Paper published in the Conference on Software Maintenance, Montreal, Canada, Semptembre 1993. IEEE Press.

Supporting Software Maintenance Processes in TEMPO

N. Belkhatir and W. L. Melo
LGI BP 53
38041 Grenoble Cedex 9 France

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 formalisme used to define software engineering activities, and we will show how contraints related to the use of objects in these activities are expressed using the role concept. We will then present our communication model. Thanks to 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 of the cooperating processes and object sharing between these processes.

Keywords: Software process, software engineering environment, communication, cooperation, synchronization, triggers, object viewpoints;

1.0 Introduction

The problems of developing large volume software are well-known. They can be classified as programming-in-the-small [11], programming-in-the-large [22] and programming-in-the-many. By programming-in-the-small we mean the development activities associated with someone who develops a module or program alone. Programming-in-the-large means development activities involving many components, and programming-in-the-many refers to software development activities involving several agents. Research has focused for some time on the first aspect with the development of the programming environments, and on the second aspect which is concerned with version and configuration management. 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. Experimental studies show that 70% to 80% of software engineering activities are based on communication and collaboration between development team members working on the same project. Such cooperation should be integrated into a software engineering environment (SEE) to offer a conceptual framework where activities involving software engineering, 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 formalisme and then implemented in the environment by a process-centered SEE [3] [6] [7] [8] [21].

Based on Adele [3], an environment for programming-in-the-large which supports software product structuring and objects versions, we started the TEMPO [5] project in order to take into account the production and evolution strategies of software systems. TEMPO is a SEE piloted by an executable formalisme which allows description of software process models, object views, and elaboration on the strategies of cooperation, communication [4]. In the following text we will stress those aspects of TEMPO which relate to cooperation, 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 with the role concept Many roles may be called on by activities which execute concurrently (cooperating processes). An activity takes place within a context called a work environment. The work environment is a unity of regrouped roles strongly linked together 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.

2.0 An overview of TEMPO

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

1. A resource manager using Adele as a persistent object base for storing objects and activities and for tracing the project's progress.
2. 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.

2.1 The software process model

TEMPO [5] 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 specialisation/generalisation 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.

2.2 The temporal contraints

Software process enacting is controlled by temporal contraints. To do so, a design structure must be provided which allows tracing and rendering persistent any previous states, and then describe the temporal contraints and verify them during software processes enacting execution.

2.2.1 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 [20], Damokles [12] and HiPAC [10] 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
```

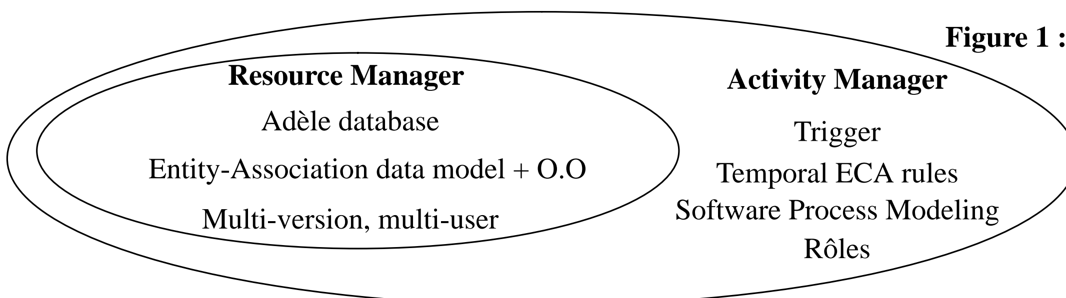


Figure 1 : Overview of TEMPO

```
        DO stop_execution;
    (2)PRE WHEN continue_execution
        IFPAST not deadline_changed
          FROM last(deadline_arrived) UNTIL
    now()
        DO ABORT;
    END_OF ModifyDesign;
```

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

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.

2.2.2 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}
   méthode
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.

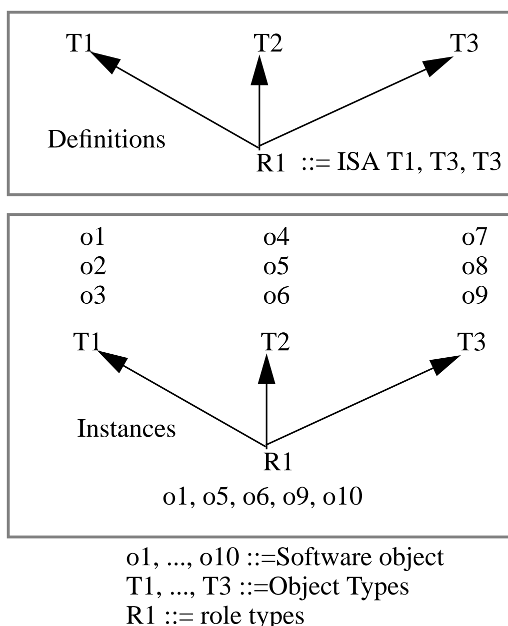
An action performed on object X will execute the triggers defined in the relationship where object X is the source (identified by the keyword ORIGIN), and the triggers defined in the relationship where object X is the destination (identified by the keyword DEST).

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.

2.3 Object 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].



o1, ..., o10 ::=Software object
T1, ..., T3 ::=Object Types
R1 ::= role types

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;
```

3.0 Resource coordination: the work environments

In classical database management systms 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 [24].

To manage the coherence (or incoherence!) of shared objects, it is necessary to provide mechanisms to coordinate the users of those objects. With relational databases, coherence is assured by the concept of transactional atomicity, and coordination is taken into account by the serialisation of these transactions. Although this type of mechanism is also necessary in software engineering, it does not provide an adequate solution since we find ourselves in a context where 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.

3.1 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] [9] [18] [21] [23]. Generally, such work results in models for long transactions similar to the check-in/check-out model [13] [16] [17]. 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.

3.2 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 [16] [17] and [19]. Thus, whenever two work environments wish to share the same object concurrently and modify it, these two environments must use the same root object.

3.3 Example of sharing

The following diagram 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 2 shows two, alternatives O.1 and O.2 of object O which are respectively reserved from occurrences WE-1 and WE-2.

When an alternative is created and made available to an occurrence, 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.

3.3.1 Branch management

Figure 3 depicts a scenario where software object O is shared by two software process occurrences, WE-1 and WE-2. The source object stored in the database has three revisions. The status attribute in the last revision is equal to not_revised. When this object is modified in the WE-1 work environment, an alternative (O.1) is created and placed under its control. In this example, the WE-1 work environment alternative acquires the contents of the last revision, as well as the status attribute value, from the database. Once under the control of the WE-1 work environment, the attribute values and the contents of the O.1 alternative from object O can be changed by making revisions when object O.1's contents are updated.

In a similar fashion, an alternative for object O is produced and made available to the WE-2 work environment whenever an update is requested by the user responsible for that environment. In a similar fashion to the WE-1 work environment, alternative O.2 from object O can change independently from alternative O.1 and source object O.

This solution allows a group of users to share a set of software objects. Each work environment belongs to a software process type; the activities which take place in the work environment are thus controlled by the descriptions providec within this type.

3.3.2 Partitioning a work environment

A work environment may be successively partitioned into several sub-environments by respecting the partitioning rules described in the software process types. Each "son" work environment may initially handle only one object subset from its "father" work environment.

The following diagram gives an example where the WE-1 work environment is partitioned into two working sub-environments. Consequently, the sub-environments WE-1/a and WE-1/b may only handle alternative O.1 from object O. An alternative is thus created for each sub-environment whenever an update operation is performed within the context of these sub-environments.

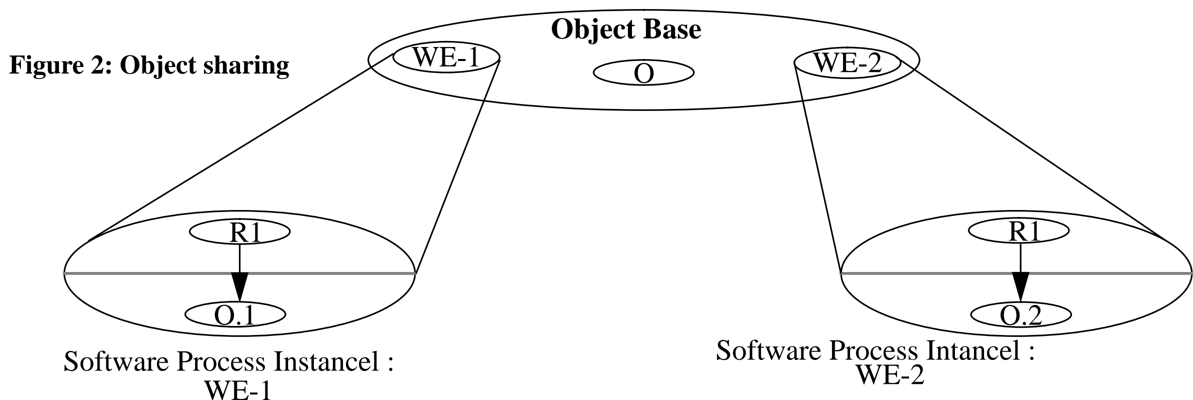
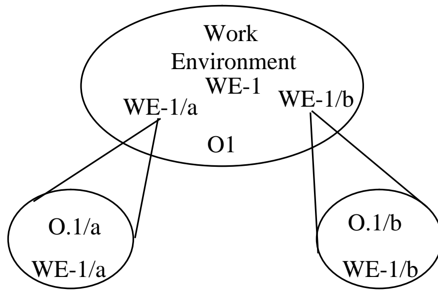


Figure 2: Object sharing



According to the procedure for creating alternatives, described in the previous paragraph, the new alternatives O.1/a and O.1/b acquire the attributes and contents of object O.

3.3.3 Unifying the work environments

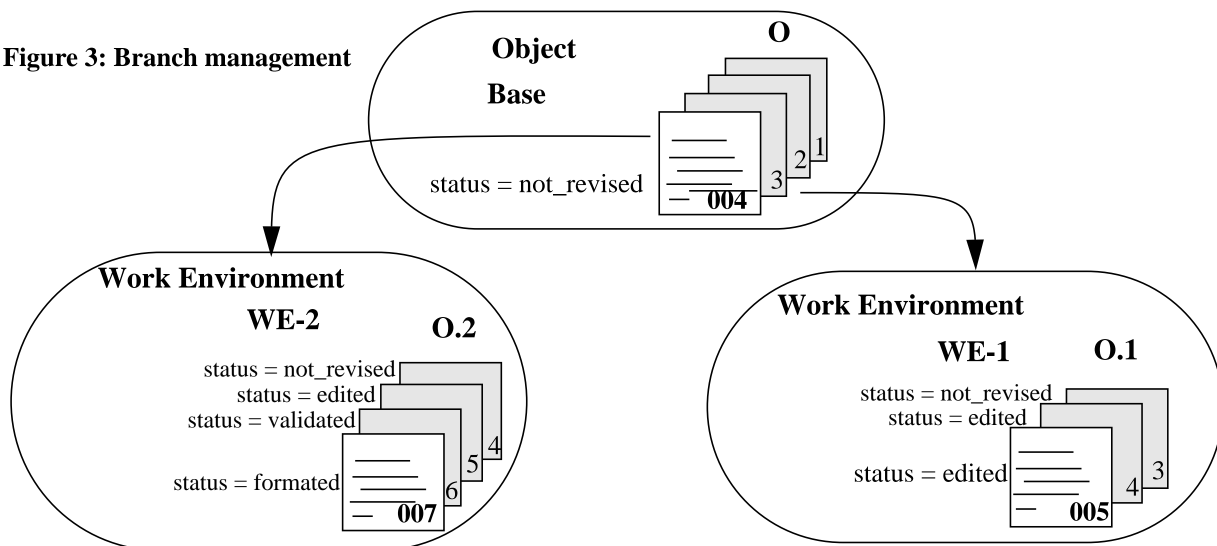
Modifications performed on objects in a "son" work environment may be transferred to the "father" work environment whenever the user so wishes. This creates a problem for integrating the results between the work environments. In other words, whenever the users attempt to propagate modifications performed on shared objects, incoherences may appear. As opposed to standard approaches where a strategy is generally imposed to account for such problems, we have provided TEMPO with a solution by which the user himself may define what the reaction should be, in case of a conflict. By using the temporal event-condition-action rules, the integration strategy the SEE must follow can be described.

Let's suppose that the WE-1 work environment is an occurrence of the Monitor-Design software process type, and that the WE-1/a and WE-1/b work environments are, respectively, occurrences of the ModifyDataFlowDesign and ModifyControlFlowDesign software process types. Let's also suppose that the WE-1/a environment was created for modifying the data flow specifications of a module's design, and WE-1/b for updating the control flow specifications.

By using the TECA rules defined in both the role and software process types, the unification strategy for results, to be used by the work environments, can be described.

```
design_documment ISA objet;
    ATTRIBUTE
        status = designed, reviewed, edited,
none := none;
        no_of_changes = INTEGER := 0;
END_OF design_documment;

ModifyDesign ISA PROCESS;
    ATTRIBUTES
        begin_date = DATE := now();
        end_date = DATE;
        deadline = DATE;
    RULES
        AFTER WHEN deadline_arrived
            DO stop_execution;
        PRE WHEN continue_execution
            IFPAST not deadline_changed
              FROM last(deadline_arrived) UNTIL
now()
            DO ABORT;
END_OF ModifyDesign;

ModifyDataFlowDesign ISA ModifyDesign;
    ROLE df;
        derived_from = desgin_document;
END_OF ModifyDataFlowDesign;

ModifyControlFlowDesign ISA ModifyDesign;
    ROLE cf;
        derived_form = design_documment;
END_OF ModifyControlFlowDesign;

MonitorDesign ISA PROCESS;
    RULES
        WHEN promote UPON under_design
        IFPAST last(%mdf.df.status) ==
```

Figure 3: Branch management



```
designed OR
              last(%mcf.df.status) ==
designed
       DO
change_attr(%under_design,status,designed);
    CONTROL mdf;
       sub = ModifyDataFlowDesign;
    CONTROL mcf;
       sub = ModifyControlFlowDesign;
    ROLE under_design;
       derived_from = design_docmment;
END_OF MonitorDesign;
```

In the MonitorDesign type, a rule is defined to control the unification of results from activities concerning data flow specification modifications and control flow modifications. This rule imposes the following restriction:

The design document will be considered as being coherent whenever the two work environments (WE-1/a and WE-1/b) have modified, respectively, the data flow and control flow specifications.

4.0 Communications protocol

Software engineering activities are characterised 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 caused by 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. Thanks 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 software process execution, there may be a sizeable number of software processes 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 SEE supportable.

4.2 Synchronization

To ensure that communication between SEE users is controlled, they must be able to synchronise to each other while they develop their activities. Without a synchronising mechanism to help, 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 synchronised 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 synchronised 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.

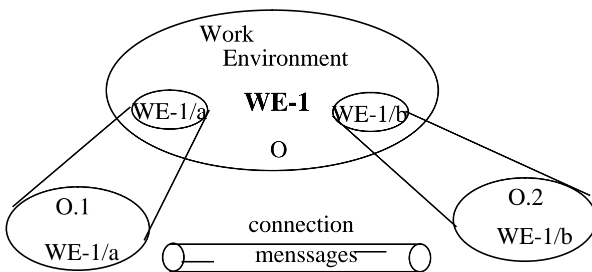
4.3 Our proposal for communications support

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 these activities. To achieve this, we furnished the concept of a connection by which the communications protocol between software process occurrences can be described, thus allowing these activities to become synchronised and contributing to an increase in the level of cooperation and collaboration between TEMPO users.

Connections are used to allow two software process occurrences to become synchronised during execution. The connections are thus a communication channel

between two occurrences. The following diagram gives an example for the occurrences WE-1/a and WE-1/b. By using the connection, the two occurrences can exchange messages during their execution.

By using connections, a software process occurrence can synchronise the sharing of its results with another occurrence which is neither a “son” nor a “father”. This means that connections allow a software process occurrence to be informed of the status of other software process occurrences, and thus authorise an occurrence to react to those events caused by other occurrences. For example, an update of object O.2 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 design database where the connections and the message exchange strategy can be described, we make a connection type. 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 synchronise 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)
   ACTIVE-RULES . . .
   PLUG-OFF-RULES . . .
END_OF designing;
```

In the example shown above, a connection of the design 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)
   ACTIVE-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 occur-

rences, 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 permit 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 provoke 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_ressouces
           UPON (SOURCE OR DEST);
     WHEN continue_execution
           UPON (SOURCE OR DEST)
    CONTROL-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 ACTIVE-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 (line3).
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.

5.0 Conclusion

In this paper we've shown how cooperative work is supported. 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 formalisme 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.

TEMPO is a research prototype conducted within the context of the Adele Project. It is implemented above the Adele database.

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. Work concerning this aspect is described in [15].

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 ten years in process-centered SEE's.

6.0 References

[1] R. Balzer. Tolerating inconsistency. In *Proc. of the 13th Int'l Conf. on Soft. Eng.g*, pp 158–165, Austin, 1991.

[2] N. S. Barghouti. Supporting cooperation in the Marvel process-centered SDE. In vol. 17 of *ACM SIGSOFT Software Engineering Notes*, pp 21–31. 1992.

[3] N. Belkhatir, J. Estublier, and W. L. Melo. Adele 2: a support to large software development process. In Dowson [14], pages 159–170.

[4] N. Belkhatir, J. Estublier, and W. L. Melo. Software process model and work space control in the Adele system. In *2nd Int'l Conf. on the Software Process*, pp 2–11, , Germany, Feb. 1993.

[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 Soft. Eng. & its Applications*, pages 733–742, Toulouse, France, Dec. 1992.

[6] K. Benali, *et. al*. Presentation of the Alf projet. In *Proc. of the 1st Int'l Conf. on System Development Env. and Factories*, pages 75–90, Berlin, May 1989.

[7] R.F. Bruynooghe, *et. al*. PSS: a system for process enactment. In Dowson [14], pp 128–141.

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

[9] W. Courington. *The Network Software Environment*. Sun Microsystems, Inc, 1989.

[10] U. Dayal *et al*. The HiPAC project: combining active databases and timing constraints. *ACM SIGMOD Record*, 17(1):51–70, March 1988.

[11] F. DeRemer and H. Kron. Programming-in-the-large verus programming in the small. *IEEE TOSE*, 2:80–86, June 1976.

[12] K.R. Dittrich. The Damokles database system for design applications: its past, its present, and its future. In *Soft. Eng. Environments: Research and Practice*, pages 151–171. Ellis Horwood Books, Durhan, 1989.

[13] K.R. Dittrich, W. Gotthard, and P. C. Lockemann. Damokles – a database system for software engineering environments. In vol. 244 of *LNCS*, pp 353–371. 1987.

[14] M. Dowson, editor. *Proc. of the First Int'l Conf. on the Software Process*, Redondo Beach, CA, Oct. 1991.

[15] J. Estublier, *et. al*. Support a l'evolution des procedes de production logiciel dans un AGL centré processus. TR, L. G. I., July 1993.

[16] P.H. Feiler. Configuration management models in commercial environments. CMU/SEI-91-TR-7, March 1991.

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

[18] W. Kim, N. Ballou J.F. Garza, and D. Woelk. A distributed object-oriented database system supporting shared and private databases. *ACM TOIS*, 9(1):31–51, January 1991.

[19] T. Miller. Configuration management with the NSE. In vol. 467 of *LNCS*, pages 99–106. Springer-Verlag, Berlin, 1990.

[20] F. Oquendo,*et. al.*. Support for software tool integration and process-centered software engineering environments. In *Proc. of the 3rd Int'l Workshop on Soft. Eng. and its Applications*, pages 135–155, Toulouse, , Dec.1990.

[21] B. Peuschel, *et. al.*A knowledge-based software development environment supporting cooperative work. *Int'l Journal of SEKE*, 2(1):79–1–6, March 1992 1992.

[22] C.V. Romamoorthy. Programming in the large. *IEEE TOSE*, 12(7):1145–1154, July 1986.

[23] S. Sarkar and V. Venugopal. Transaction mechanisms for software environment databases. In *24th Hawaii Int'l Conf. on System Sciences*, pages 511–518, Kona, 1991.

[24] R. W. Schwanke and G. E. Kaiser. Living with inconsistency in large systems. In *Int'l Workshop on Software Version and Configuration Control*, Grassau, Germany, Jan. 1988. B. G. Teubner, Stuttgart, 1988.

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