineering. In *Proc. of the 6th Int'l Conf. on Data Engineering*, pages 560–567, Los alamitos, CA, 1990. IEEE CS Press.

[16] 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.

[17] W. L. Melo. *Tempo: Un environnement de développement Logiciel Centré Procédés de Fabrication.* Thèse de Doctorat, Université Joseph Fourier (Grenoble I), Laboratoire de Génie Informatique, Grenoble, France, 22 de Octobre 1993.

[18] T. Miller. Configuration management with the NSE. In Long [16], pages 99–106.

[19] K. Narayanaswamy. Enactment in a process-centered softwre engineering environment. In W. Schafer, editor, *Proc. of the 8th Int'l Software Process Workshop*, Germany, 1993. IEEE CS Press.

[20] F. Oquendo, J.-D. Zucker, and G. Tassart. Support for software tool integration and process-centered software engineering environments. In *Proc. of the 3rd Int'l Workshop on Software Engineering and its Applications*, pages 135–155, Toulouse, France, December 3–7 1990.

[21] C.V. Romamoorthy. Programming in the large. *IEEE Transactions on Software Engineering*, 12(7):1145–1154, July 1986.

[22] 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.

[23] R. W. Schwanke and G. E. Kaiser. Living with inconsistency in large systems. In J. F. H. Winkler, editor, *Int'l Workshop on Software Version and Configuration Control*, Grassau, Germany, January 27–29 1988. B. G. Teubner, Stuttgart, 1988.

[24] 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.

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

[26] A. I. Wasserman. Tool integration in software engineering environments. In Long [16].