TEMPO and SOCCA

Concepts, modelling and comparison

by Richard Willemsen

01-05-1995

Department of Computer Science

University of Leiden

Abstract

In this thesis two software process modelling formalisms, TEMPO and SOCCA, will be discussed and compared. The TEMPO formalism has been developed at the Laboratoire de Génie Informatique, Université Joseph Fourier, Grenoble (France) and the SOCCA formalism has been developed at the University of Leiden (The Netherlands). During the modelling of some examples in both formalisms, several interesting aspects came into light. This has resulted into some extensions of the SOCCA formalism. Further more several role concepts found in the different software process modelling approaches will be presented and a proposition for the role concept in SOCCA will be given. The last part of the thesis describes the R.A.P.P. diagram. This is a diagram which can possibly be used on top of SOCCA (or other formalisms), to describe software processes on a more global level of abstraction.

Contents

Chapter 1 General Introduction

Software processes can be regarded as a set of activities, that are concurrently carried out by several agents. These agents are human as well as non-human agents (tools, document, etc.). The agents have to cooperate and to communicate with each other in order to develop software applications, deliver new releases, etc. Despite the importance of the description of the (software) process, little attention has been paid to describe and clarify these processes. By providing an explicit process description, weaknesses in the process can be discovered and corrected. Such a process description is also very useful for analysing, simulating and evaluating the (software) process. These descriptions of the processes are subject of research in the *Software Process Modelling* field, which is a relatively new branch of the software engineering tree. The purpose of developing software process modelling formalisms is to describe and clarify the (software) processes. The two software process modelling approaches which will be discussed in this thesis are: TEMPO and SOCCA.

The TEMPO formalism has been developed at the Laboratoire de Génie Informatique of the Université Joseph Fourier in Grenoble (France). Due to the fact, that the ADELE system had some drawbacks, TEMPO has been designed on top of the ADELE system to overcome these weaknesses. TEMPO is high level software process programming language based on the role concept and the connection concept. A role [3] allows to redefine dynamically the static and behavioural properties of objects depending on where the objects play their role in the process. The connection expresses how the different (sub) processes collaborate with each other.

The SOCCA formalism has been developed at the University of Leiden (The Netherlands). This specification formalism for software process modelling has not only been developed for describing the technical parts of the software process, but also for the human parts, or rather the human team members, of the software process. The formalism should reflect all kinds of interaction between the various parts, including the non-human as well as the human parts.

The purposes of my stage (research) were:

1. To understand and finally to compare the two software process modelling formalisms TEMPO and SOCCA with each other. This resulted in visiting the Laboratoire de Génie Informatique in Grenoble for three months, where I studied the TEMPO formalism. The last four months I stayed in Leiden where I studied the SOCCA formalism. During my staying in Grenoble a scenario has been invented with the purpose to use it as a frame work for comparing the two approaches. If one wants to compare two different formalisms, it is necessary to give them more attention then just reading the articles. This resulted in very interesting and difficult discussions with Jacky, Noureddine and Luuk, all having a different approach of encountering the modelling of software processes. The scenario has been used for modelling some examples in both formalisms, which made it easier to compare them with each other.

2. To study the role concept used in TEMPO and to try to apply a role concept in the SOCCA formalism. But before I wanted to propose a role concept for SOCCA, I have studied the role concepts used in some other software process modelling approaches [5, 6, 8, 14, 15, 16, 17]. This resulted into a proposition of a role concept for the SOCCA formalism. This role concept differs from other role concepts, because it's the only one which is able to restrict the extent of a role.

Before presenting the structure of this thesis I want to thank some people.

Je remercie Jacky Estublier pour m'avoir admis au sein de son équipe. Mes remerciements vont également à Noureddine Belkhatir qui m'a beaucoup aidé et souvent orienté dans la bonne direction. Qu'il me soit permis enfin d'adresser mes sincères remerciements à toute l'equipe.

I want to thank Luuk for his enthusiasm, positive attitude and the interesting discussions we had together. It was a really nice and stimulating experience to work with you, thanks!

Further I want to thank my family and friends who have supported me during my study.

This thesis has been organized as follows. After the general introduction in Chapter 1, Chapter 2 presents the TEMPO formalism with its concepts. This formalism is based upon the ADELE system, the role concept and the connection concept. In this chapter some examples which have been based upon a scenario (also presented here) and modelled in TEMPO will be given. In Chapter 3 the concepts from which SOCCA is composed are described. These concepts are object oriented, based on EER modelling, state transition diagrams combined with PARADIGM, and object flow diagrams. The examples which have been modelled in TEMPO, have been modelled in SOCCA too. During the modelling of these examples several extensions of the SOCCA formalism have been made. Chapter 4 presents various role concepts found in the different software process modelling approaches. At the end of the chapter, a role concept has been proposed and applied to the SOCCA formalism. The comparison and evaluation of TEMPO and SOCCA is subject of Chapter 5. In Chapter 6, the R.A.P.P. diagram is presented. This is a diagram which can possibly be used on top of SOCCA (or other formalisms), to describe software processes in a more global way. In Chapter 7 some conclusions and topics for future work are listed.

Chapter 2 The TEMPO Formalism

2.1 Introduction into TEMPO

The ADELE system has been in practical use for several years. However some weaknesses have been found:

1. Concept level: There is no high level concept such as software process steps, work environments.

2. Complexity: The ECA rules are fragmented among data and relation types. A clear picture of what will happen during execution is not easy.

To overcome these drawbacks, a higher level process language called TEMPO has been developed on top of the ADELE system. In this thesis the TEMPO syntax is used as described in [16]. In TEMPO the concept of process and subprocess is used. A software process can be modelled as a combination of processes and subprocesses (also called process steps). In a process step there are one or more users and a set of objects on which the users perform there actions. In TEMPO a user is defined as an object, so each process or subprocess consists of a set of objects. In a process step roles are defined to describe the object contextual behaviour, i.e. the description of the operations that can be done on the object and the rules that control these operations. A role adds temporary properties (= local attributes and methods) to the object playing this role. Each of these roles is a set of object instances sharing the same static and dynamic description. In a certain process, an object instance can play a single role, and objects of the same type can be used by different roles. TEMPO uses the TECA formalism to describe temporal events, extending the ECA formalism of ADELE with temporal operators. The ECA and TECA formalisms will be described in the following subparagraphs.

2.2 The Concept of ECA Rules in ADELE

To model software processes, the formalism of event-condition-action is used to describe the dynamic aspects.

The general expression for ECA rules is: ‘ON Event WHEN Condition DO Action’. The events are used to control the activities of the objects. The action is a set of operations activated by a trigger when an event occurs.

The ECA formalism has two basic concepts: 1. Event-Condition and 2. Action.

1. Events and Conditions

Events arise when methods or commands are executed. The events are used to control the activities in the database and the conditions are used for some constraints. The event is a complex expression, involving the method (or command) which caused the event, temporal conditions and external conditions. In the next example the definition of an event will be shown and explained.

E.g. DEFEVENT Delete_obj = (!cmd = rmobj).

The Delete_obj event is defined as being the event which raises, whenever the current command (!cmd) is an object removal command (rmobj).

2. Actions

An action is a program written in the ADELE language. This is an imperative language, an instruction can be a logical expression, an ADELE command or UNIX command. The action can call methods, and these methods will be executed as subtransactions.

2.3 The Concept of TECA Rules in TEMPO

The ECA concept doesn't deal with time, but in TEMPO it is possible to control the activities during time. So instead of the 'traditional' ECA rules, TEMPO is describing the activities with TECA (= Temporal-Event-Condition-Action) rules. These TECA rules are used to describe the order in which the activities are executed, and their synchronization.

A TECA rule is expressed in the following way: WHEN Event DO Method,

Event: An event is a predicate which expresses an event in the present or in the past of the system or of the objects.

Method: A method is a program written in an imperative language.

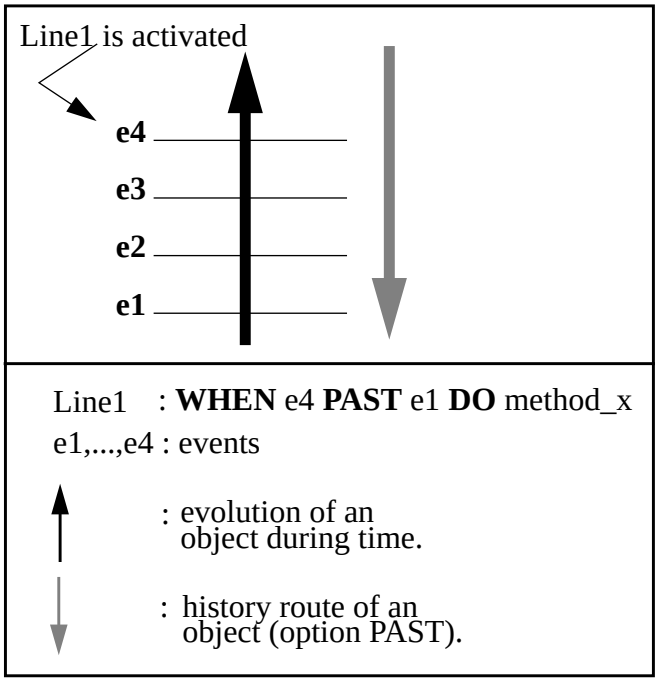
In TEMPO the operator 'PAST' is added, because that operator makes it possible to express the conditions in the past. The events and the attributes are saved in the history of an object, with the operator PAST it is possible to check if a certain event has happened in the past.

This will be clarified by the following example. When event e4 arises Line1 is activated:

Line1: WHEN e4 **PAST** e1 **DO** method_x.

The history route of the object for which the event e4 is raised, is traversed. This is necessary for the verifying of event e1 (has e1 been raised before ?(the clause PAST)). If the event e1 is registered in the history of the object, the method_x will be executed.

E.g.



2.4 The Trigger Formalism

First of all it must be clear that the trigger mechanism is used to implement the TECA concept. A TECA line is executed, when the corresponding event is true by the trigger mechanism of ADELE. The trigger mechanism allows actions to be executed automatically when some conditions hold (e.g. preconditions and postconditions). There exists four execution modes for the TECA lines: 1. PRE, 2. POST, 3. AFTER and 4. EXCEPTION (= ERROR mode in ADELE)

2.4.1 The Execution Model of Triggers

When a method is called, a transaction is opened by the system to execute the method. The PRE triggers and POST triggers are executed as part of the transaction and the AFTER triggers and ERROR triggers are executed after the transaction (see Fig. 2.1.). It must be clear, that not all the execution modes have to be used for the same object. It depends on the situation and on the object, which execution modes will be used.

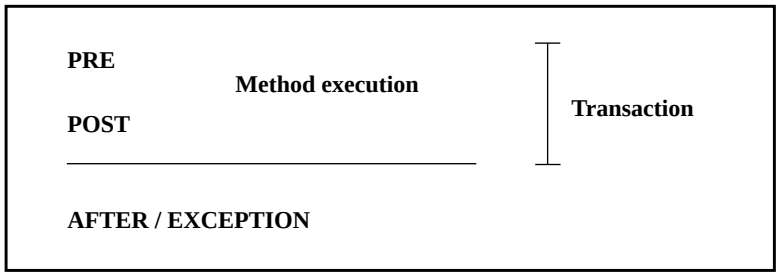


Fig. 2.1. The Execution model

1. PRE Triggers

The pre triggers are activated before the method execution. These pre triggers enable the system to check the system state and the state of the objects before the method is executed, e.g. if someone wants to modify a document, but he has no access right to the document, an ABORTION will be raised.

2. POST Triggers

The post triggers are executed after the method execution. The events are checked after the method. When the events of a post trigger is true, the post trigger is executed immediately, but before the transaction has been commitment. These triggers allow to undo (=rollback) the modifications performed by the operations of the method, e.g. if an attribute of an object has been changed and it's not allowed to change that attribute, then the operation will be aborted and the modifications are undone.

3. AFTER Triggers

The after triggers are executed after the commitment of a transaction, e.g. they can be used for recording historic information in the database.

4. EXCEPTION Triggers

The exception triggers are executed when the transaction has been aborted. These triggers can be used for an alternative strategy, e.g. if someone didn't have the permission to access an object, this user can be given the access right to that specific object now.

2.4.2 The Execution Order of Triggers

The execution of a trigger depends on the event which is always associated to a trigger. The trigger is only executed when the corresponding event is true. If two or more triggers which may run are selected, there must be made a decision which trigger has the permission to run first. In ADELE this is done in the following way. The ADELE system considers the priority which is attached to each event. The triggers which are associated to true events and with the highest priorities have to be executed first. When there are two events (or more) true, and they have the same priority, the order in which they are run is not defined.

2.5 The TEMPO Process Formalism

TEMPO is a software process programming language based on the role concept. Different software processes can share simultaneously the same objects. In each process the objects play a different role, so the object behaviour is context dependent.

TEMPO defines a process model, based on two concepts: 1. role and 2. connection.

1. **Role**: A role enables to change the static (=attributes) and behavioural (= methods) properties of objects according to the role they play in a process (= contextual behaviour).
2. **Connection**: A connection describes how the different processes collaborate with each other. A connection is a relationship between two roles.

A software process model is described as a combination of software process types. To identify and describe a set of activities a **process type** is used. A process type can be refined and specialized. It is possible to modify and overload the attributes, methods and constraints of an object type when used in a process type. A software process instance is carried out by one or more users in a Work Environment (= WE). The WE will be described in paragraph 2.6. Because of the fact, that software processes are activities executing asynchronously and concurrently, TEMPO describes the communication and synchronization protocol by temporal-event-condition-action rules. The collaboration protocol is described by a connection.

2.6 The Work Environment

TEMPO associates a work environment (= WE) to each process. A Work Environment is defined by the following tuple:

WE = (WS, PM, Tools, User),

WS: Work Space, the work space is the ‘private’ space, where users can perform software processes. The user in a WS is isolated from other WSs, in the WS he can do some activities like, e.g. designing, compiling, editing, etc.

PM: Process Model, the process model specifies what the process can do in the work space.

Tools: These tools are used in the work environment to manipulate the work space objects.

User: The user(s) who are allowed to work in this work environment.

To adapt the behaviour of objects to the corresponding work environment context, the role concept is used. A role of an object can redefine the original attributes and methods or define new ones, in order to satisfy the need of the work environment, to let the object behave according to the context in which it is used. A user is allowed to work in different WEs at the same time and the WEs may be used by different users simultaneously.

2.6.1 The Decomposition of a WE

A work environment can be decomposed into several sub (= child) work environments. Each child work environment is not allowed to manipulate the objects in their corresponding parent work environment and each child work environment corresponds with a fragment instance defined in a process type. In Fig. 2.2. an example of the work environment Monitor, which is decomposed into two sub work environments (= Designing1 and Reviewing1) is shown. The WE-Designing1 is an instance of the type Designing. The WE-Designing1 is a child of the WE-Monitor. This means that, Designing1 corresponds with the fragment Design appearing in WE-Monitor. The WE-Reviewing1 is an instance of the type Reviewing. This work environment corresponds with the fragment review in his parent work environment Monitor. When the sub work environments are created, each child gets a copy of the document doc, which is an object in the parent work environment. So doc-01 in WE-Designing1 and doc-02 in WE-Reviewing1 inherit the attributes and content of the object doc in the parent work environment. Each document in the child work environments, behaves according to the role under which it has to operate. So document doc-01 in sub work environment Designing1 has the underdesign role and the document doc-02 in sub work environment Reviewing1 has the role underreview. The roles underdesign and underreview, describe how the documents (doc-01 and doc-02) have to behave.

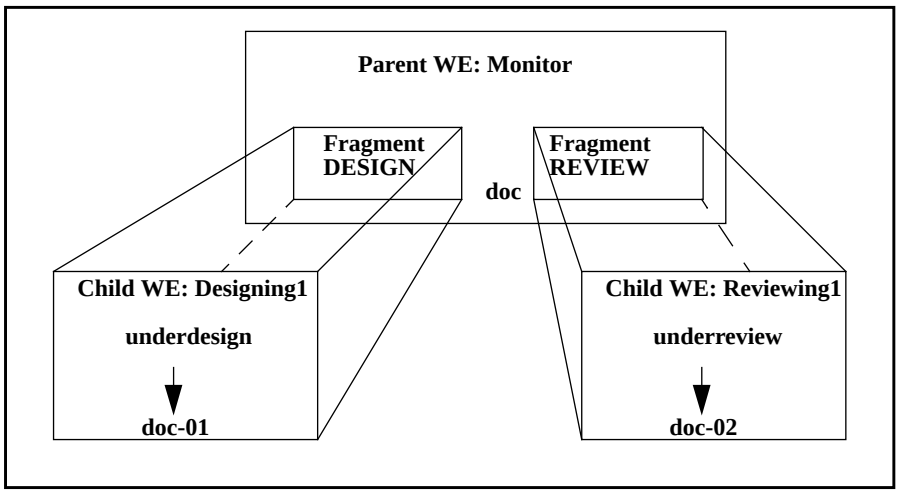


Fig. 2.2. Decomposition of WEs

2.6.2 Promoting Objects from Child WE to Parent WE

The user in a child work environment can modify his objects in isolation from other users. When the user decides to export his modified objects to the parent work environment, an integration problem can occur. This means, when the user tries to promote his modifications on the shared object, then the consistency problem will arise.

To resolve this problem, in TEMPO a user has to react by himself to the eventually conflict situations. He can write, using the TECA rules, which integration strategy has to be followed.

2.7 The Role Concept

Depending on the process where an object is used, the role of the object is different. When a process type uses a role type, this role type makes it possible to change the definition of the attributes, methods and constraints. With other words, within a process type, the role type is a new type definition of objects (properties and behaviour).

2.7.1 The Role Definition

A role is the set of object instances, having the same characteristics (= attributes) and behaviour (= methods and constraints). An object instance can only play one role in a process!, but the same object is allowed to play several roles in different processes. The syntax of a role [16] has the following structure:

The Role Syntax:

```
ROLE RoleName;
        mode := { modifiable / visualizable };
        card := Min, Max;
        derived = ObjectType;
        ATTRIBUTE    Attributes;
        METHOD       Methods;
        RULES        Rules;
```

1. RoleName = name of the role.

2. mode = the mode describes in what way the objects are used by a role. When a role has the mode modifiable, it means, that the objects can be manipulated and modified by the process step where they play a role. When the role has the mode visualized, the objects that play a role in a process cannot be modified by the process.

3. derived = gives the object type from where the role Rolename is derived.

4. card = the constructor provides a role with the cardinality (min,max). So it's possible to define the number of instances of each role.

5. ATTRIBUTE = list of attributes of the role.

6. METHOD = the list of methods of the role.

7. RULES = the list of rules of the role.

There is no strict relationship between a role and an object type, see also Fig. 2.3.:

i. An object instance plays a single role in a process.

ii. Object instances of the same type may play different roles.

iii. Instances of different types may play the same role, provided their types are compatible (is not possible in this first specification of TEMPO).

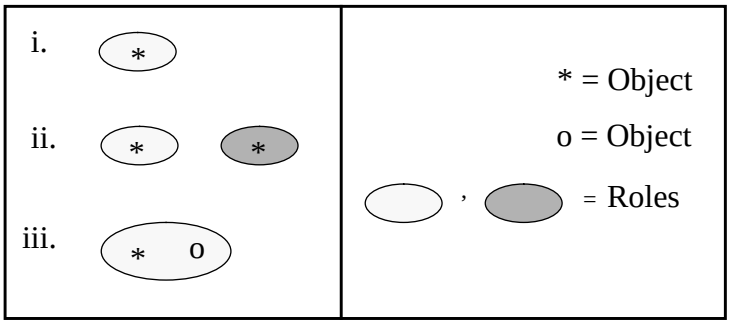


Fig. 2.3. Role and Object Type

2.7.2 The Contextual Behaviour

A role is very useful to let objects adapt their behaviour according to the environment where the objects are used. Each role can redefine the original attributes and methods or the role can define new attributes and methods. The role concept makes it possible to accommodate the original behaviour of the object to the behaviour of a work environment context. For example, the module type has methods, which are independent of the context in which they are used. The process test may test a module. In the test environment, the object will have to behave differently, namely according to the work environment context of the process test. In Fig. 2.4. is shown, how the behaviour of the module is adapted to the test environment. The figure shows the to_test role with two methods, compiling and testing. The compiling method overloads the original compiling method which is defined in the software product model, and the method testing is added to the set of methods of the object. So using the role concept, it's possible to describe the contextual behaviour of objects now.

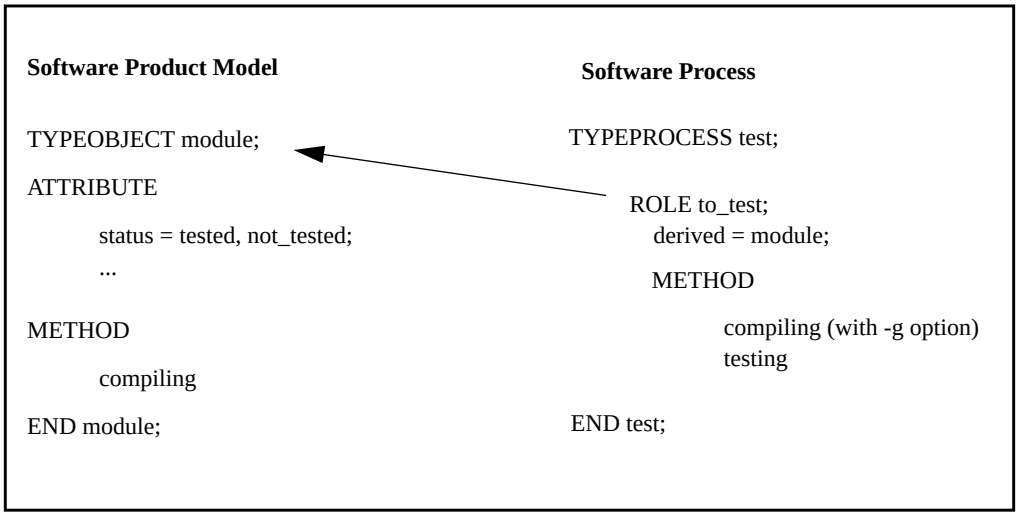


Fig. 2.4. Contextual Behaviour.

It must be clear that an object in the Product Model is independent of the context, but an object in a software process is context dependent.

2.7.3 Roles and Classes

Roles and classes look similar, so the following question can rise: is the role concept needed at all ?

A role, as well as a class, is a set of instances sharing the same definition (static and behavioural). A given object instance can be simultaneously a member of different roles (classes). Both roles and classes can be regarded as a viewing mechanism since a given object instance has a different description depending on the role (class). However the differences are the following: The association between an instance and its class(es) is statically defined at instantiation time, while an instance can be dynamically bound to an arbitrary role at any time. In an O.O. system the class definition is created first, and then the instances of the class. While in TEMPO, usually, the instances are created first, and are dynamically associated, to a (set of) role(s).

2.7.4 Roles and Processes

In TEMPO a process is a set of roles. Each role is the set of object instances sharing the same static and dynamic description of that process (attributes, methods). In TEMPO a complex process step, can be broken down in other subprocesses until the desired level of detail is achieved. Thus a complex activity can be broken down into a hierarchy of other less complex activities. However no special semantics are provided to express this policy.

2.8 The Connection

Each process model of a work environment, defines what happens in the work environment. In this way it looks like, that the work environment is performing alone, but that is clearly not true. In a process, several and different work environments are working together to reach the same goal at the end. So it is obvious, that the different work environments have to collaborate and to be synchronized with each other. The work environments can use the same objects. If the work environments are not synchronized at a certain point in time, it will be impossible to integrate the objects of their work environment with the shared objects in the other sub work environments (each sub work environment will have his own version of a shared object). A connection is relationship between two roles. The purpose of a connection is to define how each pair of connected objects is coordinated, and so to define the collaboration of the different work environments. The connection allows two roles to communicate with each other by data flow and status checking. It must be clear that connections are not symmetric, e.g. a development WE wants to get automatically new versions of objects, produced in a validation WE, and probably not the reverse. In Fig. 2.5. an example of a connection between two sub work environments is shown. Now the sub work environments (Designing1 and Reviewing1) can exchange information with each other. The information they exchange is bi-directional.

.

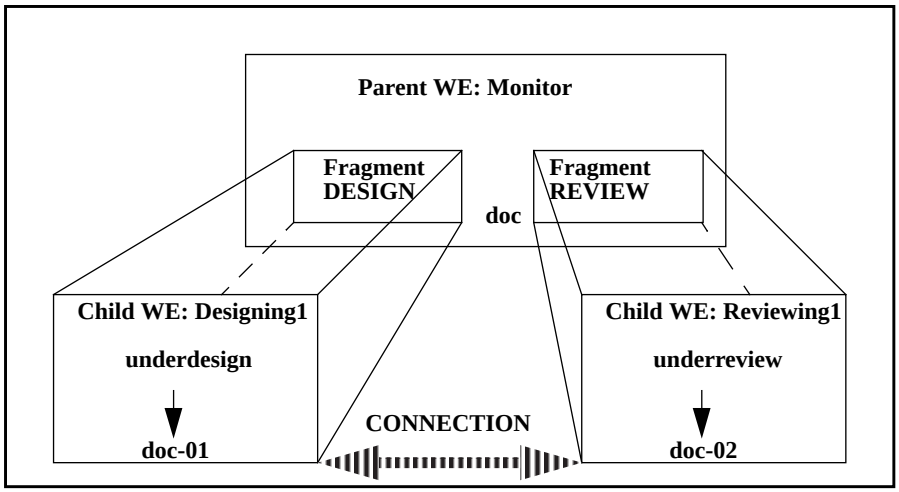


Fig. 2.5. Connection between two WEs

To describe the exchange policies of the messages between the two work environments, TEMPO uses the following structure to describe a connection:

```
The Connection :

Link ISA CONNECTION;
     DOMAIN
          Designing: underdesign ->
          Reviewing: underreview;

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

1. DOMAIN: The domain of a connection is situated in the clause DOMAIN of the type CONNECTION. The connections are always binary, this means that a connection always connect two processes with each other. A connection type describes the connection policy between a role of a process with the role of another process.

```
E.g.
     DOMAIN
          Designing: underdesign ->
          Reviewing: underreview;
```

The connection is established between the roles underdesign and underreview.

2. PLUG-ON-RULES: When two processes have to be connected, these conditions are described in the clause PLUG-ON-RULES.

```
E.g.
     PLUG-ON-RULES
       WHEN create_process UPON (SOURCE OR DEST);
```

When an occurrence of the process type Designing or Reviewing is created, the connection is instantiated.

3. PLUG-OFF-RULES: For each connection type is it possible to describe the conditions for which a connection must be disconnected. These constraints are written down in the clause PLUG-OFF-RULES.

```
E.g.
     PLUG-OFF-RULES
     WHEN finish_execution UPON (SOURCE OR DEST);
```

If one of the two cooperative processes finishes its activities, the connection between the two processes is destroyed.

4. ACTIVE-RULES: For each connection type, it is possible to write a unit of TECA rules, which are allowed to control the information exchange between two processes. When

there is an update of the object in one of the connected processes, this will raise some events in the other process. These TECA rules are defined in the clause ACTIVE-RULES

E.g. (and suppose the connection exists between Designing1 and Reviewing1)

```
ACTIVE-RULES
    WHEN design_completed UPON SOURCE
    DO allocate(% source, occurrence_of(% dest));
```

This means, when a design document is completed in the process Designing1, the document (or the modifications in the document) has (have) to be propagated to the process Reviewing1.

2.9 The Check in - Check out Mechanism

Because of the fact, that different work environments can be active at the same time, it's probable that the same objects are shared by different work environments. Figure 2.6. describes the situation for two different work environments (= WE-A and WE-B), which are using the same object (= o). The two work environments have both a copy of object o in their corresponding work space.

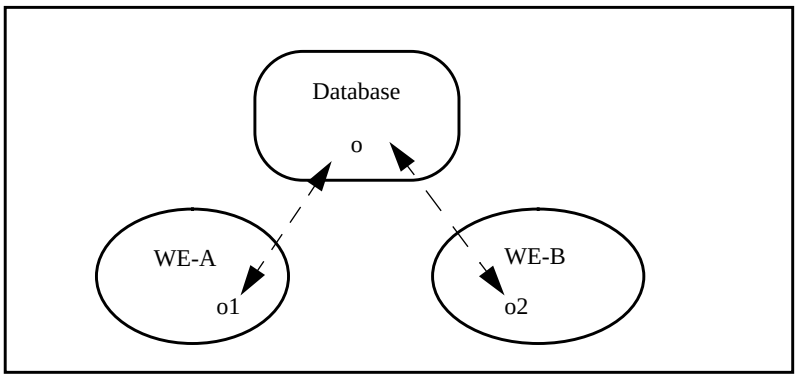


Fig. 2.6.

1. Check out

To modify objects in isolation, a branch for each object must be created. A branch is a sequential list of revisions of an object. Such a branch is created by a **check out**. This check out assures that an object with its attributes and methods is copied from the database to the work environment, or the object is copied from the parent work environment to the corresponding child work environment. The check out policy allows a user to manipulate the objects in isolation from other users in his own work environment. The manipulation of the objects can be for a long period of time, and without concurrency conflicts with other users.

2. Check in

Because of the manipulations of an object, new revisions of the object are created. If the user decides to promote the modified objects in his work environment to the database or to the parent work environment, this is performed by a **check in**. The user (in WE-A) who is responsible for the check in of the object in the data base or in the parent work environment, has to resolve eventually arising merge conflicts. If during the merge phase, other processes promote the same object, the check in operation of WE-A will be aborted. A promote operation will move

attributes values of duplicated objects from the work environment to the database (or to the parent work environment). If the user promotes an object with an attribute or method which is not defined in the type of the object, the check in will be aborted. If there are no problems with the promotion of the object, the check in will be committed.

For example, Fig. 2.7. is showing a scenario of the check in - check out concept. First a check out of object O (the 2nd revision) from the database to WE-A is performed by the user of WE-A. The user of WE-B has checked out the third revision of object O. Now the users in the work environments can modify their object without influencing the other object in a other WE (= copy of object O in WE-B) object. Because each user can modify his object, the user generates new local revisions of the object. When a user decides to promote his object to the database, he has to do this by a check in. In Fig. 2.7., object o1 in WE-A is moved to the database. The user who promotes his revisioned object, is responsible for resolving merge conflicts. So when the user in WE-A is promoting his object (= o1) to the database, he is responsible for the resolving of eventually conflicts with object o2 in inside WE-B, before the check in can be performed. Only when this is done, the check in will be committed.

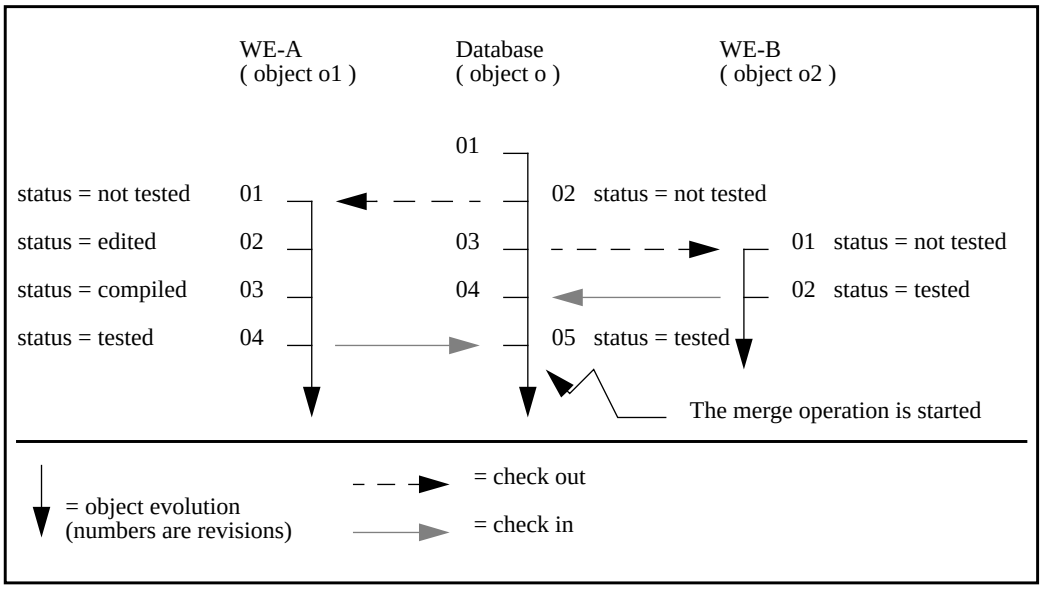


Fig. 2.7. Check in - Check out

2.10 The WS Manager and WSs

This is a small piece of writing about the ADELE Work Space Manager. The reason why writing something about it is, that TEMPO uses this concept of ADELE. In TEMPO a work environment consists of a work space, process model, user, tools. So a work environment can be seen as a tuple of the following form WE=(WS, PM, User, Tools). On top of the ADELE kernel, which consists of an object manager, version manager, transaction manager and a trigger manager, the work space manager has been implemented. The task of the work space manager is to deal with the different kind of coordinations between the different work spaces. In the next two subsections the WS and the WS Manager will be described.

2.10.1 The Work Spaces

A work space (=WS), is the place where the software engineering activities are executed. The objects that can be found in a WS can be of different types. The WS is defined by:

1. the set of components it covers and

2. the coordination policies with the other WSs involving the same objects (the only possible interactions between WSs are those defined by the coordination policies).

The WS itself is a sub database. This means that all the DBMS services are available in each WS, like local version, protection, etc. A WS can only be a sub database if it is isolated. This means that the WS has the usual ACID properties of transactions:

i. *Atomicity*: the work performed in the WS can be seen from outside as atomic (completely undone or completely done). But it is very unusual, that the work performed in a WS is completely undone, often parts of the work are made persistent.

ii. *Consistency*: the state of the database is 'consistent' when the WS was created and when it was committed.

iii. *Isolation*: a change made on an object in a WS, is only visible in that WS and a change performed on the 'same' object outside the WS isn't visible in the WS.

iv. *Durability*: the changes made on the objects are persistent when they are committed.

2.10.2 The Work Space Manager

The work space manager is an intermediate level between the ADELE kernel and the process level, e.g. TEMPO (see Fig. 2.8.). The work space manager provides two basic services:

1. Defining and controlling the WS content and

2. WS coordination.

If the coordination mechanism is used, the work performed in a WS is made visible to other WSs. So a WS can then no longer be considered as an ACID transaction.

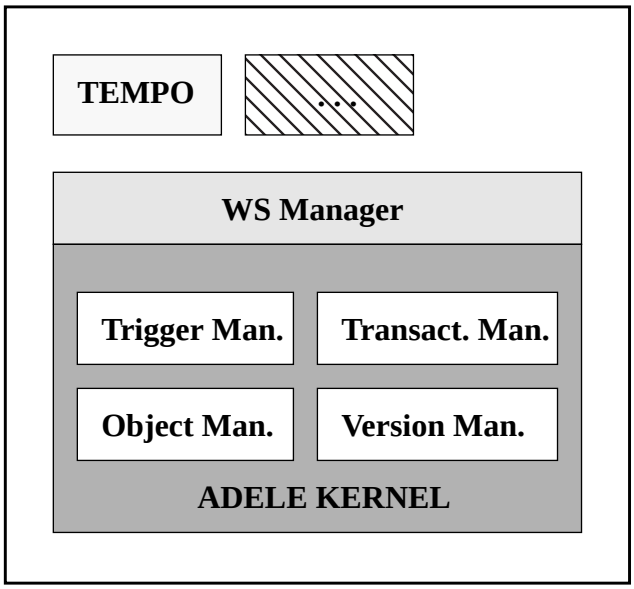


Fig. 2.8. The level of the WS manager

2.11 The WS Structure: Father and Son WSs

When a work space is created it is initialized with already existing objects. These objects belong to other work spaces, called fathers (e.g. by default the database). The ADELE WS manager (which is a program) controls these work spaces. The activities can be performed in all the work spaces, not only in the leaves of the work space structure. A work space can be atomic or composite. This distinction depends on the objects it contains in its work space, if the work space contains several objects which have a different father, the work space is called composite, otherwise it's called an atomic work space. So an atomic work space has a single father.

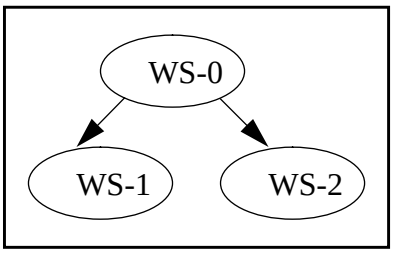


Fig. 2.9.1.

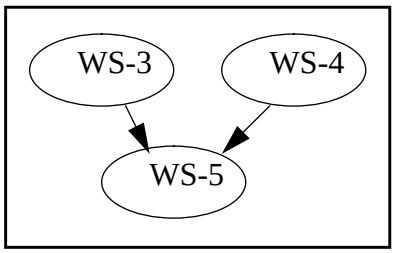


Fig. 2.9.2.

The relationship between the father and son work space describes in which conditions the changes performed on an object in the father / (son) WS have to be propagated to its sons / (father) WS. So if in Fig. 2.9.1. an object x is changed in WS-1, this change may be immediately propagated to object x in the father WS (= WS-0) and transitively to object x which is in WS-2.

Fig. 2.9.2. shows a composite WS (=WS-5), which has two different father WSs (= WS-3 and WS-4).

2.12 The Coordination of the WSs

Up till now the coordination is only defined between a father WS and his son WSs, the father contains the original object and the sons have a copy of the original object in their work space (= Father / Son coordination). There can be WSs containing the same copies of objects, so a coordination policy can be defined between these objects belonging to any arbitrary pair of WSs (= Peer to Peer coordination and Master / Slave coordination). The different kind of coordinations will be described below:

1. **Father / Son Coordination**: The son WSs contain a copy of the original object which is in the father WS. If the son work space is a composite WS it has several fathers, depending on the objects it contains.
2. **Peer to Peer Coordination**: The different instances of a WS type, contain copies of the objects in the common father WS. The peer to peer coordination allows to coordinate these WSs without going through the father WS.
3. **Master / Slave Coordination**: If necessary two different WSs (= not of the same type) have to be coordinated, because they share copies of the same objects. But without the need of going through the common father.
4. **Multiple Coordination**: In general a large number of different coordinations can exist concurrently. The WS manager is managing, supporting and controlling these coordinations.

In Fig. 2.10. is shown in a graphical way the different kind of coordinations.

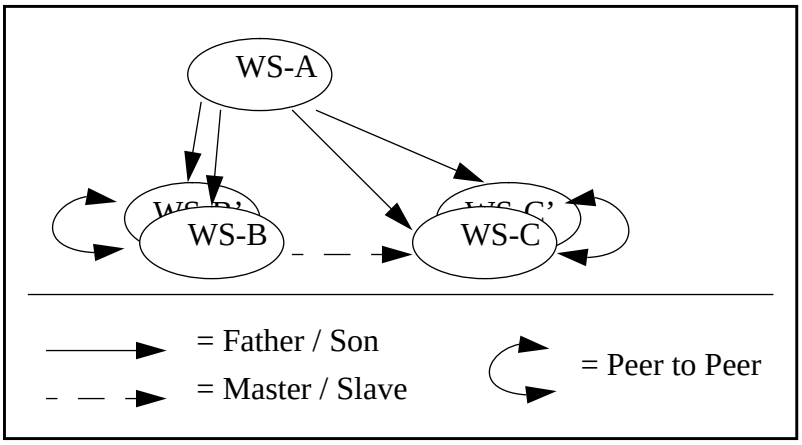


Fig. 2.10. The Different Coordinations

2.12.1 The Differences between the Coordinations

1. **Master / Slave Coordination**: This coordination is based on the principle, that only the objects in the slave WS can be modified. The slave WS integrates, the work done by its master WS, but NOT the opposite. The Master / Slave coordination is in one direction, from the master WS to the slave WS. So the slave WS can't make changes in its master WS.

2. **Peer to Peer Coordination**: This coordination is a kind of a Master / Slave coordination, but with the difference, that the coordination is in both directions. This means that the WSs which are connected by a Peer to Peer coordination can introduce changes in the other WS(s).

3. **Father / Son Coordination**: The father creates the son WSs, and these son WSs always know that they have a father WS. From time to time a son WS needs to synchronize with its father WS (the father can be seen as the master and the son as the slave). However, a father is not a common master, because it usually needs to work done in the son(s). When the work performed in the son work space is terminated, it must be transferred to the father WS, using the command 'promote'.

2.13 The Scenario

This scenario will be used for the comparison of the ADELE/TEMPO and SOCCA approach. Two examples (example 1 and example 2) of the scenario will be modelled in the ADELE/ TEMPO formalism as well as in the SOCCA formalism. The other examples which have been modelled in ADELE/TEMPO are given in Appendix B.

In this scenario the different members of the project team and their tasks they have to perform, and tools used in the software process are modelled. The purpose is to make clear what activities the members are involved in and which other activities they can perform in parallel with their current activity. E.g. you can think of a designer who is modifying a design document. He is not allowed to review his own design or is not allowed to implement the corresponding design document, but he may review and/or implement different other design document(s) developed by (an)other designer(s).

After describing the examples, it will hopefully be possible to say more about the similarities and differences between the two approaches.

There will be many different interesting aspects in the scenario. They are categorized in the following subparagraphs, where each aspect will be explained in more detail.

2.13.1 The Design Engineer

The design engineer can modify and review a document. It's also possible that the design engineer work on different documents at the same time. Of course it is not really at the same time, because he will switch between his activities. The design engineer gets some tasks of the project manager and after that he decides when and what he is going to do. E.g. A designer must design two documents and he has to review another document at the same time.

2.13.2 The Mailtool

The mailtool will be used by the human participants of the software process to communicate with each other. The mailtool comes up as a closed icon and the icon will be opened by clicking on the mailtool icon. The incoming mail is stored in the buffer of the mailtool. A mailtool consists of a main window with three push-buttons. These buttons are: a compose, a view and a reply push-button. A more detailed description of the mailtool can be found in paragraph 3.5. of the SOCCA examples.

2.13.3 Sharing Documents and Communication

There can be two or more designers working on the same document. They all have a different version of the same document. At certain points of time the different versions of the same document have to be merged, such that there won't be any inconsistencies in the different versions of the same document. The different documents in the various WEs have to be exchanged e.g. as soon as a document in the design WE has been finished, it has to be sent to the review WE. This is allowed by using the connection. The different members have to communicate with each other in order to make some parts of a problem clear. They can communicate with each other by using a mailtool. The mailtool has already been described, but it will be used in the different work environments in order to allow the communication between the different persons who participate in the software process.

2.13.4 Assigning Tools to Roles

In the project team different people work together with not the same status. A project manager has status 1, a designer has status 2, a reviewer has status 3 and an implementer has status 4. It is possible for them to use general tools at the same time e.g. FrameMaker. But there will be a problem, because for the project team only four licences of FrameMaker are available. At a certain point of time it could be possible, that all licences are occupied by some members of the project team. If someone else of the team wants to use FrameMaker, the person deals with a problem, because he can't get a licence.

If the person with no licence has a higher status than someone using FrameMaker, the user of FrameMaker with the lowest status has to give up his licence. After that, the person with no licence will get a licence. This can be arranged by sending a message to the user with the lowest status, the message will tell him to exit FrameMaker within 2 minutes. If he does not exit in those 2 minutes, the system will do this automatically. Then a message is sent to the person with the higher status, it will inform him that a licence is available and he can use FrameMaker now.

It's also possible, that the person with no licence for FrameMaker, has a lower status than the people with a licence. Then he has to wait or do something else until one of the users of FrameMaker has given up his licence.

E.g. There are 4 licences of FrameMaker available for the project team. The following persons have a licence; project manager, designer, reviewer and an implementer. At a certain moment a designer with no licence wants to use FrameMaker. A message is sent to the implementor to say he has to exit FrameMaker within 2 minutes. After the implementer has exit (voluntary or by the system), a message is sent to the designer (with no licence). This message says that he has a licence and that he is able to use FrameMaker now.

2.14 Examples Modelled in TEMPO

2.14.1 Example 1: The Design Engineer

```
################################################################################
Example 1, paragraph 2.13.1.

In the process designing the users who have a design role, can perform their activities under
the role Design. In the scenario, a designer can design a document, but he can also review dif-
ferent other documents. So the object designer, must also be in the reviewing process.
Depending on what he wants to do, he can switch between these roles. He can design a docu-
ment and he can review a different document.
################################################################################


_____________________________________________________________________________
user ISA object;

    ATTRIBUTE
        name     = String :='';              # name of the user is empty #
        Position = Pmanager, Engineer, None := None # the position of a user #
        Status   = 0,1,2,3,4:= 0;  # The status of the user, No=0, PM=1, Des=2, #
                                            # Rev=3, Impl=4 #

    METHOD
        Stop;                    # the user wants to do something else #

END_OF user;
_____________________________________________________________________________

=============================================================
Designing ISA PROCESS;

    ROLE Design;                              # designing a document #

    derived = user;              # object under this role is of type user #

    METHOD
        Work_in_doc;           # the engineer modifies the document #
        Change_r;              # change role of engineer, get role review #

    RULES
        PRE WHEN Begin_design  DO Work_in_doc;      # start design #
        PRE WHEN Stop_design   DO Stop;             # stop design #
        PRE WHEN Cont_design   DO Work_in_doc;      # continue designing #
```

```
        PRE WHEN Change_role_r DO Change_r;        # change role to review #

END_OF Designing;
=================================================================

=================================================================
Reviewing ISA PROCESS;

    ROLE Review;                              # reviewing a document #

    derived = user;              # object under this role is of type user #

    METHOD
        Work_in_doc;         # the engineer reviews the document #
        Change_d;           # change role of engineer, get design role #
        Change_i;           # change role of engineer, get implement role #

    RULES
        PRE WHEN Begin_review  DO Work_in_doc;    # start review #
        PRE WHEN Stop_review   DO Stop;          # stop review #
        PRE WHEN Cont_review   DO Work_in_doc;    # continue reviewing #
        PRE WHEN Change_role_d DO Change_d;       # change role to design #
        PRE WHEN Change_role_i DO Change_i;       # change role to implement #

END_OF Reviewing;
=================================================================
```

2.14.2 Example 2: The MailTool

```
#############################################################################
```

Example 2, paragraph 2.13.2.

In this example I try to model how the different users are communicating with each other by using a mailtool. In this specification of the mailtool it is assumed that there are high-level events like push-button and that the methods do exist.
The communication is horizontal and vertical, and of coarse in two directions:
- Horizontal communication means, that the users in the sub WEs can communicate with each other.
- Vertical communication means, that the users in the parent WE and the users in the child WEs can communicate with each other.
In Fig. 2.11. is shown how to interpret the mail tool, with its corresponding popup windows.

```
#############################################################################
```

*** Mail Tool ***

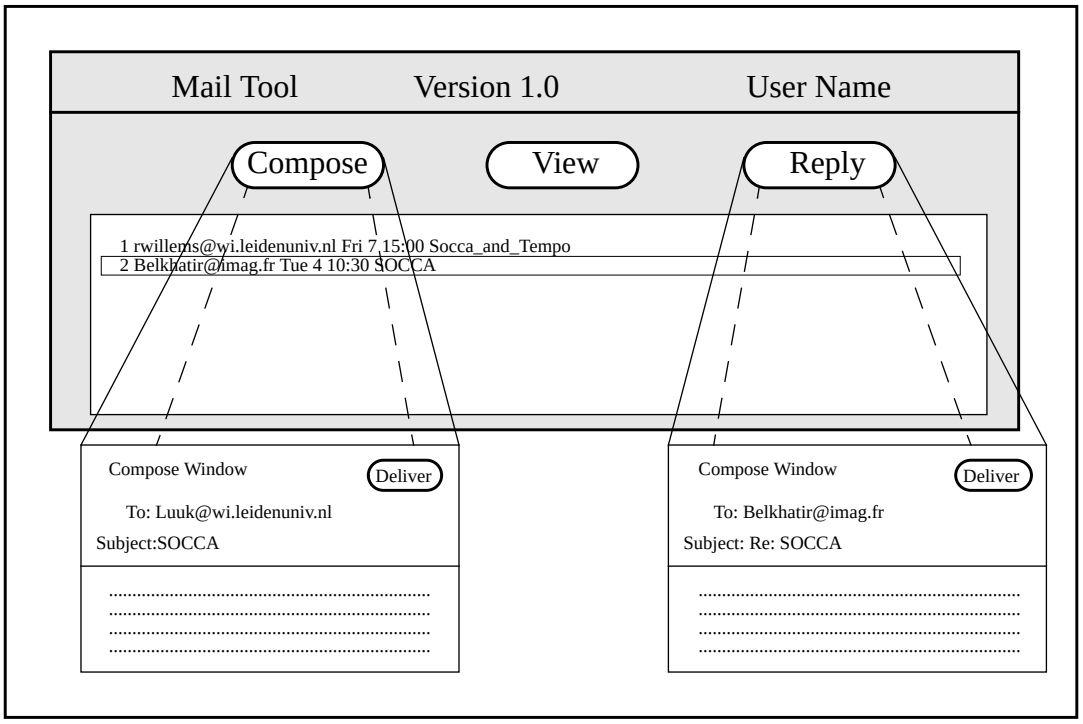


Fig. 2.11. MailTool

```
DEFEVENT Start_MT = [!cmd = ‘double click on mailtool icon’];
DEFEVENT Stop_MT = [!cmd = ‘one click on main window’];
DEFEVENT Receive = [is system event];
DEFEVENT Compose = [!cmd = ‘push button Compose’];
DEFEVENT Deliver = [!cmd = ‘push button Deliver’];
DEFEVENT View = [!cmd = ‘push button View’];
DEFEVENT Reply = [!cmd = ‘push reply button’];
```

```
mailtool ISA object;

   ATTRIBUTE
         name = String := ‘’;          # name of mailtool #
         uname = String := ‘’;          # user name#
         version = Real := 0;          # version number   #
         date = Date := 00/00/00;    # date of release #

   METHOD
         PopupMailToolWndw;   # mail tool window is popped up #
         CloseMailToolWndw;   # mail tool window is closed #

         Store;           # store the mail in the buffer #
         Send;         # send mail#;
         Sound;          # make a sound to inform user for new mail #
         PopupComposeWndw; # Compose window is popped up #
         CloseComposeWndw; # Compose window is closed #
```

```
        GiveInfo;        # information is provided to the user #
        DisplayMessage;  # message is displayed in a window #

END_OF mailtool;
```

```
==========================================================================
Designing ISA PROCESS;

   ROLE Design;                              # designing a document #

   derived = user;              # object under this role is of type user #

    METHOD
        ...

    RULES
        ...

ROLE DMailtool;                           # mailtool for a designer #

   derived = mailtool;         # object under this role is of type mailtool #

    RULES
        PRE WHEN Start_MT DO PopupMailToolWndw; # activate mailtool #
        PRE WHEN Stop_MT DO CloseMailToolWndw; # de-activate mailtool #

        PRE WHEN Receive DO Store;                  # receive new mail #
        POST WHEN Receive DO Sound;

        PRE WHEN Compose DO PopupComposeWndw; # User wants to send a message.#
                            # User writes address, subject and the message in the #
                            # compose window. #
        PRE WHEN Deliver DO Send;            # mail is sent to destination #
        POST WHEN Deliver DO CloseComposeWndw;
        ERROR WHEN Deliver DO GiveInfo;              # mail cannot be send #

        PRE WHEN View    DO DisplayMessage;    # user wants to read the mail #
        ERROR WHEN View DO GiveInfo;              # no mail in mailbox #

        PRE WHEN Reply   DO PopupComposeWndw; # now user doesn't have to write#
                            # the address of the destination#

END_OF Designing;
==========================================================================
```

Chapter 3 The SOCCA Formalism

3.1 Introduction into SOCCA

In the following paragraphs the **SOCCA** (**S**pecifications **o**f **C**oordinated and **C**ooperative **A**ctivities) approach will be described. This is not a complete introduction, because the complete one can be found in [10]. The SOCCA formalism has been developed at the University of Leiden (The Netherlands). This specification formalism for software process modelling is not only being developed for describing the technical parts of the software process, but also for the human parts, or rather the human team members of the software process. The system should reflect all kinds of interactions between the various parts, including the non-human as well as the human parts.

3.2 The 3 Different Perspectives of SOCCA

SOCCA uses three different perspectives to describe a software process. These perspectives are: the *data perspective*, the *behaviour perspective* and the *process perspective*. They will be described in the following subparagraphs.

3.2.1 The Data Perspective

The data perspective covers the **static structure** of the process. This perspective is described by an Extended Entity Relationship (**EER**) model. In this EER model two different diagrams, a Class diagram and an Import/Export diagram, can be identified. The classes in the class diagram have export operations, so a class diagram consists of attributes and export operations. Export operations can then be called from other classes where they are imported. The classes are connected by four different kind of relationships: the General relationship, the Uses relationship, the IS-A relationship and the Part-Of relationship. The *General relationship* describes the relation between the different classes. The *Uses relationship* is used to indicate where the various export operations are imported. The Import/Export diagram visualizes the Uses relationship between the several classes. The *IS-A relationship* is utilized to describe the inheritance between the different classes. The *Part-Of relationship* is used to express which class is part of another class.

3.2.2 The Behaviour Perspective

The behaviour perspective covers the **dynamic part** of the software process. The behaviour perspective and the coordination of the behaviour, will be described by State Transition Diagrams (**STDs**) and by **PARADIGM** on top of them. The STDs are used to describe the order in which the operations can be called. This means to describe the behaviour of the classes in the model. The behaviour of the class is composed of two different types of behaviour: the external behaviour and the internal behaviour.

The External Behaviour: The external behaviour of a class is described with an STD. All the export operations of the class appear as a label of a transition.

The Internal Behaviour: The export operations perform some task and to achieve this task, they have to behave in a certain way. To this aim the export operations of the external behaviour all have an internal behaviour (during the constructing of this master's thesis, it turns out to be, that not all export operations have an internal behaviour, but this will be discussed later on in paragraph 3.4.5.). In principle each export operation has an internal

behaviour. Each internal behaviour of an export operation is itself described by another STD.

It's obvious, that the cooperation between the external behaviour of a class and the internal behaviour of its exports operations must be coordinated somehow. In SOCCA this will be done by PARADIGM. Moreover, the internal behaviour of an export operation is also able to call export operations from other classes. Summarizing this means, that there is communication between the external behaviour of a class and the internal behaviour of its export operations within the same class, and there's communication between the internal behaviour of the export operations from one class and the external behaviours of other classes whose export operations are being called. Using the PARADIGM formalism, the behaviour perspective and particularly the coordination of behaviour can be modelled. In paragraph 3.3. the PARADIGM description will be presented.

3.2.3 The Process Perspective

The process perspective will be modelled by Object Flow Diagrams (**OFD**s). Because of the fact, that the integration of OFDs in SOCCA has not been completed yet, the process perspective won't be discussed any further.

In Fig. 3.1. the concepts for the three different perspectives are shown, used in the SOCCA approach.

	Data Persp.	Behaviour Persp.	Process Persp.
SOCCA Concepts	**EER**	**STD + Paradigm**	**OFD**

Fig. 3.1. The Three Perspectives of SOCCA

3.3 Introduction into PARADIGM

As PARADIGM is less commonly known compared to EER and STD concepts, it will be discussed briefly.

The **PARADIGM** (**PAR**allelism its **A**nalysis, **D**esign and **I**mplementation by a **G**eneral **M**ethod) formalism was originally developed for the specification of coordinated parallel processes, see [13, 18, 19]. The basic ideas of modelling in PARADIGM are the following:

1. The sequential behaviour of each process can be described by an STD.

2. Within each STD, subdiagrams (called subprocesses) can be identified. These subprocesses are temporary behaviour restrictions of the complete behaviour. A subprocess reflects the allowed behaviour of a process within its STD before or after communication has taken place.

3. Within each subprocess, so-called traps, are identified being a subset of the states of a subprocess. These traps are represented by shaded polygons around the states which are part of the trap. When an object has entered such a trap, it indicates that it's ready to switch from one subprocess to another one. The key property of a trap of a subprocess is, if the process is in a trap, it isn't able to leave this trap as long as the process' behaviour is restricted to that subprocess. It waits until the manager process gives permission to leave the trap.

4. A manager process, described by an STD, takes care of the coordination of the behaviour and the transitions between the different subprocesses of all objects. The employee processes, which will be controlled by the manager, are the STDs having the subprocesses. The employee's subprocess is determined by the state of the manager. This means, that in each state of the manager process, one or more subprocesses will be prescribed. The state labels of a manager process correspond to a behaviour restriction, or a so-called subprocess of each of its employees. A transition of a manager corresponds to a so-called trap of the current behaviour restriction. The manager process can only make a transition, when the relevant employees have entered these traps. An employee when entering a trap, actually prescribes the manager process a new behaviour in which the transition corresponding to this trap is allowed, which was not the case before the trap was entered.

In the following two subparagraphs the first two examples of the scenario will be modelled in the SOCCA formalism. The first example is the modelling of a design engineer. This modelling is different from the normal way of modelling in SOCCA. Generally the modelling is of type-1, but here the modelling of type-2 has been used to describe the activities of the design engineer. In the second example, a mailtool has been modelled. During the modelling of this example, interesting and new aspects of the external behaviour have been appeared. This has led to some extensions of the SOCCA formalism.

3.4 Example 1: The Design Engineer

In this example the behaviour of a design engineer is described. This example acquires type-2 communication instead of type-1 communication modelling. The type -2 communication has been discussed in [10] very briefly, so hopefully this example will make the type-2 communication better understandable.

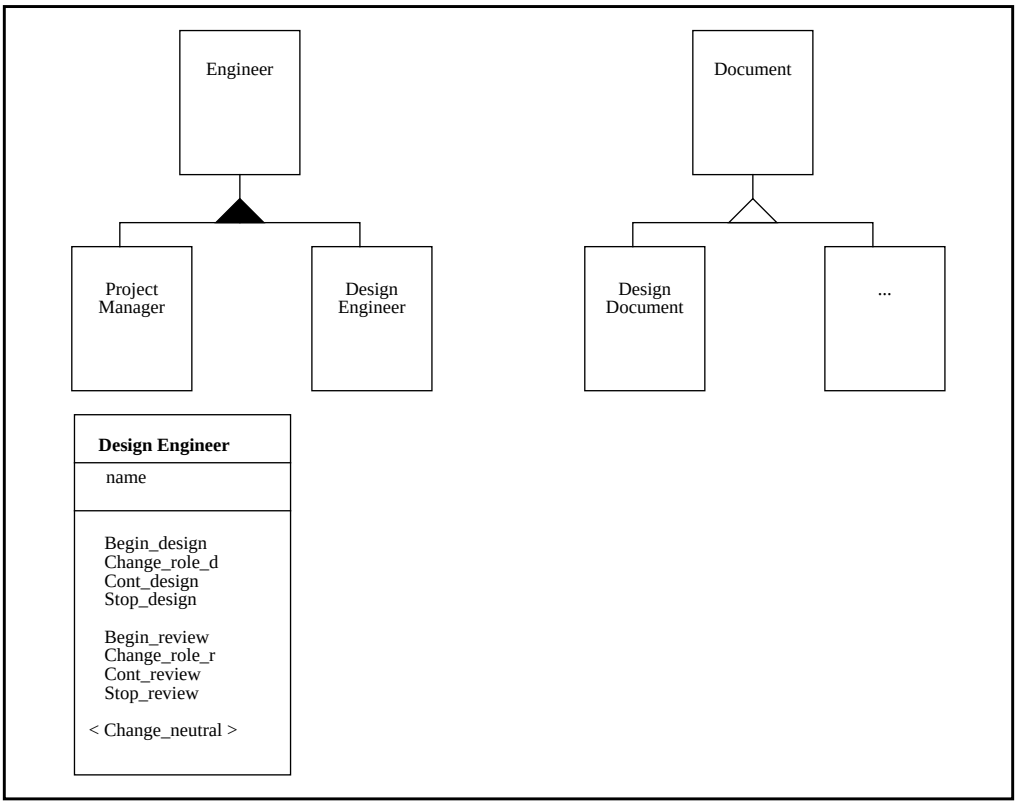


Fig. 3.4.1. Class diagram: classes, IS-A, attributes and operations

By the direction of the project manager, the design engineer has to modify or review a design document. He can decide when he wants to design or review the document(s). First the external behaviours (Fig. 3.4.4. and Fig. 3.4.5.) and then the internal behaviours (Fig. 3.4.6. - Fig. 3.4.14.) of Design Engineer will be described. The association between the manager's (Design Engineer) states and subprocesses and between the manager's transitions and traps are presented in Fig. 3.4.6.1. through to Fig. 3.4.14.1.

3.4.1 The Data Perspective of Design Engineer

First the class diagrams are presented.

It's not a complex and complete diagram, because only the design engineer is of interest in this example. The black triangle means, that the IS-A relationship is not disjoint and the hollow triangle indicates, that the IS-A relationship is disjoint. So an engineer can be a project manager as well as a design engineer. In this example the project manager and the design engineer are two different persons (see Fig. 3.4.1.).

Now the general relationships to connect the different classes from the first step are added. The general relationships used in this example are modifies and reviews. The hollow dots imply,

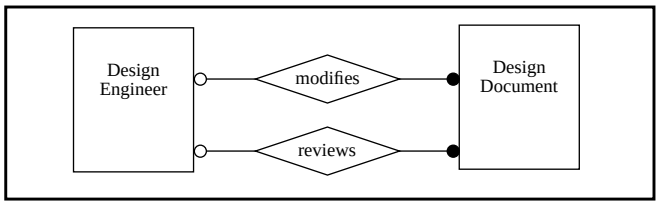


Fig. 3.4.2. Class diagram: classes and general relationships

that the design document may be modified or reviewed by a design engineer or not at all. The black dots mean that each design engineer can modify or review between zero and n design documents. Note that Fig. 3.4.1. and Fig. 3.4.2. should be read together.

The next step is to give more information which is normally not given in an EER model. A new binary relationship type, called **Uses**, denotes where the various export operations are imported.

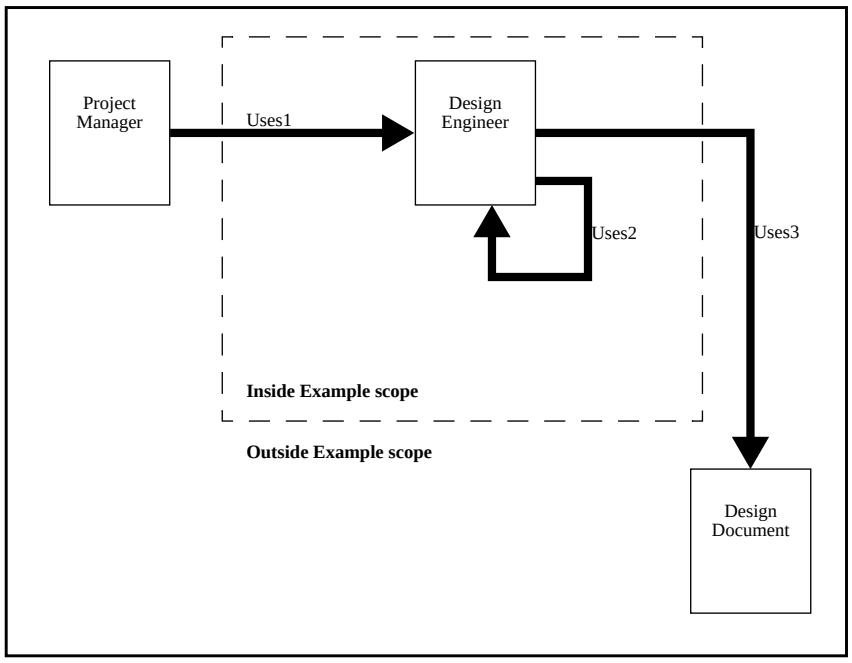


Fig. 3.4.3. Import/Export Diagram

The uses relationships can be found in the import/export diagram of Fig. 3.4.3. Each uses relationship has an attribute (import_list) for holding a list of names of imported operations. Here the imported operations of Uses3 are not given, because Uses3 is out of the scope of this example. The import_lists for uses1 and uses2 are:

Uses1	**Uses2**
Begin_design	**Change_role_d**
Begin_review	**Cont_design**
	Stop_design
	Change_role_r
	Cont_review
	Stop_review
	< Change_neutral >

3.4.2 The Behaviour Perspective of Design Engineer

In the previous subparagraph the static aspects (data aspect) have been described. Now the behaviour perspective will be demonstrated. The behaviour comprises two different types of behaviour: the external behaviour and the internal behaviour.

3.4.2.1 The External Behaviour of Design Engineer

In Fig. 3.4.4. the external behaviour of Design Engineer is depicted. The export operations Begin_design and Review_design can be called by the project manager, in this way the design engineer gets his tasks he has to perform. Up to now there is nothing unusual in the SOCCA way of modelling.

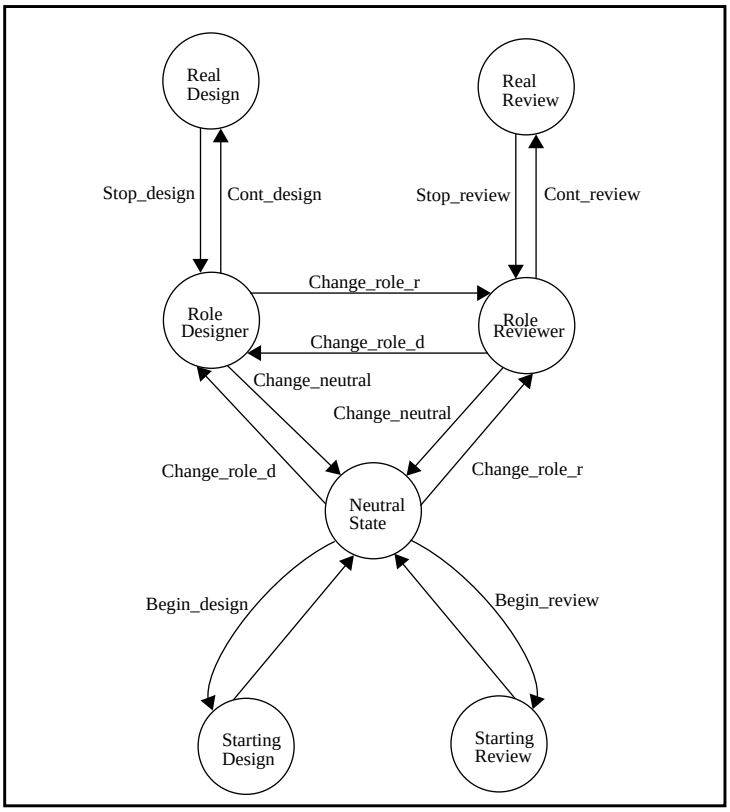


Fig. 3.4.4. Design Engineer: STD of the external behaviour

But it is interesting to see how the design engineer can decide by himself when and what he wants to do. This part of the external behaviour looks like an operating system way of modelling of the design engineer, because in the external behaviour you can see the operations *Cont_design, Stop_design, Cont_review* and *Stop_review.* This is analogous with a real OS, because it also can start, stop, continue and select other programs (tasks/activities). The differences between a real operating system and the design engineer are:

1. The design engineer is human and an operating system (OS) of course not.

2. An OS prescribes a process when it is allowed to perform it's tasks and when the process has to stop. Here the design engineer makes his own decisions.

The OS way of modelling of the design engineer is described by the states: *Role Designer, Role Reviewer, Real Design* and *Real Review* and by the operations: *Change_role_d, Cont_design, Stop_design, Change_role_r, Cont_review, Stop_review* and *Change_neutral*. All the export operations of Design Engineer will be explained in the internal behaviour STDs (Fig. 3.4.6. - Fig. 3.4.14.).

The state called *Neutral State,* is the state which the design engineer returns to after the design or review activity has been started by him. The project manager calls the operations *Begin_design* and *Begin_review,* so Design Engineer makes the transition from state *Neutral State* to state *Starting Design* or state *Starting Review.*

The state *Starting Design* can be considered as the commencement of the design activity, and the state *Starting Review* can be regarded as the beginning of the review activity. At the time that the Design Engineer is in one of these states it means, that the design activity or review activity is ready to start, but stopped! The Design Engineer has only received the task(s) design or review, but he doesn't start the real designing or reviewing. The transitions from these states back to the state *Neutral State* have no export operations. So these transitions are made without any calling initiative from abroad.

Now the state and transitions made by the design engineer are described. The state *Role Designer* means that the design engineer can act as a designer. This state will be reached by calling the *Change_role_d* operation. The state *Role Reviewer* means that the design engineer can behave as a reviewer. This state will be attained by calling the *Change_role_r* operation. Note that the design engineer is still not really active in the sense of modifying or reviewing a document. The design engineer can switch between his roles by using the operations *Change_role_d* and *Change_role_r.* If the design engineer calls the *Cont_design* or *Cont_review* operations, he will transit to the states *Real Design* or *Real Review.* In these states he can really start with the design or the review of a document. So in these states he is actually active. After a while the designer wants to stop designing or reviewing, therefore he uses the operations *Stop_design* or *Stop_review* to stop his activity and go back to his state *Role Designer* or state *Role Reviewer.* When the design engineer wants to go back to state *Neutral State*, he uses the operation *Change_neutral*. As soon as the design engineer is back in its neutral state, he can get new tasks of the project manager. It is possible, that the design engineer wants to start his activities again. Then the same operations have to be called by him as described above. This finishes the description of the external behaviour of Design Engineer.

3.4.2.2 An Alternative for the External Behaviour of Design Engineer

During the modelling of this example, an alternative for the external behaviour of Design Engineer arose (see Fig. 3.4.5.). In this alternative external behaviour, Design Engineer is reduced with the operation *Change_neutral* and the state *Neutral state*. The operation *Change_neutral* is represented in the class diagram between < >, because this means that it must **not** be used in this alternative external behaviour. In Fig. 3.4.5., the states *Role Designer* and *Role Reviewer* can be regarded as the neutral states. Further, nothing has been changed in comparison with the external behaviour of Design Engineer in Fig. 3.4.4. The states *Starting Design* and *Starting Review* are added and connected with state *Role Reviewer.* This implies, that the same export operations and the same internal behaviours have been utilized.

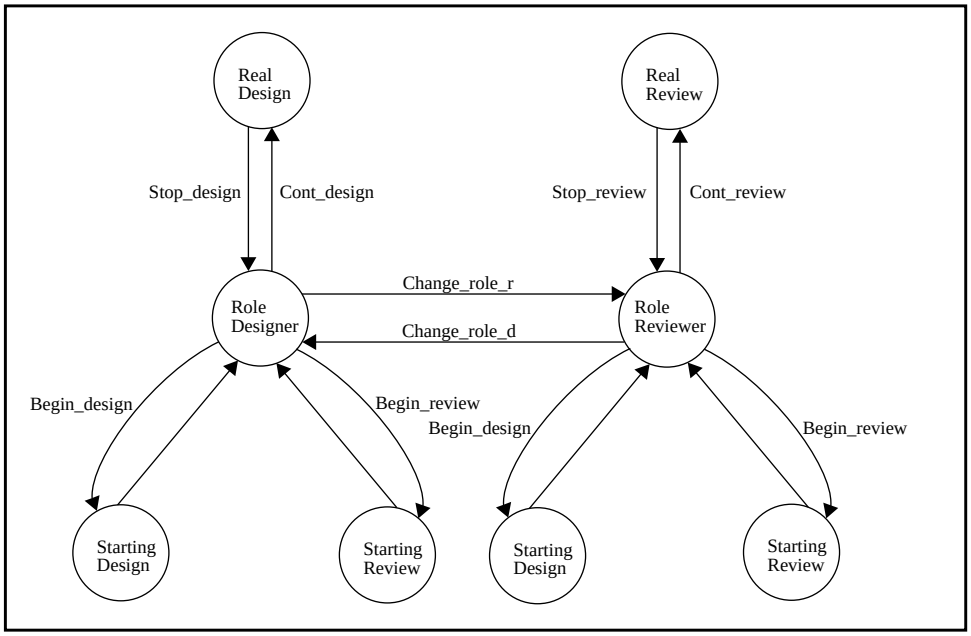


Fig. 3.4.5. Design Engineer2: alternative STD of the external behaviour

3.4.2.3 The Internal Behaviour of Design Engineer

After the specification of the external behaviour, the internal behaviours of all operations have to be specified. These operations have to correspond to the various export operations called from elsewhere and by the design engineer himself. Before the specification of the internal behaviours, a list of possibly imported operations for each uses relationship will be presented.

Uses1	**Uses2**
Begin_design Begin_review	Change_role_d Cont_design Stop_design Change_role_r Cont_review Stop_review < Change_neutral >

This enumeration of operation names shows all possibly imported operations. Note that the internal behaviours are described with respect to Design Engineer. These internal behaviours have also been used in the two alternative managers (Design Engineer2 and Design Engineer3). In design Engineer2 and Design Engineer3 not all internal behaviours and states have been used

as in Design Engineer. In Design Engineer3 the way of modelling differs from the original way of modelling in SOCCA. These differences will all be explained and described in the corresponding subparagraphs. Now the specification of the internal behaviours will be presented. The nine internal behaviour specifications can be found in Fig. 3.4.6. through to Fig. 3.4.14. Note, that below each internal behaviour specification the corresponding subprocesses and traps are given. The subprocesses and traps can be found in Fig. 3.4.6.1. through to Fig. 3.4.14.1.

In Fig. 3.4.6. the internal behaviour of operation *Begin_design* (int-Begin_design) is shown.

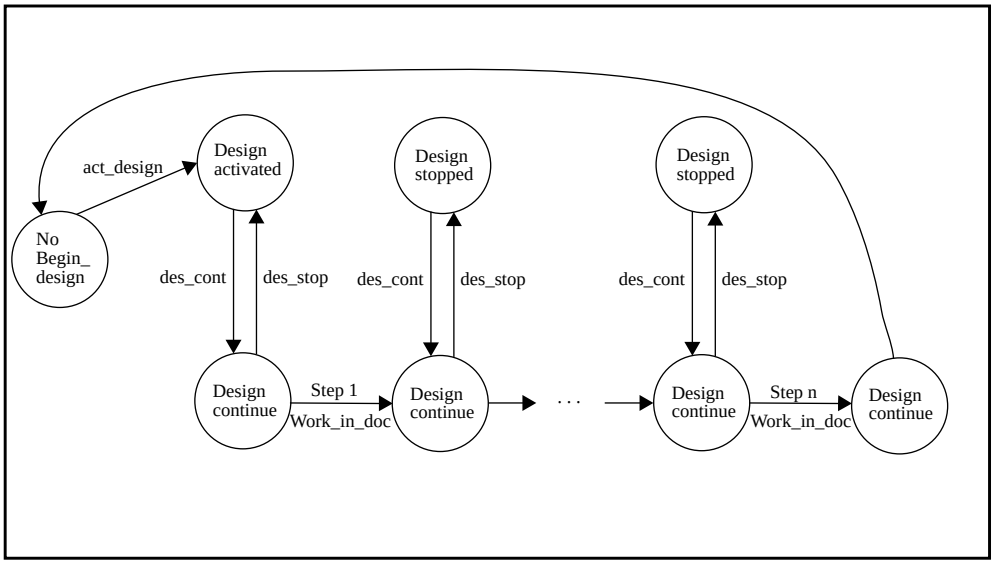


Fig. 3.4.6. int-Begin_design: STD of its internal behaviour

The three subprocesses (S-1, S-2 and S-3) and the four traps (t-1, t-1a, t-2 and t-3) belonging to int-Begin_design can be found in Fig. 3.4.6.1. In the beginning the design engineer has nothing to do and waits for the tasks the project manager is going to give him. If the project manager calls e.g. *Begin_design*, the manager Design Engineer (see Fig. 3.4.15.) is requested to transit from *State Neutral* to *Starting Design*. Nevertheless, Design Engineer is still outside the state *Starting Design*. This indicates, that int-Begin_design is in its subprocess S-1, and hopefully within trap t-1. The manager Design Engineer can (and will) make the transition from *State Neutral* to *Starting Design*, only after int-Begin_design has reached trap t-1. When the state *Starting Design* has been entered, Design Engineer assigns subprocess S-2 to int-Begin_design. Considering that int-Begin_design was in trap t-1, it directly starts inside subprocess S-2 from state *No Begin_design*, and then nearly right away entering trap t-2. When trap t-2 has been entered, Design Engineer leaves the state *Starting Design* and transits to state *Neutral State*. Because of the transition Design Engineer prescribes subprocess S-1 to int-Begin_design. For int-Begin_design, this means that it is waiting in its state *Design activated* of subprocess S-1. Only when the design engineer really wants to perform the design activity, he must use the operation *Change_role_d* to reach the state *Role Designer* of the manager Design Engineer. When the design engineer calls the operation *Cont_design*, Design Engineer is invited to transit from state *Role Designer* to state *Real Design*. Because int-Begin_design is in subprocess S-1 and in trap t-1a, Design Engineer makes the transition to state *Real Design*. Upon entering *Real Design*, Design Engineer now prescribes subprocess S-3 (instead of S-1) to int-Begin_design. As int-Begin_design was in trap t-1a, it directly starts within subprocess S-3 from state *Design activated* thereby nearly immediately entering trap t-3, now the real design is occurring. The design can be considered as a sequence of activity steps. In Fig. 3.4.6.1. this is represented by *step1 Work_in_doc ... step n Work_in_doc*. If the designer wants to stop his design activity,

Design Engineer will prescribe subprocess S-1 for int-Begin_design. As int-Begin_design was in trap t-3, it starts inside S-1 from state *Design Continue*, thereby nearly directly entering trap t-1a of subprocess S-1. When the design engineer wants to go on with the design (operation *Cont_design*), Design Engineer will prescribe subprocess S-3 for him and so he is allowed to perform his design activity again.

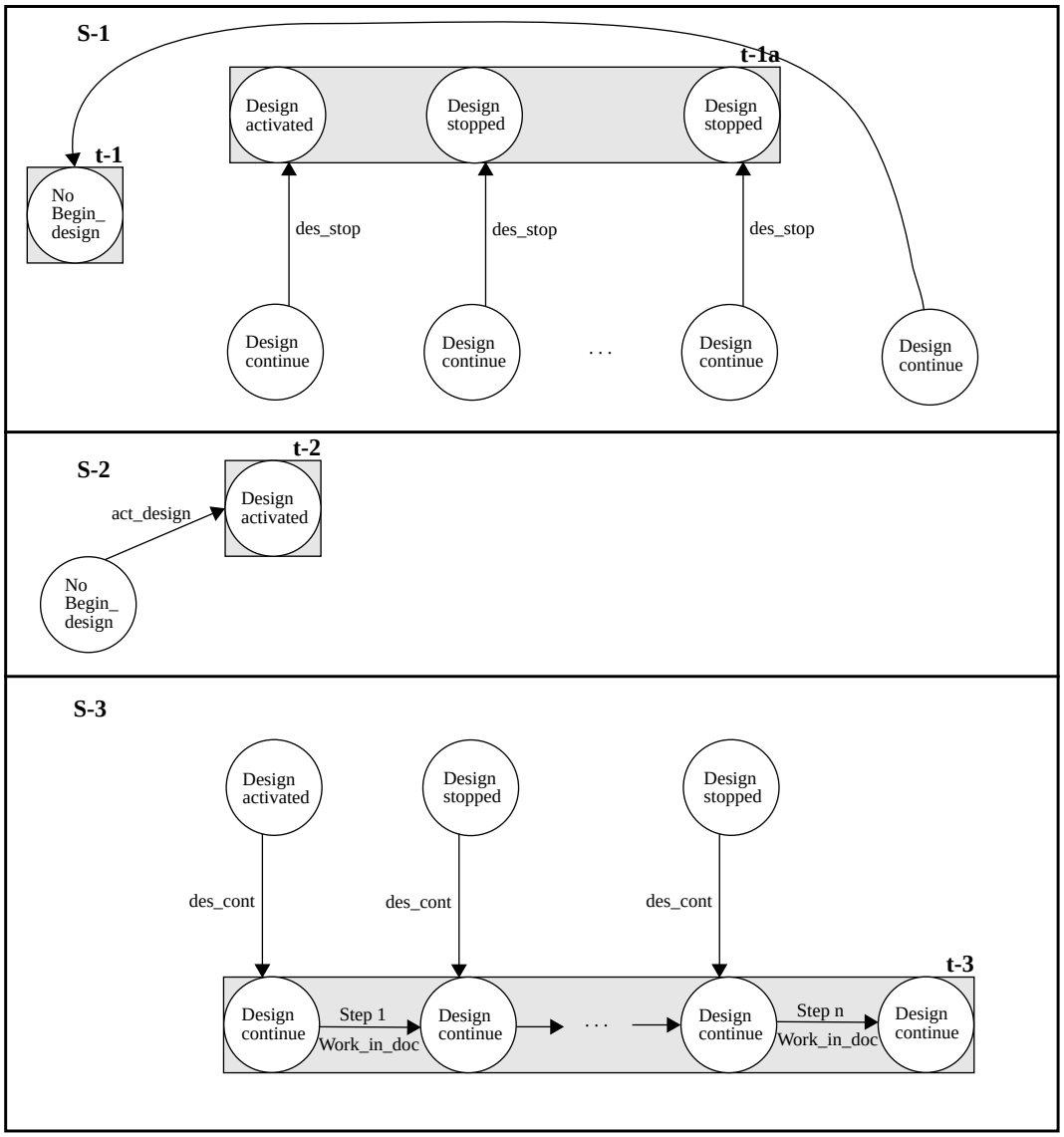


Fig. 3.4.6.1. int-Begin_design's subprocesses and traps with respect to Design Engineer

In Fig. 3.4.7. the internal behaviour of the operation *Change_role_d* (int-Change_role_d) is depicted.

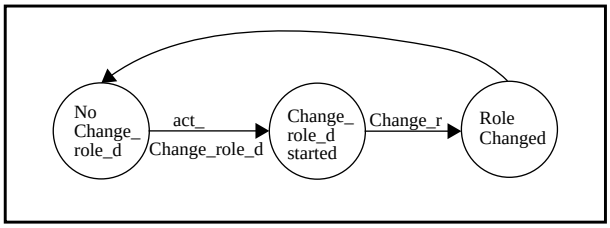


Fig. 3.4.7. int-Change_role_d: STD of its internal behaviour

This operation is called by the design engineer, as soon as he wants to change to the designer role. The two subprocesses (S-4 and S-5) and the two traps (t-4 and t-5) can be found in Fig. 3.4.7.1. If the design engineer calls *Change_role_d*, the manager Design Engineer, is asked to transit from state *Neutral State* to state *Role Designer.* Notwithstanding, Design Engineer is still outside the state *Role Designer.* This means, that int-Change_role_d is in its subprocess S-4 and expectantly in trap t-4.

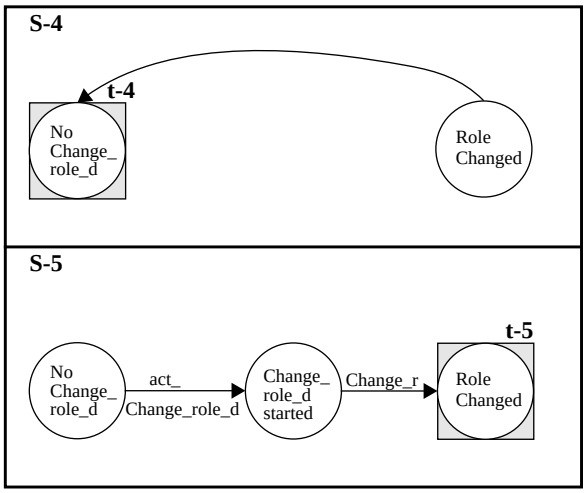


Fig. 3.4.7.1. int-Change_role_d's subprocesses and traps with respect to Design Engineer

The manager Design Engineer can only make the transition, if int-Change_role_d is in trap t-4, and if the Design Engineer is prescribing other subprocesses and traps, but this will be explained later on. As soon as Design Engineer is in state *Role Designer*, subprocess S-5 is prescribed and the role has been changed to the designer role. When the Design Engineer returns back to the state *Neutral State,* int-Change_role_d must be in subprocess S-5 and in trap t-5.

In Fig. 3.4.8. the internal behaviour of the operation *Cont_design* (int-Cont_design) is displayed, and in Fig. 3.4.8.1. the subprocesses belonging to int-Cont_design are represented. The operation *Cont_design* is also an operation which can be called by the design engineer himself. This operation allows a design engineer to really start his design activity. of course before calling the operation *Cont_design*, Design Engineer is in its state *Role Designer*, and prescribes subprocess S-6 to int-Cont_design.

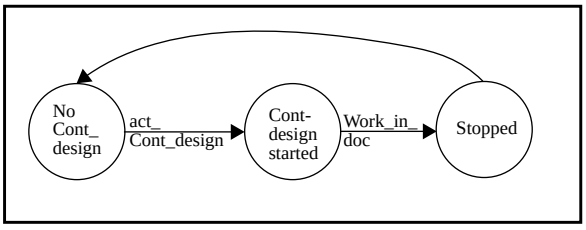


Fig. 3.4.8. int-Cont_design: STD of its internal behaviour

The Design Engineer can only make a transition to its state *Real Design*, if int-Cont_design is in subprocess S-6 and in trap t-6.

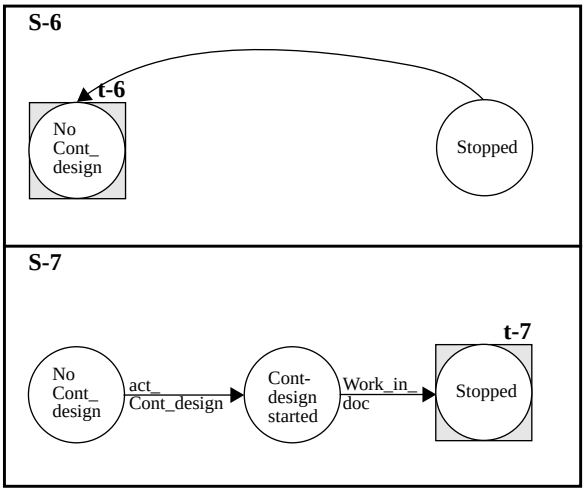


Fig. 3.4.8.1. int-Cont_design's subprocesses and traps with respect to Design Engineer

Now the last operation in the context of the design activity is explained. In Fig. 3.4.9. the internal behaviour of the operation *Stop_design* (int-Stop_design) and in Fig. 3.4.9.1. its subprocesses and traps are described. This operation will be called by the design engineer as soon as he wants to stop his design activity.

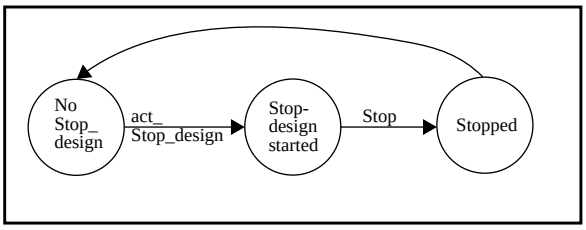


Fig. 3.4.9. int-Stop_design: STD of its internal behaviour

The Design Engineer can only make the transition back to its state *Role Designer*, if int-Stop_design is in subprocess S-8 and in trap t-8.

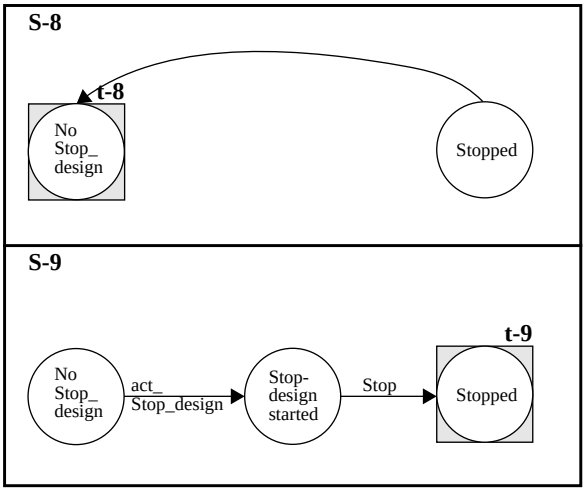


Fig. 3.4.9.1. int-Stop_design's subprocesses and traps with respect to Design Engineer

This finishes the explanation of the internal operations which corresponds to the design activity context of the design engineer. Now will be continued with the clarification of the review activity which can also be performed by the design engineer.

In Fig. 3.4.10. the internal behaviour of operation *Begin_review* (int-Begin_review) is repre-

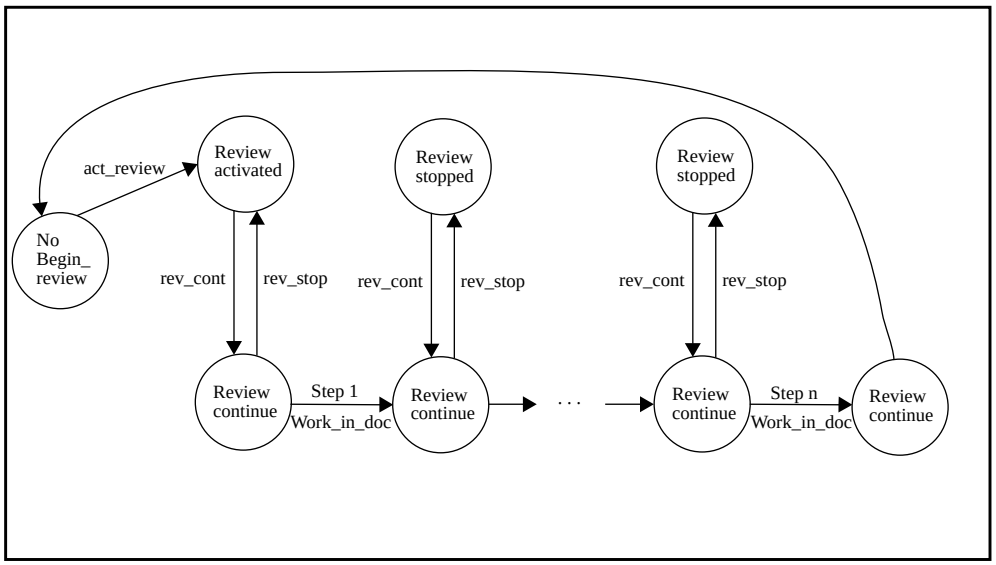


Fig. 3.4.10. int-Begin_review: STD of its internal behaviour

sented. The three subprocesses (S-10, S-11 and S-12) and the four traps (t-10, t-10a, t-11 and t-12) belonging to int-Begin_review can be found in Fig. 3.4.10.1. The design engineer waits for the tasks the project manager is going to give him. If the project manager calls e.g. *Begin_review*, the manager Design Engineer (see Fig. 3.4.15.) is requested to transit from *State Neutral* to *Starting Review.* However, Design Engineer is still outside the state *Starting Review.* This indicates, that int-Begin_review is in its subprocess S-10, and hopefully within trap t-10. The manager Design Engineer can (and will) make the transition from *State Neutral* to *Starting Review*, only after int-Begin_review has reached trap t-10. When the state *Starting Review* has been entered, Design Engineer appoints subprocess S-11 to int-Begin_review. Considering that int-Begin_design was in trap t-10, it directly starts inside subprocess S-11 from state *No Begin_review*, and then nearly right away entering trap t-11. When trap t-11 has been entered, Design Engineer leaves the state *Starting Review* and moves to state *Neutral State*. Because of the transition Design Engineer prescribes subprocess S-10 to int-Begin_review. For int-Begin_review, this means that it is waiting in its state *Review activated* and trapped in trap t-10a of subprocess S-10. Only when the design engineer really wants to perform the review activity, he must utilize the operation *Change_role_r* to reach the state *Role Reviewer* of the manager Design Engineer.

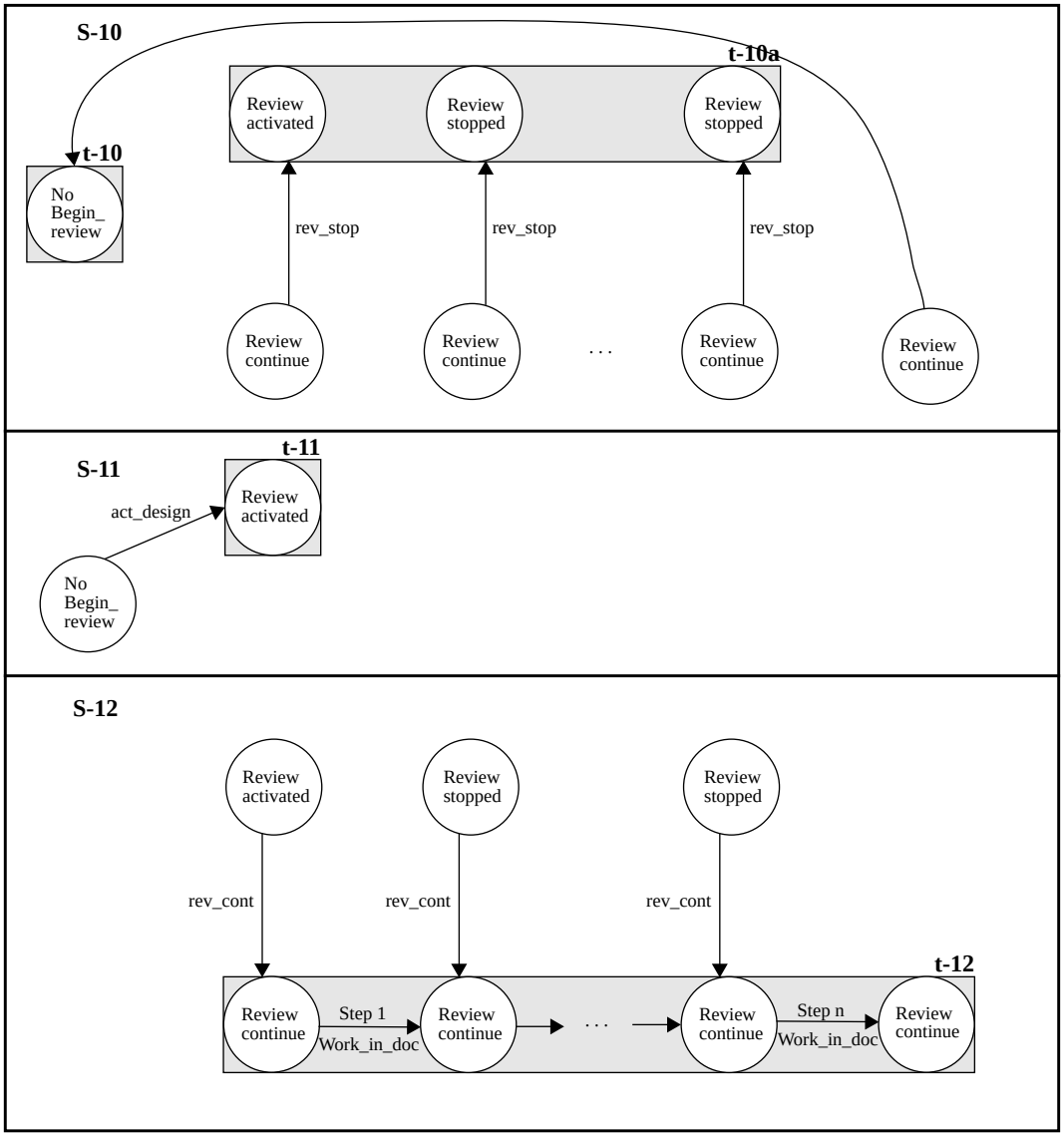


Fig. 3.4.10.1. int-Begin_review's subprocesses and traps with respect to Design Engineer

When the design engineer calls the operation *Cont_review*, Design Engineer is invited to transit from state *Role Reviewer* to state *Real Review.* Because int-Begin_review is in subprocess S-10 and in trap t-10a, Design Engineer makes the transition to state *Real Review.* Upon entering. *Real Review,* Design Engineer now prescribes subprocess S-12 (instead of S-10) to int-Begin_review. As int-Begin_review was in trap t-10a, it directly starts within subprocess S-12 from state *Review activated* thereby almost immediately entering trap t-12. Now the real review is happening. The review can be regarded as a sequence of activity steps. In Fig. 3.4.10.1. this is represented by *step 1 Work_in_doc ... step n Work_in_doc*. If the designer wants to stop his review activity, Design Engineer will prescribe subprocess S-10 for int-Begin_review. As int-Begin_review was in trap t-12, it starts inside S-10 from state *Review Continue*, thereby nearly immediately entering trap t-10a of subprocess S-10. When the design engineer wants to go on with the review (operation *Cont_review*), Design Engineer will prescribe subprocess S-12 for him and so he is allowed to perform his review activity again. In Fig. 3.4.11. the internal behaviour of the operation *Change_role_r* (int-Change_role_r) is represented.

This operation is called by the design engineer, as soon as he wants to change to the reviewer role.

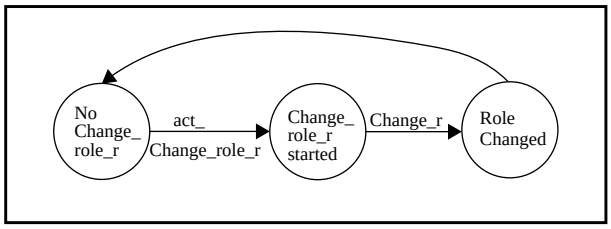


Fig. 3.4.11. int-Change_role_r: STD of its internal behaviour

The two subprocesses (S-13 and S-14) and the two traps (t-13 and t-14) are depicted in Fig. 3.4.11.1. If the design engineer calls *Change_role_r*, the manager Design Engineer, is asked to transit from state *Neutral State* to state *Role Reviewer*. Although, Design Engineer is still outside the state *Role Reviewer*, it means, that int-Change_role_r is in its subprocess S-13 and hopefully in trap t-13.

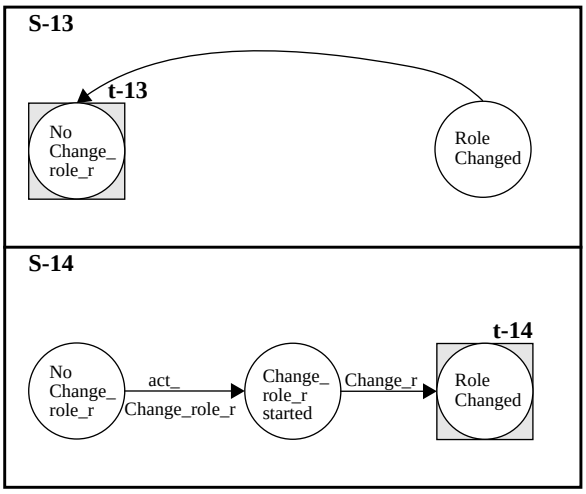


Fig. 3.4.11.1. int-Change_role_r's subprocesses and traps with respect to Design Engineer

The manager Design Engineer can only make the transition, if int-Change_role_r is in trap t-13, and if the Design Engineer prescribes other subprocesses and traps, but this will be explained later on. Then subprocess S-14 is prescribed and the role has been changed to the reviewer role. When the Design Engineer returns back to the state *Neutral State*, int-Change_role_r must be in subprocess S-14 and in trap t-14.

In Fig. 3.4.12. the internal behaviour of the operation *Cont_review* (int-Cont_review) is showed, and in Fig. 3.4.12.1. the subprocesses belonging to int-Cont_review are demonstrated.

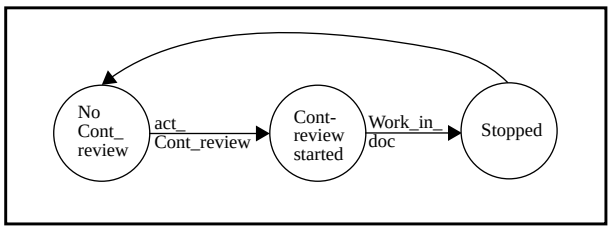


Fig. 3.4.12. int-Cont_review: STD of its internal behaviour

The operation *Cont_review* is also an operation which can be called by the design engineer himself. This operation allows a design engineer to really start his review activity. Of course before calling the operation *Cont_review*, Design Engineer is in its state *Role Reviewer*, and prescribes subprocess S-15 to int-Cont_review. The Design Engineer can only make a transition to its state *Real Review*, if int-Cont_review is in subprocess S-15 and in trap t-15.

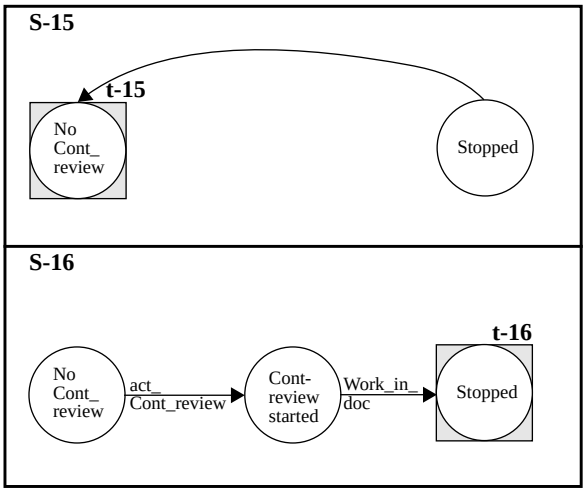


Fig. 3.4.12.1. int-Cont_review's subprocesses and traps with respect to Design Engineer

The last operation in the context of the review activity will be explained. In Fig. 3.4.13. the internal behaviour of the operation *Stop_review* (int-Stop_review) and in Fig. 3.4.13.1. its subprocesses and traps are described. This operation will be called by the design engineer as soon as he wants to stop his review activity.

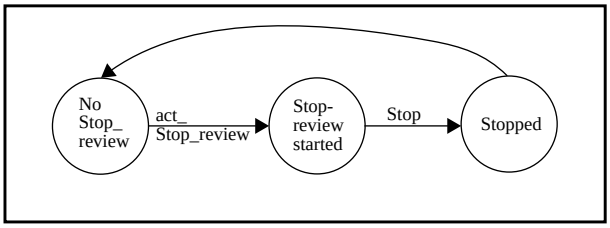


Fig. 3.4.13. int-Stop_review: STD of its internal behaviour

The Design Engineer can only make a transition back to its state *Role Reviewer*, if int-Stop_review is in subprocess S-17 and in trap t-17.

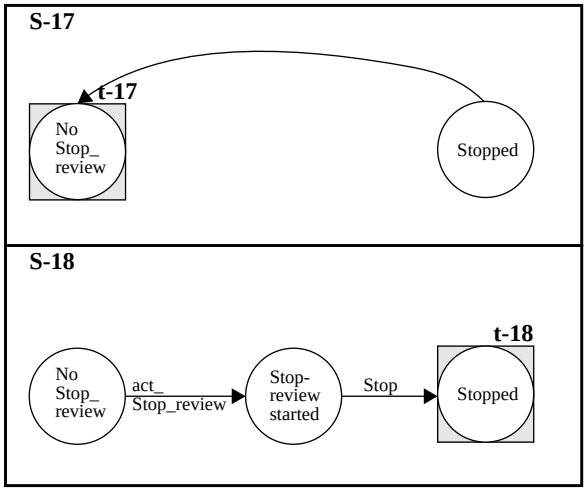


Fig. 3.4.13.1. int-Stop_review's subprocesses and traps with respect to Design Engineer

This finalizes the explanation of the internal operations which corresponds to the review activity context of the design engineer.

In Fig. 3.4.14. the internal behaviour of the extra added operation *Change_neutral* (int-Change_neutral) is presented. It's an extra operation because in the TEMPO description this operation hasn't been used. In Fig. 3.4.14.1. the subprocesses belonging to int-Change_neutral are demonstrated. This operation is necessary to let Design Engineer make the transition back to state *Neutral State*. When Design Engineer is in state *Role Designer* or in state *Role*

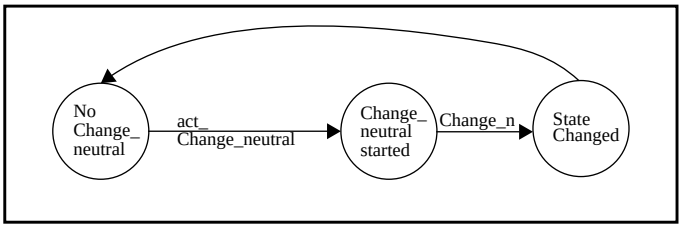


Fig. 3.4.14. int-Change_neutral: STD of its internal behaviour

Reviewer, the design engineer can make the decision to go on with the design or review. This means he has to use the operation *Cont_design* or *Cont_review.* If he wants to go back to his neutral state in order to wait for some new tasks of the project manager, he must use the operation *Change_neutral*.

The manager Design Engineer can only make a transition to its state *Neutral State*, if int-Change_neutral is in subprocess S-19 and in trap t-19.

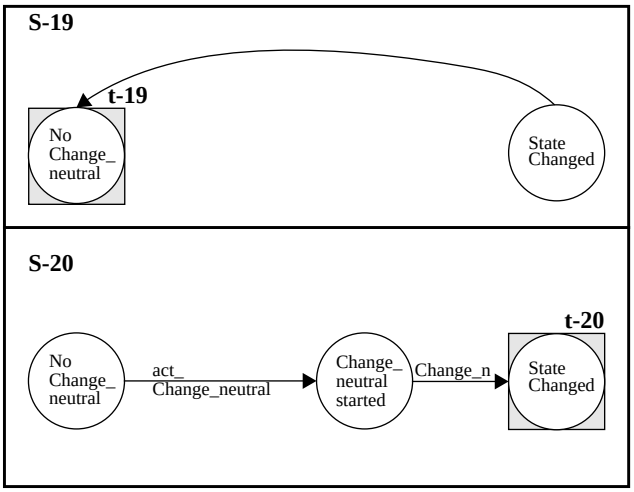


Fig. 3.4.14.1. int-Change_neutral's subprocesses and traps with respect to Design Engineer

3.4.3 The Manager Process: Design Engineer

The external behaviour of Design Engineer is used as the manager process. The employees are the nine internal behaviours of Design Engineer (int-Begin_design, int-Change_role_d, int-Cont_design, int-Stop_design, int-Begin_review, int-Change_role_r, int-Cont_review, int-Stop_review and int-Change_neutral). The internal behaviours of Project Manager are employees too, but they are not used because the Project Manager is not inside the scope of this example.

The manager's starting state is state *Neutral State*. Suppose that the project manager calls the operation *Begin_design* of design engineer. So after the calling, Design Engineer is requested

to make the transition to the state Starting Design. This is only possible when int-Begin_design is in subprocess S-1 and in trap t-1. As soon as int-Begin_design is in subprocess S-2 and in trap t-2, the manager Design Engineer is leaving *Starting Design* and returns to its *Neutral State*. It is possible for the design engineer to start another design or review, but only if he is called to do so.

Up to now, the design engineer was able to obtain his tasks from the project manager. When he wants to perform his activities, he must change to the role under which he can accomplish these activities. So in case of a design activity, he has to change to the state *Role Designer* and in case of a review activity he has to change to the state *Role Reviewer.* Now the design engineer can decide if he wants to do the real design. In this case he must first change to the state *Role Designer.* This can only be done if int-Begin_design is in subprocess S-1 and in trap t-1a, int-Change_role_d must be in subprocess S-4 and in trap t-4, and int-Change_neutral is in subprocess S-20 and trapped in trap t-20. When the design engineer wants to review a document, Design Engineer must be able to change to state *Role Reviewer.* This transition only can be made if int-Begin_review is in subprocess S-10 and in trap t-10a, int-Change_role_r is in subprocess S-13 and in trap t-13, and int-Change_neutral is in its subprocess S-20 and in trap t-20.

To return back to state *Neutral state*, Design Engineer must prescribe subprocess S-5, trap t-5 for int-Change_role_d and subprocess S-19, trap t-19 for int-Change_neutral if the manager is in state *Role designer.* When the Design Engineer is in state *Role Reviewer,* the manager must prescribe subprocess S-14, trap t-14 for int-Change_role_r and subprocess S-19, trap t-19 for int-Change_neutral.

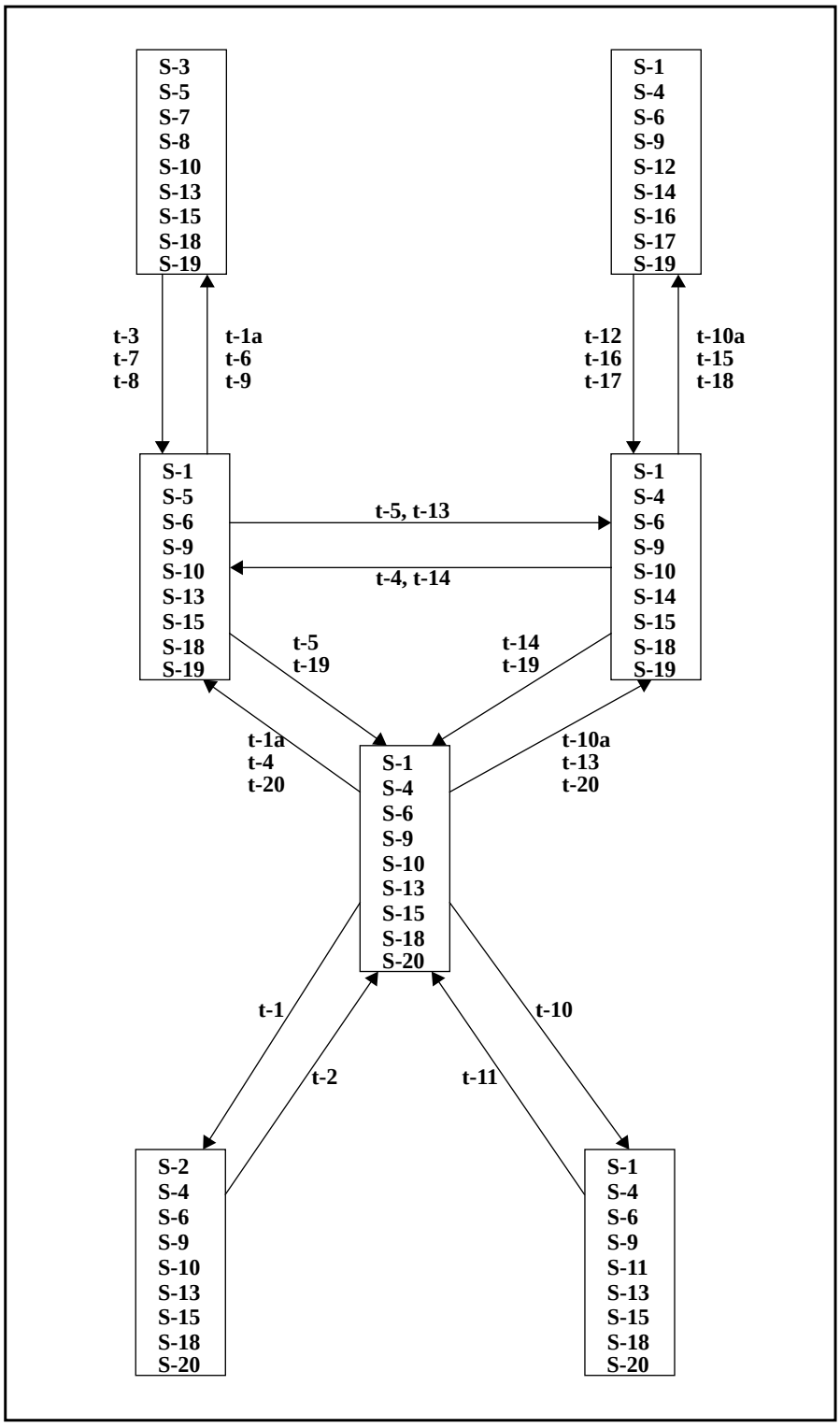


Fig. 3.4.15. Design Engineer, manager of nine employees

Of course the design engineer can change between his roles (designer role and reviewer role). This is possible by using the operations *Change_role_d* and *Change_role_r*. If the manager is in state *Role Designer*, and the design engineer wants to do the review activity, then Design Engineer must prescribe subprocess S-5, trap t-5 for int-Change_role_d and subprocess S-13, trap t-13 for int-Change_role_r to transit to state *Role Reviewer*. If the manager is in state *Role Reviewer*, and the design engineer wishes to do the design activity, then Design Engineer must prescribe subprocess S-4, trap t-4 for int-Change_role_d and subprocess S-14, trap t-14 for int-Change_role_r to make the transition to state *Role Designer*.

To perform his real activities, the Design Engineer must perform the transitions to the states *Real Design* and *Real Review*. To transit to state *Real Design*, the Design Engineer must prescribe the next subprocesses and traps: S-1, t-1a for int-Begin_design, S-6, t-6 for int-Cont_design and S-9, t-9 for int-Stop_design. When the design engineer wants to stop his design activity, he has to use the operation *Stop_design*. To make the transition to the state *Role Designer*, the manager prescribes the following subprocesses and traps: S-3, t-3 for int-Begin_design, S-7, t-7 for int-Cont_design and S-8, t-8 for int-Stop_design.

If the manager is in state *Role Reviewer* and wants to change to state *Real Review*, the Design Engineer must assign the next subprocesses and traps: S-10, t-10a for int-Begin_review, S-15, t-15 for int-Cont_review and S-18, t-18 for int-Stop_review. When the design engineer wants to stop his review work, he has to use the operations *Stop_review*. To make the transition to the state *Role Review*er, the manager prescribes the following subprocesses and traps: S-12, t-12 for int-Begin_review, S-16, t-16 for int-Cont_review and S-17, t-17 for int-Stop_review. In the following two paragraphs two alternative managers will be described.

3.4.4 An Alternative for Manager Design Engineer: Design Engineer2

In Fig. 3.4.16., an alternative manager Design Engineer2 is given. This manager process is based upon the external behaviour of Design Engineer (see Fig. 3.4.5.). Design Engineer2 has been modelled, in order to prevent the adding of one extra operation (operation *Change_neutral)*. So in Design Engineer2, the state *Neutral State* and the operation *Change_neutral* have been removed. Note that this manager reflects the resemblance with the modelling of the design engineer in the ADELE-TEMPO example in a more accurate way, because the same operations have been used to describe the design engineer.

3.4.5 A Second Alternative for Design Engineer: Design Engineer3

The second alternative manager (Design Engineer3) is presented in Fig. 3.4.17. The structure of this manager and the export operations are the same as those used for describing manager Design Engineer2 (see Fig. 3.4.16.). The underlying idea of this manager is, that all original export operations will be utilized. But not every operation has an internal behaviour, this means that the operations which are typical for type-2 communication don't have subprocesses and traps. So the internal behaviours: int-Change_role_d, int-Cont_design, int-Stop_design, int-Change_role_r, int-Cont_review and int-Stop_review, and the corresponding subprocesses and traps aren't used. In fact the real activities of the design engineer are described by the internal behaviours: int-Begin_design and int-Begin_review. Therefore in Design Engineer3 only two internal behaviours are of interest, instead of the eight internal behaviours used in Design Engineer2. It is clear, that this alternative is not really corresponding to the original SOCCA formalism, but it could be a useful adaptation to simplify the manager and to reduce the number of employee processes of the manager process. So it is a serious option for simplifying SOCCA models.

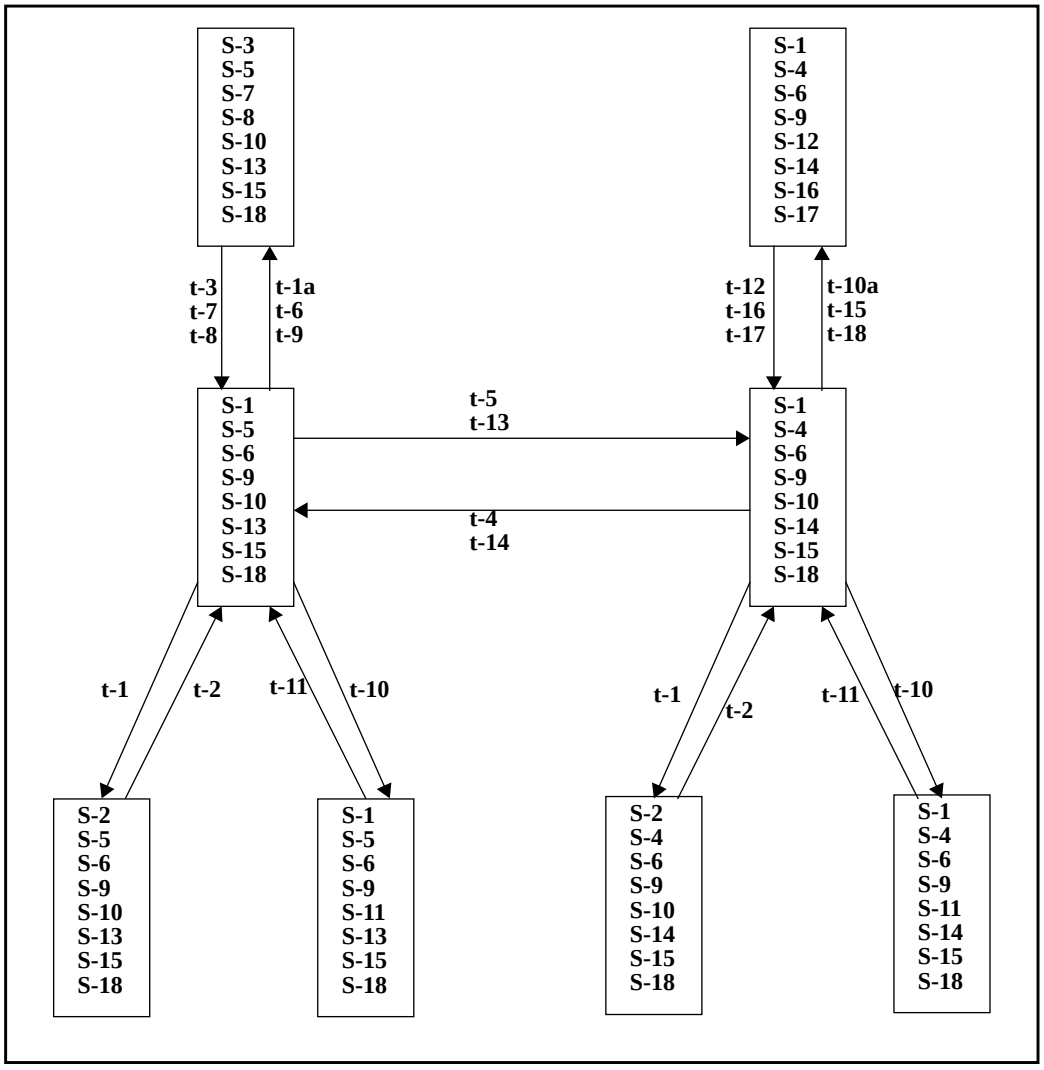


Fig. 3.4.16. Design Engineer2, alternative manager of eight employees

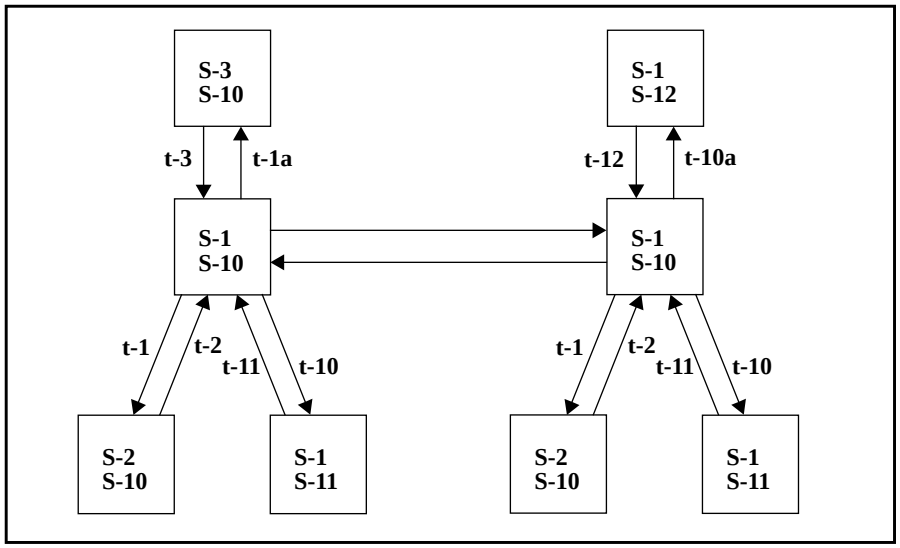


Fig. 3.4.17. Design Engineer3, alternative manager of two employees

3.5 Example 2: The MailTool

In this example a mailtool is described. In Fig. 3.5.1. a graphical representation of the mailtool is given. The mailtool comes up as a closed icon and the icon will be opened by clicking on the mailtool icon. The incoming mail is stored in the buffer of the mailtool. A mailtool consists of a main window with three push-buttons. These buttons are: a compose, a view and a reply push-button.

In case of pushing the compose button a compose window will be popped up. This window is a text window, in which the user composes messages to send. It moreover has mail header lines such as ‘To:’ and ‘Subject:’.

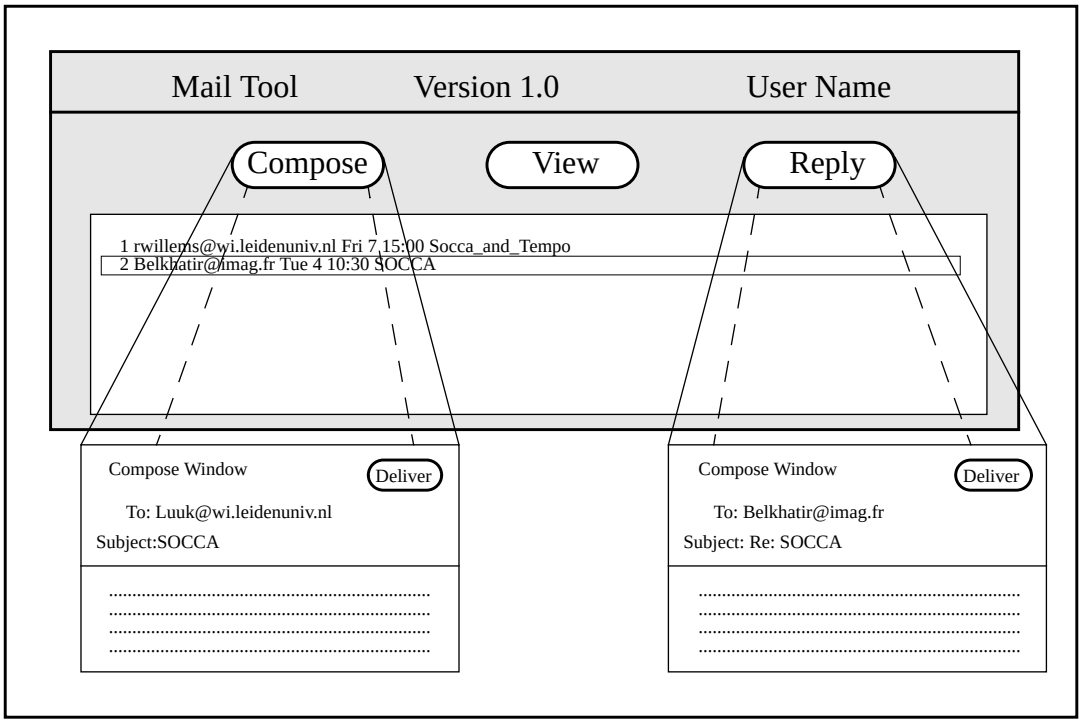


Fig. 3.5.1. MailTool

When the user pushes the view button a view window will be popped up. In the view window the user reads the mail he had received.

The pushing of the reply button causes the opening of a compose window to reply to a selected message e.g. Belkhatir@imag.fr (see Fig. 3.5.1.). The reply window is also a text window, in which the user writes messages to send. This window has mailheader lines such as ‘To:<address>’ and ‘Subject: Re:<subject>’. The <...> indicate, that the address and the subject are inserted behind the header lines automatically.

A deliver push-button, can be found in the compose window and in the reply window. This button must be pushed when the user wants to send (deliver) the message to someone else. An ok push-button is part of the view window, the pushing of this button causes the closing of the view window.

3.5.1 The Data Perspective of MailTool

In Fig. 3.5.2. a class diagram for the MailTool is given.

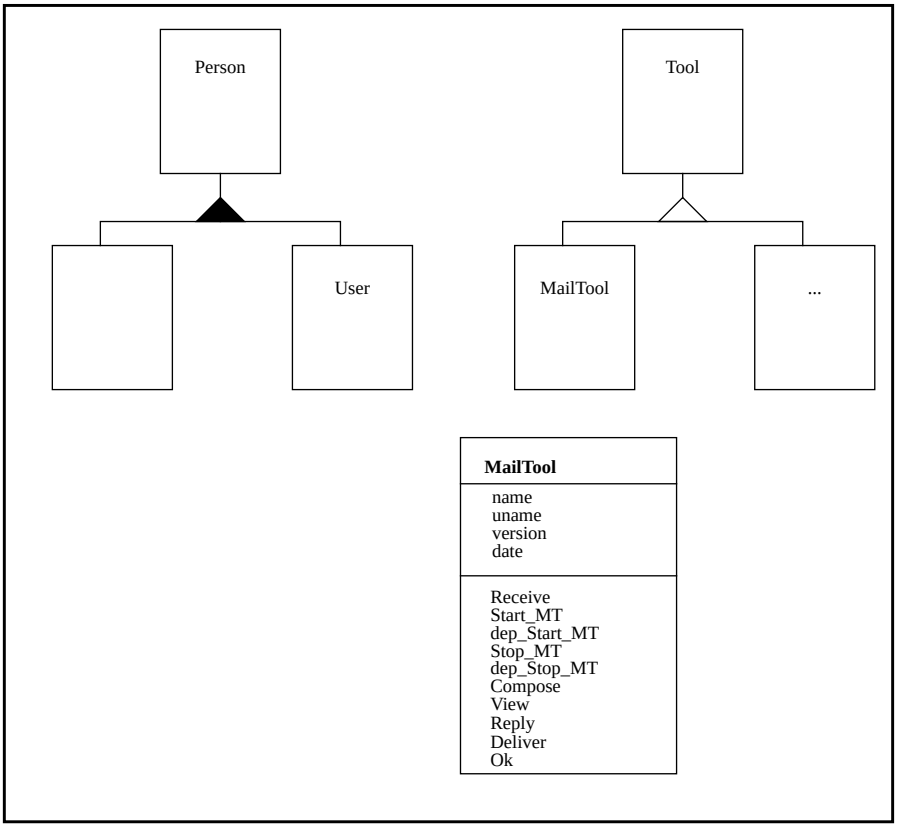


Fig. 3.5.2. Class Diagram: classes, IS-A, attributes and operations

In the class diagram is shown, that a User **IS-A** Person and a MailTool **IS-A** Tool. The User represents the person who is using the mailtool, this can be anyone in the process. In the software process several tools are involved, e.g. compiler, editor, mailtool.

In the next diagram (Fig. 3.5.3.), the general relationship is added in order to connect the different classes of Fig. 3.5.2., by means other than the relationship **IS-A**. The general relationship in Fig. 3.5.3. is called utilizes. Note that Fig. 3.5.2. and Fig. 3.5.3. must be read together. The hollow dot implies, that the MailTool may be used by a user or not at all. The black dot indicates, that the user can utilize between zero and n mailtools.

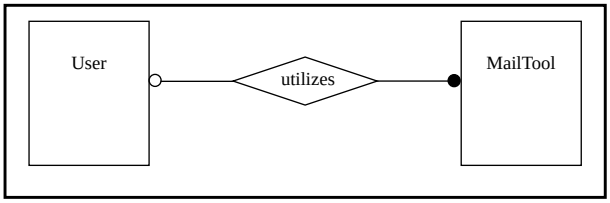


Fig. 3.5.3. Class Diagram: classes and general relationship

Now, more information will be provided which is usually not given in an EER model. The Uses relationship, expresses where the various export operations are imported. In Fig. 3.5.4., the Uses relationships can be found.

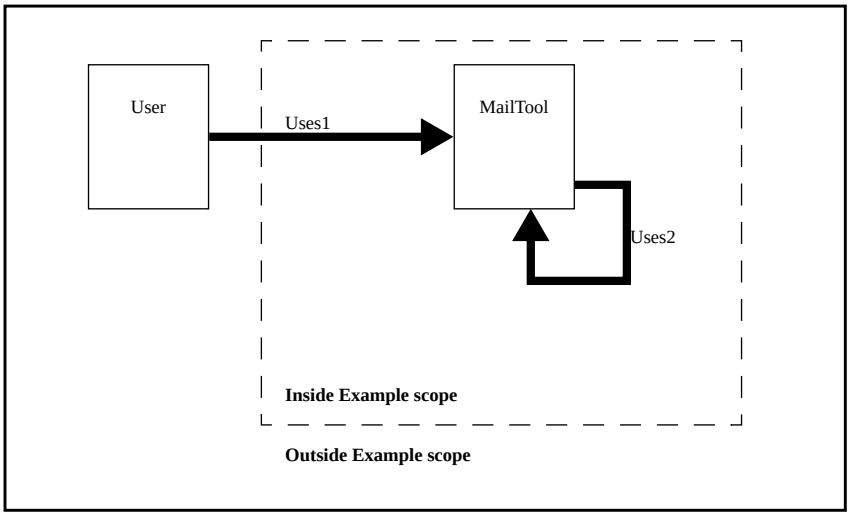


Fig. 3.5.4. Import/Export diagram

The import_lists for uses1 and uses2 are:

Uses1	**Uses2**
Start_MT	**Receive**
Stop_MT	**dep_Start_MT**
Compose	**dep_Stop_MT**
View	
Reply	
Deliver	
Ok	

3.5.2 The Behaviour Perspective of MailTool

In this paragraph, the dynamic aspects of the mailtool will be described. These aspects consist of two different types of behaviour: the external behaviour and the internal behaviour.

3.5.2.1 The Extension of the External Behaviour: Several STDs.

Up to now, the external behaviour was described by one STD. This allows only the description of sequential behaviour in the external behaviour. But in some cases it is necessary to have a parallel description in the external behaviour. So the way of modelling has to be adapted slightly. As soon as parallel behaviour has to be expressed in the external behaviour, several STDs must be used instead of one STD. Two different STD types can be distinguished to describe the external behaviour. There is a STD type called **Main-STD** and some STDs which are dependent of this Main-STD, these STD types are called **Dep-STDs**. The dependencies between a Main-STD and its Dep-STDs have to be expressed in the external behaviour. This can be done by choosing the ‘right’ export-operation names of the Dep-STDs. This means, that an export operation name of a Dep-STD, which has to make clear the dependency between the Main-STD and its own STD, must have the same name as the export operation of the Main-STD and must be preceded by the prefix ‘**dep_**’.

E.g.

STD type	***STD name***	***export operation***
Main-STD	STD1	exp_op1
Dep-STD	STD2	**dep_**exp_op1

Dependently upon on the number of different parallel behaviours which someone wants to model, 2 up to n STDs can be found in the external behaviour.

3.5.2.2 The External Behaviour of MailTool

In Fig. 3.5.5. the external behaviour of MailTool is depicted. The external behaviour consist of four STDs, each describing a different part of the behaviour of a mailtool.

1). In the first STD (**MailTool/Main Window**) the opening of the mailtool icon and the receiving of mail messages has been described. The receiving of mail messages (operation *Receive*) is expressed by the states *Starting Receive*. Although the states are the same, there is a tiny difference. In the highest state *Starting Receive*, the mailtool can receive mail when the mailtool icon is closed. In the lowest state *Starting Receive* the mailtool can receive mail when the mailtool's icon is opened and the main window has been popped up. The neutral state of the mailtool is state *Icon Closed*. As soon as a mail message has reached the mailtool, the MailTool/ Main Window is making the transition to state *Starting Receive*. In this state the mail message will be stored in the mailtool buffer. The MailTool/Main Window returns to the states *Icon Closed* or *Icon Open* after the internal receive activity has really started. When the user clicks on the mailtool icon (operation *Start_MT*), the icon is opened and a mailtool main window is opened. To close the mailtool, the user has to click on the main window. This matches to the export operation *Stop_MT.*

2). In the second STD (**ext-Compose Window**) the external behaviour of the compose window has been depicted. The manager ext-Compose Window is dependent of manager MailTool/ Main Window, because this second window can only be popped up as the main window has been popped up too. The dependency between these windows is expressed by the export operations *dep_Start_MT* and dep*_Stop_MT.* The states *Icon Closed* and *Icon Open* are the same as in the first STD. The user can pop up a compose window by pushing the compose button in the main window. Pushing of the compose button corresponds to the export operation *Compose*. When the user has finished writing his message in the compose window and decides to send it, he has to push the deliver button. The export operation *Deliver* corresponds to the pushing of the deliver button. After the pushing of the deliver button, ext-Compose Window has to return to state *Icon Open* when the main window is open at that moment or to state *Icon Closed* when the main window has been closed.

3). In the third STD (**ext-View Window**) the external behaviour of the view window has been described. This ext-View Window is also dependent of MailTool/Main Window, because this third window can only be popped up as the main window has been popped up too. The dependency between these windows is expressed by the export operations *dep_Start_MT* and *dep_Stop_MT.* The states *Icon Closed* and *Icon Open* are also the same as in the first STD. When the user pushes the view button in the main window, the view window is popped up. Pushing of the view button corresponds to the export operation *View.* When the user has finished reading the message in the view window and he wants to close it, he has to push the ok button. After the pushing of the ok button, ext-View Window has to return to state *Icon Open*

when the main window is open at that moment or to state *Icon Closed* when the main window has been closed.

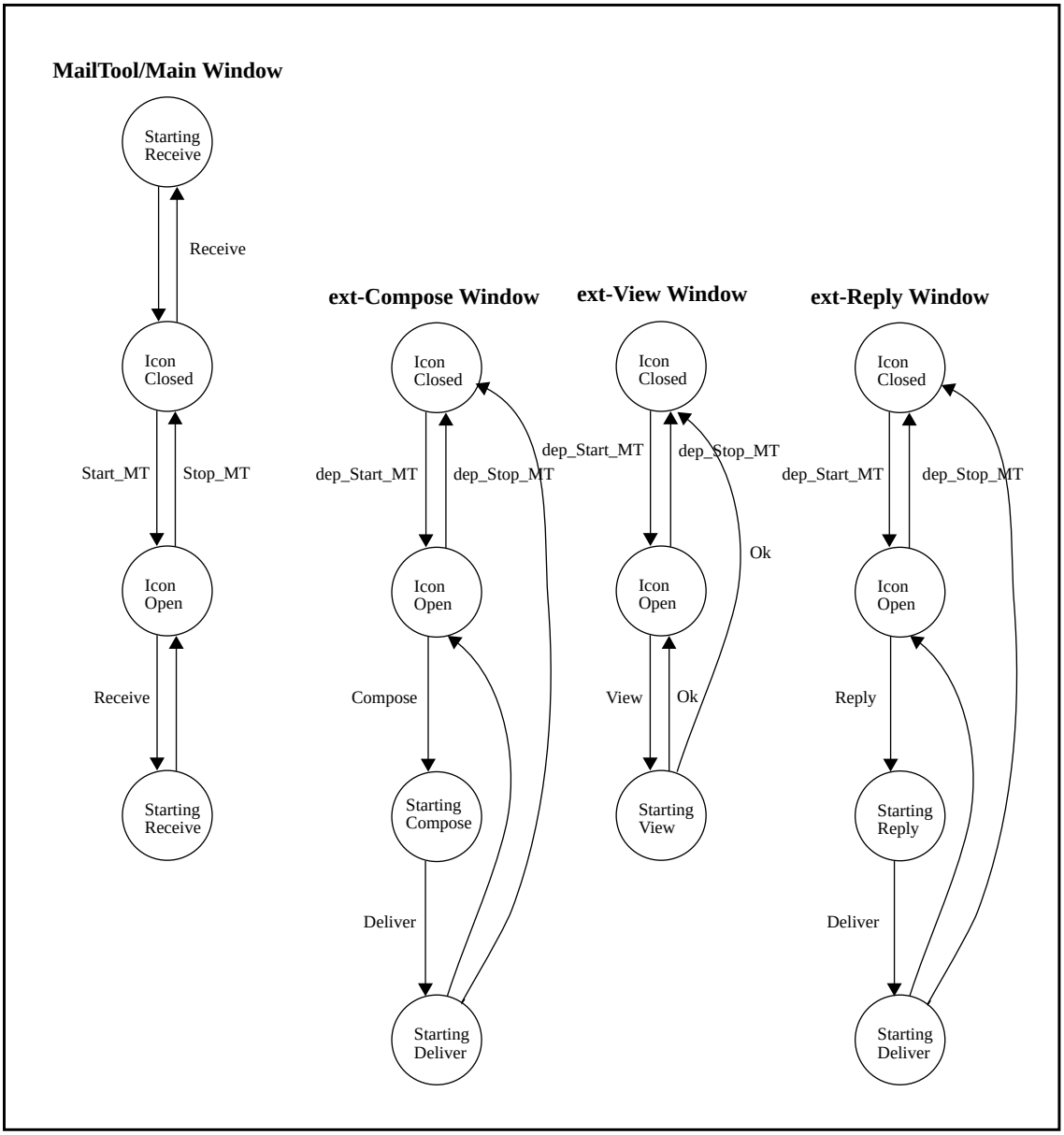


Fig. 3.5.5. MailTool: STD of the external behaviour

4). The fourth and last STD (**ext-Reply Window**) is nearly the same as the second STD (ext-Compose Window), only one export operation is different. Instead of export operation *Compose*, the export operation *Reply* has been used. In this window it's possible to reply to someone's message. By pushing the reply button, a reply window is popped up and the address of the sender is automatically added at the place where the user normally had to type the email address. Because of the great resemblance with the second STD this finishes the discussion of this Dep-STD (ext-Reply Window).

Describing the external behaviour with more than one STD influences the way of describing the subprocess and traps. The idea is, that the Main-STD is a manager of the Dep-STDs. This

means, that the Main-STD is a manager of a part of the manager (Dep-STDs). Here the Main-STD is: MailTool/Main Window and the Dep-STDs are: ext-Compose Window, ext-View Window and ext-Reply Window. So for the Dep-STDs the right subprocesses and traps with respect to the Main-STD have to be chosen. These subprocesses are called manager subprocesses (**MS**) and the traps are called manager traps (**mt**). The three parts of the external behaviour with respect to the Main-STD (MailTool/Main Window) are: ext-Compose Window, ext-View Window and ext-Reply Window. The convention for naming these external behaviours prefixes '**ext-**' to the possibly abbreviated name of the corresponding Dep-STD.

In Fig. 3.5.5. the external behaviour of the Compose Window (ext-Compose Window) is presented. The two manager subprocesses (MS-1 and MS-2) and the two manager traps (mt-1 and mt-2) belonging to ext-Compose Window can be found in Fig. 3.5.5.1.

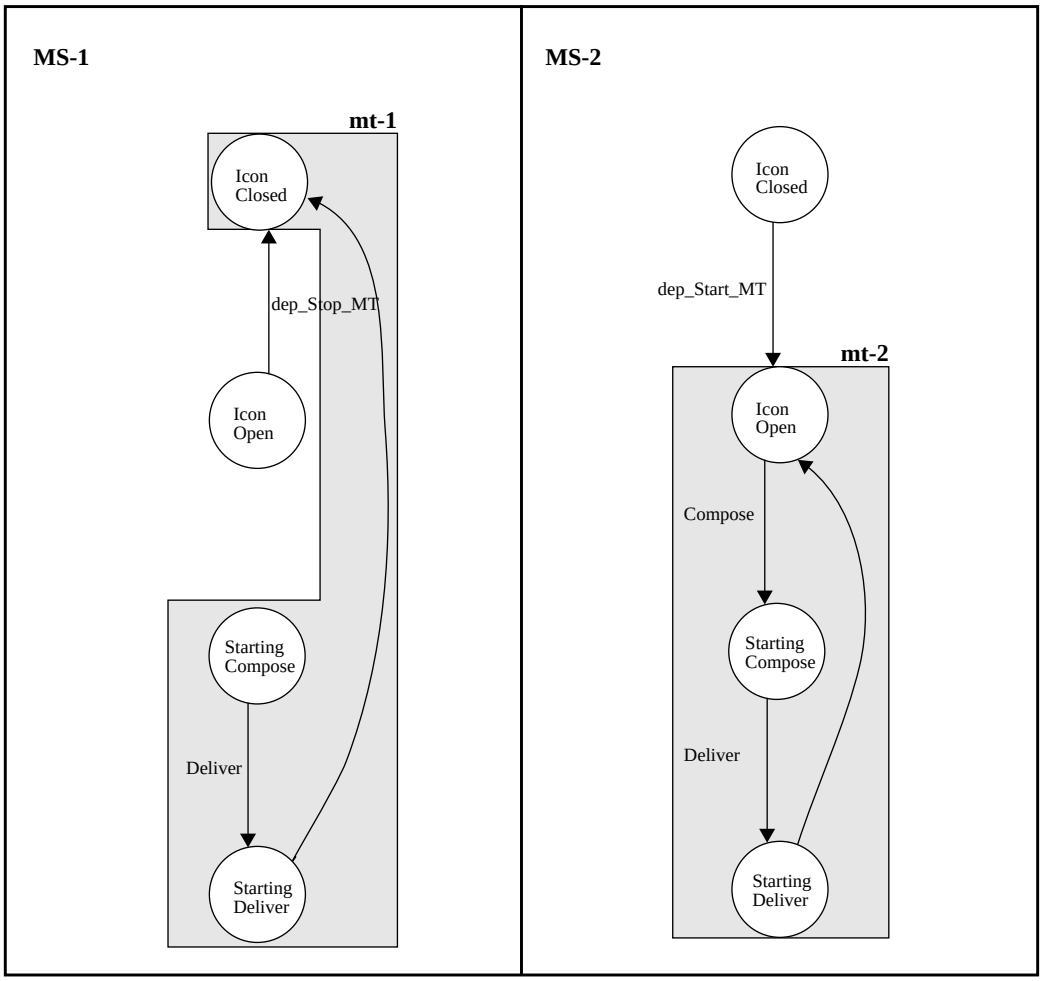


Fig. 3.5.5.1. ext-Compose Window's subprocesses and traps with respect to MailTool/Main Window

The starting state of MailTool/Main Window is *Icon Closed*. In this state the MailTool/Main Window prescribes manager subprocess MS-1 for ext-Compose Window. As soon as the user has opened the mailtool icon, MailTool/Main Window is invited to transit from state *Icon Closed* to state *Icon Open*. The manager can (and will) make the transition from Icon Closed to Icon Open, only when ext-Compose Window is trapped in manager trap mt-1. The large manager trap mt-1 allows the MailTool/Main Window to make the transition to manager subprocess MS-2 as soon as possible. When the transition has been made, MailTool/Main Window assigns manager subprocess MS-2 to ext-Compose Window. Considering that ext-Compose Window was in manager trap mt-1, it directly starts inside manager subprocess MS-2, and thereby nearly right away entering manager trap mt-2. If the user wants to close the mail-

tool icon, he has to click on the main window. By doing this, the manager process is invited to go back to its state *Icon Closed*. For ext-Compose Window, it's no problem to return to subprocess MS-1, because the manager trap mt-2 is chosen to be as large as possible, to allow the manager making the transition to its state *Icon Closed* as soon as possible. The manager process prescribes manager subprocess MS-1 in case of closeness of the mailtool icon, and manager subprocess MS-2 in case of openness of the mailtool icon.

The external behaviour of the View Window (ext-View Window) is given in Fig. 3.5.5. The two manager subprocesses (MS-3 and MS-4) and the two manager traps (mt-3 and mt-4) belonging to ext-View Window are presented in Fig. 3.5.5.2.

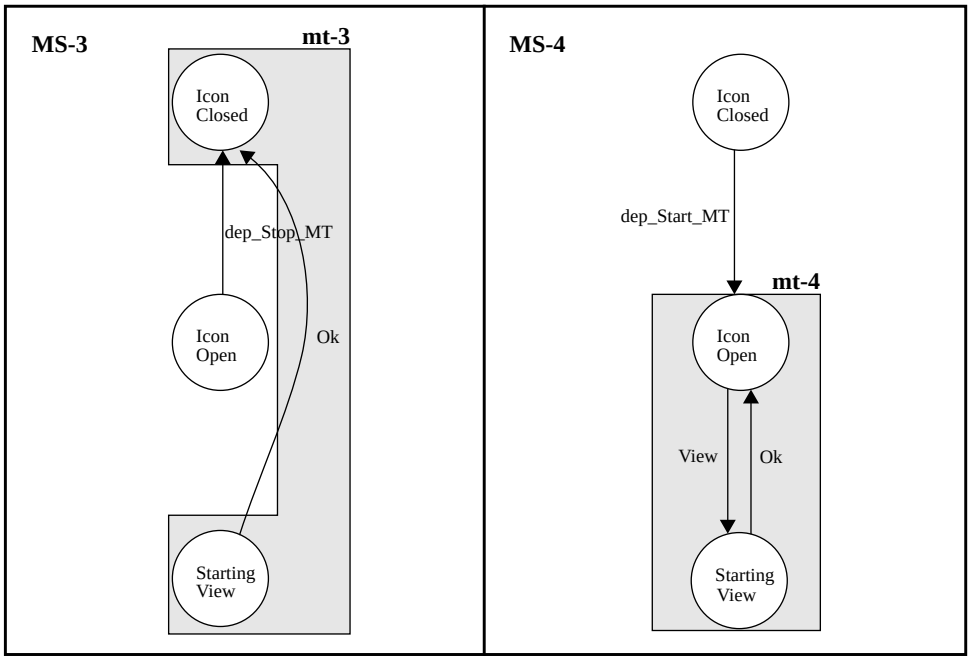


Fig. 3.5.5.2. ext-View Window's subprocesses and traps with respect to MailTool/Main Window

The starting state of MailTool/Main Window is *Icon Closed*. In this state MailTool/Main Window prescribes manager subprocess MS-3 for ext-View Window. As soon as the user has opened the mailtool icon, MailTool/Main Window is invited to transit from state *Icon Closed* to state *Icon Open*. The manager MailTool, can (and will) make the transition from Icon Closed to Icon Open, when ext-View Window is trapped in manager trap mt-3. When the transition has been made, MailTool/Main Window assigns manager subprocess MS-4 to ext-View Window. Keeping in view, that ext-View Window was in manager trap mt-3, it directly starts inside manager subprocess MS-4, and thereby nearly right away entering manager trap mt-4. If the user wants to close the mailtool icon, he has to click on the main window. By doing this, the manager process MailTool/Main Window is invited to go back to its state *Icon Closed*. For ext-View Window, it's no problem to return to subprocess MS-3, because the manager trap mt-4 covers the states *Icon Open* and *Starting View*, to allow the manager making the transition to its state *Icon Closed* as soon as possible. The manager process prescribes manager subprocess MS-3 when the mailtool icon is closed, and manager subprocess MS-4 in case of the mailtool icon being open.

In Fig. 3.5.5. the external behaviour of the Reply Window (ext-Reply Window) is shown. The two manager subprocesses (MS-5 and MS-6) and the two manager traps (mt-5 and mt-6) belonging to ext-Reply Window can be found in Fig. 3.5.5.3.

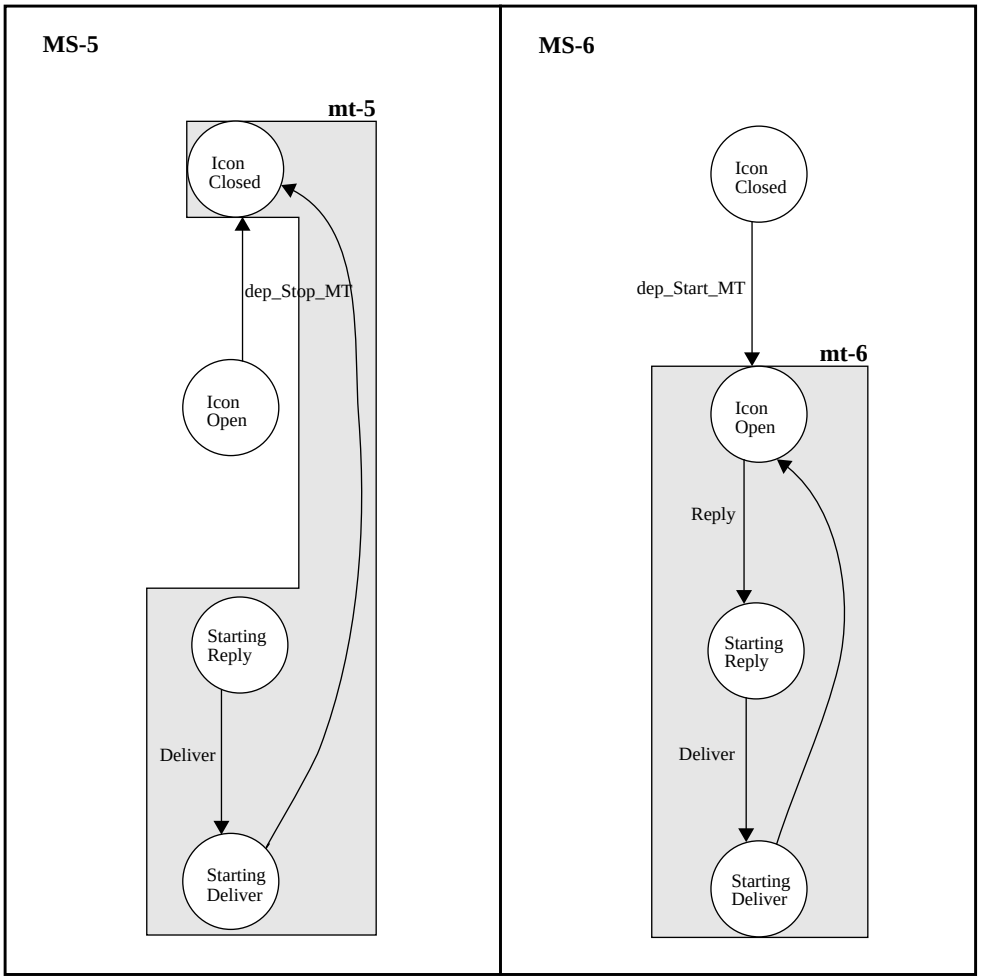


Fig. 3.5.5.3. ext-Reply Window's subprocesses and traps with respect to MailTool/Main Window

The starting state of MailTool/Main Window is *Icon Closed*. In this state the MailTool/Main Window prescribes manager subprocess MS-5 for ext-Reply Window. As soon as the user has opened the mailtool icon, MailTool/Main Window is invited to transit from state *Icon Closed* to state *Icon Open*. The manager MailTool/Main Window, can (and will) make the transition from Icon Closed to Icon Open, only after ext-Reply Window is trapped in manager trap mt-5. Of course, other (manager) subprocesses and (manager) traps are involved to let the MailTool make its transition to state *Icon Open*. When the transition has been made, MailTool/Main Window assigns manager subprocess MS-6 to ext-Reply Window. Bearing in mind, that ext-Reply Window was in manager trap mt-5, it directly starts inside manager subprocess MS-6, and thereby nearly right away entering manager trap mt-6. If the user decides to close the mail-tool icon, he has to click on the main window. By doing this, the manager process is invited to transit to its state *Icon Closed*. For ext-Reply Window, it's no problem to return to subprocess MS-5, because the manager trap mt-6 has been chosen as large as possible, to allow the manager making the transition to its state *Icon Closed* as soon as possible. The manager process prescribes manager subprocess MS-5 when the mailtool icon is closed, and manager subprocess MS-6 when the mailtool icon is open.

3.5.2.3 The Internal Behaviour of MailTool

Now the internal behaviours of all export operations have to be specified. These operations can be called by the user who uses the mailtool and by the mailtool itself. Before the specification of the internal behaviours, a list of possibly imported operations for each uses relationship will be presented.

Uses1	**Uses2**
Start_MT	**Receive**
Stop_MT	**dep_Start_MT**
Compose	**dep_Stop_MT**
View	
Reply	
Deliver	
Ok	

This list of operation names present all possibly imported operations. The user is allowed to call the export operations *Start_MT*, *Stop_MT*, *Compose*, *View*, *Reply*, *Deliver* and *Ok*. The export operations with the prefix '**dep_**' don't have any subprocesses and traps. They are just used to express the dependency between the Main-STD and the Dep-STD. So they are not the same as the normally used export operations. The seven (normal) internal behaviour specifications are depicted in Fig. 3.5.6. through to Fig. 3.5.13. Below each internal behaviour specification the corresponding subprocesses and traps are presented. These subprocesses and traps are shown in Fig. 3.5.6.1. through to Fig. 3.5.13.1.

In Fig. 3.5.6. the internal behaviour of the operation *Start_MT* (int-Start_MT) is expressed.

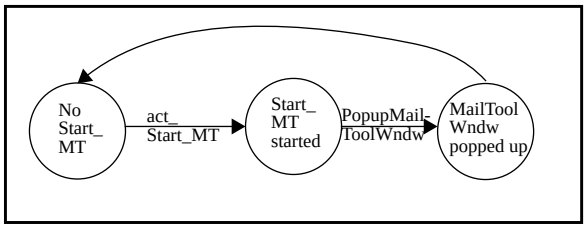


Fig. 3.5.6. int-Start_MT: STD of its internal behaviour

This operation is called by the user, when the user wants to open the mailtool icon. The opening of the mailtool icon is achieved by clicking twice on the icon. The two subprocesses (S-1 and S-2) and the two traps (t-1 and t-2) are given in Fig. 3.5.6.1. When the operation *Start_MT* is called, the manager MailTool/Main Window, is asked to transit from state *Icon Closed* to state *Icon Open*. The manager can only make the transition, if int-Start_MT is in subprocess S-1 and trapped in trap t-1.

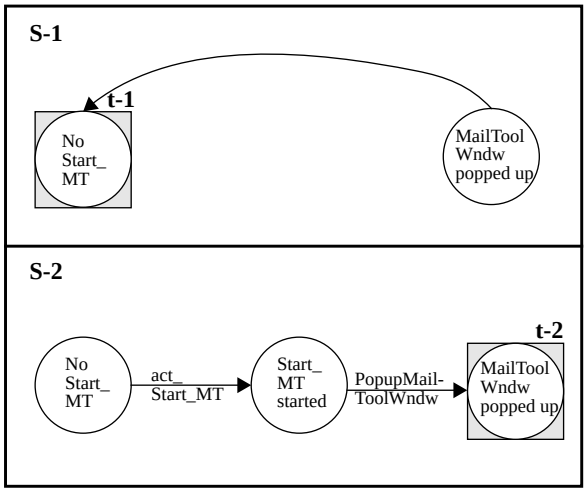


Fig. 3.5.6.1. int-Start_MT's subprocesses and traps with respect to MailTool/Main Window

In subprocess S-2 a mailtool main window is popped up. The MailTool/Main Window has to prescribe subprocess S-2 and trap t-2 for int-Start_MT, when the MailTool/Main Window makes the transition to its neutral state *Icon Closed*.

In Fig. 3.5.7. the internal behaviour of the operation *Stop_MT* (int-Stop_MT) and in Fig. 3.5.7.1. the subprocesses and traps being part of int-Stop_MT are depicted.

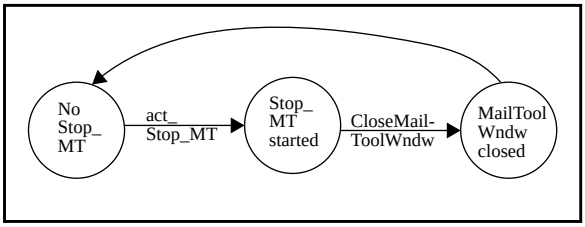


Fig. 3.5.7. int-Stop_MT: STD of its internal behaviour

This operation is called by the user, when the user wants to close the mailtool icon. The closing of the mailtool icon is achieved by clicking twice on the main window. When the operation *Stop_MT* is called, the manager MailTool/Main Window, is asked to transit from state *Icon Open* to state *Icon Closed*. The manager can only make the transition, if int-Stop_MT is in subprocess S-3 and in trap t-3. This operation takes care for the closing of the main window of the mailtool.

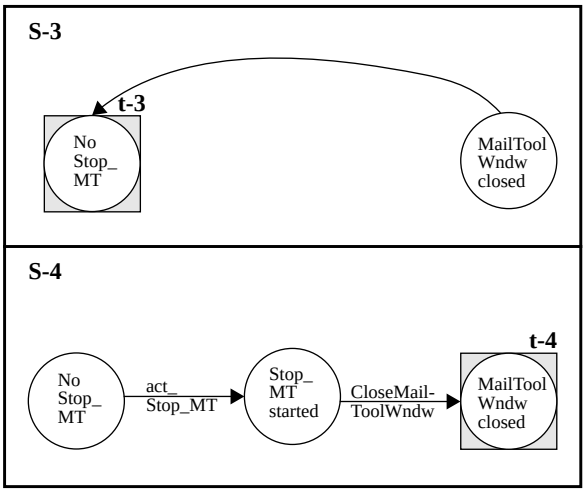


Fig. 3.5.7.1. int-Stop_MT's subprocesses and traps with respect to MailTool/Main Window

In Fig. 3.5.8. the internal behaviour of the operation *Receive* (int-Receive) is given. This operation is called by the mailtool, when the mailtool receives some mail. The two subprocesses (S-5 and S-6) and the two traps (t-5 and t-6) are given in Fig. 3.5.8.1. When the operation *Receive* has been called, the MailTool/Main Window is requested to transit from state *Icon Closed* or from state *Icon Open* to state *Starting Receive*. The manager can only make the transition, if int-Receive is in subprocess S-5 and trapped in trap t-5. As soon as int-Receive is in subprocess S-6, the mailtool can store the new mail in its buffer and make a sound to make the user aware of the new mail. The trap t-6 is as large as possible, because this allows the MailTool/Main Window to return as soon as possible to the state *Icon Closed* or to the state *Icon Open*.

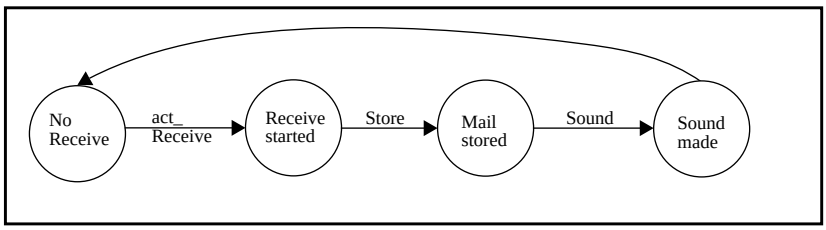


Fig. 3.5.8. int-Receive: STD of its internal behaviour

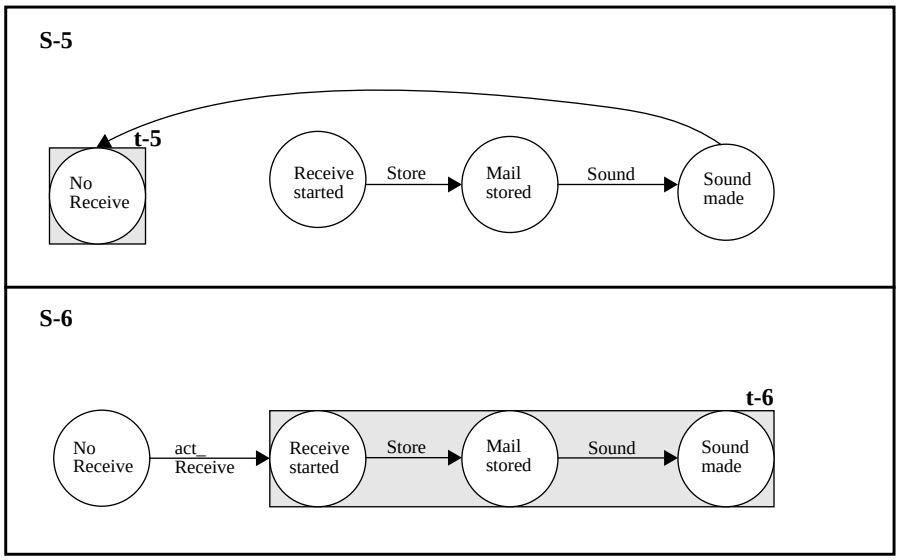


Fig. 3.5.8.1. int-Receive's subprocesses and traps with respect to MailTool/Main Window

In Fig. 3.5.9. the internal behaviour of the operation *Compose* (int-Compose) is presented. This operation is called by the user, when he wants to send a message. By pushing the compose button, a compose window will be popped up. The two subprocesses (S-7 and S-8) and the two traps (t-7 and t-8) can be found in Fig. 3.5.9.1.

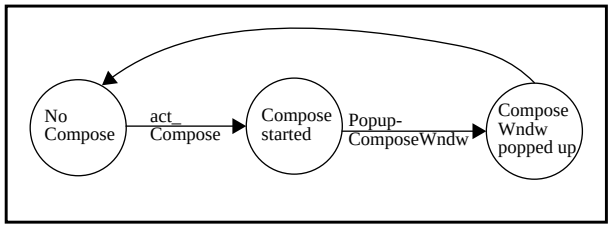


Fig. 3.5.9. int-Compose: STD of its internal behaviour

When the operation *Compose* has been called, the manager ext-Compose Window is requested to transit from state *Icon Open* to state *Starting Compose*. The manager can only make the transition, if int-Compose is in subprocess S-7 and in trap t-7. As soon as int-Compose is in subprocess S-8, the mailtool pops up a compose window in which the user can write his message.

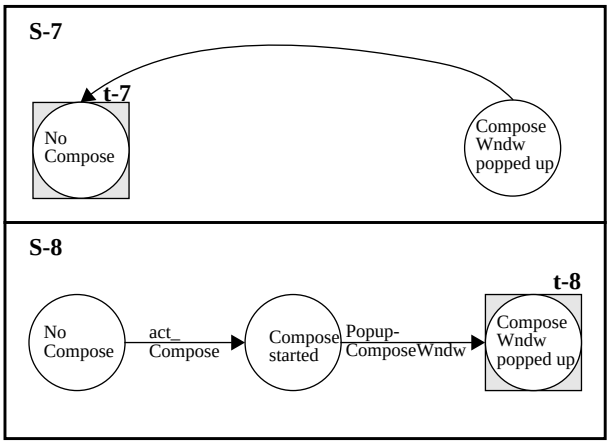


Fig. 3.5.9.1. int-Compose's subprocesses and traps with respect to ext-Compose Window

The internal behaviour of the operation View (int-View) is depicted in Fig. 3.5.10. The subproc-

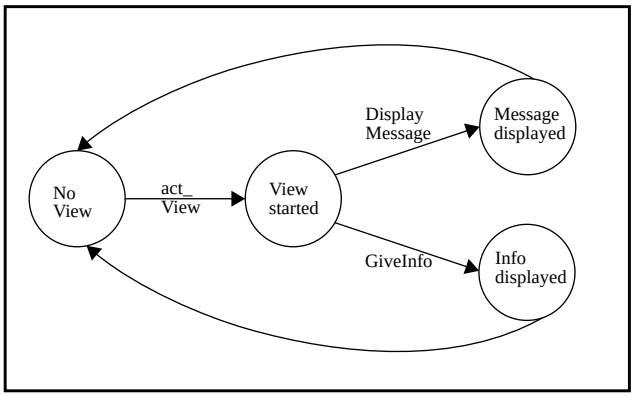


Fig. 3.5.10. int-View: STD of its internal behaviour

esses and traps belonging to int-View are presented in Fig. 3.5.10.1. The operation *View* is called by the user when he wants to read a mail message. If no problems have been detected by the mailtool, the message will be displayed in a view window by the internal operation *Display Message*. But if a message cannot be displayed or some other problems occur, some information will be given. This is done by the internal operation *GiveInfo*. As soon as the operation *View* has been called, the manager ext-View Window is requested to transit from state *Icon Open* to state *Starting View.* The manager can only make the transition, if int-View is in subprocess S-9 and trapped in trap t-9.

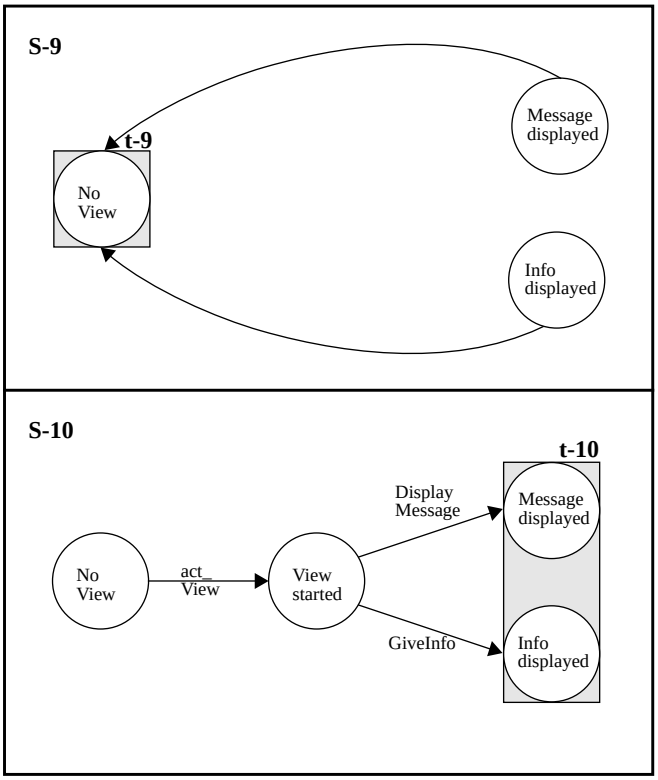


Fig. 3.5.10.1. int-View's subprocesses and traps with respect to ext-View Window

In subprocess S-10 the manager really pops up the view window with the corresponding message or displays a window with information about the problems. To let the manager make the transition from state *Starting View* to state *Icon Open* or to state *Icon Closed*, the manager ext-View Window must prescribe subprocess S-10 and trap t-10 for int-View.

In Fig. 3.5.11. the internal behaviour of the operation *Reply* (int-Reply) is given. The corre-

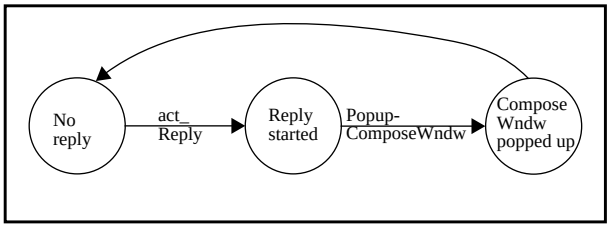


Fig. 3.5.11. int-Reply: STD of its internal behaviour

sponding subprocesses and traps of int-Reply are depicted in Fig. 3.5.11.1. This operation *Reply* is called by the user if he wants to reply to a certain message. The reply window will be popped up as soon as the user clicks on the reply button in the main window. Immediately after the calling of operation *Reply*, the manager ext-Reply Window is requested to transit from state *Icon Open* to state *Starting Reply*. The manager can only make this transition, if int-Reply is in subprocess S-11 and trapped in trap t-11.

In subprocess S-12 the manager really pops up a reply window. This reply window is a compose

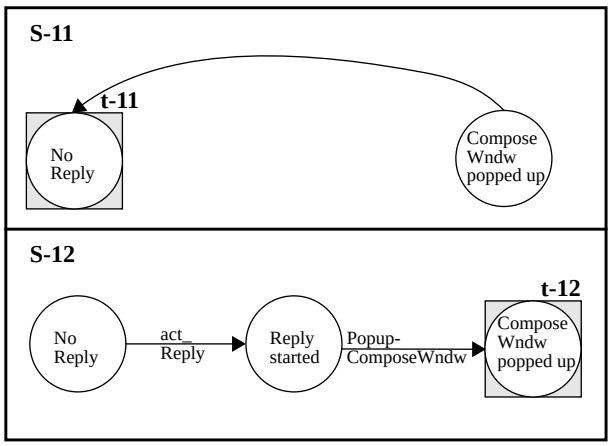


Fig. 3.5.11.1. int-Reply's subprocesses and traps with respect to ext-Reply Window

window, but with a small difference. In the reply window the address of the person to whom the user wants to send the mail is automatically added. To let the manager make the transition from state *Starting Reply* to state *Starting Deliver*, the manager ext-Reply Window must prescribe subprocess S-12 and trap t-12 for int-Reply.

The internal behaviour of the operation *Deliver* (int-Deliver) is presented in Fig. 3.5.12. The subprocesses and traps which belong to this operation are given in Fig. 3.5.12.1. The operation *Deliver* is called by the user. When the user is satisfied about what he had written in his mail message, he can send it by clicking on the deliver button in the compose window or reply window.

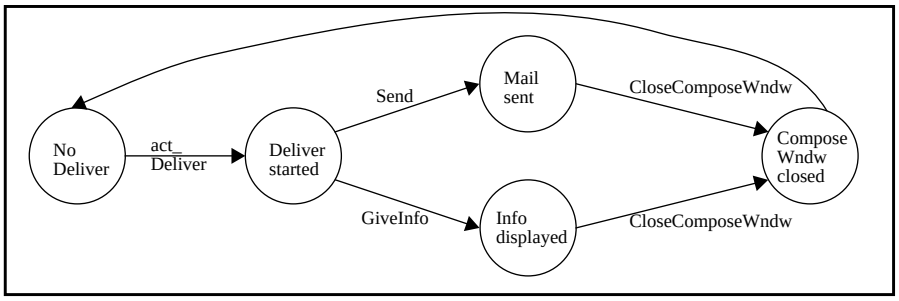


Fig. 3.5.12. int-Deliver: STD of its internal behaviour

Right after the calling of operation *Deliver*, the corresponding manager (ext-Compose Window or ext-Reply Window) is asked to transit from state *Starting Compose* or state *Starting Reply* to state *Starting Deliver.* The manager can only make this transition, if int-Deliver is in subprocess S-13 and trapped in trap t-13.

In subprocess S-14 the mailtool really sends the message (operation *Send*) or displays a window with information when an error occurs (operation *GiveInfo*). After this, the mailtool closes the windows and int-Reply ends up in its state *Compose Wndw closed*. To let the manager make the transition from state *Starting Deliver* to state *Icon Open* or to state *Icon Closed*. The manager must prescribe subprocess S-14 and trap t-14 for int-Deliver.

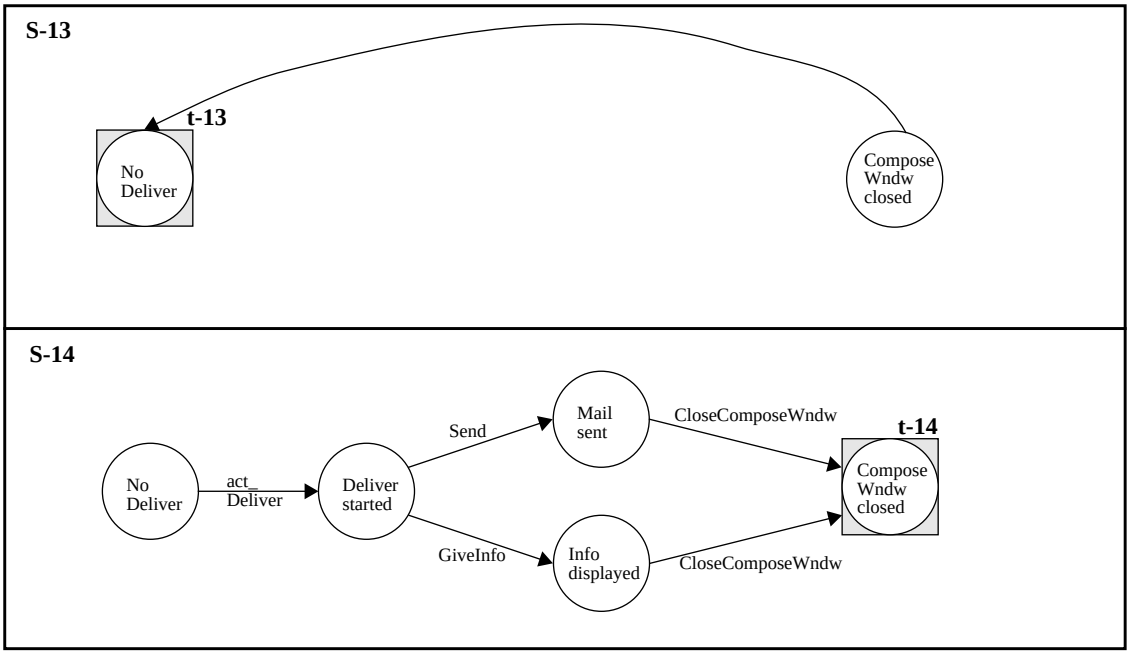


Fig. 3.5.12.1. int-Deliver's subprocesses and traps with respect to ext-Compose Window and ext-Reply Window

In Fig. 3.5.13. the internal behaviour of the operation *Ok* (int-Ok) is shown. The corresponding subprocesses and traps of int-Ok are depicted in Fig. 3.5.13.1.

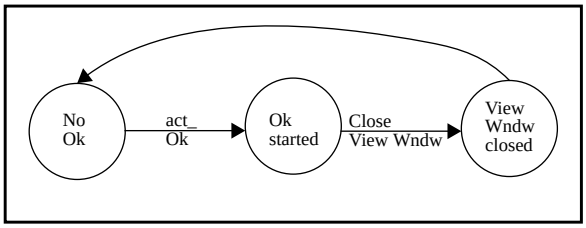


Fig. 3.5.13. int-Ok: STD of its internal behaviour

This operation *Ok* is called by the user if he wants to quit the view window. Of course, this ok button could have been modelled in the compose window and in the reply window. The ok button is not modelled in those windows, because they are closed by the operation *Deliver* (see Fig. 3.5.12.1.). The view window will be closed as soon as the user clicks on the ok button in the view window. Immediately after the calling of operation *Ok*, the manger ext-View Window is requested to transit from state *Starting View* to state *Icon Open* or to state *Icon Closed*. The manager can only make this transition, if int-Ok is in subprocess S-15 and trapped in trap t-15. The internal operation *Close View Wndw* of int-Ok is used to close the view window.

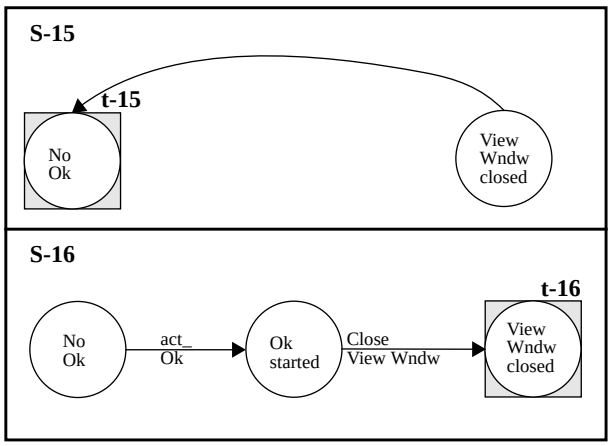


Fig. 3.5.13.1. int-Ok's subprocesses and traps with respect to ext-View Window

3.5.3 The Manager Process of MailTool

The external behaviour of MailTool will be used as the manager process (Fig. 3.5.14.). Note, that MailTool is considered to be the aggregation of the four manager processes (MailTool/ Main Window, ext-Compose Window, ext-View Window and ext-Reply Window). The employee processes being associated with this manager process are: ext-Compose Window, ext-View Window, ext-Reply Window, int-Start_MT, int-Stop_MT, int-Receive, int-Compose, int-View, int-Reply, int-Deliver and int-Ok. Of course the internal behaviours of the user are employees too, but the user isn't within the scope of this example. The manager process Mail-Tool will be described, by giving a explanation of each STD (Main-STD and Dep-STDs).

Main-STD: MailTool/Main Window

The manager's starting state is *Icon Closed*. When the user decides to use the mailtool, he has to click on the mailtool icon. This means, that the user calls the operation *Start_MT.* After the calling, MailTool/Main Window is requested to make the transition to state *Icon Open*. This is only possible when int-Start_MT is in subprocess S-1 and trapped in trap t-1, int-Stop_MT is in subprocess S-4 and in trap t-4, ext-Compose Window is in manager subprocess MS-1 and in manager trap mt-1, ext-View Window is in manager subprocess MS-3 and trapped in manager trap mt-3, and ext-Reply Window is in manager subprocess MS-5 and in manager trap mt-5. Because of the transition from state *Icon Closed* to *Icon Open* of the Main-STD (MailTool/Main Window), the Dep-STDs (ext-Compose Window, ext-View Window and ext-Reply Window) make automatically the transition from state *Icon Closed* to state *Icon Open* in their own STD (operation *dep_Start_MT*). As soon as the user decides to stop using the mailtool, he has to click twice at the main window, which causes the calling of operation *Stop_MT.* Now the manager MailTool/Main Window is invited to go back to state *Icon Closed*. Only after int-Start_MT is in subprocess S-2 and has reached trap t-2, int-Stop_MT is in subprocess S-3 and trapped in trap t-3, ext-Compose Window is in manager subprocess MS-2 and in manager trap mt-2, ext-View Window is in manager subprocess MS-4 and in manager trap mt-4, and ext-Reply Window is in manager subprocess MS-6 and trapped in manager trap mt-6, the manager can make this transition.

It must be clear, that the mailtool can receive mail messages independently of the state of the mailtool icon. This means, that it must be possible for the mailtool to receive mail when the mailtool icon is closed and the mailtool must be able to receive mail when the mailtool

icon is opened. The mailtool can receive mail, when MailTool/Main Window prescribes subprocess S-5 and trap t-5 for int-Receive. When the mailtool has stored the mail and has made a sound, it can go back to state *Icon Closed* or *Icon Open*. This depends on the state of the mailtool icon (icon opened or icon closed). To let manager MailTool/Main Window transit to state *Icon Closed* or *Icon Open*, int-Receive must be in subprocess S-6 and trapped in trap t-6.

Dep-STD: ext-Compose Window

At the time that the user wants to send a message he must pop up a compose window. To get a compose window, he has to click on the compose button in the main window. The click on this compose button, causes the calling of operation *Compose*. After calling the manager ext-Compose Window is invited to transit from state *Icon Open* to state *Starting Compose*. At the moment however, the manager is still outside *Starting Compose*. This means that int-Compose is in subprocess S-7 and sanguinely in trap t-7. Only when int-Compose has reached trap-7, ext-Compose Window will make the transition to state *Starting Compose*. Now the compose window can be popped up.

After writing the message in the compose window, the user has to send the message. This is done by pushing the deliver button in the compose window. The pushing of the deliver button, provokes the calling of operation *Deliver*. Right after the calling of operation *Deliver*, the manager is asked to make the transition from state *Starting Compose* to state *Starting Deliver*. The manager can only make this transition, if int-Compose is in subprocess S-8 and trapped in trap t-8, and int-Deliver is in subprocess S-13 and in trap t-13. As soon as int-Deliver has performed its operations (*Send* or *GiveInfo* and *CloseComposeWndw*), ext-Compose Window transits to state *Icon Open* or to state *Icon Closed*. This will be done when int-Deliver has been arrived at trap t-14.

Dep-STD: ext-View Window

To read a mail message, the user has to start a view window. To get such a view window, he has to click on the view button in the main window. The pushing of this view button, induces the calling of operation *View*. After the calling of this operation, the manager ext-View Window is invited to transit from state *Icon Closed* to state *Starting View*. However, ext-View Window is still outside *Starting View*. This indicates, that int-View is in subprocess S-9 and hopefully within trap t-9. Only when int-View has reached trap-9 and int-Ok is in subprocess S-16 and in trap t-16, the manager will make the transition to state *Starting View*. Now the view window will be popped up by the internal operation *Display Message* of int-View. If an error is detected by the mailtool, then a window is displayed with the corresponding information (operation *GiveInfo* of int-View).

After reading the message, the view window has to be closed. This is performed by pushing the ok button in the view window. After the calling of operation *Ok*, the manager is provoked to transit from state *Starting View* to state *Icon Open* or to state *Icon Closed*. The manager ext-View Window can only make this transition, if int-View is in subprocess S-10 and trapped in trap t-10, and int-Ok is in subprocess S-15 and in trap t-15.

Dep-STD: ext-Reply Window

When the user decides to reply to a certain mail message, he must pop up a reply window. This window is nearly the same as a compose window. The only difference is, that the address has been added in the reply window already e.g. Belkhatir@imag.fr (see Fig. 3.5.1.). To get a reply window, the user has to click on the reply button in the main window. The pushing of this reply button, causes the calling of operation *Reply*. After the calling, the

manager ext-Reply Window is asked to transit from state *Icon Open* to state *Starting Reply.* Notwithstanding, ext-Reply Window is still outside *Starting Reply.* This indicates, that int-Reply is in subprocess S-11 and expectantly in trap t-11. Only when int-Reply has entered trap-11, ext-Reply Window will transit to state *Starting Reply.* Now the reply window will be popped up by the internal operation *PopupComposeWndw* of int-Reply.

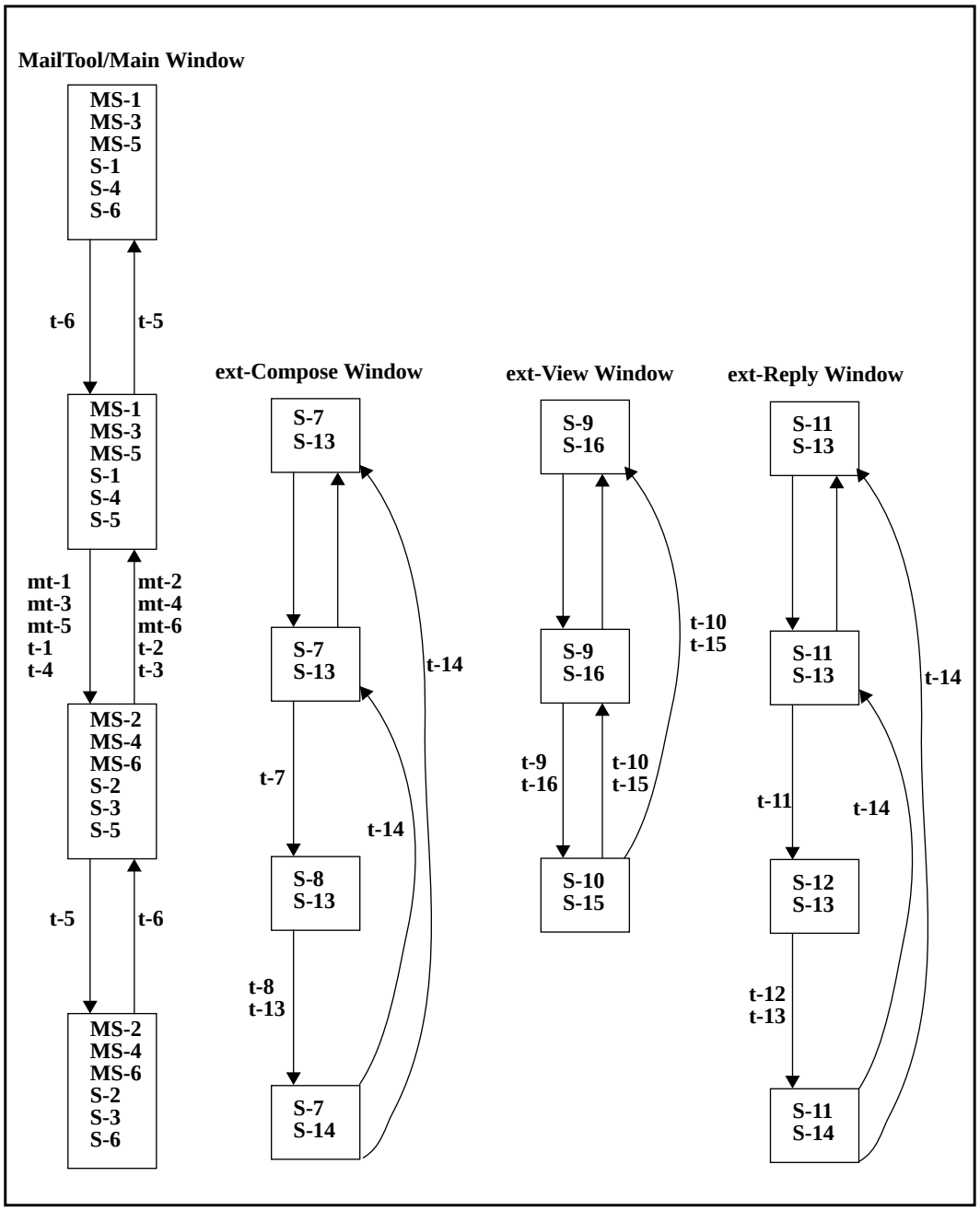


Fig. 3.5.14. MailTool/Main Window, manager of six employees
ext-Compose Window, manager of two employees
ext-View Window, manager of two employees
ext-Reply Window, manager of two employees

After writing the message in the reply window, the user has to send the message. This is achieved by pushing the deliver button in the reply window. The pushing of the deliver button, evokes the calling of operation *Deliver.* Immediately after the calling of operation *Deliver*, the manager is asked to transit from state *Starting Reply* to state *Starting Deliver.*

The manager can only make this transition, if int-Reply is in subprocess S-12 and trapped in trap t-12, and int-Deliver is in subprocess S-13 and in trap t-13. As soon as int-Deliver has performed its operations (*Send* or *GiveInfo* and *CloseComposeWndw*), the manager ext-Reply Window transits to state *Icon Open* or to state *Icon Closed*. This transition will be performed as soon as int-Deliver has been arrived at trap t-14.

3.5.4 An Alternative for MailTool: MailTool2

In this paragraph an alternative for MailTool is presented. This means, that the Main-STD (MailTool/Main Window) is used for the alternative way of modelling (see Fig. 3.5.16). The idea is to combine the internal behaviours of two different export operations into one STD. This reduces the number of employee processes in the manager process. Here the internal behaviours of the two export operations *Start_MT* and *Stop_MT* are combined. The new subprocess S-01 substitutes the two old subprocesses S-1 and S-4. Subprocess S-02 replaces the two old subprocesses S-2 and S-3. In Fig. 3.5.15. the combination of these two internal behaviours is depicted.

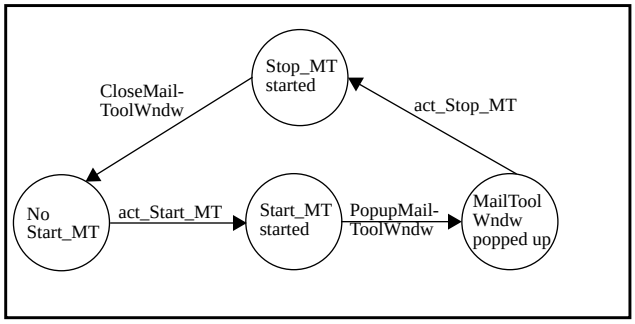


Fig. 3.5.15. int-Start_Stop_MT: STD of its internal behaviour

The corresponding subprocesses (S-01 and S-02) and traps (t-01 and t-02) are expressed in Fig. 3.5.15.1. The calling of the export operations remain the same as in the original way of modelling in SOCCA. But instead of the four subprocesses (S-1, S-2, S-3 and S-4) and the four traps (t-1, t-2, t-3 and t-4), only two subprocesses (S-01 and S-02) and two traps (t-01 and t-02) are used now.

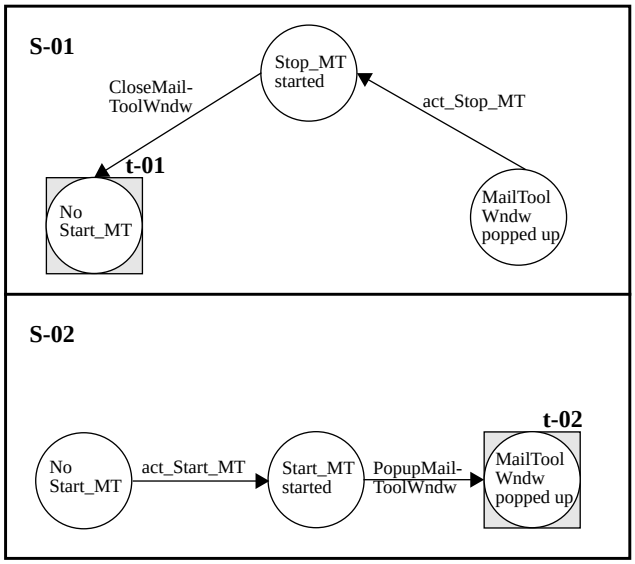


Fig. 3.5.15.1. int-Start_Stop_MT's subprocesses and traps with respect to MailTool/Main Window2

The starting state of MailTool/Main Window2 is the same as for the previous manager, namely state *Icon Closed*. Only the manager prescribes subprocess S-01 instead of the subprocesses S-1 and S-4. When the operation *Start_MT* has been called, the manager MailTool/Main Window2 is invited to make the transition from state *Icon Closed* to *Icon Open*. This transition can only be made if int-Start_Stop_MT is in subprocess S-01 and in trap t-01. The closing of the mailtool will be performed by clicking twice on the main window. This causes the calling of operation *Stop_MT*. Now MailTool2 is asked to transit from state *Icon Open* to state *Icon Closed*. The manager can only make this transition, is int-Start_Stop_MT in subprocess S-02 and in trap t-02.

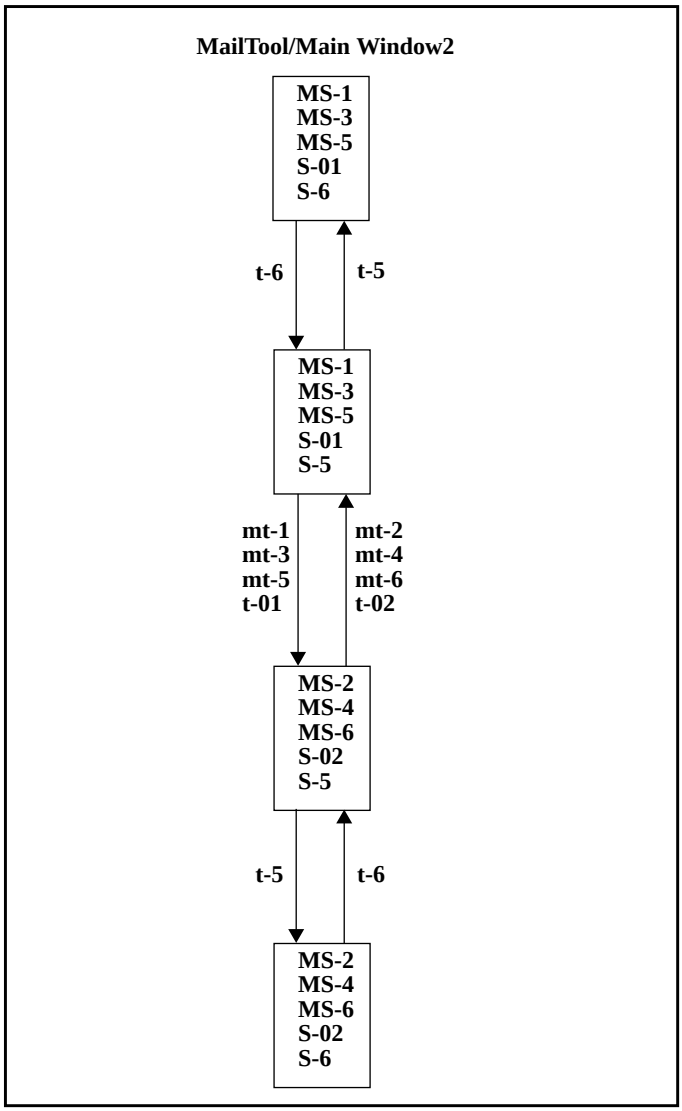


Fig. 3.5.16. Part of alternative manager MailTool/Main Window2, manager of five employees

In Fig. 3.5.16. the alternative manager MailTool/Main Window2 is shown. This external behaviour should have been complete when the other parts of the external behaviour (ext-Compose Window, ext-View Window and ext-Reply Window) were presented too. But only the changed part of the external behaviour has been depicted. The employee processes being associated with this part of the alternative manager are: ext-Compose Window, ext-View Window, ext-Reply Window, int-Start_Stop_MT and int-Receive. The combination of two different internal behaviours into one internal behaviour, reduces the number of employee processes of the manager

process. So it is also possible to combine some internal behaviours of Design Engineer with each other. I don’t present the STDs, but I will only give the internal behaviours which could be combined with each other. The following internal behaviours can be combined with each other: the internal behaviours of int-Cont_design and int-Stop_design (Fig. 3.5.8. and Fig. 3.5.9.), the internal behaviours of int-Change_role_d and int-Change_role_r (Fig. 3.5.7. and Fig. 3.5.11.), and the internal behaviours of int-Cont_review and int-Stop_review (Fig. 3.5.12. and Fig. 3.5.13.).

Chapter 4 Various Role Concepts

4.1 Introduction into Role Concepts

In the following subparagraphs several role concepts used in the different process modelling formalisms and found in several articles will be presented. It's an interesting topic in the process modelling area, because the role concept is used to describe the behaviour or parts of the behaviour of the human as well as the non-human participants in a (software) process. Because of the fact that agents, roles and activities are closely related to each other, the definitions of the agents and activities are taking into account. After these descriptions a proposition will be presented in paragraph 4.6. how to integrate the role concept into the SOCCA formalism, and in paragraph 4.6.1. the role concept will be applied on some examples modelled in SOCCA.

4.2 MERLIN

In [14] MERLIN is described. It is a prototype Process-centred Software Development Environment (PSDE). The concepts used in MERLIN are:

- **Resources**: The resources in the software process are people who participate in the production of software and the technical resources such as tools supporting the software development activities (e.g. compiler and editors).
- **Activities**: An activity is a collection of tasks which achieve a goal related to the production of a software product (e.g. specify, edit, compile or test a module).
- **Role**: A role is a group of activities which are logically highly related (e.g. project management, design, etc.).

The users in the software process are associated with one or more roles. The users are supported primarily by the display of the relevant information in a working context associated with their current role. In MERLIN the working context is displayed on the screen. The working context shows documents which have to be manipulated, their dependencies with other documents and the activities which have to be performed on each document. In this way you have a clearly overview of how the concepts of agent, role and activities are used in the MERLIN working context.

4.3 OIKOS

In [17] the software process modelling formalism OIKOS is described. The purpose of the OIKOS project is to leisure the construction of process-centred software development environments. There are three subgoals in OIKOS and one of them is interesting in the context of the role concept:

(2). 'to define a systematic way to model the process at different levels of abstraction such that each actor playing a role in the process can be naturally provided with a personalized view of the process which is more and more detailed as it is nearer and nearer to the focus of the actor's role.'

The concepts used in OIKOS are:

- The terms like **Agent, Task** and **Activity** are used in the everyday meaning and not in the precise meaning of the software process modelling classification.

- **Role**: A role is used to model the actions of the actors performing in the software process. Each actor is allowed to play several roles and each role defines a coherent set of contributions to the process and a coherent view of the state of the process that permits the actor to provide the expected contributions. During the enactment of the process each role corresponds to a window on the actor's display. The managing and technical roles are considered to be the same. In both roles, the contributions are in terms of tool application or inputs in the process. Only the kind of information provided or the kind of tools used in the process, vary between the managing and technical roles.

4.4 ALF

In [6] ALF is described. ALF is a research project which wants to provide computerized facilities to support software development activities.

The concepts relevant for roles found in the ALF formalism are:

- **Agent**: An agent or actor is a human participant in the process. The agent playing a specific role can execute its activities. Actors work in work contexts, in these work contexts an actor works with objects, tools, has to follow certain policies, etc.

- **Role**: A role is a set of **activities** enactable by an agent. Agents are playing the same role if they are working in the same working context, if they can access the same objects, if they can invoke the same operators and if they are subject to the same prescriptions. Role cooperation occurs when agents are working in the same team. A role can prescribe constraints to another one, e.g. in the role *designer* a rule can be defined to trigger an action (modify-design) if the attribute approved=false of a document is set by the role *reviewer.*

4.5 ADELE-TEMPO

In [1, 2, 3, 4, 5, 16] ADELE-TEMPO is depicted.

The concepts in the context of roles in ADELE-TEMPO are:

- **User**: A user is a human person who can work in a work environment (WE), he can use tools to perform his activities.

- **Methods**: The methods are the activities in the process. When a certain event is true the corresponding method will be executed.

- **Role**: The role concept is used to define the common behaviour and characteristics of the objects in the software process. The terms characteristics and behaviour mean, valid attributes respectively methods and constraints. The role type allows to change the definition of attributes, methods and constraints of an object type, depending on where the object 'plays' its role.

4.6 Several Articles

In [8] a software production process is characterized as a partially ordered network of activities which are interacting with each other, in order to produce a software product. The activities are performed by human agents supported by tools. The basic definitions in the context of roles are:

- **Agent**: A human or a specific person, performing the activities related to a role. An agent is characterized by the role properties and his/her availability. In the article they also mention some key roles of agents in the software process like, process owner, process technology provider, process designer, process manager and process agent.
- **Activity**: An activity incorporates procedures, rules and policies and its purpose is to generate and modify a given set of artifacts. The activities may be organized into networks with both horizontal (chaining) and vertical (decomposition) dimensions. Activities are associated with roles, artifacts and tools.
- **Role**: A role describes a set of responsibilities, rights and skills necessary to accomplish a specific activity in the software process.
- **Tool**: A tool can be a computer program supporting or automating a part of the work related to an activity.
- **Artifact**: The artifact are the (sub)products of a process. An artifact produced by a process may be used as input by the same or a different process to produce other artifacts. The artifacts in a process are often persistent and versioned.

In [15] the importance of a clear conceptual and terminological framework for software process engineering is discussed. The concepts and definitions related to roles are:

- **Agent**: *Definition*: 'A performer of a process. It may be a human or a computerized tool'. *Comments*: A set of agents can perform elementary activities. The automatic activities only rely on computerized agents, interactive activities need both types of agents, interacting to reach the appointed goal. The manual activities only involve human agents. Computerized agents are tools being associated with the domain support technology.
- **Activity**: *Definition*: 'An elementary process step. At the level of abstraction of the process description, an activity has no visible substructure'.
- **Task**: *Definition*: 'A managed process step. Managed means that resources are allocated to it, it is scheduled, it is assigned to one or several agents, and it is monitored'.

  Comments: Tasks are well known entities in the area of project management. In the field of software process engineering some analysis-oriented approaches and guidance-oriented approaches particularly focus on this level of abstraction (i.e. their elementary activities are the managed tasks). In some other approaches, they are split into smaller activities which are not managed. Clearly, there's no essential distinction between a task and an activity. It's just a human decision about whether a process step should be managed or not. Moreover, when the distinction between tasks and activities has been made, it gives rise to the question whether an activity is consequently a subprocess of some task or not.

- **Role**: *Definition*: 'A set of permissions and obligations associated to a functional objective'. *Comments*: E.g. the permissions to achieve the set of activities of the process aimed to reach the objective and obligations to 'satisfy' the corresponding constraints. The objectives and the related roles may be refined from general roles (e.g. software developer) into specific ones (e.g. designer). At a certain level of refinement, it is possible to recognize three aspects:

  1. the *generic role* (e.g. team leader),

  2. the *specific role* (e.g. the team leader of the team, developing the component x within the project y) and

  3. the *role occupants* (e.g. Jacques, team leader of team, developing the product x of project y).

  A human or a computerized agent occupies (plays) a role when the corresponding role is assigned to him. A dynamic mapping between role occupants and processes is possible by using roles. The role occupants are allowed to play several roles at different times and a given role may be played by different occupants at various times.

4.7 SOCCA

After having presented several concepts of different process modelling formalisms, the concepts related to roles in SOCCA will be discussed in this paragraph. The SOCCA formalism is described in [9, 10]. The main focus will be the *role concept* in this paragraph. This means how to integrate the role concept into the SOCCA formalism in a flexible, understandable and useful way. Although the terms agent, activity and role do exist in the SOCCA formalism already, they will be mentioned explicitly and a proposition will be given how to interpret them. The concepts relevant for roles and the role concept in SOCCA will be given. The concepts in the context of roles in SOCCA are:

- **Agent**: An agent can be a human or a non-human participant of the software process. The human participants are e.g. project manager, design engineer, etc. The non-human participants of the software process are e.g. tools (mailtool, compiler, etc.), documents, etc.

- **Activity**: The activities are the export operations and the internal operations. They all describe a certain part (activity) of the software process. As in [8], the activities may be organized into horizontal (chaining) and vertical (decomposition) dimensions. The *horizontal dimension* means, that the activities can be connected with each other. This corresponds to the states and the transitions in an STD. The *vertical dimension* indicates, that the export operations all have an internal behaviour which will be described by internal operations (activities).

- **Role**: A role is a collection of activities which are highly related with each other. Of course a role must have a name to make a distinction between the several roles. A role can be played by several agents e.g. a design engineer and a review engineer can both have the role *Reviewing.* An agent can have several roles e.g. a design engineer can play the role *Designing* and the role *Reviewing*. In SOCCA the external behaviour will be a perfect candidate for defining roles. Because the external behaviour is the behaviour visible from outside and each part of this visible behaviour comes close to the intuitive feeling of a 'role'. A role can be the complete external behaviour or a part of the external behaviour. It must be clear, that a role comprises some states and transitions of the external behaviour and therefore the role is a restriction of the external behaviour. The role

description can be found in the internal behaviours belonging to the export operations. Now some restrictions with respect to the role concept in SOCCA will be presented. These restrictions are:

1. Each external behaviour has at least one role. The number of roles of the external behaviour varies between 1 and n, depending on the preferences of the modeller.

2. The role concept provides the facility to prevent the explosion of states and transitions of the external behaviour. This is achieved by restricting the number of states of the external behaviour which can be part of a role. Although the number of states for a role is intuitively determined, I will propose to restrict the number of states of a role between 2 and 10. There must be at least two states in a role, because the external behaviour has a neutral state and another state to which it has to switch to start a role.

Probably after more experiences of modelling in SOCCA, the number of states of a role can be tuned to attain a more acceptable restriction for the role concept.

Summarizing, a role in SOCCA consist of the following ingredients: **a**. rolename, **b**. list of states, **c**. list of transitions and **d**. number of roles.

4.7.1 Using the Role Concept in SOCCA

In this subparagraph the role concept will be applied to three examples which have been modelled in the SOCCA formalism.

In the first example the role concept will be applied to a design document: Design. The document Design has been modelled in [10]. In Table 1. the roles which belong to the design document Design are presented. The roles Under design and Under review are induced by the roles Designing and Reviewing of Design Engineer2 (see Table 2.).

In the second example the role concept will be applied to an agent (human): Design Engineer. The external behaviour of Design Engineer2 will be used and can be found in Fig. 3.4.5. Table 2. presents the roles belonging to Design Engineer2.

In the third example the role concept will be applied to a tool: MailTool. The external behaviour of MailTool will be utilized and is presented in Fig. 3.5.5. In this external behaviour it is easy to distinguish the different role, because for each separate STD a role can be defined. In Table 3. the roles corresponding to MailTool are depicted.

Number of roles	Rolename	States	Transitions
3	**Under design**	non existing	prepare
		creatable	create_first
		starting creation	create_next
		created	open_for_modify
		modifiable	modify
		starting modify	close_modify
	Under review	pre review	open_for_review
		reviewable	review
		starting review	close_and_review_not_ok
			close_and_review_ok
	Under copy	readable	copy
		starting copying	

Table 1. Roles of Design

Number of roles	Rolename	States	Transitions
2	**Designing**	Role Designer	Begin_design
		Starting Design	Cont_design
		Real Design	Stop_design
			Change_role_r
	Reviewing	Role Reviewer	Begin_review
		Starting Review	Cont_review
		Real Review	Stop_review
			Change_role_d

Table 2. Roles of Design Engineer2

Number of roles	Rolename	States	Transitions
4	**Main**	Icon Closed	Start_MT
		Icon Open	Stop_MT
		Starting Receive	Receive
	Composing	Icon Closed	dep_Start_MT
		Icon Open	dep_Stop_MT
		Starting Compose	Compose
		Starting Deliver	Deliver
	Viewing	Icon Closed	dep_Start_MT
		Icon Open	dep_Stop_MT
		Starting View	View
			Ok
	Replying	Icon Closed	dep_Start_MT
		Icon Open	dep_Stop_MT
		Starting Reply	Reply
		Starting Deliver	Deliver

Table 3. Roles of MailTool

Chapter 5 TEMPO versus SOCCA

After modelling some examples in TEMPO and SOCCA, it will be interesting to compare the two software process modelling approaches. In the following four subparagraphs the similarities, differences, advantages and drawbacks of the TEMPO formalism and SOCCA formalism will be discussed. This is very useful, because in this way you get can get a quick overview of both approaches. The results of the comparison have been presented in Table 1 through to Table 4 and eventually some comments will be given to explain the tables.

5.1 The Similarities between TEMPO and SOCCA

In Table 1 the similarities between TEMPO and SOCCA are given.

nr.	TEMPO	SOCCA
1	EER	EER
2	Object	Class
3	Attribute	Attribute
4	Event	Export operation
5	Method	Internal behaviour
6	Connection	Relationship
7	Role	Role

Table 1: The similarities between TEMPO and SOCCA

The terms which have been mentioned in Table 1. will be explained in more detail now.

1. **TEMPO**: TEMPO has been build on top of the ADELE database and this database is based upon an entity relationship database, complemented by object-oriented concepts (e.g. multiple inheritance, object versioning).

SOCCA: In SOCCA the EER model is appropriate for the data perspective of the process model. The data perspective is described by classes (attributes and operations) and relationships (part-of, is-a, general and uses).

2. **TEMPO**: The users, tools, documents, etc. are described by objects in TEMPO. The objects contain attributes and methods.

SOCCA: In SOCCA the classes which contain the attributes and operations represent the users, tools, documents, etc. Concrete instances of a class are being called objects according to object oriented terminology.

3. For both approaches the attribute is a name with a domain (integer, real, string, etc.).

4. **TEMPO**: The events in TEMPO are the commands executed by tools, mostly users or tools.

SOCCA: The export operations of the external behaviour in SOCCA are the commands or actions performed by other objects, particularly users or tools. This is represented by transitions from one state to another.

The examples which have been modelled express the resemblance between the events and the export operations in a clearly way, because they are equal in number and have the same names.

5. **TEMPO**: As soon as an event arise the corresponding method(s) are executed in TEMPO.

SOCCA: In SOCCA, as it has been extended in this thesis, most export operations have an internal behaviour. In the original SOCCA version, all export operation have an internal behaviour. The export operations perform some task and to carry out this task, separate behaviour has been modelled, mostly as internal behaviour.

6. **TEMPO**: The connection used in TEMPO is a special kind of relationship, assumed to be instantiated between pairs of roles. A connection is meant to define how each pair of connected objects in the different WEs is coordinated. The connection is a dynamic relationship, because it will be instantiated or deleted when necessary between the pairs of roles.

SOCCA: In SOCCA the different classes are connected by general relationships and uses relationships. The instances of the general relationship have a dynamic character, see Section 7 in [10]. The uses relationship is a static relationship and this relationship indicates where the various export operations are imported.

7. **TEMPO**: In TEMPO the role concept is used to describe the behaviour of the object. The role makes it possible to change the attributes, methods and constraints of an object depending on where the object plays its role (contextual behaviour).

SOCCA: In SOCCA, as it has been extended in this thesis, the role concept is used to aggregate the states and transitions of the external behaviour which are relevant for a specific role. So the external behaviour can be dived into several roles.

5.2 The Differences between TEMPO and SOCCA

The following table (Table 2.) presents the differences between the TEMPO and SOCCA formalisms. Below the table the differences will be described.

nr.	TEMPO	SOCCA
1	Progr. language	Graph. language
2	Rule/Event based	
3	Triggers	
4		STD+PARADIGM
5	Work Environments	
6	Bottom up	Top down

Table 2: The differences between TEMPO and SOCCA

1. **TEMPO**: TEMPO is a high level programming language based on the role and connection concepts.

SOCCA: SOCCA is a graphical language based on class diagrams, STDs and PARADIGM.

2. **TEMPO**: The TEMPO language is rule/event based, because the TECA formalism is used to describe the rules by: temporal events, their conditions and corresponding actions.

3. **TEMPO**: The triggers are executed each time a corresponding event is true.

4. **SOCCA**: The STDs describe the behaviour and PARADIGM expresses the coordination of the agents and documents in the software process in a graphical way. On a more abstract level one can argue however, that SOCCA's communication corresponds to events and triggers.

5. **TEMPO**: In TEMPO each software (sub) process is associated by a work environment. This work environment is defined by the following tuple: WE = (WS, PM, Tools, User). At this moment a similar grouping of classes/instances is not present in SOCCA.

6. **TEMPO**: The ADELE database has already been used for several years. TEMPO's prototype is/has been built on top of the ADELE database. This indicates, that the TEMPO formalism has been based on the ADELE database and concepts. This typically is a bottom up approach for developing abstract concepts.

SOCCA: On the contrary SOCCA is independent of a specific database. The SOCCA approach is to start first with abstract concepts, this typically is a top down approach for developing abstract concepts.

5.3 The Advantages and Drawbacks of TEMPO

First the advantages and thereafter the drawbacks of TEMPO will be discussed.

nr.	Advantages	nr.	Drawbacks
1	Progr. language	1	Progr. language
2	Role concept	2	Examples
3	Prototype	3	Syntax
4	WE	4	WE (WS)
5	Shared objects	5	User modelling

Table 3: The advantages and drawbacks of TEMPO

The advantages of TEMPO are:

1. For people who are used to work with programming languages, the TEMPO language can and will be grasped rather easily, at least in principle.

2. The role concept provides the facility to describe the contextual behaviour of the objects. This means, the role concept prescribes for the same object (without losing identity) its behaviour depending on where the object 'plays' its role.

3. Although the prototype has not been finished yet, it will be easier to describe and understand software processes by using a prototype.

4. A WE is a grouping of user(s), tool(s) and work space. Using a WE simplifies a software process, because the software process is split into smaller and coherent parts of the process. This improves the understanding and simplifying of the process.

5. TEMPO supports the sharing of objects. An object shared by multiple work environments can be modified within each work environment. In this way for each shared object a revision list or branch exists. The problems which will arise when exchanging or promoting the objects can be solved by TEMPO.

The following drawbacks of TEMPO have been found:

1. The TEMPO language has been based upon the ADELE language. The ADELE language is not really developed for the ease of use of the users, because a lot of symbols are used and therefore the language is hard to grasp. So TEMPO is suffering of these drawbacks, and therefore not really userfriendly.

2. The few examples which have been modelled in TEMPO are small and incomplete. This causes also the difficulties in understanding the TEMPO approach. Probably if the ISWP-6 and ISWP-7 or more examples have been modelled, a better and faster understanding of TEMPO will be achieved.

3. A huge drawback of TEMPO is the changing of the used syntax. In nearly every article a different syntax has been proposed and used. This does not really contribute to the understanding and clarity of the TEMPO formalism.

4. In each work space (WS) an agent or document performs its tasks. This has been modelled by using the role concept. The disadvantage is that several WSs and therefore WEs must be created when an agent has different roles. This indicates, that the roles which belong to the same person or tool are spread over the different WEs, e.g. a design engineer can have different roles like designing and reviewing. So two work environments have to be created. Another possibility which isn't supported by TEMPO is that only one work environment must be created in which several role s of a agent can be defined.

5. The examples which have been modelled in TEMPO did not focus on user modelling, but more attention has been paid to the modelling of the documents in the software process. This makes TEMPO, through ADELE, perhaps too DB oriented.

5.4 The Advantages and Drawbacks of SOCCA

In this subparagraph the advantages and drawbacks of the SOCCA formalism will be discussed.

nr.	Advantages	nr.	Drawbacks
1	Modular	1	Large
2	Graphical	2	Complex
3	Role concept	3	No prototype
4	User modelling		
5	Examples		

Table 4: The advantages and drawbacks of SOCCA

The advantages of SOCCA are:

1. The modular modelling of SOCCA is expressed by the different perspectives: data perspective, behaviour perspective and process perspective. By using these perspectives a software process can be described in a modular way.

2. The STDs in SOCCA are used to describe the behaviour (external behaviour and internal behaviour) in a graphical manner. This is a userfriendly approach of representing the software process.

3. The role concept which has been applied to SOCCA in this thesis, is useful for several reasons. This role concept allows to group relevant parts of the external behaviour into roles. This grouping expresses a more global view of the external behaviour, such that it will be more easy to understand. Another advantage is, that this role concept is unique in that sense that it can restrict the extent of the role, by restricting the number of states in the role. This implicitly indicates, that the description of the software process (level of detail) can be restricted by using this role concept.

4. Much attention has been paid to the modelling of users. In this thesis a user (design engineer) has been described in an operating system way by using the type-2 communication of SOCCA.

5. Rather a lot of examples have been modelled in SOCCA like the following examples: ISWP-6, ISWP-7, MERLIN Process Transactions, Petri Nets, some problems in companies (e.g. Philips) and the examples in this thesis. This is useful for evaluating the SOCCA formalism, because of the differences and varieties of the examples. By this evaluation the advantages, weaknesses and possibly some extensions of SOCCA can be found. Another advantage is, that when someone wants to understand SOCCA, he has some good and complete examples to focus on. This increases the comprehension of SOCCA.

The drawbacks of SOCCA are:

1. The specifications in SOCCA are very large, because of the PARADIGM part. A number of propositions have been raised e.g. using state charts, combing different internal behaviours, not every export operation has an internal behaviour. These topics can be subject for further research.

2. The specifications are complex, because of the great number of STDs which are used. The manager process can also be complex as soon as it contains a lot of employees and transitions.

3. The lack of a prototype of SOCCA is a drawback. There is no environment, no tools, this makes it more difficult to model in SOCCA.

Chapter 6 The RAPP Diagram

The R.A.P.P. (= Roles of Agents and their Position in a Process) diagram describes in a clearly way how the different agents, positions, roles and processes have to be interpreted.

6.1 Role, Agent, Position and Process

The terms role, agent, position and process are the key-words upon which the R.A.P.P. diagram has been based. In this section the different terms will be explained.

Process: A process can consists of N agents which have position(s) and role(s) in a specific process. A process comprises the activities performed by the agents (human as well as non-human agents). The process can be decomposed into several subprocesses, so a hierarchy of processes and subprocesses can be created.

Agent: The term agent refers to human as well as to non-human agents. The human agents are the persons involved in the process and the non-human agents are for example the tools used by the human agents in the process. Each agent can have N positions, e.g. an agent can have the position of team leader and the position of design engineer.

Position: A position is an intermediate level of abstraction between an agent and a role. A position comprises a set of N roles for an agent, e.g. the position design engineer has the following roles: designing and reviewing. A position can belong to N agents, e.g. the position design engineer can belong to two different persons (agents).

Role: Each agent has one or more roles in a process. A role can be owned by N positions, e.g. the role reviewing can belong to the positions design engineer and review engineer. Each role consists of N activities, e.g. the role designing comprises the activities: modify design, review design, etc.

The cardinalities and the dependencies between the different terms are represented in Fig. 6.1. and Fig. 6.2.

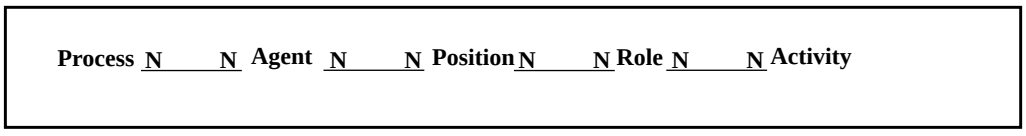


Fig. 6.1. Cardinalities

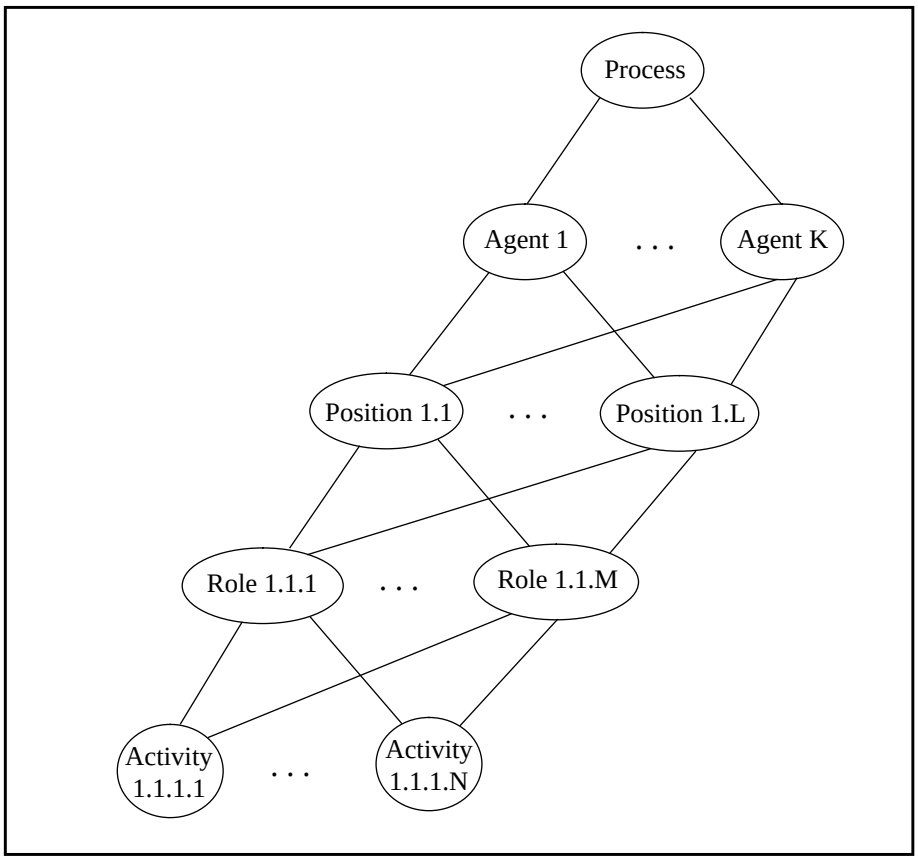


Fig. 6.2. Dependencies

6.2 The R.A.P.P. Diagram

The R.A.P.P. diagram describes the general overview of a software process. It is an informal and intuitive description of how to express a software process in a more global way. This means, that only processes, their dependencies/connections, agents, positions and roles are described. Note that the two figures, Fig. 6.3. and Fig. 6.4. should be read together. Fig. 6.4. is a more detailed description of each process and it also contains the description of the Tool Depot.

In Fig. 6.3. the different processes are connected with each other. There are three kinds of connections.

1. The first connection is the *Parent-Child* connection which is represented by a black arrow. This connection expresses, that the parent takes care of the child. This means, that the parent process monitors its child processes. The work performed in the parent process and child process can be exchanged with each other. E.g. the monitor process (parent) manages the processes Design, Review and Implement (children).

2. The second connection is called the *Brother-Sister* connection and is represented by a dashed arrow. This connection reflects the cooperation between processes of the same parent, e.g. the work done in a design process has to be reviewed in the review process.

3. The third connection is called the *Twins* connection and is represented by a bi-directional dashed arrow. This connections connects two processes of the same type, e.g. two design processes are cooperating with each other in order to produce a software product.

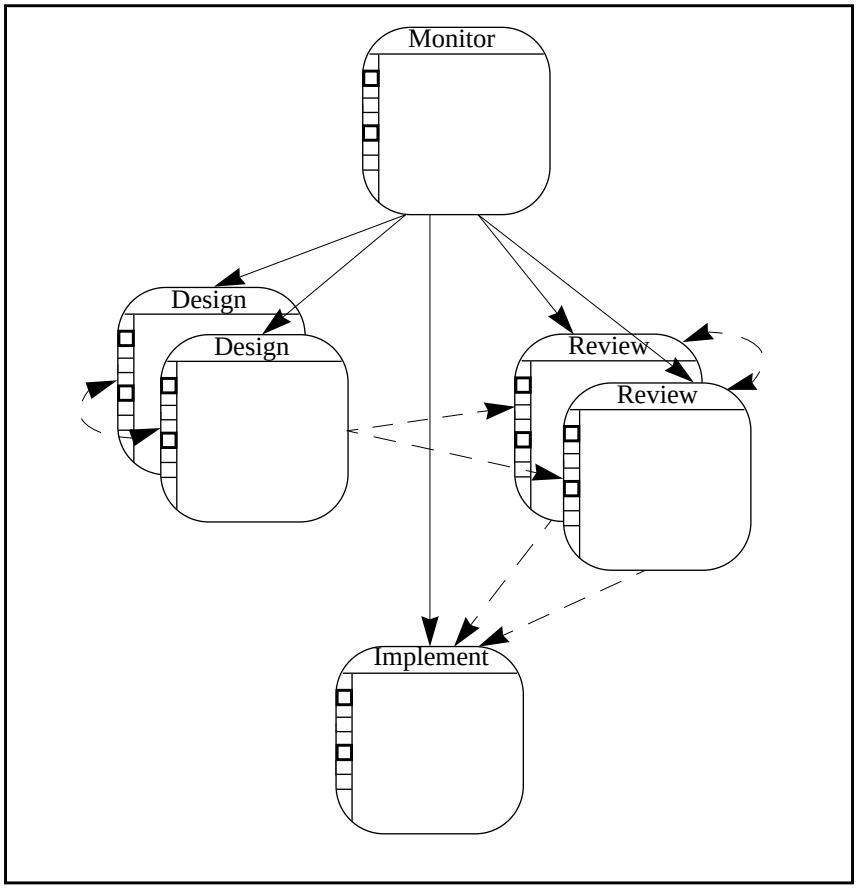


Fig. 6.3. RAPP diagram

In Fig. 6.4. a representation for an agent, its position and its roles in a process has been given. This is described by a rectangle which consists of two fields ‘A: <Agent name>’, a field ‘P:<Position name>’ to represent to corresponding name and position of a person or tool. In the same rectangle some ellipses can be distinguished which represent the roles the agent can/must perform.

At the left side of each process, a rectangle is depicted with information about what kind of tools are used by the agents in the process. This is represented with characters below TD (Tool Depot), each character reflects a certain tool which can be found in the Tool Depot. In the Tool Depot (TD) all tools are stored which can be used in a process. There is no explicit connection between a process and the TD, because the character which reflects the tool, implies that the agent can use this tool.

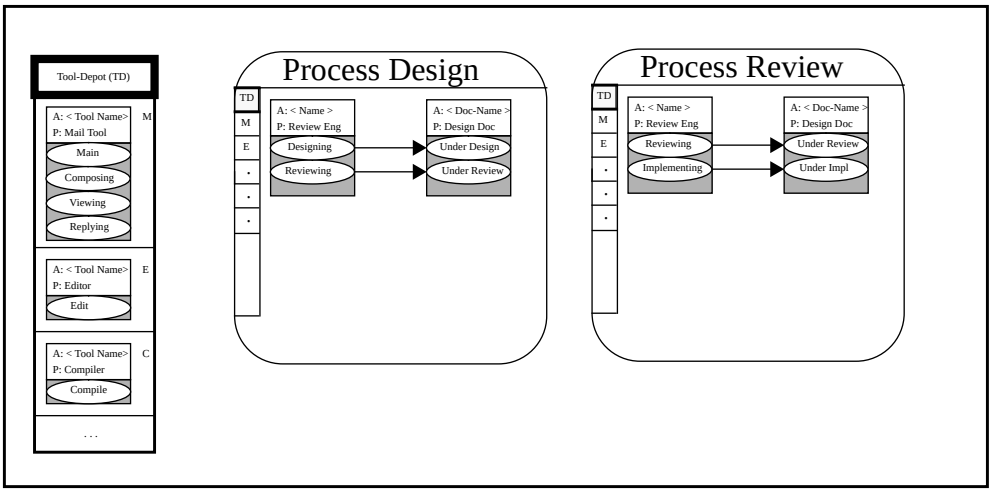


Fig. 6.4. RAPP diagram

In Fig. 6.5. a legend of the used symbols and connections is presented.

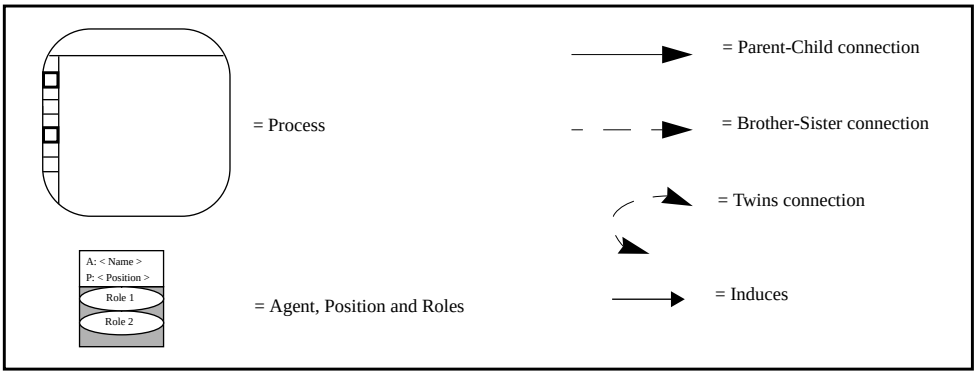


Fig. 6.5. Legend

Considering the roles of the mailtool and the roles of the design engineer in the SOCCA examples, is it clear that the RAPP diagram can be used on top of SOCCA as a general and more global description of a software process.

Chapter 7 Conclusions

In this thesis TEMPO and SOCCA have been discussed. Although they don't use the same concepts, it was possible to find some similarities. The main similarities between TEMPO and SOCCA were: both are object oriented, the events of TEMPO correspond to SOCCA's export operations of the external behaviour and the methods used in TEMPO correspond to the internal behaviour of the export operations in SOCCA. It turns out to be, that by using these similarities the examples modelled in TEMPO could be translated into a SOCCA description.

During the modelling of the examples (Design Engineer and MailTool) interesting aspects and extensions came into light. The modelling of Design Engineer wasn't of type-1 communication, but of type-2 communication. This resulted in an operating system way of describing the design engineer. In paragraph 3.4.5. a second alternative for the manager Design Engineer has been given. In this paragraph a proposition has been made to simplify the description of the manager process. Normally for each export operation the corresponding internal behaviour with its subprocesses and traps were given, but here not every export operation has an internal behaviour. The export operations which were typical for type-2 communication weren't described by any internal behaviour (subprocesses and traps). Although this proposition doesn't correspond to the original SOCCA formalism, it could be a useful adaptation to simplify the manager and to reduce the number of employee processes of the manager process. So it is a serious option for simplifying SOCCA models. During the modelling of the MailTool (see paragraph 3.5.) some extensions of the SOCCA formalism have been proposed and made. In the original SOCCA formalism, only one STD was used to represent the external behaviour. But in this thesis, the external behaviour of MailTool has been presented by more than one STD. This allows to describe parallel behaviour even in the external behaviour. Of course, this extension needs more adaptations of the SOCCA formalism, so more details can be found in paragraph 3.5. and its subparagraphs. This extension is an interesting topic for future research, because it could be applied for several other aspects of the software process e.g. representing associations, team representation. In the paragraph 3.7.5. an alternative for the manager MailTool has been given. The idea was to combine the internal behaviours of two closely related export operations into one internal behaviour. This allows to reduce the number of employee processes of the manager process too.

In Chapter 4, various role concepts were presented and finally a role concept for the SOCCA formalism has been proposed. This role concept admits to divide the external behaviour into several roles (at least one). By using this role concept the extent of a role can be restricted and therefore also the size of the total model of a software process.

In the last chapter, the RAPP diagram has been presented. The purpose of this diagram is to represent the software process on a more global level. So only four terms are of interest namely Roles, Agents, Position and Process. Despite this diagram is an informal and intuitive description of a software process, it could be used on top of SOCCA and probably on top of other formalisms too. So the RAPP diagram can be an interesting topic of future research.

A final remark will be, that for both software process modelling approaches TEMPO and SOCCA, it would be highly preferable that an environment will be implemented in the near future.

Appendix A: Mnemonics

ACID: Atomicity, Consistency, Isolation and Durability

DB: Database

DBMS: Database Management System

DEST: Destination

ECA: Event, Condition and Action

EER: Extended Entity-Relationship

MS: Manager Subprocess

mt: manager trap

OFD: Object Flow Diagram

OO: Object Oriented

PARADIGM: PARallelism its Analysis, Design and Implementation by a General Method

PSDE: Process-centred Software Development Environment

RAPP: Roles of Agents and their Position in a Process

SOCCA: Specifications of Coordinated and Cooperative Activities

STD: State Transition Diagram

TD: Tool Depot

TECA: Temporal Event, Condition and Action

WE: Work Environment

WS: Work Space

Appendix B: More TEMPO Examples

```
##########################################################################
Example 3, paragraph 2.13.3.

In this example the problem is: what to do if two designers work on the same document. They
have to solve eventually merge conflicts, when they check in their documents.
##########################################################################

_____________________________________________________________________________
user ISA object;

    ATTRIBUTE
        name     = String :='';              # name of the user is empty #
        Position = Pmanager, Engineer, None := None # the position of a user #
        Status   = 0,1,2,3,4:= 0;  # The status of the user, No=0,PM=1,Des=2 #
                                   # ,Rev=3, Impl=4 #

    METHOD
        Stop;                  # the user wants to do something else #

END_OF user;
_____________________________________________________________________________

_____________________________________________________________________________
document ISA object;

    ATTRIBUTE
         Name     = String := '';            # document name is empty #
        Status   = designed, reviewed, approved, implemented, none := none; # status of
doc#
        Date     = Date;                    # the date of the document #

END_OF document;
_____________________________________________________________________________

=============================================================
Monitor ISA PROCESS;

    CONTROL DESIGN;
        Fragment = Designing; # Designing corresponds with the fragment DESIGN #
                              # appearing in the type Monitor #

    CONTROL REVIEW;
        Fragment = Reviewing; # Reviewing corresponds with the fragment REVIEW #
                              # appearing in the type Monitor #

    ROLE Undermonitor;                    # role of the document #
```

```
    derived = document;        # object under this role is of type document #

    METHOD
        Update;              # Update the status of the document #
                                 # IF the date of status designed < date #
                                 # of status reviewed THEN the document #
                                 # gets the status approved. #

    RULES
        PRE WHEN Promote DO Update;            # document is promoted #

END_OF Monitor;
=================================================================

=================================================================
Designing ISA PROCESS;

   ROLE Design;                          # designing a document #

    derived = user;            # object under this role is of type user #

    METHOD
        Checkout;        # check out document from parent WE (= Monitor) #
        Checkin;         # check in document to parent WE (= Monitor)    #
        Decide;          # if conflict, make decision what to do #

    RULES
        PRE WHEN Get_doc  DO Checkout;          # check out document #
        PRE WHEN Give_doc DO Checkin;           # check in document  #
        !?PRE WHEN Conflict DO Decide;          # decide what to do  #

    ROLE Underdesign;                       # role of the document #

    derived = document;        # object under this role is of type document #

    METHOD
        Mod_doc;         # document can be modified #
        Stop_mod_doc;    # document can't be modified and status document #
                             # is designed. #

   RULES
        PRE WHEN Open_doc  DO Mod_doc;          # document open   #
        PRE WHEN Close_doc DO Stop_mod_doc;        # document closed #

END_OF Designing;
=================================================================
```

```
##############################################################################
```
Example 4, paragraph 2.13.4.

In this example I try to model what a tool has to do in case of a problem with licence rights. If someone with a lower priority wants to use the tool while all licences are occupied, he has to wait until one of the users give up his licence.
But if someone has a higher priority and he wants to use the tool, someone else has to give up his licence. The user (with the lower priority) has to give up his license within 2 minutes, if he doesn't do that, the system will do it. After that a message is sent to inform the user with the higher priority that he can use the tool now. In this example the tool will be an editor (= FrameMaker).

```
##############################################################################

DEFEVENT  Start_FM = [!cmd = start_fm];

------------------------------------------------------------------------------
FrameMaker ISA object;

    ATTRIBUTE
        name    = String    := '';      # name of FrameMaker              #
        uname   = String    := '';      # user name                       #
        uprio   = 0,1,2,3,4 := 0;       # priority of the user,1 is highest #
        version = Integer   := 0;       # version number                  #
        date    = Date      := 00/00/00; # date of release                #

    METHOD
        CheckLicenseFree;               # check if there is a free license #
        PopUpFMWndw;                    # FrameMaker menu is started      #
        Wait;                       # wait                              #
        GetLicense;                 # user gets a license               #
        GiveUpLicense;              # user's license is given up        #

END_OF FrameMaker;
------------------------------------------------------------------------------

==============================================================
Designing ISA PROCESS;

    ROLE Design;                            # designing a document #

    derived = user;             # object under this role is of type user #

    METHOD
        ...

    RULES
        ...

    ROLE D_FrameMaker;                      # FrameMaker for a designer #
```

```
    derived = FrameMaker;      # object under this role is of type FrameMaker #

    RULES
        PRE   WHEN Start_FM DO {IF CheckLicenceFree THEN
                                        PopUpFMWndw
                                    ELSE
                                        ABORT;}

        ERROR WHEN Start_FM DO {IF SearchLowPrio(user,user*) THEN
                                        {sendmessage('give up licence in 2 min',user);
                                        Wait(2min);

                                        IF CheckLicenceFree THEN
                                            GetLicence(user);
                                            sendmessage('you've a licence',user);
                                        ELSE
                                            GiveUpLicence(user*);
                                            sendmessage('you've a licence,user);}
                                       ELSE
                                        sendmessage('you've to wait, try again',user);
                                       ABORT;}

END_OF Designing;
=====================================================================
```

```
#############################################################################
Example 5

In this example will be described how the different project members communicate with each
other. The project members use a mailtool to communicate.
The cooperation between the documents is defined by a connection.
#############################################################################
```

```
_____________________________________________________________________________
user ISA object;

    ATTRIBUTE
        name     = String := '';                # username is empty      #
        Position = Pmanager,Engineer,None := None   # the position of a user #
        Status   = Integer := 0;   # The status of the user, No=0,PM=1, Des=2#
                            # ,Rev=3, Impl=4                    #

    METHOD
        ...

END_OF user;
_____________________________________________________________________________
```

```
document ISA object;

    ATTRIBUTE
        name     = String := ‘’;                # document name is empty #
        Status   = designed,reviewed,approved,implemented,none := none;
                                                                    # status of the document
#

    METHOD
        ...

END_OF document;
```

```
mailtool ISA object;

    ATTRIBUTE
        name    = String  := ‘’;         # name of mailtool #
        uname   = String  := ‘’;          # user name       #
        version = Integer := 0;           # version number   #
        date    = Date    := 00/00/00;    # date of release #

    METHOD
        PopupMailToolWndw;# mail tool window is popped up #
        Store;           # store the mail in the buffer #
        Send;            # send mail #
        Sound;           # make a sound to inform user for new mail #
        PopupComposeWndw; # Compose window is popped up #
        CloseComposeWndw; # Compose window is closed #
        GiveInfo;        # information is provided to the user #
        DisplayMessage;   # message is displayed in a window #

END_OF mailtool;
```

```
============================================================
Monitor ISA PROCESS;

    CONTROL DESIGN;
        Fragment = Designing; # Designing corresponds with the fragment DESIGN #
                              # appearing in the type Monitor              #

    CONTROL REVIEW;
        Fragment = Reviewing; # Reviewing corresponds with the fragment REVIEW #
                              # appearing in the type Monitor              #
```

```
    ROLE Manage;                                # role of the user #

    derived = user;              # object under this role is of type user #

    METHOD
          ...

    RULES
          ...

    ROLE Undermonitor;                          # role of the document #

          derived = document;        # object under this role is of type document #

    METHOD
          Merge_doc;          # merge doc. of child WE with doc. in parent WE #

   RULES
          PRE  WHEN Promote DO Merge_doc;            # document is promoted #

    ROLE M_Mailtool;                          # mailtool for a manager #

    derived = mailtool;         # object under this role is of type mailtool #

    RULES
          PRE   WHEN Start_MT DO PopupMailToolWndw;      # activate mail tool #
          PRE   WHEN Receive  DO Store;                  # receive new mail #
          POST  WHEN Receive  DO Sound;
          PRE   WHEN Compose DO PopupComposeWndw;# User wants to send a message#
                                                 # User writes address, subject #
                                                 # and the message in the com- #
                                                  # pose window.              #
          PRE   WHEN Deliver Do Send;          # mail is sent to destination #
          POST  WHEN Deliver DO CloseComposeWndw;
          ERROR WHEN Deliver DO GiveInfo;             # mail cannot be send #

          PRE   WHEN View    DO DisplayMessage;  # user wants to read the mail #
          ERROR WHEN View DO GiveInfo;               # no mail in mailbox #

          PRE   WHEN Reply   DO PopupComposeWndw;# user doesn't have to write  #
                                                 # the address of the destination#

END_OF Monitor;                           # end of process Monitor #
=================================================================

=================================================================
Designing ISA PROCESS;

    ROLE Design;                          # designing a document #
```

```
    derived = user;              # object under this role is of type user #

    METHOD
          ...

    RULES
          ...

    ROLE D_Mailtool;                          # mailtool for a designer #

    derived = mailtool;          # object under this role is of type mailtool #

    RULES
          PRE   WHEN Start_MT DO PopupMailToolWndw;       # activate mail tool #

          PRE   WHEN Receive  DO Store;                  # receive new mail #
          POST  WHEN Receive  DO Sound;

          PRE   WHEN Compose DO PopupComposeWndw;# User wants to send a message#
                                                  # User writes address, subject #
                                                  # and the message in the com- #
                                                   # pose window.              #

          PRE   WHEN Deliver Do Send;          # mail is sent to destination #
          POST  WHEN Deliver DO CloseComposeWndw;
          ERROR WHEN Deliver DO GiveInfo;             # mail cannot be send #

          PRE   WHEN View    DO DisplayMessage;  # user wants to read the mail #
          ERROR WHEN View DO GiveInfo;               # no mail in mailbox #

          PRE   WHEN Reply   DO PopupComposeWndw;# user doesn't have to write  #
                                                  #the address of the destination #

END_OF Designing;
=================================================================

=================================================================
Reviewing ISA PROCESS;

    ROLE Review;                              # reviewing a document #

    derived = user;              # object under this role is of type user #

    METHOD
          ...

    RULES
          ...
```

```
    ROLE R_Mailtool;                          # mailtool for a designer #

    derived = mailtool;        # object under this role is of type mailtool #

    RULES
        PRE   WHEN Start_MT DO PopupMailToolWndw;      # activate mail tool #

        PRE   WHEN Receive  DO Store;                  # receive new mail #
        POST  WHEN Receive  DO Sound;

        PRE   WHEN Compose DO PopupComposeWndw;# User wants to send a message#
                                                        # User writes address, subject #
                                                       # and the message in the com- #
                                                        # pose window.              #

        PRE   WHEN Deliver Do Send;          # mail is sent to destination #
        POST  WHEN Deliver DO CloseComposeWndw;
        ERROR WHEN Deliver DO GiveInfo;              # mail cannot be send #

        PRE   WHEN View    DO DisplayMessage;  # user wants to read the mail #
        ERROR WHEN View DO GiveInfo;                 # no mail in mailbox #

        PRE   WHEN Reply   DO PopupComposeWndw;# user doesn't have to write  #
                                                     # the address of the destination#

END_OF Reviewing;
=================================================================

-----------------------------------------------------------------------------------------------------------
des_rev_doc ISA CONNECTION      # connection between the roles of the document #
                                  # in the WE-Designing and the WE-Reviewing     #

    DOMAIN  Designing: Underdesign ->
                Reviewing: Underreview;

    PLUG-ON-RULES
        WHEN Begin_design   UPON SOURCE;
        WHEN Continu_design UPON SOURCE;
        WHEN Begin_review   UPON DEST;
        WHEN Cont_review    UPON DEST;

    ACTIVE-RULES
        WHEN Design_completed UPON SOURCE
        DO   Send_doc;                       # send document to reviewer #

        WHEN Design_reviewed UPON DEST
        DO   Send_doc;              # if changes, send document to designer #
                                      # else to the manager                 #
```

```
    PLUG-OFF-RULES
        WHEN Stop_design   UPON SOURCE;
        WHEN Finish_design UPON SOURCE;
        WHEN Stop_review   UPON DEST;
        WHEN Finish_review UPON DEST;

END_OF des_rev_doc;                  # end of connection des_rev_doc #
-------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------------------
des_des_doc ISA CONNECTION     # connection between the roles of the document #
                                # in the WE-Designing and other WE-Designing   #

    DOMAIN  Designing: Underdesign ->
        Designing: Underdesign;

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

    ACTIVE-RULES
        WHEN Design_problem UPON (SOURCE OR DEST);
        DO   Send_doc;               # send document to other designer #

        WHEN Problem_solved UPON (SOURCE OR DEST);
        DO   Send_doc;               # send document back with updates #

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

END_OF des_des_doc;                  # end of connection des_des_doc #
-------------------------------------------------------------------------------------------------------------
```

References

[1] Belkhatir, N., Melo, W.L. and Estublier, J.: Adele 2: A Support to Large Software Development Process. In Dowson, M. (editor): Proceedings of the First International Conference on the Software Process, Redondo Beach, CA, October 21-22, 1991.

[2] Belkhatir, N., Melo, W.L., Estublier, J. and Nacer, M.A.: Supporting Software Maintenance Evolution Processes in the Adele System. In Proceedings of the 30th Annual ACM Southeast Conference, Raleigh, NC, April 08-10, 1992.

[3] Belkhatir, N. and Melo, W.L.: TEMPO: a Software Process Model Based on Object Context Behavior. In Proceedings of the 5th International Conference on Software Engineering & its Applications, Toulouse, France, December 07-12, 1992.

[4] Belkhatir, N., Estublier, J. and Melo, W.L.: Software Process Model and Work Space Control in the Adele System. In Osterweil, L. (editor): Proceedings of the 2nd International Conference on the Software Process, Berlin, Germany, February, 1993.

[5] Belkhatir, N., Estublier, J. and Melo, W.L.: ADELE-TEMPO: An Environment to Support Process Modelling and Enaction. In Finkelstein, A., Kramer, J. and Nuseibeh, B. (editors): Software Process Modelling and Technology, 1994.

[6] Canals, G., Boudjlida, N., Derniame, J.C., Godart, C. and Lonchamp, J.: ALF: A Framework for Building Process-Centred Software Engineering Environments. In Finkelstein, A., Kramer, J. and Nuseibeh, B. (editors): Software Process Modelling and Technology, 1994.

[7] Casallas, R.: Using Triggers in a Software Configuration Manager, Techn. Rep., Laboratoire de Génie Informatique, Grenoble, 1992.

[8] Conradi, R., Fernström, C. and Fuggetta, A.: Concepts for Evolving Software Processes. In Finkelstein, A., Kramer, J. and Nuseibeh, B. (editors): Software Process Modelling and Technology, 1994.

[9] Engels, G. and Groenewegen, L.P.J.: Specification of Coordinated Behaviour in the Software Development Process (Position Paper). In Derniame, J.C. (editor): Proceedings of the 2nd European Workshop on Software Process Technology (EWSPT 92), Tronheim, Norway , Springer-Verlag, Berlin, LNCS 635, 1992.

[10] Engels, G. and Groenewegen, L.P.J.: SOCCA: Specifications of Coordinated and Cooperative Activities. In Finkelstein, A., Kramer, J. and Nuseibeh, B. (editors): Software Process Modelling and Technology, 1994.

[11] Estublier, J.: The Adele Configuration Manager, Techn. Rep., Laboratoire de Génie Informatique, Grenoble, 1992.

[12] Estublier, J.: The Adele Work Space Manager, Techn. Rep., Laboratoire de Génie Informatique, Grenoble, 1994.

[13] Groenewegen, L.P.J.: Parallel Phenomena 1 - 14, Techn. Rep. 86-20, 87-01, 87-05, 87-06, 87-11, 87-18, 87-21, 87-29, 87-32, 88-15, 88-17, 88-18, 90-18, 91-19, University of Leiden, Department of Computer Science, 1986-1991.

[14] Junkermann, G., Peuschel, B., Schäfer, W. and Wolf, S.: MERLIN: Supporting in Software Development Through a Knowledge-Based Environment. In Finkelstein, A., Kramer, J. and Nuseibeh, B. (editors): Software Process Modelling and Technology, 1994.

[15] Lonchamp, J.: A Structured Conceptual and Terminological Framework for Software Process Engineering. In Osterweil, L. (editor): Proceedings of the 2nd International Conference on the Software Process, Berlin, Germany, February, 1993.

[16] Melo, W.L.: TEMPO: Un Environnement de Développement Logiciel Centré Procédés de Fabrication. Ph.D. Thesis. (French), Laboratoire de Génie Informatique, l'Université Joseph Fourier, Grenoble, October 22, 1993.

[17] Montangero, C. and Ambriola, V.: OIKOS: Constructing Process-Centred SDEs. In Finkelstein, A., Kramer, J. and Nuseibeh, B. (editors): Software Process Modelling and Technology, 1994.

[18] Morssink, P.J.A.: Behaviour Modelling in Information Systems Design: Application of the PARADIGM Formalism. Ph.D. Thesis, University of Leiden, Department of Computer Science, 1993.

[19] Steen van, M.R.: Modelling Dynamic Systems by Parallel Decision Processes. Ph.D. Thesis, University of Leiden, Department of Computer Science, 1988.