EVOLUTIONARY CHANGE
the evolution of change management

by Jeroen van der Zon

University of Leiden, Department of Computer Science

April 24, 1996

Abstract

In this thesis, evolutionary change is studied by describing the evolution of Change Management (CM). CM is one of the important aspects of the software process modelling lifecycle. It describes the organizational aspects of changing a software product. In order to be able to describe the evolution of CM, two models describing CM have been developed first. Both models have been designed with the use of Socca. The first model describes the basic requirements for a CM-model. The second model is an extension of the first model. The CM-process will undergo an evolutionary transformation from this first model to this second model. This will be called evolutionary change. The evolutionary change is described by means of an extra process component called WODAN, first introduced in [2]. In order to relate the WODAN approach to similar approaches used by others, also PMMS [9] has been used to describe the change.

Table of contents

Abstract . . . 3

Table of contents . . . 5

1 General introduction . . . 7

2 Socca and Paradigm . . . 9
- 2.1 Introduction into Socca . . . 9
  - 2.1.1 The data perspective . . . 9
  - 2.1.2 The behaviour perspective . . . 9
  - 2.1.3 The process perspective . . . 10
- 2.2 Introduction into Paradigm . . . 10

3 Modelling Change Management . . . 13
- 3.1 Introduction . . . 13
- 3.2 Data perspective . . . 13
- 3.3 Behaviour perspective . . . 16
  - 3.3.1 Designing the external behaviours of the classes . . . 16
  - 3.3.2 Designing the internal behaviours of the export-operations . . . 21
  - 3.3.3 Adding Paradigm to model the communication . . . 26

4 The new model . . . 45
- 4.1 Introduction . . . 45
- 4.2 Designing the new model . . . 45
  - 4.2.1 Designing the new external behaviours of the classes . . . 50
  - 4.2.2 Designing the new internal behaviours of the export-operations . . . 53
- 4.3 The communication in the new model . . . 59

5 WODAN, a method to describe change . . . 79
- 5.1 Introduction . . . 79
- 5.2 Types of change . . . 80
- 5.3 Problems as a consequence of change . . . 80

6 Changing the software process model using WODAN . . . 83
- 6.1 A setup for WODAN . . . 83
- 6.2 Designing WODAN to manage the change . . . 84

7 PMMS . . . 99
- 7.1 Introduction . . . 99
- 7.2 Change Management modelled by means of PMMS . . . 100
- 7.3 Behaviour of the basic components . . . 101
- 7.4 Communication between the components . . . 109

8 Conclusions and future research . . . 123

9 References 125

Appendix A. Simultaneous calls described in more detail 127

Appendix B. Logical transitions 131

Appendix C. List of subprocesses w.r.t. WODAN 133

Appendix D. List of figures 135

1 General introduction

Whenever a software process model does not answer to the expectations or cannot cope with the actual situation anymore, it will have to be changed. This type of change is called evolutionary change. It was first introduced in [2]. In [2] also a method to describe this type of change was introduced and a small example was presented. This thesis will present a larger, more complicated and realistic example of the so-called evolutionary change of a software process model. The software process model on which this example will be based, is the software process model describing Change Management (CM). CM is one of the important aspects of the software process modelling lifecycle. It describes the organizational aspects of changing a software product. This type of change is called configuration change in order to distinguish it from the evolutionary change.
In order to be able to present the example describing the evolutionary change of CM, CM has to be modelled first. A base scenario for CM has been given in ISPW-9 [6]. Based on this scenario the CM-aspect of the lifecycle will be modelled. The scenario also involves the aspect of problem reporting, however the models in this thesis will not consider this issue. Also the actual change of the software will not be considered. The models in this thesis will focus upon the way the configuration change is embedded into CM and the organizational aspects of CM involving the Change Advisory Board (CAB). The CAB can be seen as the coordinator of the configuration change management process and consists of several (human) team members.
There are many ways to model (aspects of) a software process. Socca (and Paradigm, an integrated part of Socca) is one of them. Socca is especially suited to model CM, as it 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. Therefore the models, that describe Change Management, will be designed with the use of Socca. The first model describes the basic requirements for a CM-model. Whenever a change of a software product is requested, a meeting will be prepared and opened. In this meeting the request will be discussed and eventually accepted or rejected. In case the request has been accepted, it will trigger a change in the software. The model will be an extension of the original example presented in ISPW-6 [5]. Parts of the model to be designed can also be applied to other situations, as it in fact describes a way to plan a meeting.
As mentioned above this thesis elaborates further upon the type of change called evolutionary change. Therefore a second model, that describes Change Management too, has to be designed. The first model described just the basic requirements for a Change Management model. The second model is a more realistic extension of the first model. There will be made a distinction between big and small changes, moreover more than one request can be handled in a meeting. After designing the second model the enactment of this model is desired. In order to accomplish this one has to switch from the first to the second model. This is the type of change called evolutionary change. The first model will represent the first evolution phase, and the second model will represent the second evolution phase. In this thesis these models will be discussed more or less separately, without the evolutionary transformation from the first into the second model. That will be discussed in another chapter.
In order to facilitate the modelling of both CM models, Socca will be extended with some new concepts. These concepts are "layered visibility" (first introduced in [3]), logical transitions and simultaneous calls. These new concepts will of course be clarified in this thesis.

Before presenting the structure of this thesis, I want to thank Luuk Groenewegen for the excellent guidance and the many discussions which led to a better understanding of the ideas behind Change Management, WODAN and PMMS.

This thesis has been organized as follows. The next chapter consists of a short introduction into Socca and Paradigm. Those readers familiar with Socca and Paradigm may skip this chapter. Chapter 3 describes the first Change Management model. Also the concept of "layered visibility" is introduced. In chapter 4 the model as presented in chapter 3 will be extended in order to meet more realistic requirements. In chapter 5 a method [2] with which the change from the first to the extended model can be described is introduced and in chapter 6 this change is modelled with the use of that method. Finally, in chapter 7, another method to describe evolutionary change is discussed. It will clarify the way evolutionary change is conducted. Appendices describing simultaneous calls (Appendix A) and logical transitions (Appendix B) have been added too.

2 Socca and Paradigm

2.1 Introduction into Socca

In [3] a complete introduction of Socca has been given. Those readers familiar with it may skip this chapter.
Socca, which stands for **S**pecifications **o**f **C**oordinated and **C**ooperative **A**ctivities, is a software process modelling methodology, developed at the University of Leiden, department of Computer Science. Socca 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 idea behind Socca is the separation of concern. A Socca model describes the software process from three different perspectives; the data perspective, the behaviour perspective and the process perspective. To achieve this, Socca consists of (parts of) several formalisms combined together to describe the software process models.

2.1.1 The data perspective

The data perspective describes the static structure of the system and its relation to its environment by means of object-oriented class diagram models, based on Extended Entity-Relationship (EER) models. One of the features of the classes is that they can have export-operations. These export-operations are imported in other classes in order to be called (or used) from there. To visualize this with respect to the class diagram in Socca an extra relationship is introduced between the various classes; the uses relationship. Consequently the class models have been extended with an extra diagram, the import/export diagram, to display the uses relationship. The other relationships that can be identified in the class diagram are the IS-A relationship, the Part-Of relationship and the general relationship. The IS-A relationship is used to describe the inheritance between the different classes and the Part-Of relationship expresses which class is part of another class. The general relationship describes the relation between the different classes and is usually presented in a separate diagram.

2.1.2 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 (STD's) and Paradigm on top of them. The STD's are used to describe the order in which the export-operations of a class can be called. So in fact, the behaviour of a class is described with an STD of which some transitions are labelled with the export-operations of that class. This is called *the external behaviour* of the class. Also each export-operation has an *internal behaviour*; this internal behaviour actually achieves the task the corresponding export-operation is supposed to perform. To describe these internal behaviours also STD's have been used. Together the external behaviour of a class and the internal behaviours of its export-operations form the behaviour of that class.
It is obvious that the cooperation between the external behaviour of a class and the internal behaviours of its export-operations has to be coordinated somehow. To this aim Paradigm has been incorporated into Socca. Moreover, the internal behaviour of an export-operation can also call export-operations from other classes. Therefore it is not only necessary to have communication between the external behaviour of a class and the internal behaviours of its export-operations, but there should also be communication between the external behaviour of one class and the internal behaviours of another class from where the there imported export-operations are being called. This communication is also modelled by means of Paradigm. Section 2.2 gives a short introduction into Paradigm.

2.1.3 The process perspective

The process perspective will be modelled by Object Flow Diagrams (OFD's). As the integration of OFD's into Socca has not been completed yet, the process perspective will not be discussed any further in this thesis.

2.2 Introduction into Paradigm

Paradigm (**Par**allelism, its **A**nalysis, **D**esign and **I**mplementation by a **G**eneral **M**ethod) is a specification mechanism originally developed for the specification of coordinated parallel processes. A Paradigm model can be designed in the following manner:

1. Describe the sequential behaviour of each process by means of an STD. In case of a Socca model the processes will be the classes and the export-operations, and the STD's will describe the external and internal behaviours.
2. Within an STD that describes internal behaviour a set of subdiagrams (called *subprocesses*) is indicated. 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 certain sets of states, so-called *traps*, can be identified. They are usually represented by a shaded polygon drawn around the states which form the trap. By entering such a trap, the STD indicates that it is ready to switch from the subprocess to which this trap belongs to another subprocess. The set of traps of an STD is called the *trap structure* of that STD. An important property of a trap is that, within the subprocess the trap belongs to, there are no transitions leading from one of the states of a trap to another state outside the trap. Consequently when an STD has entered a trap, the STD cannot leave its trap as long as the same subprocess restriction remains valid.
4. An STD that decribes external behaviour is called a *manager process*. It coordinates the behaviour restrictions of some of the STD's having subprocesses. An STD that is being coordinated by the manager process is called an *employee process* of that manager process. Depending on the state it is in, the manager process prescribes subprocesses to each of its employees. Every employee may only behave according to the subprocess which is currently being prescribed by its manager process. Next, the manager process monitors the behaviour of its employees. Whenever an employee has entered a trap to another subprocess, the manager will make the corresponding transition to another state where it prescribes the subprocesses the employee wants to enter. However it can also postpone the prescribing of the new subprocess to its employee as long as it wants to, possibly depending on traps having been reached by other employee processes. The mapping of the states of the manager process to the subprocesses of its various employees and the mapping of the transitions of the manager process to the traps of its employees is called the *state-action interpreter* of the manager process with respect to its employees. So the state-action interpreter labels each state of the manager process with the subprocesses it prescribes in that state to its employees and labels the transitions of the manager process with those traps that have to be entered for this transition to be selected.

Each manager process, i.e. external behaviour, is manager over:

1. each of its own internal behaviours and also over
2. each internal behaviour containing a call to one or more of the export-operations of that manager process.

An individual STD may be the employee of more than one manager. In that case the STD will have a separate set of subprocesses and a separate trap structure with respect to each of its man-

agers. The behaviour of the STD will be controlled by all of its manager processes together and the STD will be restricted to the intersection of the different subprocesses prescribed by the different manager processes.
More information on Paradigm can be found in [7].

3 Modelling Change Management

3.1 Introduction

In order to be able to present the evolutionary change of Change Management, models describing Change Management are necessary. Only then Change Management can evolve from one evolution phase to another. Therefore in this chapter a model describing Change Management will be designed. This model will represent the first evolution phase.
The model will be relatively simple, just clarifying some essentials of Change Management. For every change-request, i.e. a request for changing a software product, a meeting, in which the request is discussed, will be prepared and opened. Consequently every request corresponds with a meeting. There will be made no difference between requests. This implies that all requests will be treated the same way. Because of "layered visibility" (explained later) simultaneous requests are possible, i.e. requests can be made simultaneously. Because of parametrization of operations meetings can also be held concurrently. The model itself is an extension of the original example (ISPW-6). In the next sections the data-perspective of the model and the behaviour perspective of the model will be given. To describe the behaviour perspective the external behaviours of the classes, the internal behaviours of the export-operations and the communication between these behaviours will be given. The process-perspective will not be discussed. Later on, in Chapter 4, the model will be extended and better tuned to what is needed in reality. Then more than one request can be discussed in a meeting and also there will be made a difference between big and small changes. That model will represent the second evolution phase.

3.2 Data perspective

In order to describe the data perspective of the model an EER based class diagram has to be defined. In this class diagram three extra classes have been defined compared to the original example (ISPW-6 [5]):

- Change Advisory Board (CAB): the coordinator of the change management process, it consists of several (human) team members; a ProjectManager, two DesignEngineers (for design expertise), two Quality-AssuranceEngineers (for testing expertise) and two User-Representatives. The ProjectManager is also the board-leader, who prepares a meeting after it has been requested.
- CABMember: a superclass, which possesses all properties members of the board should possess. Its subclasses are Engineer and UserRepresentative. Note that not every Engineer always acts as a CABMember, although it has all the properties of the class CABMember, because not every instance of the class Engineer is related to every instance of the class CAB (see also Figure 1).
- UserRepresentative: one of the members of the board and a subclass of the class CABMember.

Figure 1 shows the class diagram of the model. Not all details are given on the data perspective level. Later on in the model, at the behaviour perspective level (section 3.3.1), a parallel description in the external behaviour of CAB will be introduced. CAB will be split into two (sub)classes: MainCAB (has only one instance) and DepCAB (has several instances). MainCAB will be the manager of DepCAB. This is called "layered visibility" and was first introduced in [3]. By this means parallel behaviour in one and the same external behaviour is modelled. MainCAB and several instances of DepCAB can be executed concurrently, thereby making simultaneous requests possible. The separation of MainCAB and DepCAB is not visible at the data perspective level.

Note that every change, not only the first change, by which the software is created, is considered to be a project.

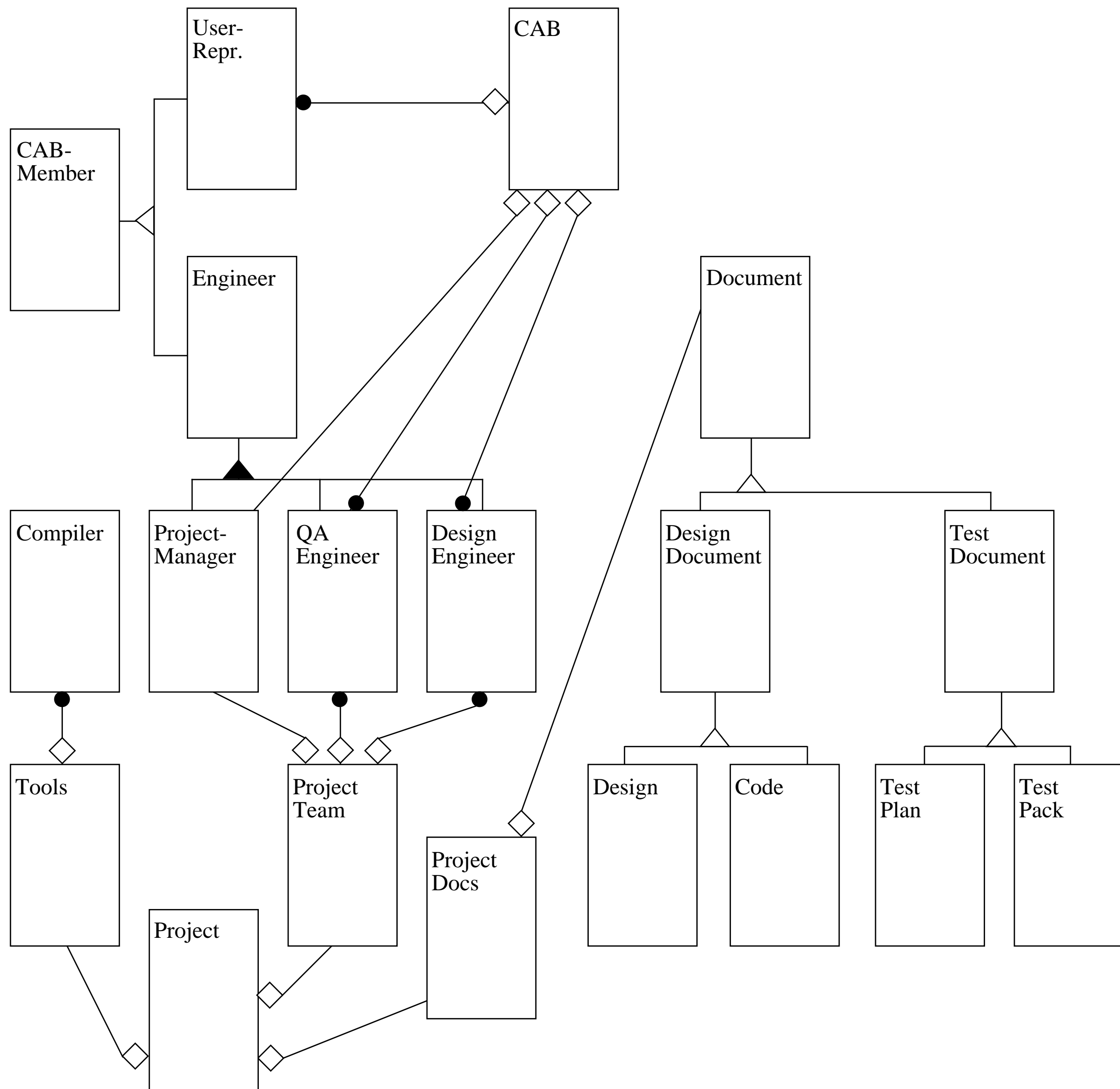


Figure 1. Class diagram: classes and IS-A and Part-Of relationships

Figure 2 shows the operations of the three extra classes and the additional operation of the class ProjectManager. In the latter case for the sake of completeness the original attributes and operations have also been mentioned.

ProjectManager	CAB	UserRepresentative	CABMember
			name
assign_and_schedule_t. monitor prepare_meeting	request_for_change open_meeting do_meeting close_meeting do_change authorize cancel		join_meeting leave_meeting check_agenda receive_confirmation

Figure 2. Class diagram: attributes and operations

The export-operations of the class CAB have all (implicitly) been parametrized with a parameter *request-id*, just like the operations *assign_and_schedule_tasks* and *monitor* of ProjectManager have (implicitly) been parametrized with a parameter *doc_name* (see [1]). Also the operation *prepare_meeting* has been parametrized with the parameter *request-id*. In section 3.3.1 the external behaviour of CAB will be modelled and a reason for using the parameter *request-id* will be given.
Note that the classes ProjectManager, DesignEngineer, QualityAssuranceEngineer (through the class Engineer) and UserRepresentative all inherite the operations *join_meeting*, *leave_meeting*, *check_agenda* and *receive_confirmation* from the class CABMember.
In the following step of describing the data perspective of the model the general relationships between the classes have to be defined. Only the new general relationships are shown in Figure 3.

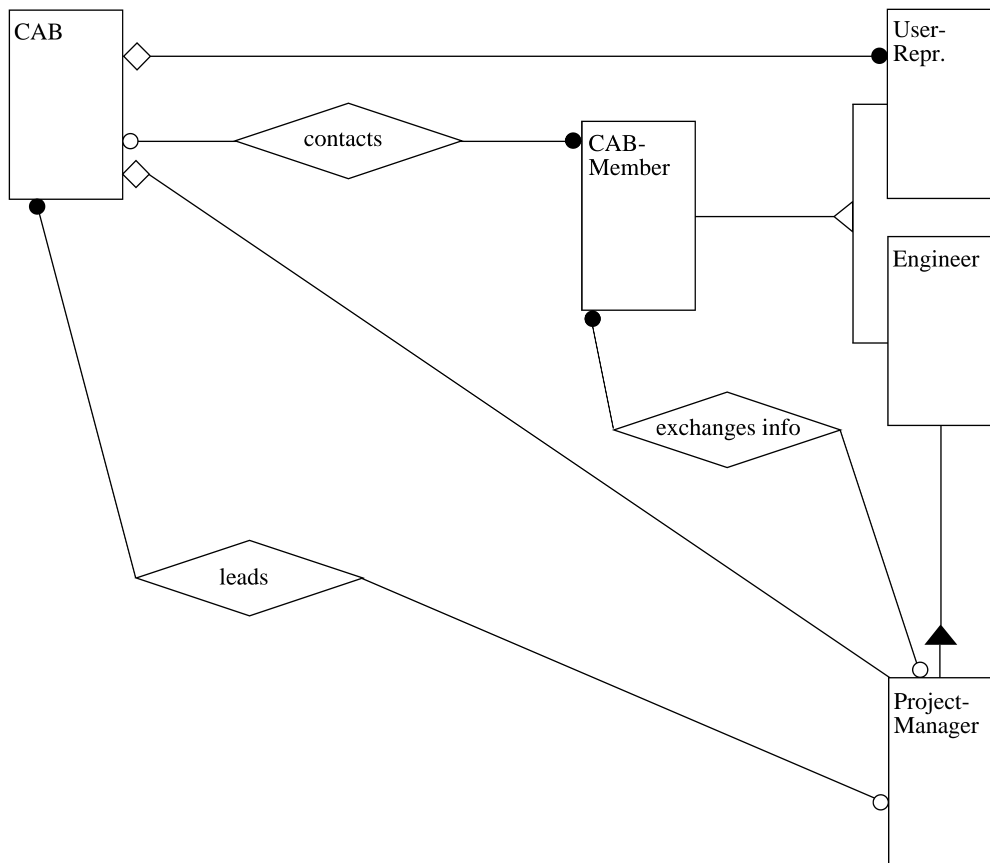


Figure 3. Class diagram: classes and general relationships

As a last step in the description of the data perspective of the model, the uses relation is given (Figure 4) together with the corresponding import list (Figure 5). Note that only the new uses relations are given in the figures, as the old uses relations will not be useful for our discussion.

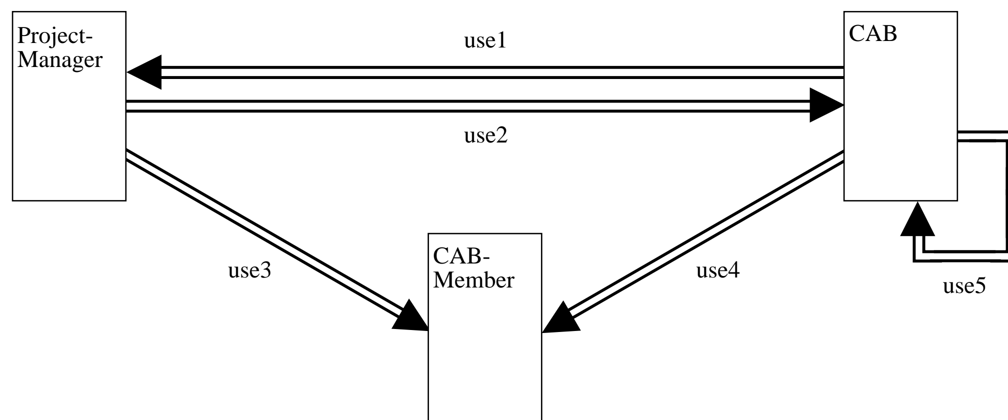


Figure 4. Import/export diagram

use1	use3
prepare_meeting	*check_agenda (member)*
schedule_and_assign_tasks	*receive_confirmation (member)*
use2	use4
open_meeting	*join_meeting (member)*
do_meeting	*leave_meeting (member)*
close_meeting	use5
	do_change

Figure 5. Import list

Note that the operations *check_agenda*, *receive_confirmation*, *join_meeting* and *leave_meeting* are parametrized with the parameter *member* as an explicit reminder of the fact that some details in the calling operations have been omitted. These details, which actually are rather complicated, will be discussed in Appendix A.

3.3 Behaviour perspective

3.3.1 Designing the external behaviours of the classes

First the external behaviours of the classes will be specified. From then on, the order in which the export-operations can be called will be known. Not only the classes CAB and UserRepresentative have to be modelled, but also the classes ProjectManager, DesignEngineer and QualityAssuranceEngineer have to be remodelled because some new operations have to be added to model Change Management. Note that the external behaviour of the class CABMember will not be given here, because it is a generalization of some specialized classes. The class itself has no instances, only through its subclasses.

A possible STD for the class CAB is given in the following figure.

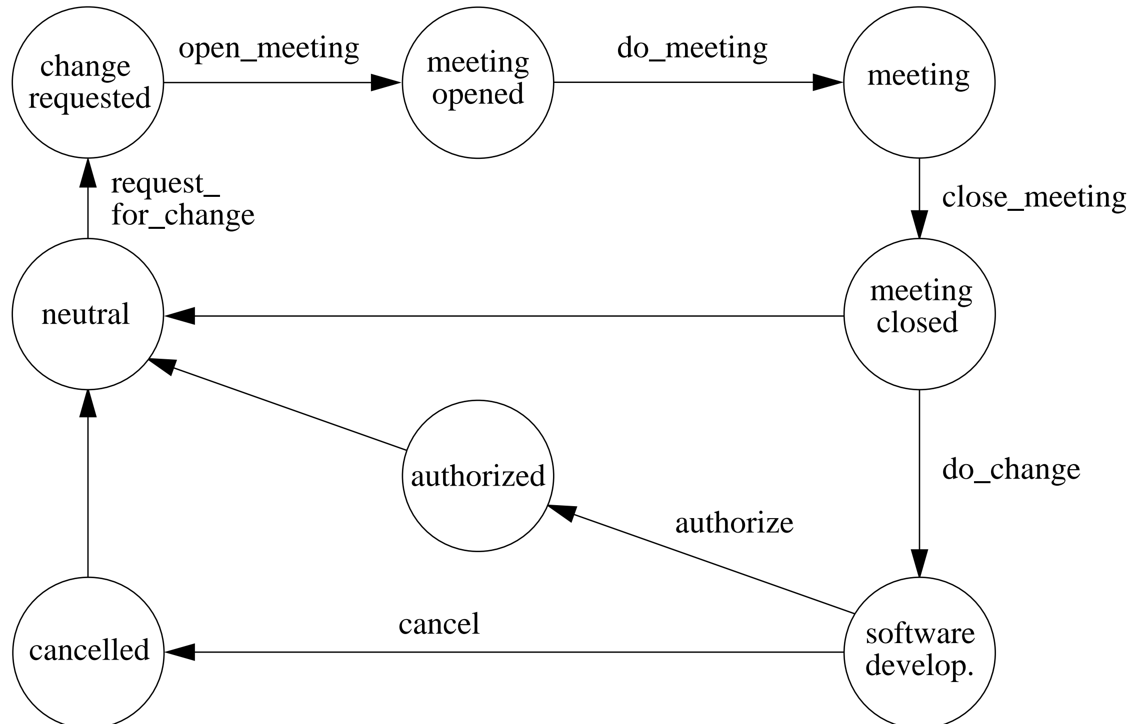


Figure 6. CAB: possible STD of the external behaviour

The STD, presented in Figure 6, provides all the necessary export-operations, but there is one major disadvantage: because of the sequentiality of the model one has to wait for a previous change being implemented and after that authorized or cancelled before requesting a new change.

To solve this problem 3 other ways to model the external behaviour, thereby modelling Change Management, are possible:

1 The first solution is the interleaved version. As one can see in Figure 7, it is not immediately clear in what order the operations are called.

2 The second solution has to do with an n-dimensional state-space (n = number of simultaneous requests). For large n this will be very complicated.

3 The third solution has to do with “layered visibility” or "zooming", as you are zooming in on an operation, in this case *request_for_change*. If one uses this solution to describe the external behaviour, the STD of the external behaviour will be split in two layers. So actually one gets two STD’s when describing external behaviour: one STD modelling the first layer of visibility and one STD modelling the second layer of visibility (see Figures 8 and 9). Layered visibility is not supported by the original SOCCA approach.

Note that whatever solution is chosen, for every request a separate meeting has to be prepared and opened. But these three solutions have the advantage that requests, meetings and the realisation of changes can be executed simultaneously (unless e.g. the same person is supposed to join two or more overlapping meetings).

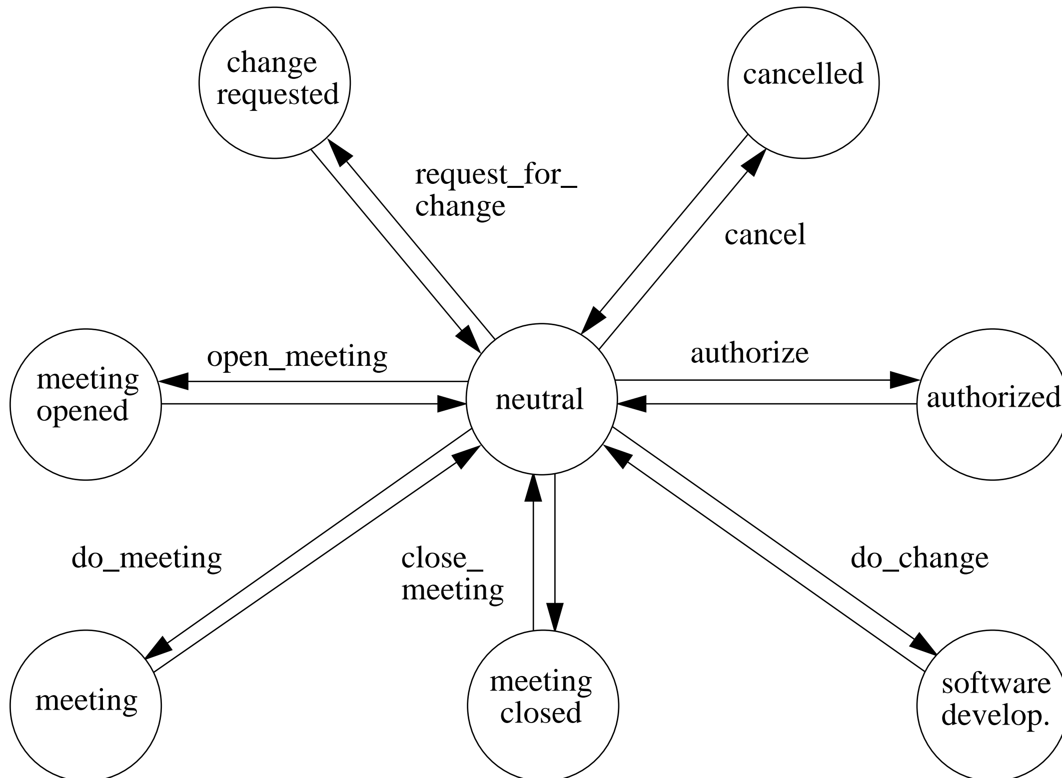
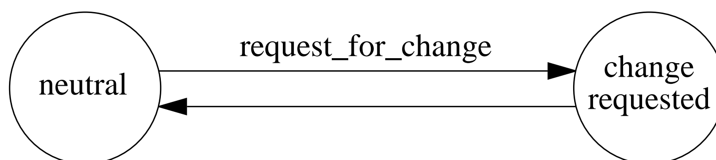


Figure 7. CAB (solution 1): the interleaved version

In order to model Change Management the third solution will be used. Normally the external behaviour of a class is described by one STD. This allows only the description of sequential behaviour in the external behaviour. In this case more than one STD is used to describe the external behaviour. This allows a parallel description in the external behaviour. Two STD-types can be distinguished to describe the external behaviour. There is one STD type called Main-STD and some STD's which are dependent of this Main-STD, these STD-types are called Dep-STD's. In this case however there is only one Dep-STD (actually there are as many Dep-STD's as there are requests, but the external behaviours of these Dep-STD's are all similar). The dependencies between the Main-STD and the Dep-STD have to be expressed in the external behaviour. This can be done by giving the export-operation of the Dep-STD, that makes clear the dependency between the Main-Std and the Dep-STD, the same name as the export-operation of the Main-STD preceded by the prefix *dep* (see also [3]).



w.r.t WODAN this is subprocess s-111 and its state space is trap t-111

Figure 8. MainCAB (solution 3): Main-STD of the external behaviour

Every time a change is requested the external behaviour of DepCAB will be activated. For that reason the first transition of the external behaviour of DepCAB is labeled with *dep_request_for_change*. Simultaneous requests activate their own external behaviour of DepCAB, so for every request a new meeting will be prepared and opened. To that aim the operation *request_for_change* has been parametrized with a parameter *request-id*. So there are as many operations *request_for_change*, states *change requested* and instances of DepCAB as there are

requests. As every instance of DepCAB uses the same set of export-operations, these export-operations also have to be parametrized (with the parameter *request-id*), otherwise two or more instances of DepCAB could not operate concurrently when they want to use the same operation at the same time. Note that this parameter *request-id* has not been indicated explicitly in Figure 8 and Figure 9.

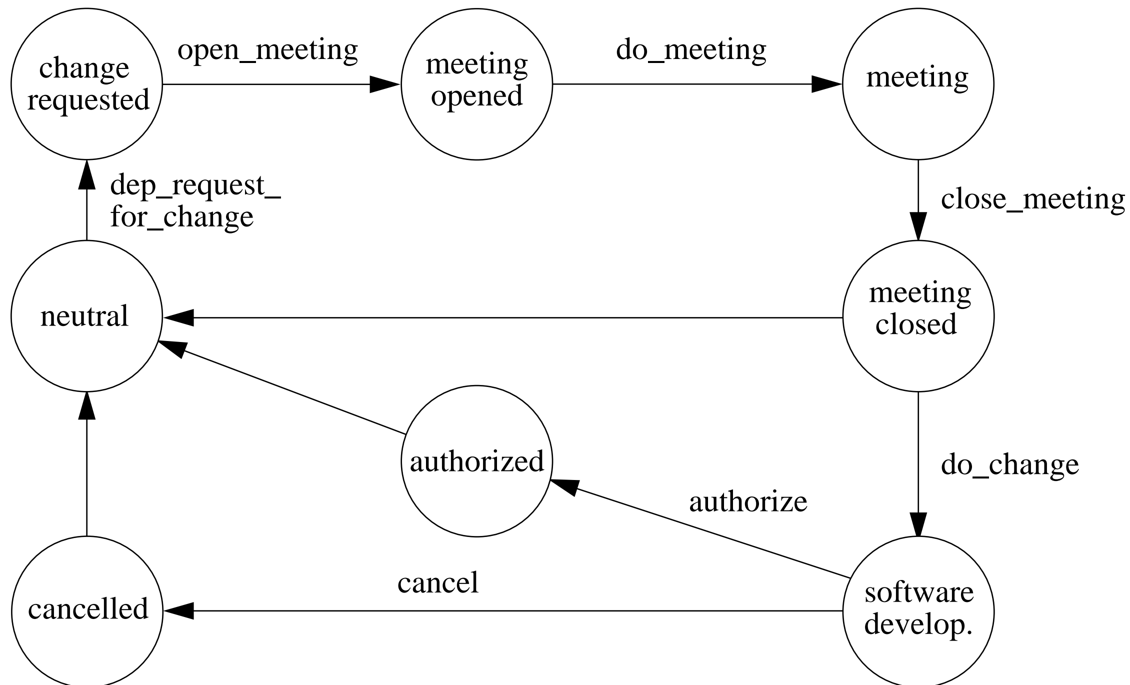


Figure 9. DepCAB (solution 3): Dep-STD of the external behaviour

The class UserRepresentative contains only export-operations relevant to Change Management. These operations are all inherited from the class CABMember. The internal behaviours of other export-operations are less relevant for our discussion, so they will not be shown in the external behaviour of the class UserRepresentative.

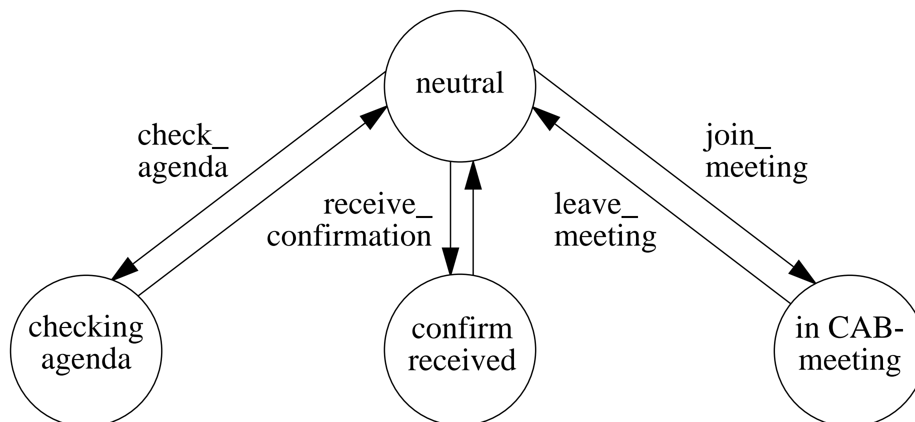


Figure 10. UserRepresentative: STD of the external behaviour

In addition the external behaviour of the class ProjectManager as well as the external behaviour of the class DesignEngineer is changed, because new operations have been added in comparison with the original example (ISPW-6). These operations are *check_agenda*, *join_meeting*, *leave_meeting* and *receive_confirmation*. They are inherited, via the class Engineer, from the class CABMember. Figure 11 shows the changed external behaviour of ProjectManager. The changed external behaviour of the class DesignEnigineer is given in Figure 12.

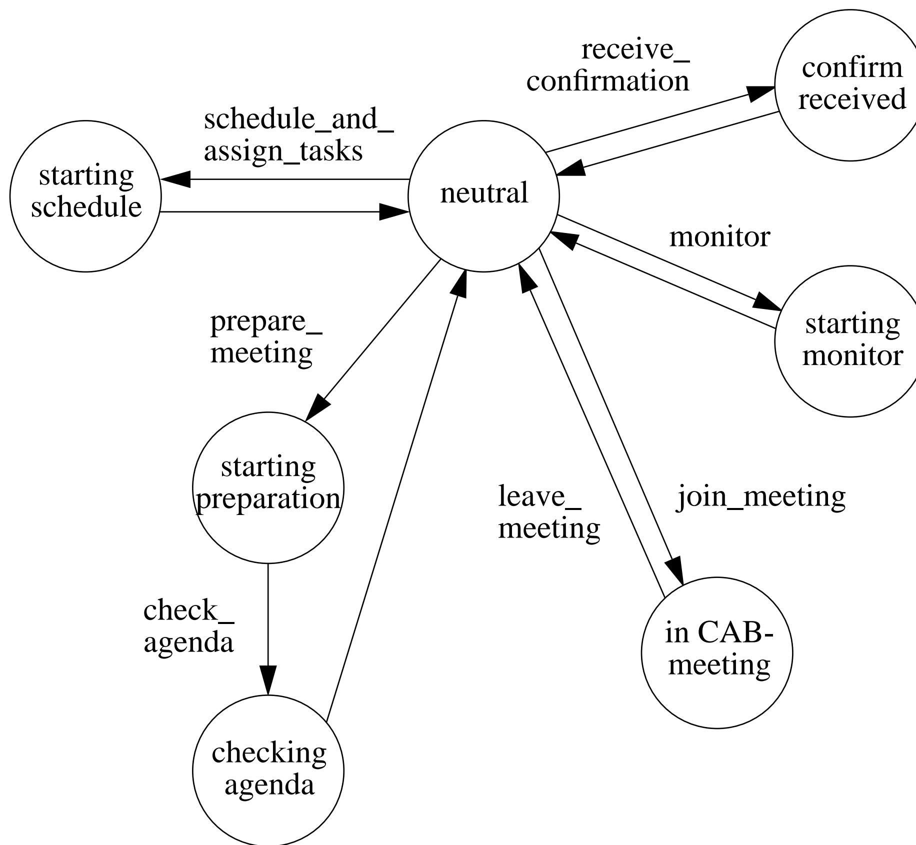


Figure 11. ProjectManager: STD of the external behaviour

w.r.t. WODAN this is subprocess s-113 and its state space is trap t-113

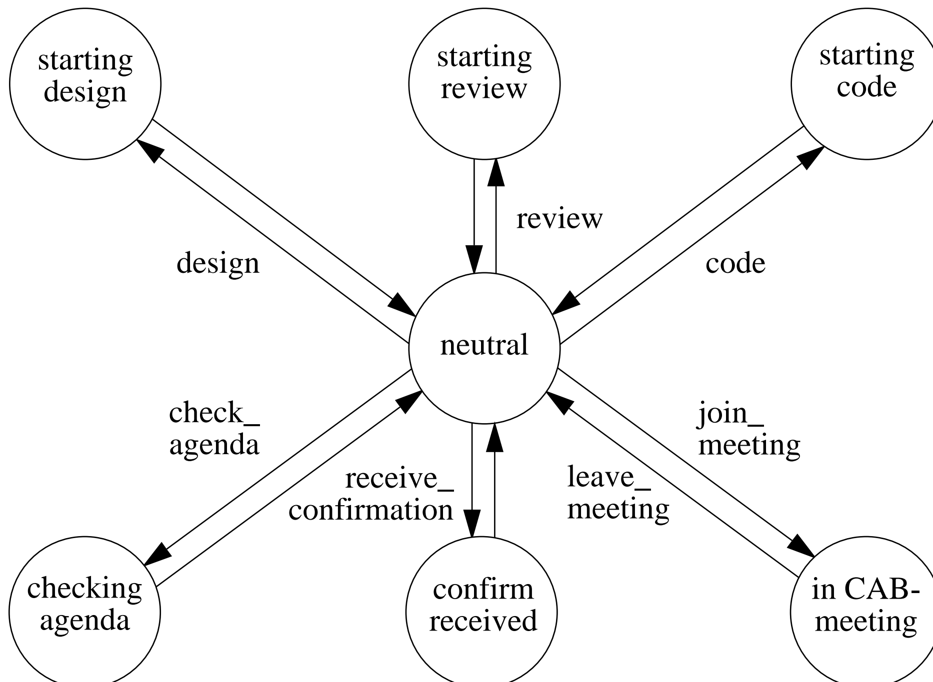


Figure 12. DesignEngineer: STD of the external behaviour

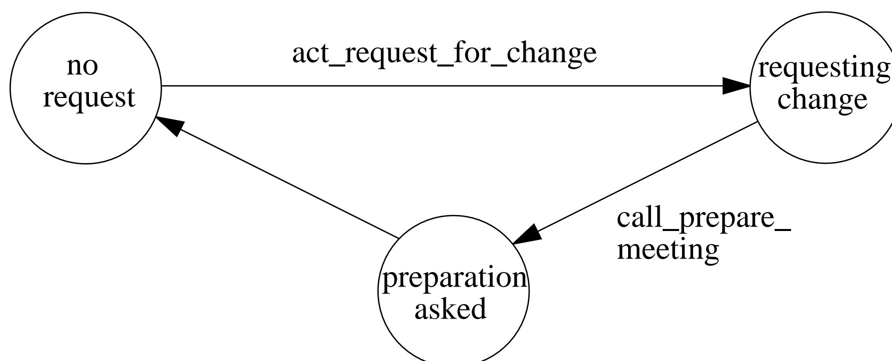
Note that only the ProjectManager can decide which tasks a DesignEngineer has to perform.

The order, in which these tasks have to be performed, is not controlled by the ProjectManager. As soon as an operation has been activated, it will run on the background and a new operation can be activated. If one wants a DesignEngineer to be able to switch between its tasks explicitly, one has to extend the model. Such an extended model is given in [3, figure 3.4.4 or figure 3.4.5].

The class QualityAsurranceEngineer also inherits the export-operations of the class CABMember. Therefore its external behaviour has to be changed too. As the class QAEngineer is not that important for our discussion, its changed external behaviour will not be given here. The behaviour can be remodelled the same way the external behaviour of the class DesignEngineer has been remodelled.

3.3.2 Designing the internal behaviours of the export-operations

After specifying the external behaviours of the classes, the internal behaviours of the operations can be specified. The conventions used to specify the internal behaviours of the operations are the same as the conventions used in [1]. Two different types of operations can occur within an internal behaviour specification. First of all, imported operations can be used. They are preceeded by the prefix *call*. The second type of operations are the internal operations within the internal behaviour and will not be worked out further. The internal operation having the prefix *act* (short for *activate*) reflects the fact that the internal behaviour is activated, which not means it is actually going.
The export-operations of the class CAB are *request_for_change* (called by a UserRepresentative or perhaps the Configuration Control Board (CCB)), *cancel*, *authorize* (both called by the CCB), *open_meeting*, *do_meeting*, *close_meeting* (all three called by the class ProjectManager from within *prepare_meeting*) and *do_change* (called by the class CAB itself from within *do_meeting*). The internal behaviours of *cancel* and *authorize* will not be given here as they are irrelevant to the problem. Also the model would grow unnecessarily big.
The internal behaviours of *request_for_change*, *open_meeting*, *do_meeting*, *close_meeting* and *do_change* are shown in figures 13, 14, 15, 16 and 17 respectively. As mentioned before these operations have been parametrized implicitly with the parameter *request-id*.



w.r.t. WODAN this is subprocess s-114

Figure 13. int-request_for_change

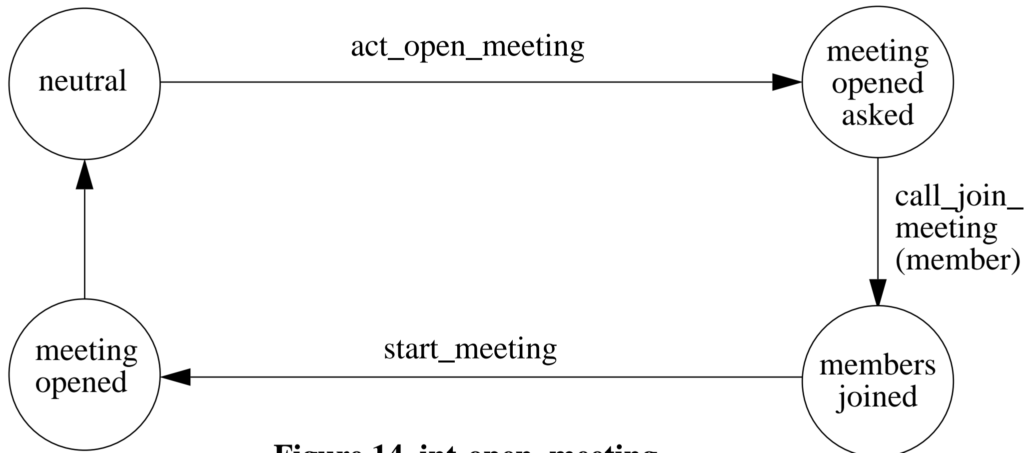
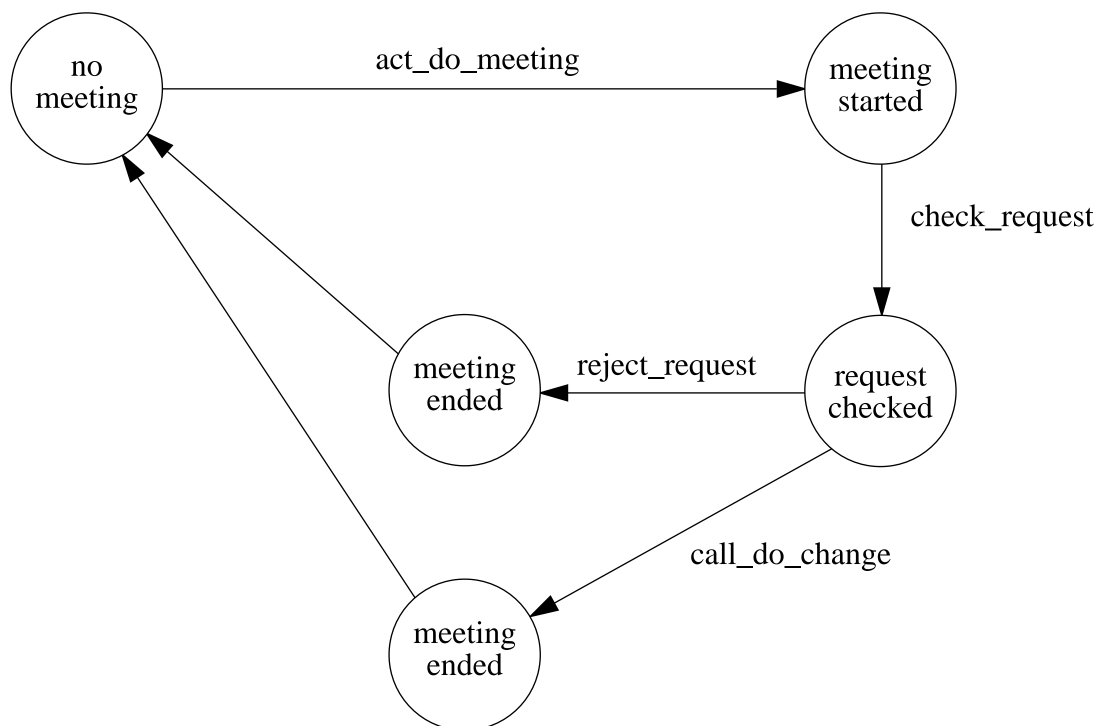


Figure 14. int-open_meeting

Note that the state *members joined* is a simplified representation of an (sub)STD in which all members of the board are called in some order. In order to remind one that some details have been omitted, the parameter in *call_join_meeting* has been indicated explicitly. The exact representation of the (sub)STD does not matter here, more information can be found in Appendix A.



w.r.t. WODAN this is subprocess s-116 and the state space is trap t-116

Figure 15. int-do_meeting

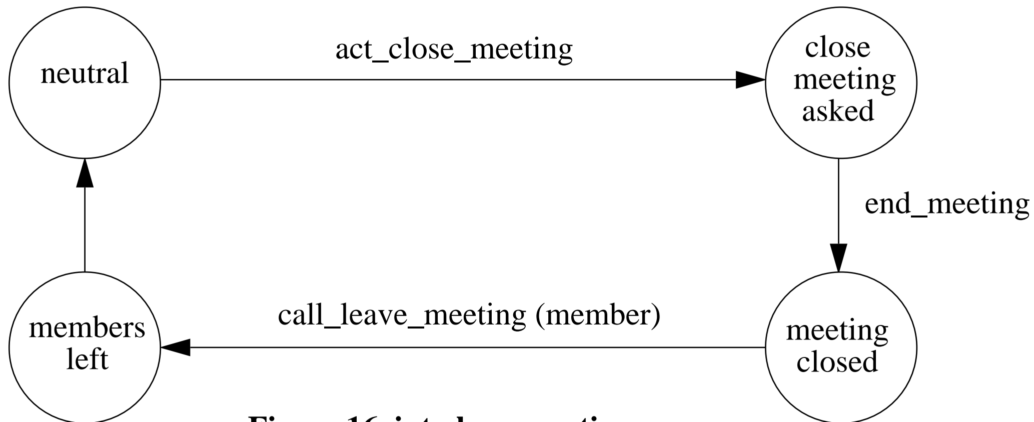


Figure 16. int-close_meeting

Note that the state *members left* is a simplified representation of an (sub)STD in which all members of the board are called in some order. In order to remind one that some details have been omitted, the parameter in *call_leave_meeting* has been indicated explicitly. The exact representation of the (sub)STD does not matter here, more information can be found in Appendix A.

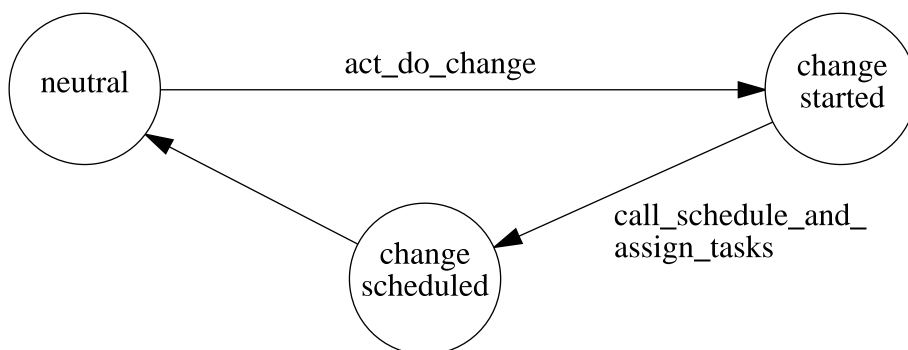


Figure 17. int-do_change

Note that *call_schedule_and_assign_task*s implicitly has been parametrized with the parameter *doc_name* (see also [1]).

The export-operations of the class CABMember are *join_meeting*, *leave_meeting*, *check_agenda* and *receive_confirmation* (shown in figures 18, 19 and 20 respectively). As mentioned before these operations are all inherited by the classes ProjectManager, DesignEngineer, QAEngineer (all through Engineer) and UserRepresentative.

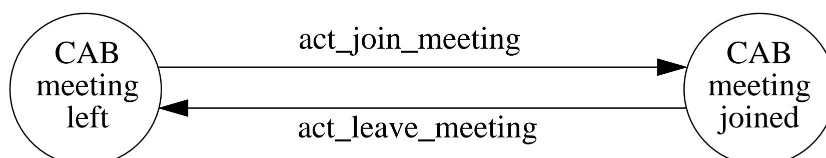


Figure 18. int-join/leave_meeting

Note that the internal behaviours originally corresponding to the operations *join_meeting* and *leave_meeting* have been merged into one STD. The reason to do so is that the operations have a very strong influence on each other. The internal behaviours originally corresponding to the operations are exactly opposite to each other, i.e. activation of one operation implies that the other operation has to return to its neutral state. This property has made it possible to merge the operations.

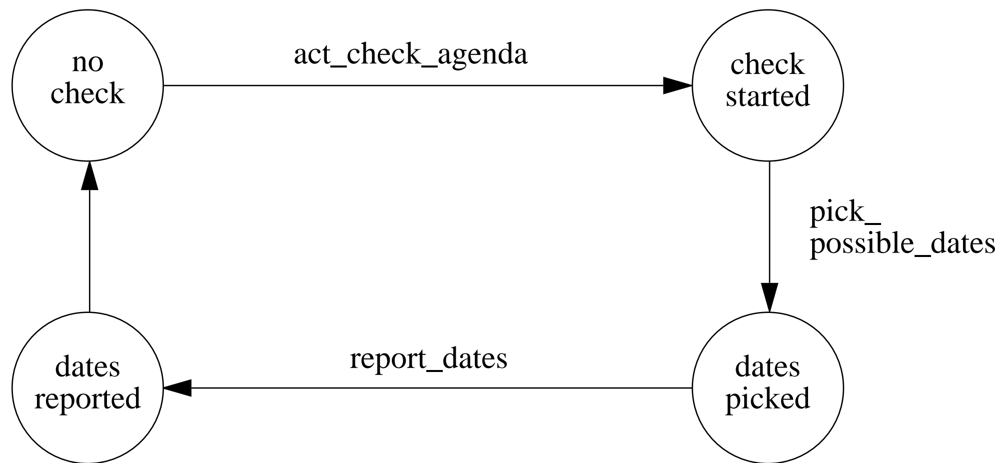


Figure 19. int-check_agenda

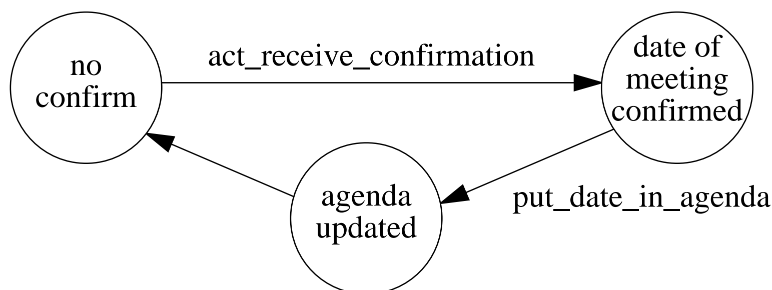
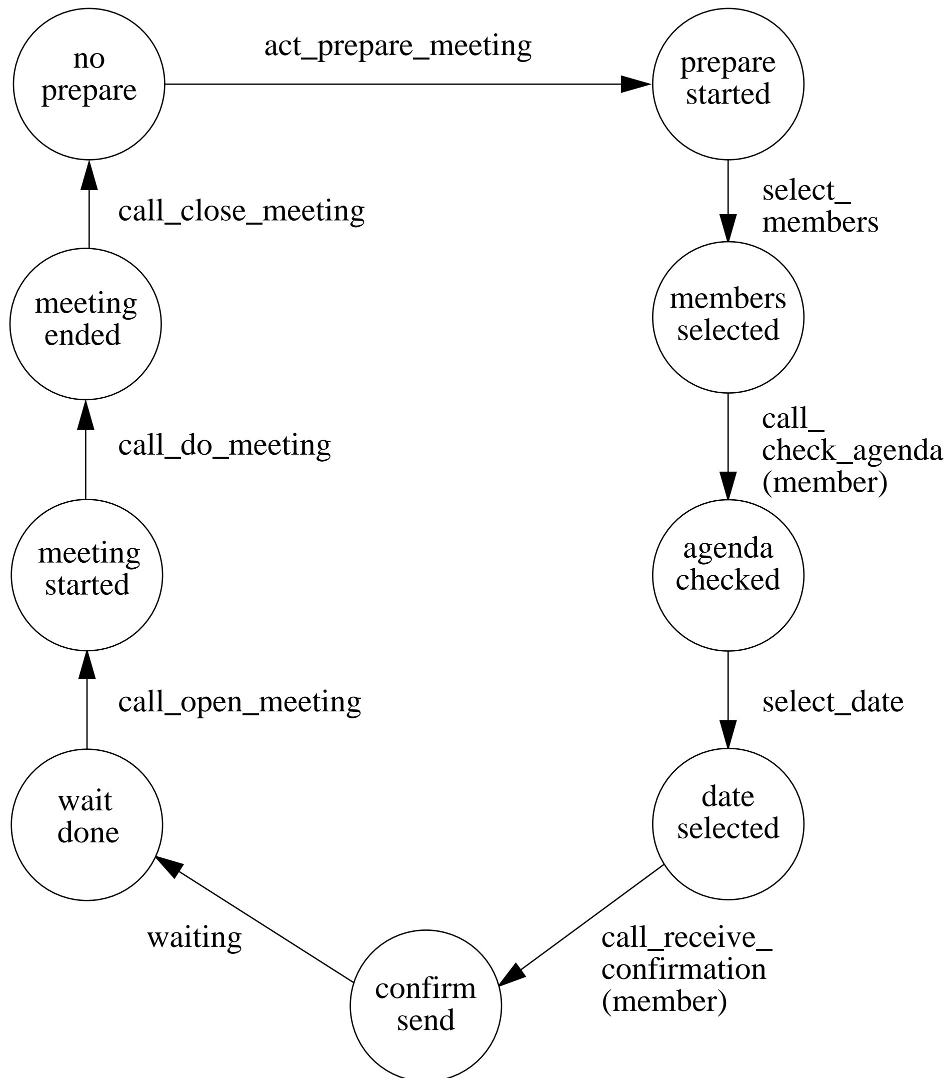


Figure 20. int-receive_confirmation

The export-operations of the class ProjectManager are *schedule_and_assign_tasks* (called by the class CAB from within *do_change*), *monitor* (called by the class ProjectManager itself from within *schedule_and_assign_tasks*) and *prepare_meeting* (called by the class CAB from within *change_for_request*). Also the class ProjectManager inherites the operations *join_meeting*, *leave_meeting*, *check_agenda* and *receive_confirmation* from the class CABMember. The STD's of the internal behaviours of *schedule_and_assign_tasks* and *monitor* can be found in [1]. As they are not very important for our discussion, we simply omit them here. It is from the internal behaviour of *prepare_meeting* (see Figure 21) that *check_agenda*, *receive_confirmation* and also *open_*, *do_* and *close_meeting* are called. The last three operations have been parametrized with the parameter *request-id*. In order to pass this parameter through, *int-prepare_meeting* has been parametrized too. Another reason to do so is that one has to be able to prepare more than one meeting at a time.



w.r.t. WODAN this is subprocess s-115 and the state space is trap t-115

Figure 21. int-prepare_meeting

The result of *call_check_agenda* is a list of possible dates and times. We simply assume that always a suitable date and time can be selected. Note that the state *agenda checked* is a simplified representation of an (sub)STD in which all members of the board are called in some order. In order to remind one that some details have been omitted, the parameter in *call_check_agenda* has been indicated explicitly. The exact representation of the (sub)STD does not matter here, more information can be found in Appendix A. The same holds for the state *confirm send* .

The export-operations of the class DesigEngineer are *design*, *code* and *review* (all three called by the class ProjectManager from within *schedule_and_assign_tasks*). The STD's of the internal behaviours of these operations can be found in [1]. Also the class DesignEngineer inherites the operations *join_meeting*, *leave_meeting*, *receive_confirmation* and *check_agenda* from the class CABMember.
The export-operations of the class UserRepresentative relevant to Change Management are *join_meeting*, *leave_meeting*, *receive_confirmation* and *check_agenda* (inherited from the class CABMember). The internal behaviours of other export-operations are less relevant for our discussion. So they will be omitted.

3.3.3 Adding Paradigm to model the communication

After the specification of the external and internal behaviours of the classes and operations, the communication between these behaviours has to be modelled. This communication is presented in five parts. Before it is presented, the standard way of modelling communication in Paradigm will be discussed.

The external behaviours act as the manager processes and the internal behaviours of the export-operations act as the employee processes. A manager process can have two kinds of employee processes: the internal behaviours of export-operations being called (called operations, they belong to the same class acting as the manager process) and the internal behaviours of operations performing these calls (calling operations). In Paradigm in the case of called operations the internal behaviour is usually split into two subprocesses. Both subprocesses contain all states. One subprocess has a small trap, which contains only the state preceding the transition labeled with *act_name_of_operation* (the neutral state). The other subprocess has a large trap containing all other states in order to enable the manager to continue as soon as possible to go to its next state. In both subprocesses the transitions coming out of the traps have of course been removed. Also in the case of calling operations the internal behaviour is usually split into two subprocesses. Again both subprocesses contain all states. One subprocess has a small trap containing only the state preceded by the transition labeled with *call_name_of_operation* and *name_of_operation* being an operation of the class acting as manager. The other subprocess has a large trap, which contains all other states. Again of course in both subprocesses the transitions coming out of the traps have been removed. Note that these standards are the same as the Socca conventions used in [1]. Whenever the model deviates from these standards, it will be mentioned and a reason will be given.

When the external behaviour of a class is described by more than one STD, the external behaviour described by the Main-STD will be the manager of the external behaviours described by the Dep-STD's (see also [3]). The external behaviour of a Dep-STD will also be split into two so-called manager subprocesses. They both contain all states. One manager subprocess has a small manager trap, containing the (neutral) state preceding the transition labeled by *dep_name_of_operation.* The other manager subprocess has a large manager trap, which contains all other states. In both manager subprocesses the transitions coming out of the manager traps have been removed. This is very similar to the standards described above to model the communication between external behaviours and internal behaviours of called operations. Note that the external behaviour of the MainCAB can also be the manager of some internal behaviours. Also the external behaviours of the DepSTD's can be the manager of some internal behaviours.

Let us now return to our model. The first part of the communication specification presents the communication between the manager process MainCAB and its employee processes *int-request_for_change* and DepCAB. *Int-request_for_change* is the internal behaviour of an operation of MainCAB itself, DepCAB is the external behaviour of a part of the class CAB.

MainCAB (Figure 24) starts in its state *neutral.* When a change is requested, the manager waits for *int-request_for_change* to be trapped in its trap t-1, which means that a possible previous request has been dealt with, and DepCAB to be trapped in mt-1. When these traps have been entered, CAB makes the transition to the state *change requested* and prescribes subprocess s-2 to *int-request_for_change*, thereby making it possible to start the preparations, and manager subprocess MS-2 to DepCAB, so that the request can be discussed and handled if necessary. If *int-request_for_change* has entered its trap t-2, which means that the preparations have been started, if moreover DepCAB has entered its manager trap mt-2, which means the request is being discussed or handled, then MainCAB will return to its state *neutral*.

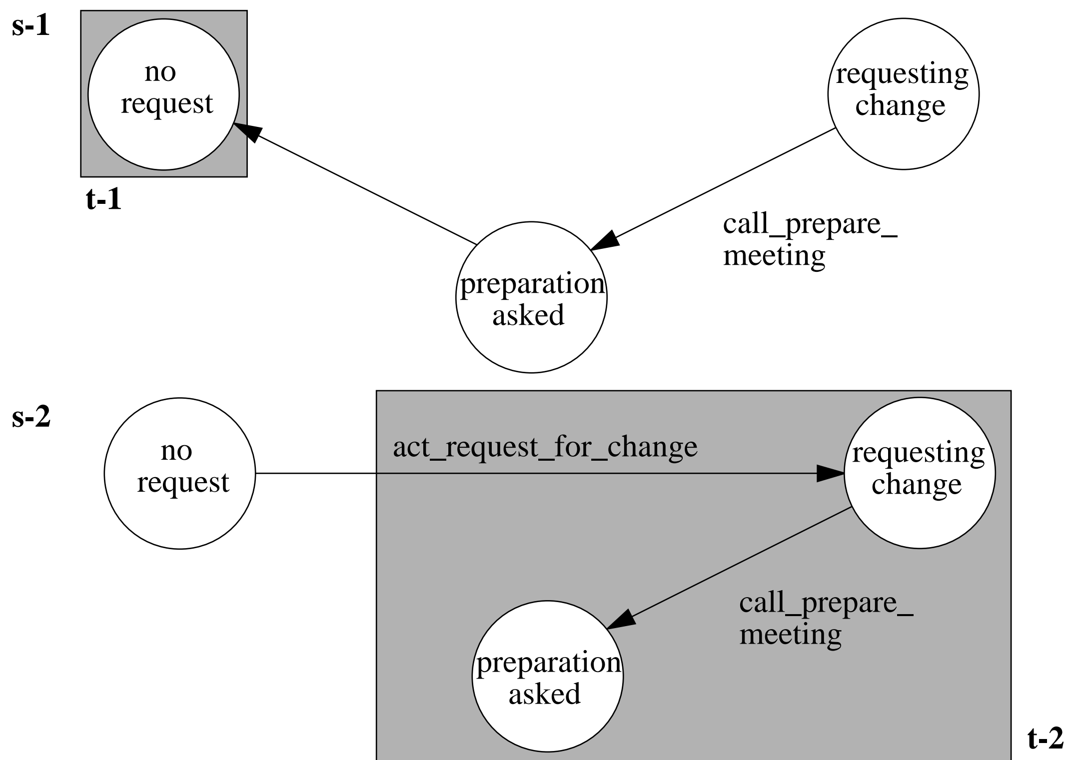


Figure 22. int-request_for_change’s subprocesses and traps w.r.t. MainCAB

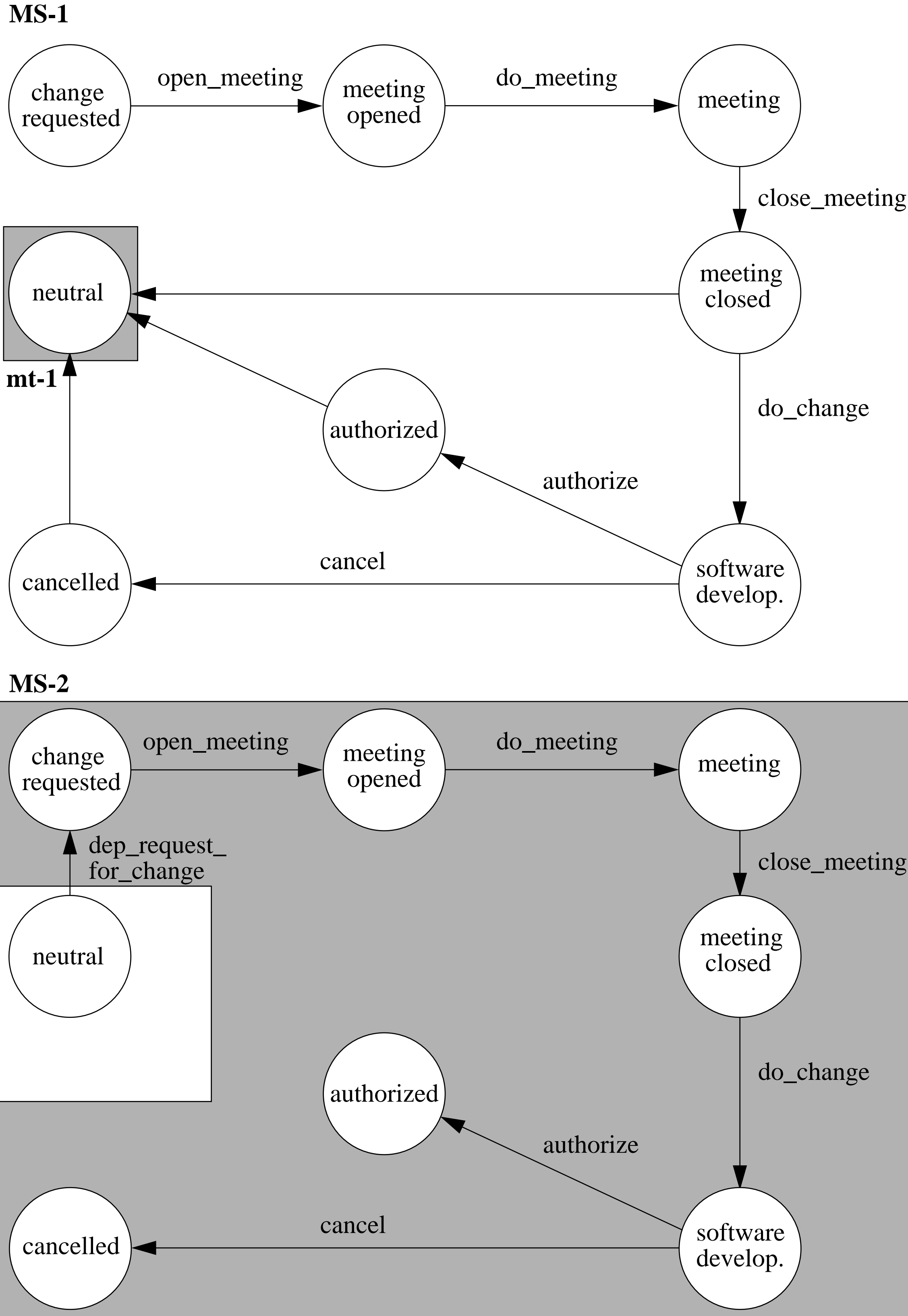


Figure 23. DepCAB's subprocesses and traps w.r.t. MainCAB

Figure 24 shows MainCAB as manager of DepCAB and *int-request_for_change* (a called operation).

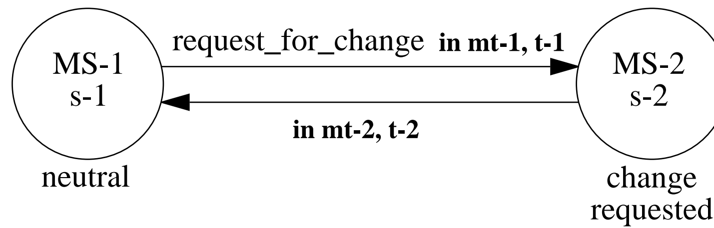


Figure 24. MainCAB: viewed as manager of 2 employees

Remember that the operation *request_for_change* has been parametrized with the parameter *request-id* (not explicitly indicated in Figure 24). So there are as many operations *request_for_change*, states *change requested* and external behaviours of DepCAB as there are requests. When MainCAB is in one of its states *change requested*, the manager subprocess MS-2 will be prescibed to only one external behaviour of DepCAB. The manager subprocess MS-1 will remain prescribed to all other external behaviours of DepCAB.

The second part of the communication specification presents the communication between the manager process DepCAB and its employee processes *int-open_meeting*, *int-do_meeting*, *int-close_meeting*, *int-do_change* and *int-prepare_meeting*. The export-operations *cancel* and *authorize* have not been modelled and also the calling of these operations has been left out, so the communication between DepCAB and these operations will not be specified. *Int-open_meeting*, *int-do_meeting*, *int-close_meeting* and *int-do_change* all are internal behaviours of operations of DepCAB itself. *Int-prepare_meeting* is the internal behaviour of an operation that calls operations of DepCAB.
Every DepCAB (Figure 30) starts in its state *neutral*. Whenever MainCAB goes to one of its states *change requested*, one particular DepCAB goes to its state *change requested* too, as the dependency between MainCAB and DepCAB is modelled that way. If *int-open_meeting* has entered its trap t-3, which means that the meeting can be opened, if moreover *int-prepare_meeting* has entered its trap t-16, which means that the preparations have come to the point the meeting actually can be asked to start, then *open_meeting* is performed and DepCAB will transit from its state *change requested* to its state *meeting opened*. There DepCAB will prescribe the subprocesses s-4 and s-17 to *int-open_meeting* and *int-prepare_meeting* respectively. The members will be called to join the meeting and after that the meeting will be started. If *int-prepare_meeting* has entered its trap t-17, if also *int-open_meeting* has entered its trap t-4, which means the meeting has been opened, and *int-do_meeting* is in its trap t-9, then DepCAB will go to its state *meeting*. There subprocesses s-3, s-5 and s-18 are going to be prescribed to *int-open_meeting*, *int-do_meeting* and *int-prepare_meeting* respectively. The meeting can take place and the request will be discussed now. If *int-do_meeting* has entered its trap t-5 or t-6, which means the meeting has ended and a result with respect to the request has been established, if also *int-close_meeting* has entered its trap t-10, which means the meeting can be closed, if furthermore *int-prepare_meeting* has entered its trap t-18, then *close_meeting* can and will be performed and DepCAB transits to its state *meeting closed*. There the subprocesses s-11 and s-16 will be prescribed to *int-close_meeting* and *int-prepare_meeting* respectively. To *int-do_meeting* subprocess s-5 is still being prescribed. So *int-do_meeting* will still be waiting in its trap t-5 or t-6. In this state of DepCAB the meeting will be ended. As soon as *int-close_meeting* enters its trap t-11, which means the meeting has been closed, DepCAB will make its next transition. In case *int-do_meeting* is waiting in its trap t-5, the request had been rejected and DepCAB will go back to its state *neutral.* In case *int-do_meeting* is waiting in its trap t-6, the request had been accepted, and when *int-do_change* is in its trap t-12, DepCAB will

transit to its state *software developing*. In both state *neutral* and state *software developing* subprocess s-9 is prescribed to *int-do_meeting* and subprocess s-10 to *int-close_meeting*. Normally, as soon as the change has been established *authorize* will follow. However, sometimes the software development may be interrupted by *cancel*. In both cases *int-do_change* has to be in its trap t-13.

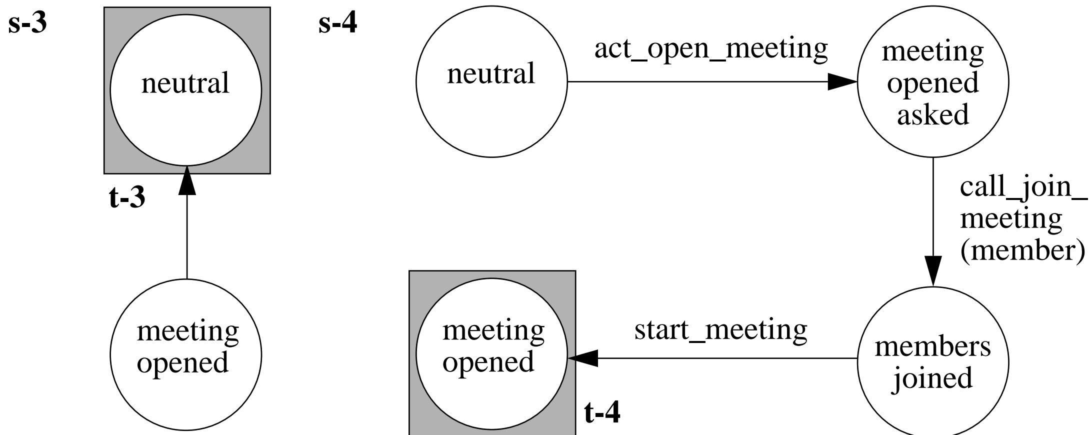


Figure 25. int-open_meeting's subprocesses and traps w.r.t. DepCAB

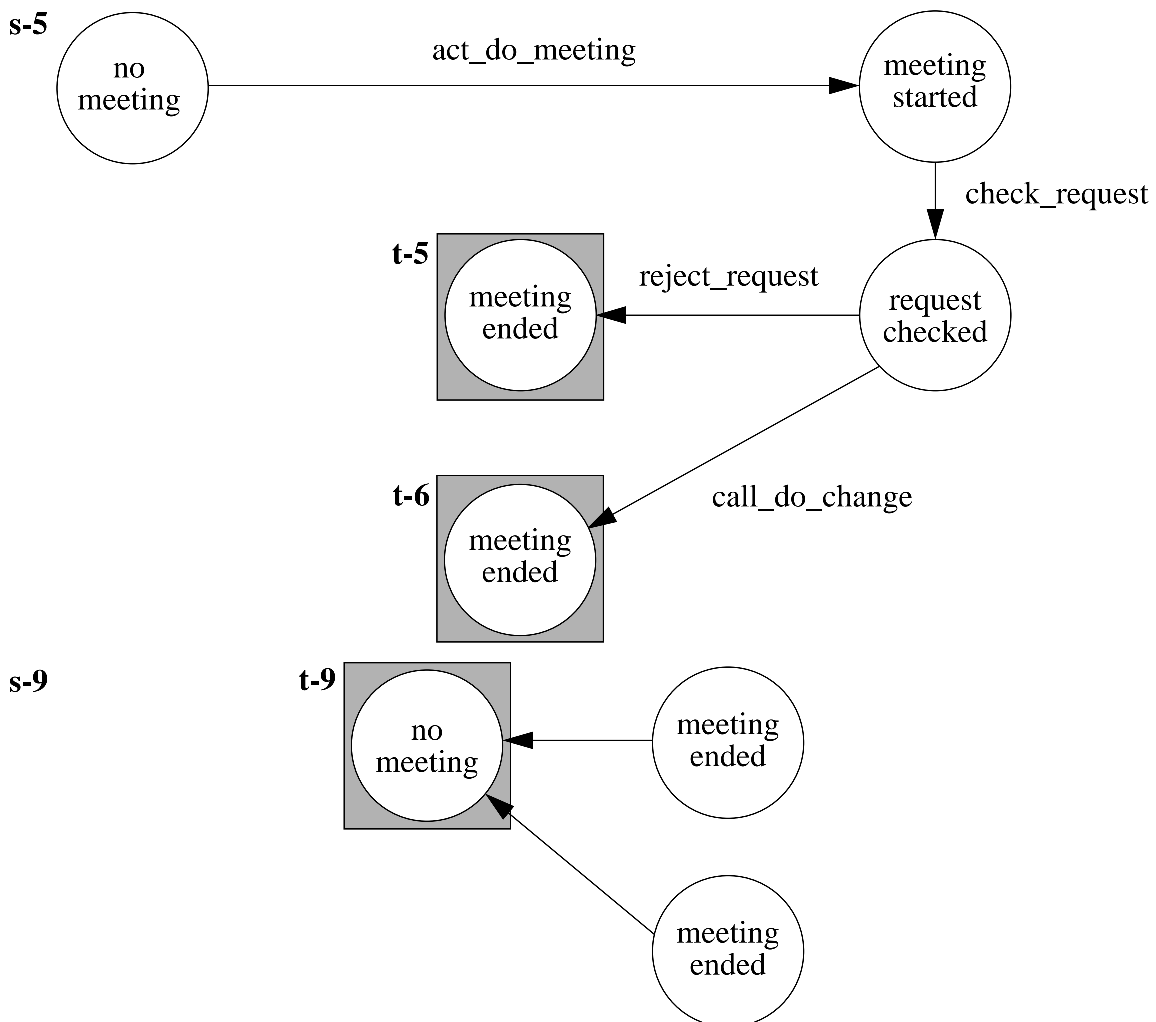


Figure 26. int-do_meeting's subprocesses and traps w.r.t. DepCAB

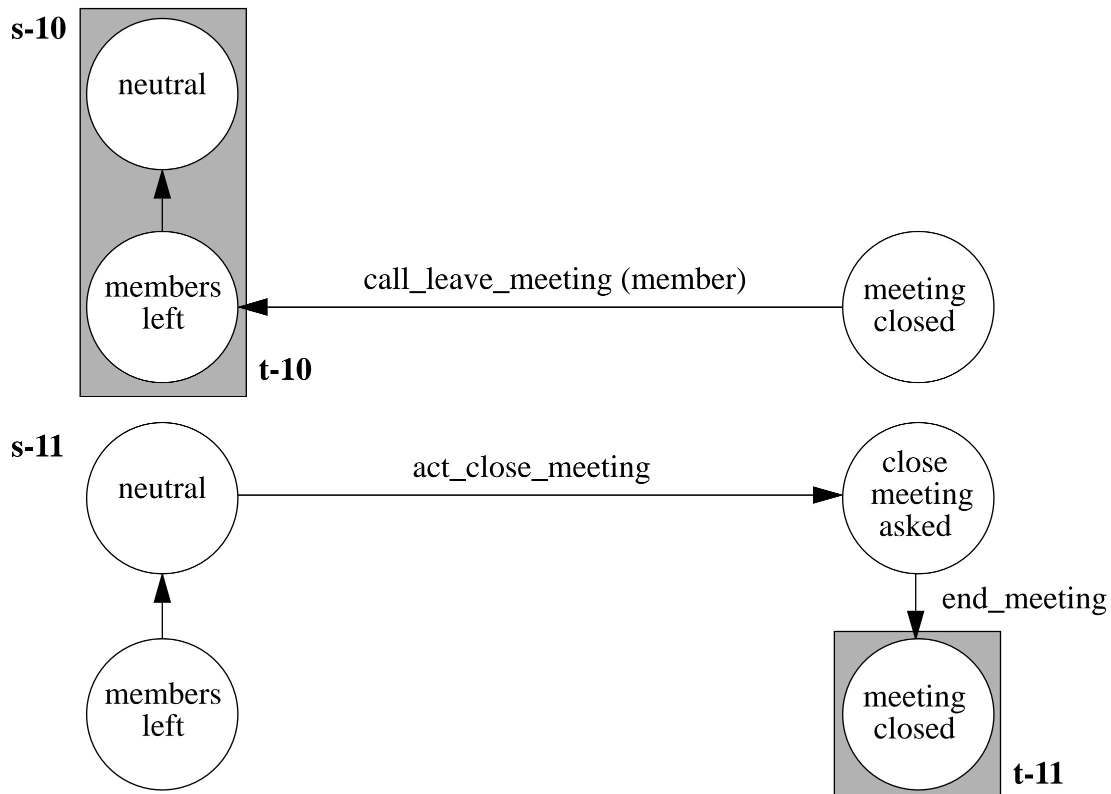


Figure 27. int-close_meeting's subprocesses and traps w.r.t DepCAB

Note that the trap-structures of *int-open_meeting*, *int-do_meeting* and *int-close_meeting* (figures 25, 26 and 27 respectively) deviate from the standards described above. As *int-open_meeting*, *int-do_meeting* and *int-close_meeting* are actually part of one big operation that has been split into three parts, these three operations cannot operate in a parallel way. In order to sequentialize these operations only small traps are used, because only then the operation is ready before the trap is entered and after that the manager will make its next move.

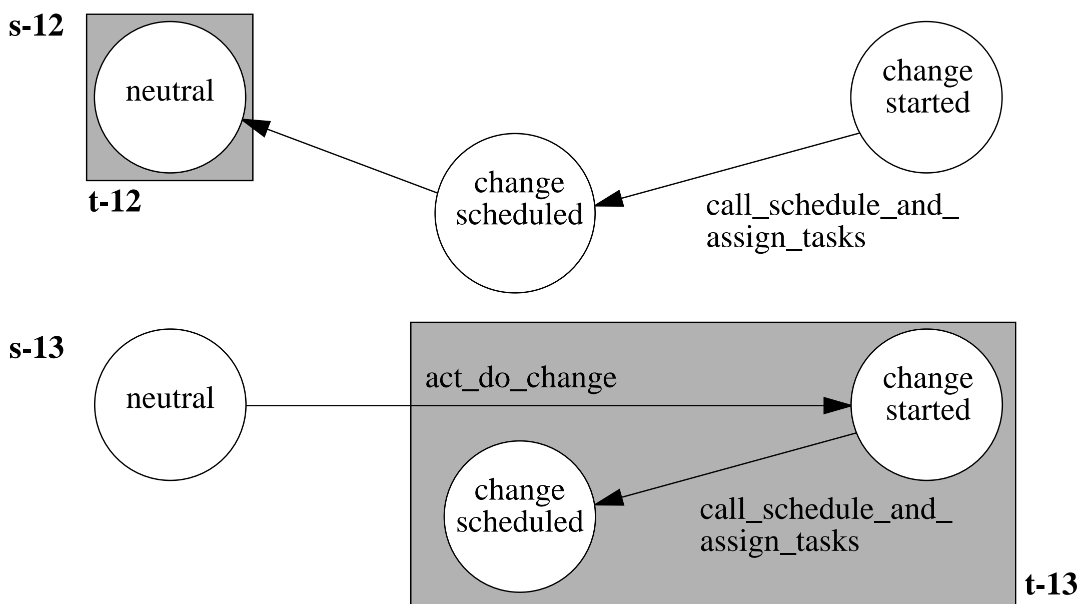


Figure 28. int-do_change's subprocesses and traps w.r.t. DepCAB

Note that the trap-structure of *int-prepare_meeting* (Figure 29) deviates from the standards de-

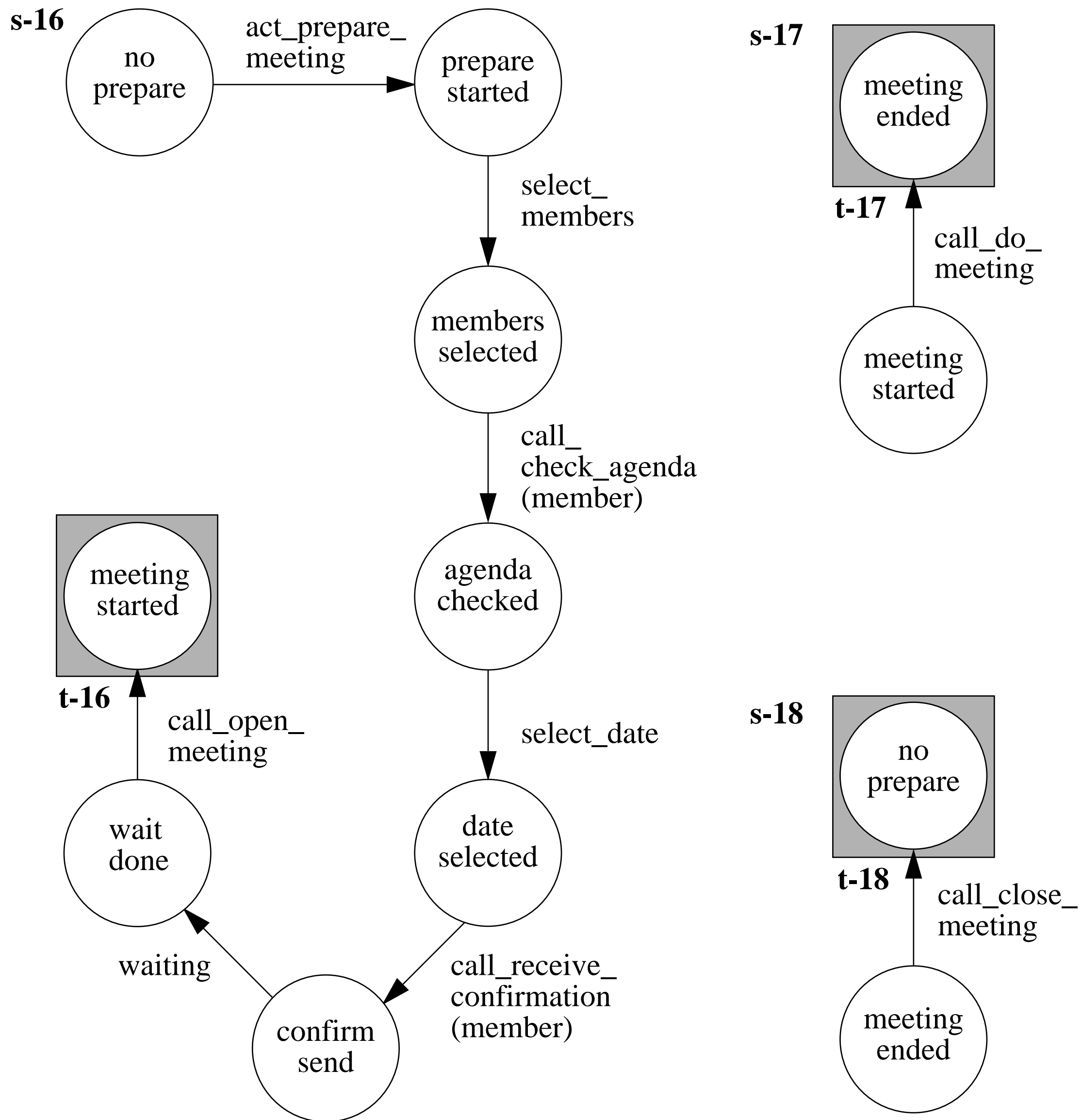


Figure 29. int-prepare_meeting's subprocesses and traps w.r.t. DepCAB

scribed before. This time there are three subprocesses. The reason to do so is that *int-prepare_meeting* calls three different operations of the DepCAB. As *prepare_meeting* is a calling operation, every subprocess hass a small trap containing one state preceded by the transition labeled with the call. For the calling itself this is in accordance with the standards.

Figure 30 shows the class DepCAB as manager of *int-open_meeting*, *int-do_meeting*, *int-*

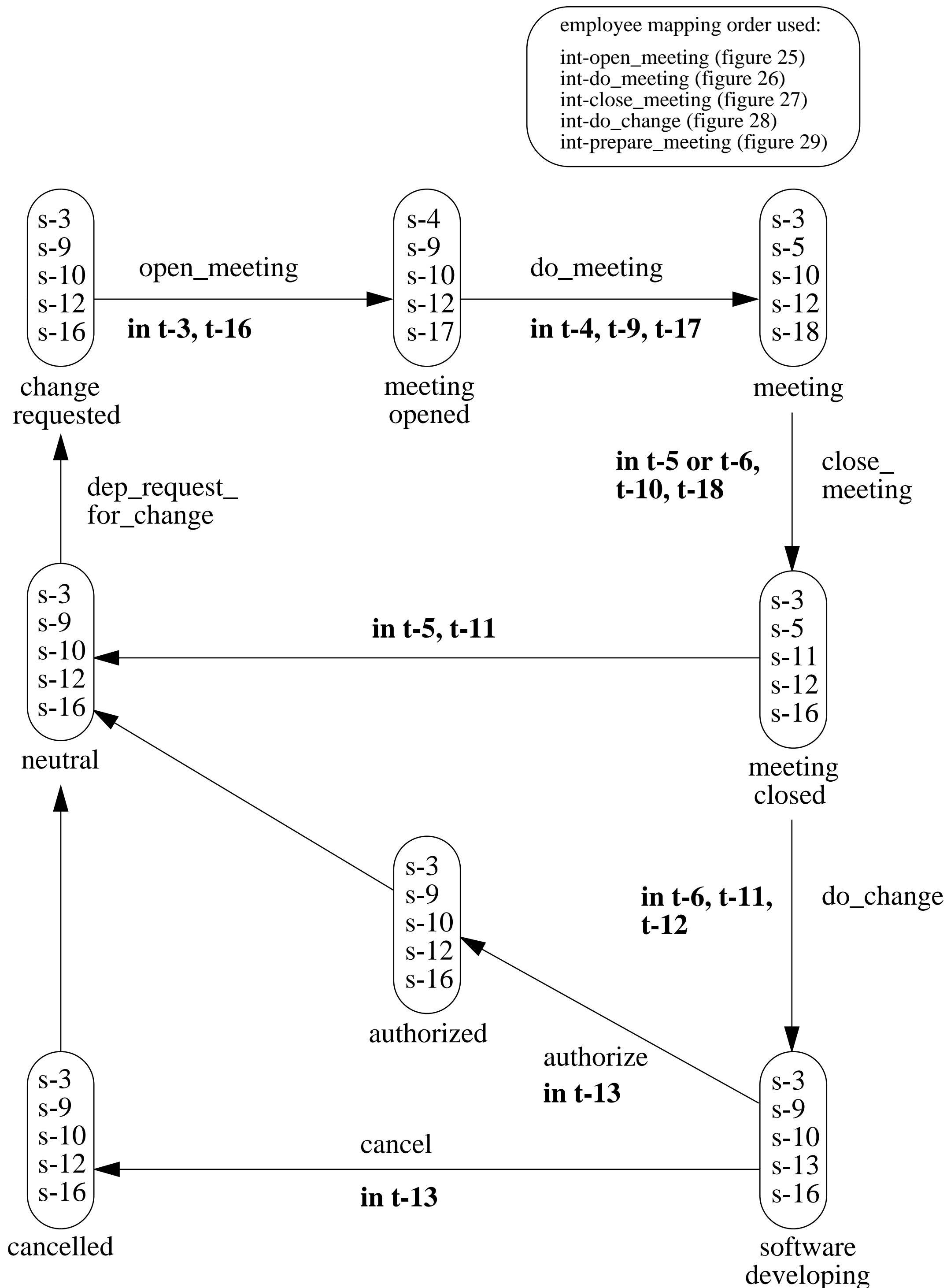


Figure 30. DepCAB: viewed as manager of 5 employees

close_meeting, *int-do_change* (called operations) and *int-prepare_meeting* (calling operation). Note that the states *neutral*, *software developing*, *cancelled* and *authorized* are equal w.r.t. each of these employees (except for *do_change*), that is the subprocesses of these employees remain unchanged in all states.

The third part of the communication specification shows the communication between the manager process ProjectManager and its employee processes *int-check_agenda*, *int-prepare_meeting*, *int-receive_confirmation*, *int-request_for_change*, *int-join_meeting*, *int-leave_meeting*, *int-open_meeting*, *int-close_meeting* and *int-do_change*. The internal behaviours of the export-operations *int-check_agenda*, *int-receive_confirmation*, *int-prepare_meeting*, *int-request_for_change*, *int-join_meeting* and *int-leave_meeting* belong to the class ProjectManager itself. *Int-open_meeting*, *int-close_meeting* and *int-do_change* are the internal behaviours of operations that call operations of the class ProjectManager. As the subprocesses and traps of *int-schedule_and_assign_tasks* and *int-monitor* are not very important for our discussion, we simply omit this part of the communication specification.
Note that the trap-structure of *int-check_agenda* (Figure 31) deviates from the standards described before. It is clear that before a meeting can be started a date has to be picked. Small traps are used in *int_check_agenda*, so that every member, including ProjectManager, will report its possible dates before returning to its neutral state. When using large traps they can return to their neutral states before they have reported their possible dates. Consequently, when a meeting is opened, members could be called to join the meeting before they have reported their dates.

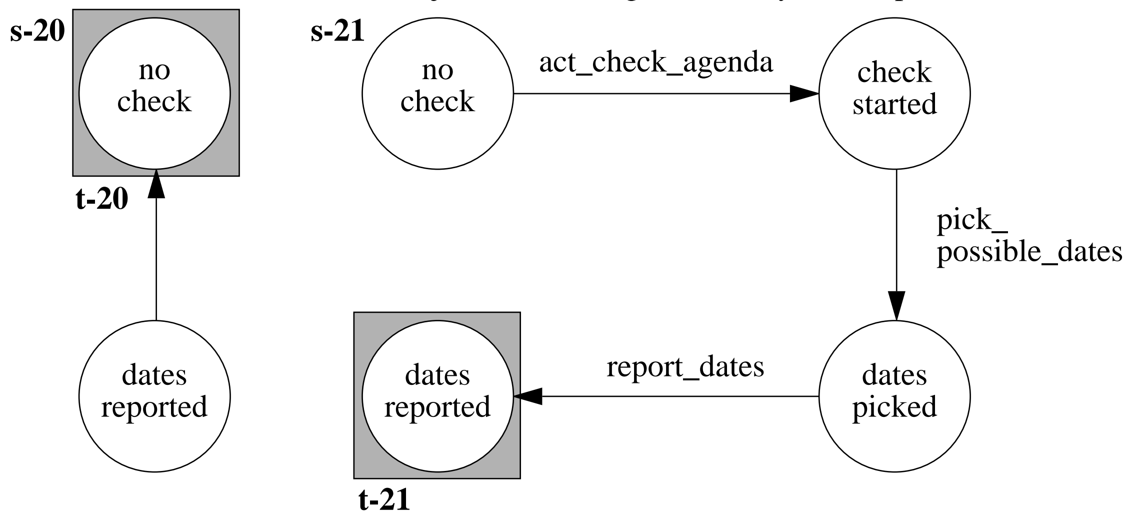


Figure 31. int-check_agenda's subprocesses and traps w.r.t. CABMember

ProjectManager starts in its state *neutral*. There subprocess s-22 has been prescribed to *int-prepare_meeting*. If this behaviour has entered its trap t-22a, which means a preparation can be started, if also *int-request_for_change* has entered its trap t-25, which means a preparation has been asked, then *prepare_meeting* will be executed and ProjectManager will go to its state *starting preparation*. There subprocess s-23 is prescribed to *int-prepare_meeting*, so that the preparations will be started, and subprocess s-26 to *int-request_for_change*, so that a new request can be made. If *int-prepare_meeting* has entered its trap t-23, which means all members of the board including ProjectManager itself have been asked to check their agenda and send a list of possible dates, and *int-check_agenda* is in its trap t-20, which means the agenda can be checked, then ProjectManager can go to its state *checking agenda*. There the subprocesses s-21 and s-24 are prescribed to *int-check_agenda* and *int-prepare_meeting* respectively. ProjectManager will check its agenda now. If *int-check_agenda* has entered its trap t-21 and *int-prepare_meeting* has entered its trap t-24, which means the own agenda has been checked, if furthermore *int-request_for_change* has entered its trap t-26, then ProjectManager can go to its state *neutral*. There ProjectManager continues to prepare the meeting or doing whatever is was doing. Note that *int-prepare_meeting* will remain in its trap t-24, therefore subprocess s-24 is still being prescribed. As soon as *int-prepare_meeting* has entered its trap t-24a and *int-receive_confirmation* has entered its trap t-40, ProjectManager can and will go to its state *confirm received*. In this state subprocess s-22 is prescribed again to *int-prepare_meeting* and subprocess s-41 to *int-*

receive_confirmation. The date that has been picked for the meeting will be put in the agenda. When *int-receive_confirmation* enters its trap t-41 and *int_prepare_meeting* enters its trap t-22, which means the confirmation has been received, ProjectManager will return to its state *neutral*. *Int-prepare_meeting* will remain in its subprocess s-22. To *int-receive_confirmation* subprocess s-40 will be prescribed again. In the neutral state ProjectManager can now start the preparations for a new meeting, schedule and assign tasks and monitor the design process.

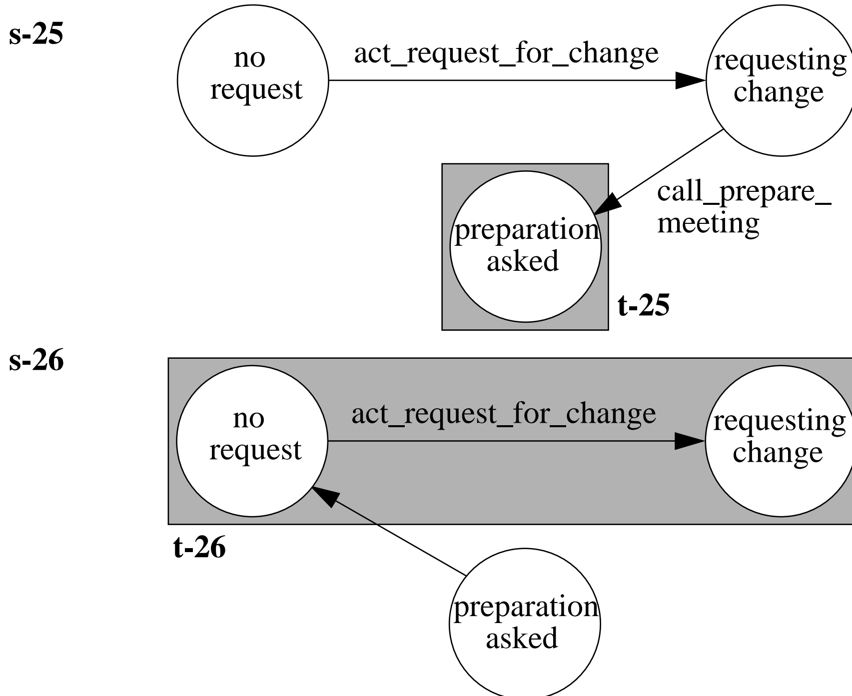


Figure 32. int-request_for_change's subprocesses and traps w.r.t. ProjectManager

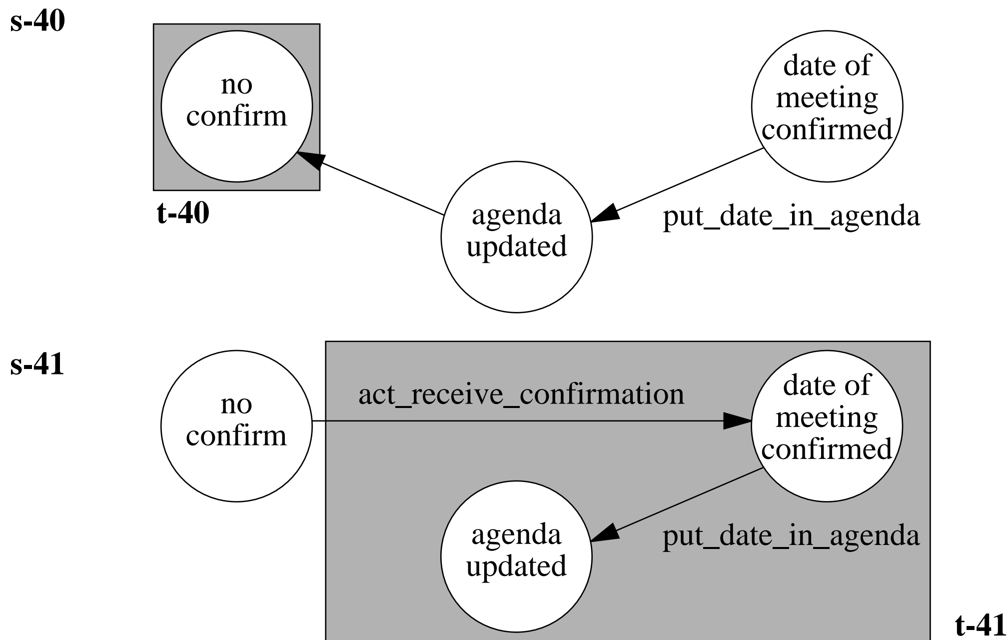


Figure 33. int-receive_confirmation's subprocesses and traps w.r.t CABMember

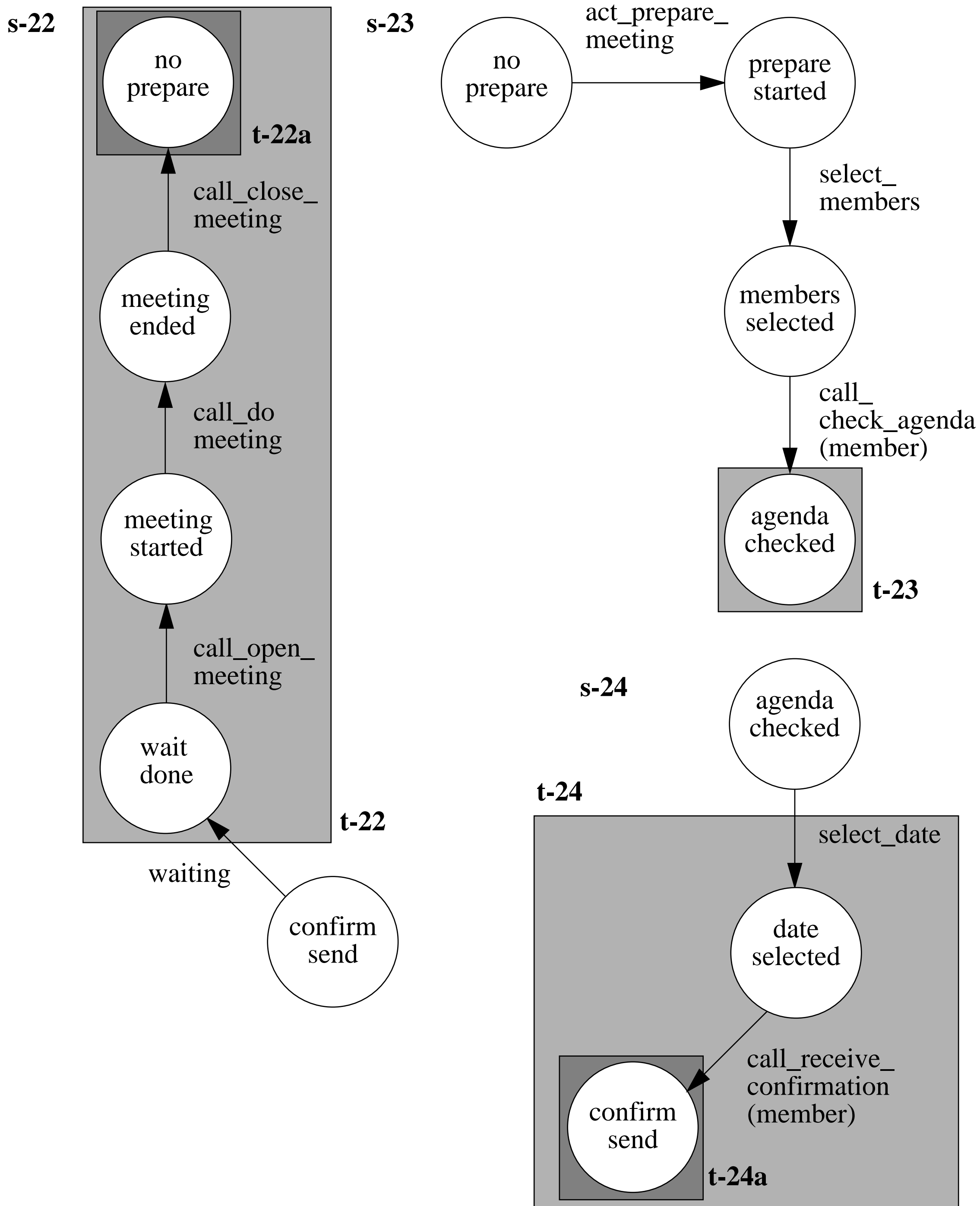


Figure 34. int-prepare_meeting's subprocesses and traps w.r.t. ProjectManager

Note that the trap-structure of *int-prepare_meeting* (Figure 34) deviates from the standards described before. *Prepare_meeting* is of course a called operation, as it belongs to the class ProjectManager itself, but it is also a calling operation as it performs a call to the operations *check_agenda* and *receive_confirmation* which belong to the same (instance of the) class as *prepare_meeting*. As it is a called operation there must be a subprocess having a small trap, which contains only the neutral state (*no prepare*), and as it is a calling operation there must be subprocesses having a small trap, which contains only the state preceded by *call_check_agenda* or *call_receive_confirmation*. Therefore there are three subprocesses containing a small trap. Two of these subprocesses also have a large trap embedding the small trap in order to allow ProjectManager to go to its next state without waiting for the small trap to be entered. In the next

state of ProjectManager the same subprocess will be prescribed. Only after the small trap has been entered this next state can be left.
Note also that the partition of subprocesses in Figure 34. overrides the partition of subprocesses in Figure 40, which means that although ProjectManager is a subclass of the class CABMember, the original partition (Figure 40) will not be inherited from the class CABMember and that *int-prepare_meeting*'s subprocesses and traps are redefined for the class Projectmanager in Figure 34.
Note also that the internal behaviour of *prepare_meeting* cannot go to the state *date selected*, even if *int-prepare_meeting* already is in its subprocess s-24 prescribed by the ProjectManager, until every other member of the board has reacted to trap t-42 and consequently has prescribed subprocess s-44 (see also Figure 40).
Also note that this operation has been parametrized implicitly with the parameter *request-id*. This implies that two or more meetings can be prepared simultaneously. ProjectManager however can prepare a new meeting, i.e. execute the next *prepare_meeting*, only when it has returned to its state *neutral*. This implies that *int-prepare_meeting* initiated by the previous request has had to pass its state *date selected* and enter its trap t-24 (see also Figure 39). If not, ProjectManager will not be in its state *neutral* and *int-prepare_meeting* initiated by the new request will have to wait in its state *no prepare*.

Now let us return to the manager process. After the actual preparations have been finished, i.e. the agendas have been checked, the date has been picked and the confirmations have been received, the meeting eventually will take place and ProjectManager will have to join it. When *int-open_meeting* is in its trap t-31, which means all members including the ProjectManager are called to join the meeting, and *int-join_meeting* is in its trap t-27, which means ProjectManager is able to join the meeting, and also *int-close_meeting* is in its trap t-34, which means a possible previous meeting in which ProjectManager was involved has been closed, *join_meeting* will be performed and ProjectManager will make the transition to its state *in cab-meeting*. If *int-close_meeting* has entered its trap t-33, which means all members including the ProjectManager are called to leave the meeting, if moreover *int-leave_meeting* has entered its trap t-28, which means the Projectmanager is able to leave the meeting, if furthermore *int-open_meeting* has entered its trap t-32, then *leave_meeting* is executed and ProjectManager will return to its state *neutral*. There ProjectManager can continue to do whatever it was doing.

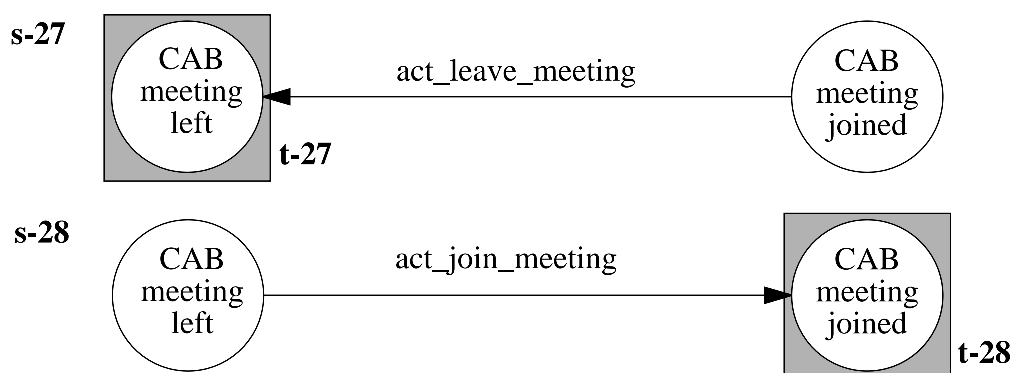


Figure 35. int-join/leave_meeting's subprocesses and traps w.r.t. CABMember

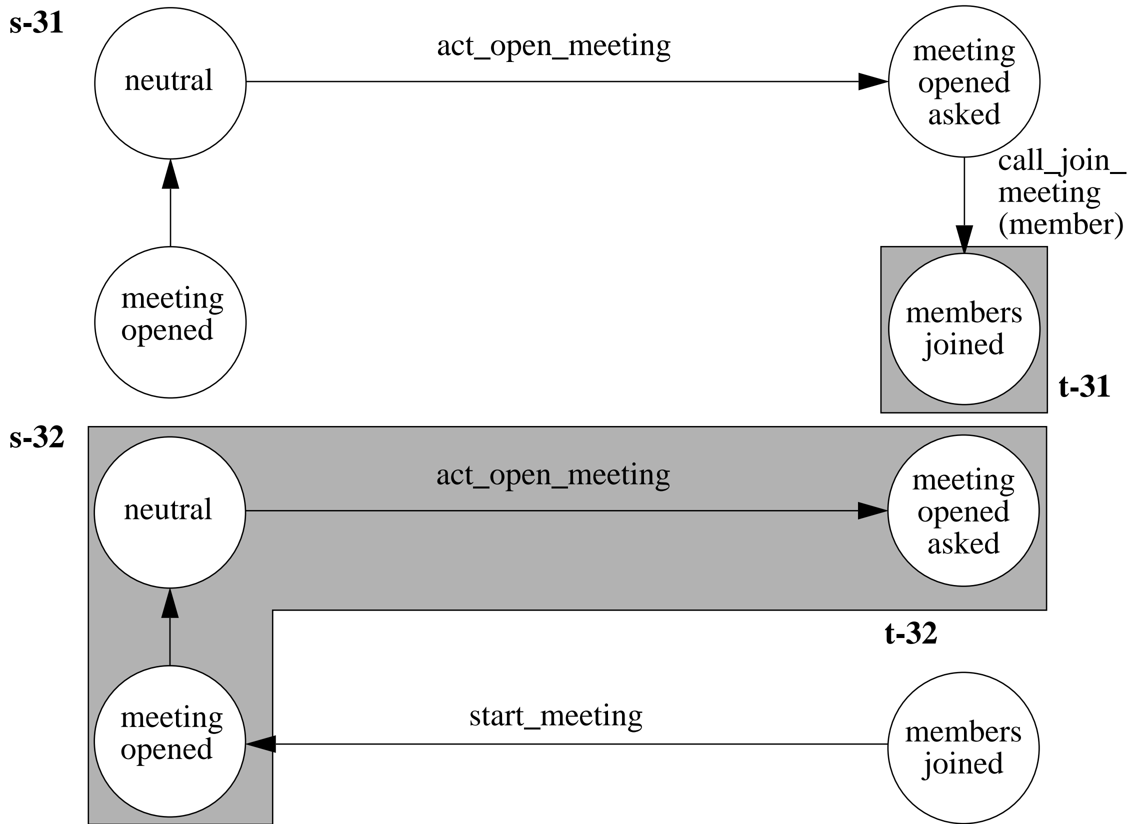


Figure 36. int-open_meeting's subprocesses and traps w.r.t. CABMember

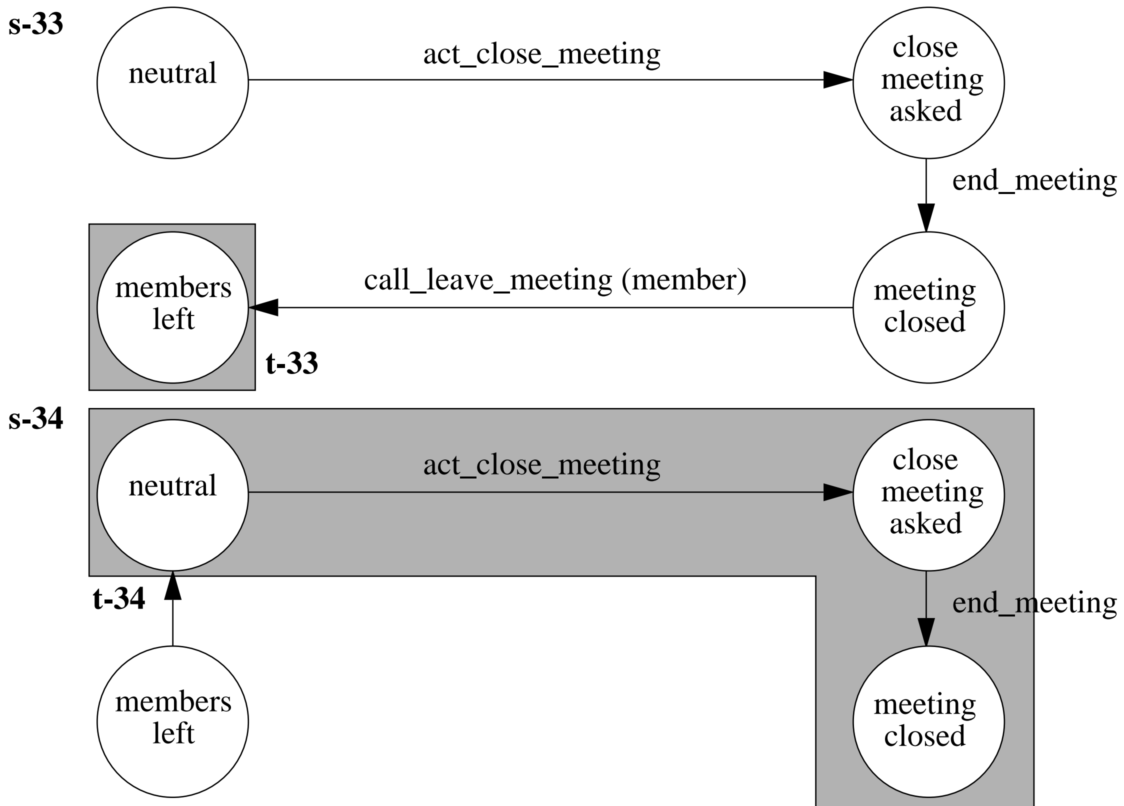


Figure 37. int-close_meeting's subprocesses and traps w.r.t. CABMember

Note that the partitions in figures 36 and 37 (and 40 too) are valid for every member of the board. These partitions are partitions of internal behaviours of operations of another class. The states in which these operations remain, are not only dependent on the subprocesses the other class prescribes, but are also dependent on the subprocesses every member prescribes. Only when all members have reacted to trap t-31 and consequently have prescribed subprocess s-32 to *int-open_meeting*, the internal behaviour of *int-open_meeting* can continue to go to its next state and enter trap t-32. Only when *int-open_meeting* has entered trap t-32, any particular member is allowed to go to its next state. So it seems that every member has to wait until all other members have prescribed subprocess s-32, before going to its next state. However this is not the case. As the state *members called* is a simplified representation of an (sub)STD, in which all members are called, the trap-structure is in fact more complicated. This implies that every member can react to trap t-32 without waiting for the other members. More information can be found in Appendix A. The same applies to *int-close_meeting*.
The partitions in figures 31, 33 and 35 are also valid for all members of the board. These partitions are partitions of internal behaviours of operations (*receive_confirmation, join_meeting*, *leave_meeting* and *check_agenda)* of the class itself, in this case ProjectManager, DesignEngineer, UserRepresentive or QAEngineer. The states in which these operations remain, are dependent on the subprocesses one particular member prescribes, as these operations do not call to other members, in contradistinction to the operations described above.

Now let us return to the manager process again. One detail still has to be discussed. In state *neutral* ProjectManager can also execute the operations *schedule_and_assign_tasks* and *monitor*. If *int-do_change* has entered its trap t-35, which means the operation *schedule_and_assign_tasks* has been called, then ProjectManager will go to its state *starting schedule*. As soon as *int-do_change* enters its trap t-36 and naturally the tasks have been scheduled, Projectmanager will go back to its state *neutral*. As mentioned before the communication between the manager ProjectManager and its other employees *int-schedule_and_assign_tasks* (Figure 62) and *int-monitor* [2, figure 12] falls outside the scope of the problem of modelling Change Management and the subprocesses and traps of these operations w.r.t. ProjectManager will not be given here. However they can be found in [2, figures 53 and 55 respectively].

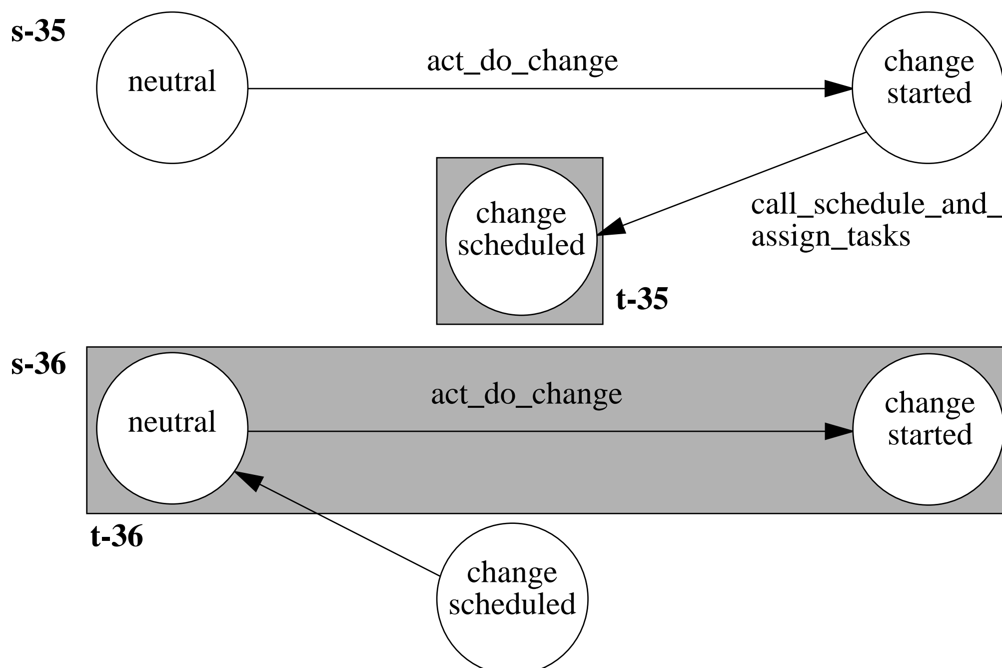


Figure 38. int-do_change's subprocesses and traps w.r.t. Projectmanager

Figure 39 shows the class ProjectManager as manager of *int-join_meeting*, *int-leave_meeting*,

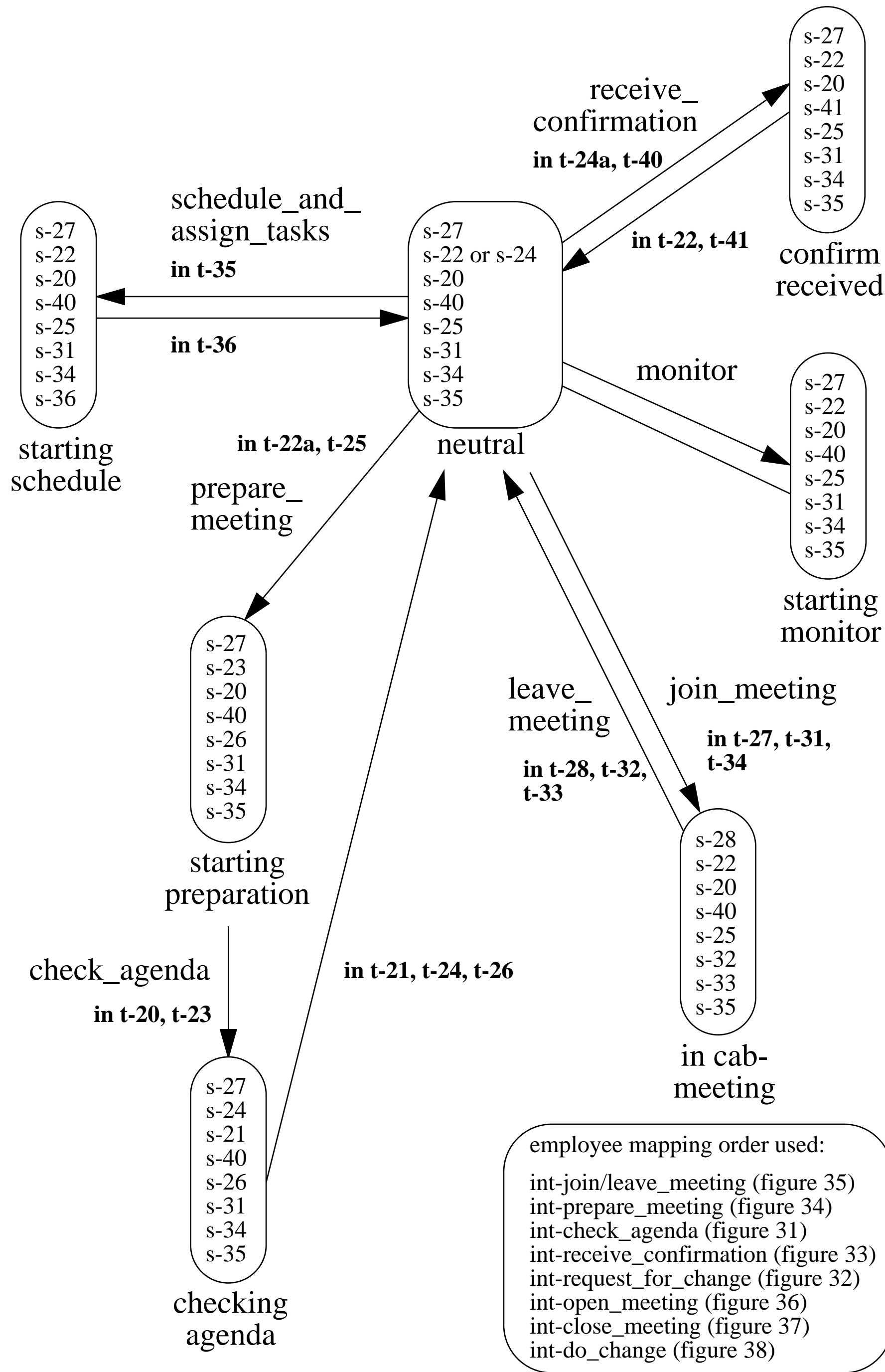


Figure 39. ProjectManager: viewed as manager of 8 employees

int-prepare_meeting, *int-check_agenda*, *int-receive_confirmation* (called operations), *int-request_for_change*, *int-open_meeting*, *int-close_meeting* and *int-do_change* (calling operations).

The fourth part of the communication specification shows the communication between the manager process DesignEngineer and its employee processes *int-join_meeting*, *int-leave_meeting*, *int-check_agenda*, *int-receive_confirmation*, *int-prepare_meeting*, *int-open_meeting* and *int-close_meeting*. The operations *design*, *code* and *review* have not been modelled, so the communication will not be specified. However the STD's of the internal behaviours of these operations and the subprocesses and traps of these internal behaviours can be found in [1]. The internal behaviours of the export-operations belonging to the class DesignEngineer are *int-check_agenda*, *int-join_meeting*, *int-leave_meeting* and *int-receive_confirmation*. *Int-open_meeting*, *int-close_meeting* and *int-prepare_meeting* are the internal behaviours of operations that call operations of the class DesignEngineer. As the class DesignEngineer is, just like ProjectManager, a subclass of CABMember, a major part of the communication specification is equal to the communication specification of the class ProjectManager. This part consists of the communication between DesignEngineer and the employee processes *int-join_meeting*, *int-leave_meeting*, *int-check_agenda*, *int-open_meeting* and *int-close_meeting*. So only the communication between DesignEngineer and *int-prepare_meeting* will be given here.
DesignEngineer (Figure 41) starts in its state *neutral*. When a particular DesignEngineer has been selected by ProjectManager to be a member of the board, DesignEngineer will be asked to check its agenda, i.e. to execute its operation *check_agenda*. If *int-prepare_meeting* has entered its trap t-42, if moreover *int-check_agenda* is in its trap t-20, then DesignEngineer can and will go to its state *checking agenda.* In this state the subprocesses s-44 and s-21 are going to be prescribed to *int-prepare_meeting* and *int-check_agenda* respectively. DesignEngineer will check its agenda now. As soon as *int-check_agenda* enters its trap t-21 and *int-prepare_meeting* enters its trap t-44, DesignEngineer can return to its state *neutral*. There subprocesses s-42 and s-20 will prescribed to *int-prepare_meeting* and *int-check_agenda* respectively. As soon as the date of the meeting is selected, which is when *int-prepare_meeting* has entered its trap t-43, and *int-receive_confirmation* is in its trap t-40, DesignEngineer can go to its state *confirm received*. There DesignEngineer will put the date in its agenda. DesignEngineer will return to its state *neutral* when *int-receive_confirmation* enters its trap t-41 and moreover *int-prepare_meeting* enters its trap t-45.
The traps that have to be entered to reach and leave the state *in CAB-meeting* are the same as for ProjectManager.

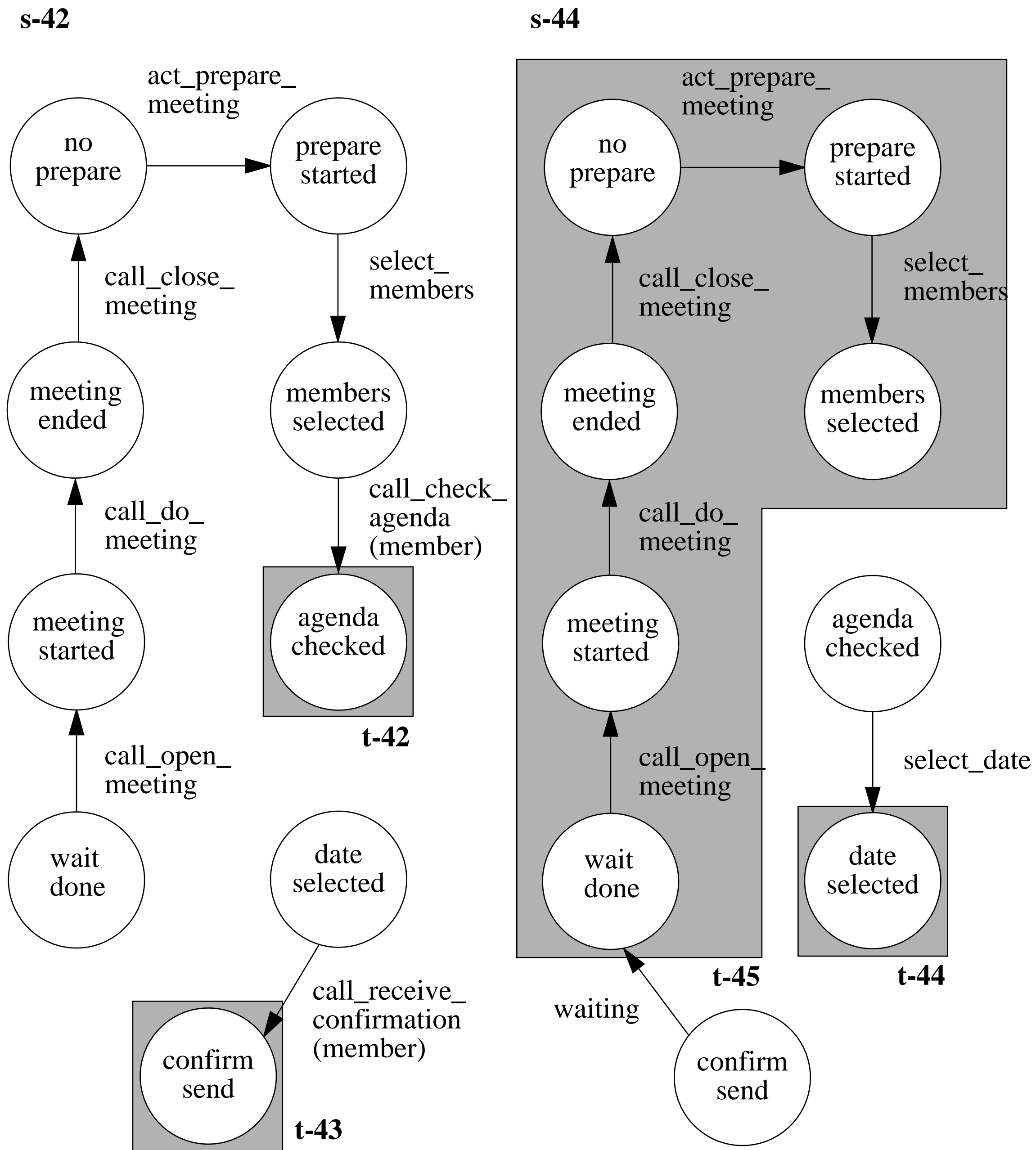


Figure 40. int-prepare_meeting's subprocesses and traps w.r.t. CABMember

Note that, as it is the internal behaviour of an operation of another class, only when every member has reacted to trap t-42 (or trap t-43) and consequently has prescribed subprocess s-44, *int-prepare_meeting* can continue to go to the state *date selected* (or state *wait done*). So it seems that every member has to wait for all other members to be called before it can return to its neutral state as t-44 (or t-45) can only be entered if state *date selected* (or state *wait done*) can be reached. However this is not the case. As the states *agenda checked* and *confirm send* are simplified representations of an (sub)STD, in which all members are called, the trap-structures are in fact more complicated. This implies that every member can react to trap t-44 and trap t-45 without waiting for the other members. More information can be found in Appendix A.

Figure 41 shows the class DesignEngineer as manager of *int-join_meeting*, *int-leave_meeting*,

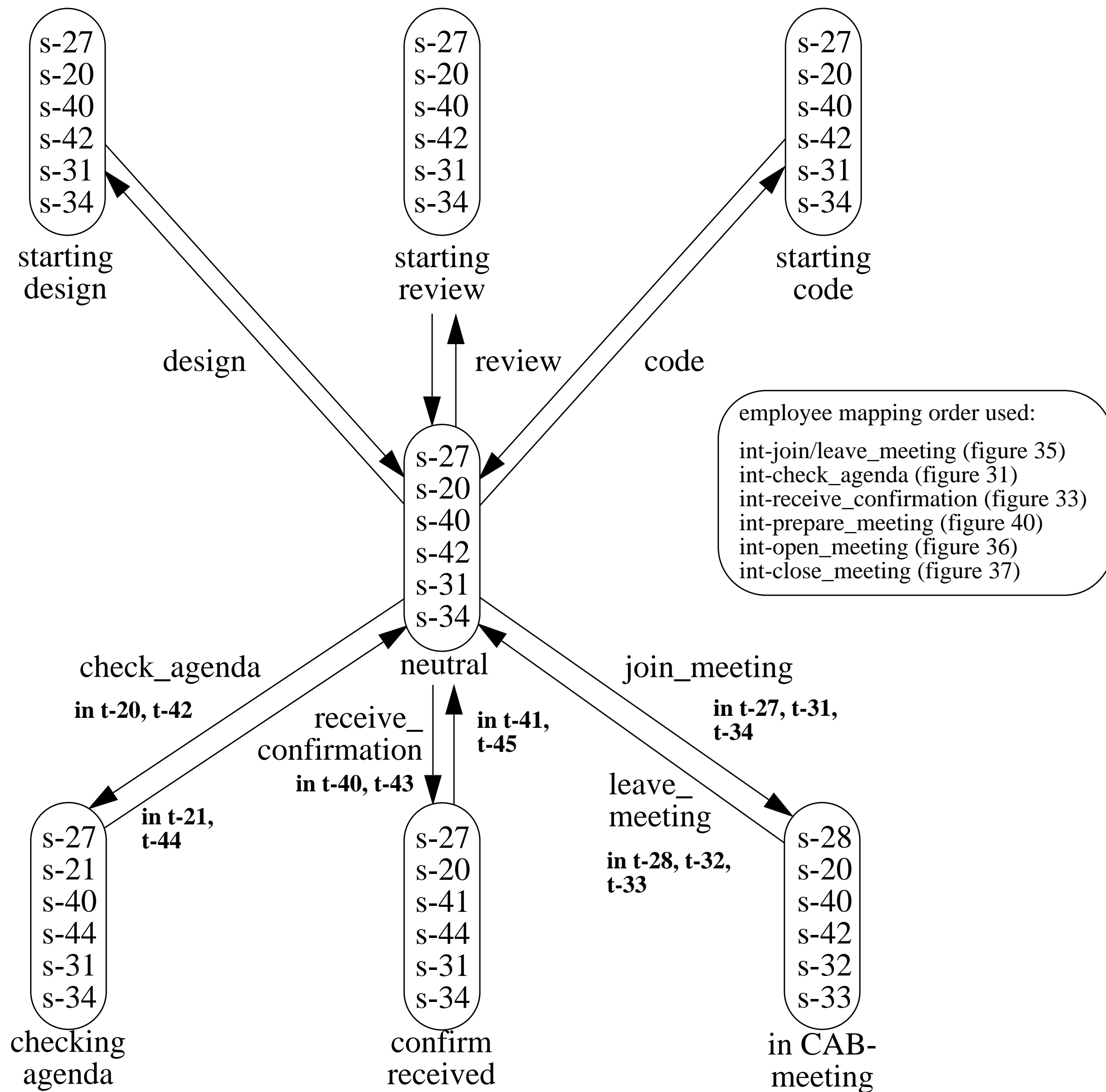


Figure 41. DesignEngineer: viewed as manager of 6 employees

int-check_agenda, *int-receive_confirmation* (called operations), *int-prepare_meeting*, *int-open_meeting* and *int-close_meeting* (calling operations). Note that the states *neutral*, *starting design*, *starting review* and *starting code* are equal w.r.t. these employees, that is the subprocesses remain unchanged in all states.

The fifth part of the communication specification shows the communication between the manager process UserRepresentative and its employee processes *int-join_meeting*, *int-leave_meeting*, *int-check_agenda*, *int-receive_confirmation*, *int-prepare_meeting*, *int-open_meeting* and *int-close_meeting*. The internal behaviours of the export-operations belonging to the class UserRepresentative itself are *int-check_agenda*, *int_receive_confirmation*, *int-join_meeting* and *int-leave_meeting*. The internal behaviours of operations that call operations of the class UserRepresentative are *int-open_meeting*, *int-close_meeting* and *int-prepare_meeting*. Because the class UserRepresentative is, just like DesignEngineer, a subclass of CABMember, the communication specification is equal to the communication specification of DesignEngineer and will therefore not be repeated here.
Figure 42 shows the class UserRepresentative as manager of *int-join_meeting*, *int-*

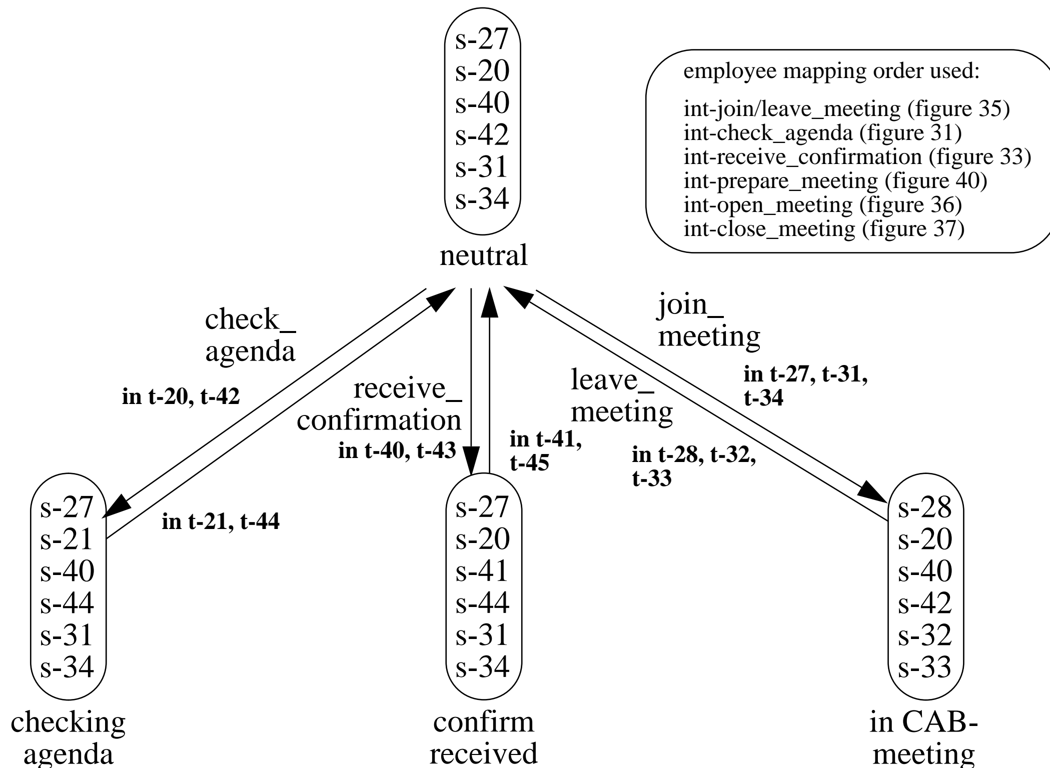


Figure 42. UserRepresentative: viewed as manager of 6 employees

leave_meeting, *int-check_agenda*, *int-receive_confirmation* (called operations), *int-prepare_meeting*, *int-open_meeting* and *int-close_meeting* (calling operations).

4 The new model

4.1 Introduction

The model presented in the previous chapter implies a lot of meetings in which only one request can be discussed, as for every request a new meeting will be prepared and opened. It would be more practical and efficient, and more realistic too, to discuss more than one request in a meeting. Also in the model presented in the previous chapter there is made no difference between small and big changes. Again it would be more practical, efficient and realistic to model such a difference, because small and big changes each have a different effect on the software process model. Therefore in this chapter a new model, based upon the previous model, will be designed. This new model is also necessary in order to be able to describe the evolutionary change of Change Management. The previous model represents the first evolution phase, the new model will represent the second evolution phase.
In the new model more than one request can be discussed in a meeting. This will be done by placing requests on a list. As soon as a list is full a meeting will be requested, so every list will correspond with a meeting. After that new requests will be placed on a new list. Of course also during a meeting new requests will have to be handled.
In order to model this some new classes will be introduced and other classes will have to change. The new class CABSecretary will administrate and handle the incoming change-requests and will place them on a list. It will also handle the outgoing, i.e. accepted, change-requests. Another new class will be the class Request, in which the status of the request is kept. The class CAB will be changed as a part of the external behaviour becomes internal behaviour of an export-operation of the new class CABSecretary (*handle_change_request*). In this new model meetings cannot be held concurrently. However it is not necessary to model this possibility, as we can assume that during a meeting a list will not grow so full that a new meeting already is necessary. Consequently the parallel behaviour modelled by using a parallel description in the external behaviour of CAB, as has been used in the previous model, will be removed. This is consistent with the original SOCCA approach.
In the new model also the difference between small and big changes will be modelled. When modelling this difference, there are two possibilities:

1. split behaviours up: create separate STD's for the external behaviour of some classes (in this case Design) and for the internal behaviour of some operations in case of a big change and in case of a small change. The result will be a lot of STD's that will be almost identical, for instance *schedule_and_assign_big_tasks* and *schedule_and_assign_small_tasks*. It also means that managers get a lot of employees. This implies the model grows very big.
2. parametrize transitions: make one STD and parametrize with a parameter *size* all transitions that would cause creating separate STD's when using the first possibility (see for instance Figure 59). This implies the amount of STD's will not grow too big.

In the new model the second possibility is used. From now on the model in the previous chapter will be referred to as the old model.

4.2 Designing the new model

As mentioned in the previous section, some new classes will be added to the old model in order to describe Change Management more precisely:

- CABSecretary: handles some administrative business for the Change Advisory Board.
- Request: for every request the status is kept and a possible change is initiated.

Also the classes CAB and Design will change, and from now on they will be referred to as the

class NewCAB and the class NewDesign respectively.
As some classes have been added to the model, the static structure of the model, i.e. the class-diagram, will change too. Figure 43 shows the class diagram of the new model. Note that the new classes CABSecretary and Request do not have a IS-A relationship or Part-Of relationship with any other class.

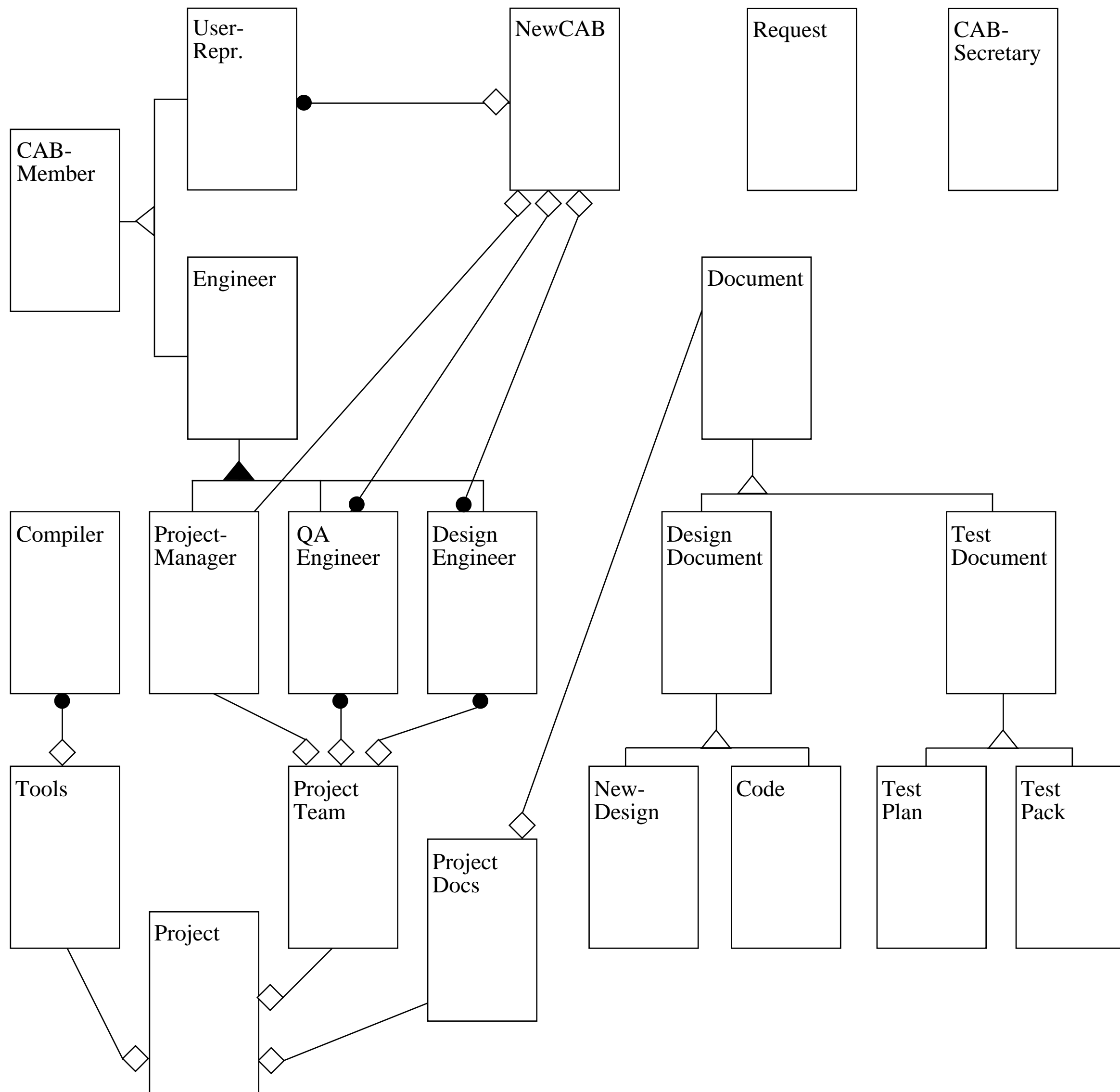


Figure 43. Class diagram of the new model: classes and IS-A and Part-Of relationships

Figure 44 shows the attributes and operations of the new classes and the changed class New-CAB. The export-operations of the class NewDesign have not been changed. Therefore this class will not be mentioned in the figure. The operations and attributes of all other classes will not be given too.

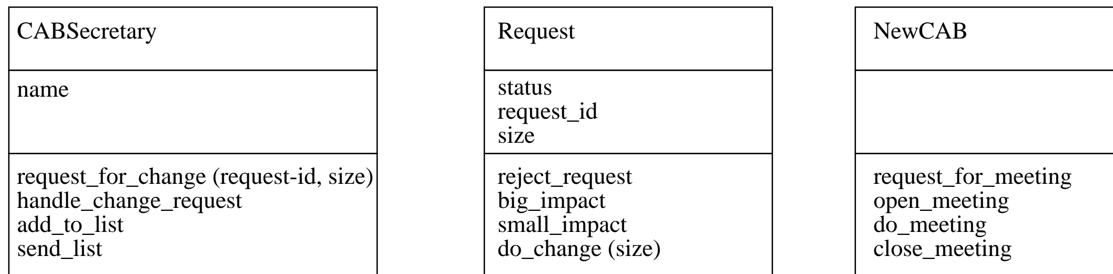


Figure 44. Class diagram of the new model: attributes and operations

Note that the operations of the classes Request and CABSecretary (except for *send_list*) have all been parametrized (explicitly or implicitly) with the parameter *request-id*. The operations *add_to_list* and *send_list* have been parametrized implicitly with a parameter *list-id*. The operations *request_for_change* and *do_change* have been parametrized with a parameter *size*.
As new classes have been added to the model and others have been changed, some new general relationships between the classes can be defined. These new general relationships are shown in Figure 45. Note that the old general relationships remain valid. Therefore they are also shown in Figure 45.

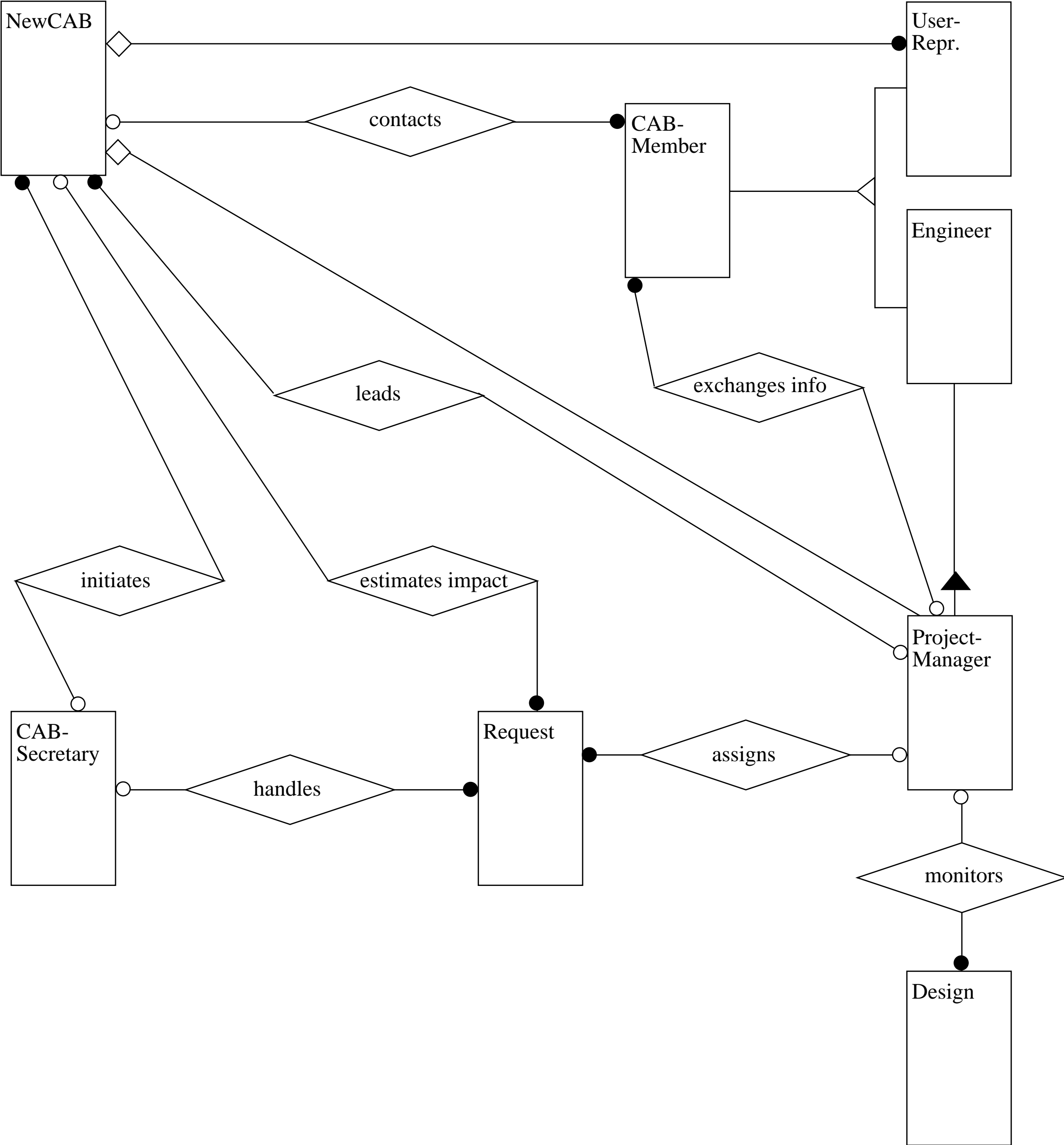


Figure 45. Class diagram of the new model: classes and general relationships

By adding and changing classes, new uses relationships will appear too. In Figure 46 all uses relations important to our discussion, i.e. the uses relations from the old model and all new uses relations, are given. In Figure 47 the corresponding import list is given.

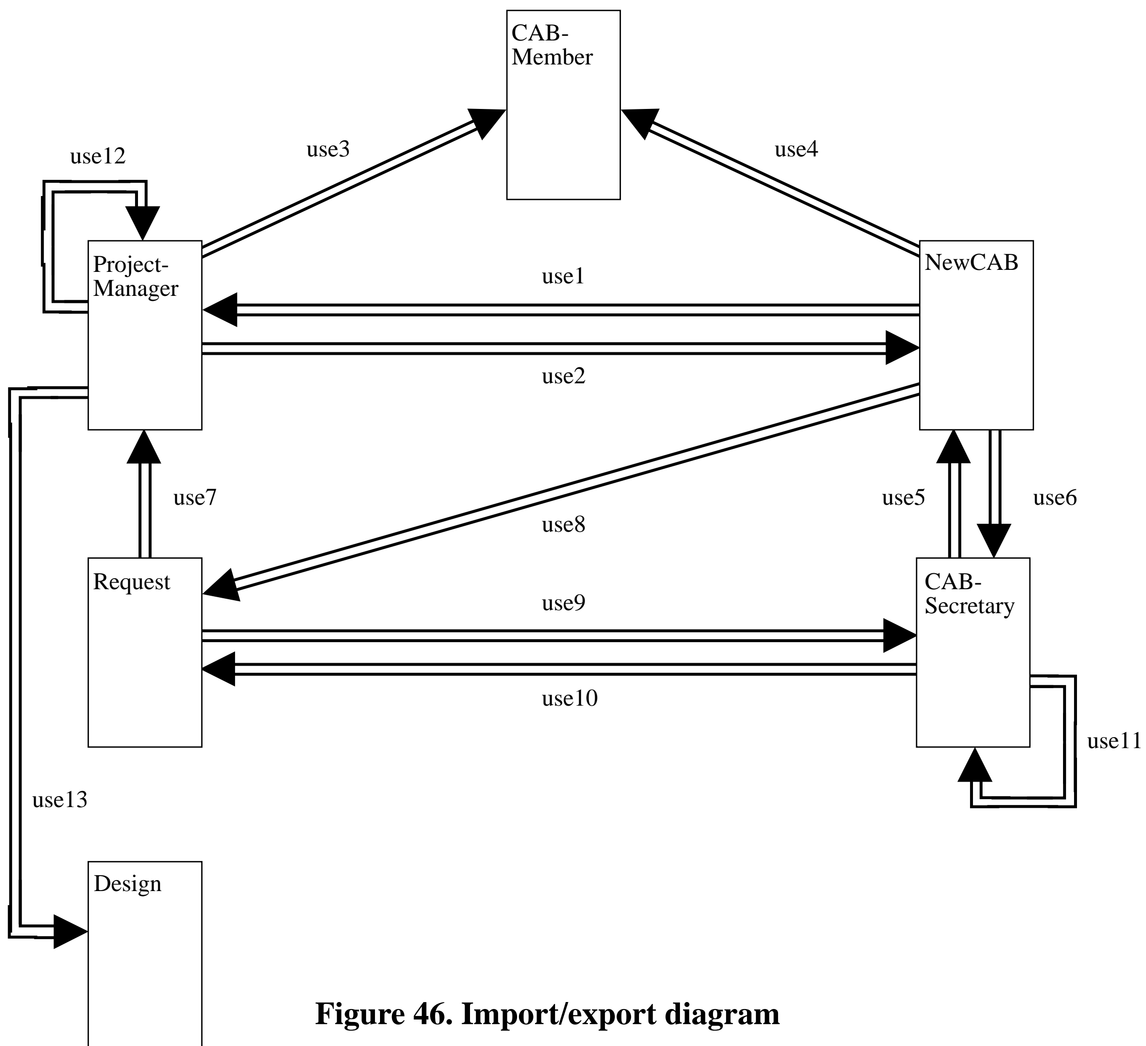


Figure 46. Import/export diagram

use1
prepare_meeting
use2
open_meeting
do_meeting
close_meeting
use 3
check_agenda (member)
receive_confirmation (member)
use4
join_meeting (member)
leave_meeting (member)

use5
request_for_meeting
use6
send_list
use7
sched_and_assign_tasks
use8
big_impact
small_impact
use9
handle_change_request (req-id, size)

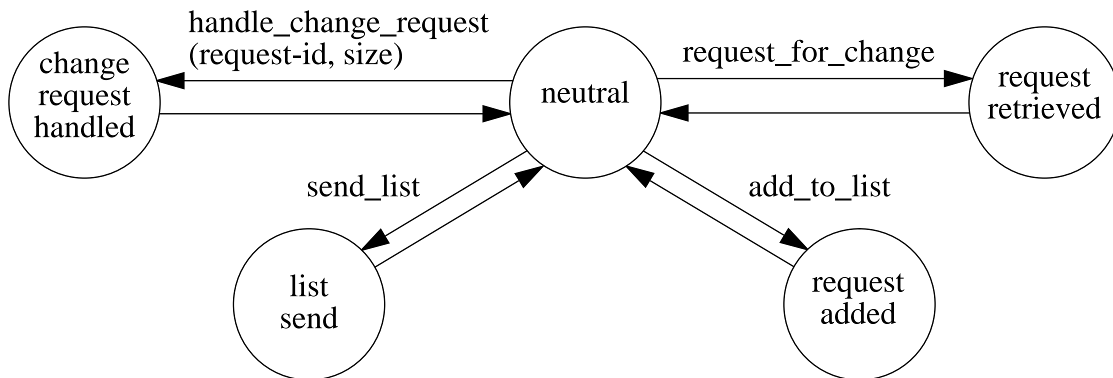
use10
do_change (size)
use11
add_to_list
use12
monitor
use13
notify_modif_opened
notify_modif_closed
notify_review_opened
report_review_result

Figure 47. Import list

Note that the operations *check_agenda*, *receive_confirmation*, *join_meeting* and *leave_meeting* are parametrized with the parameter *member* as an explicit reminder of the fact that some details in the calling operations have been omitted. As mentioned in the previous chapter, these details, which actually are rather complicated, will be discussed in Appendix A.

4.2.1 Designing the new external behaviours of the classes

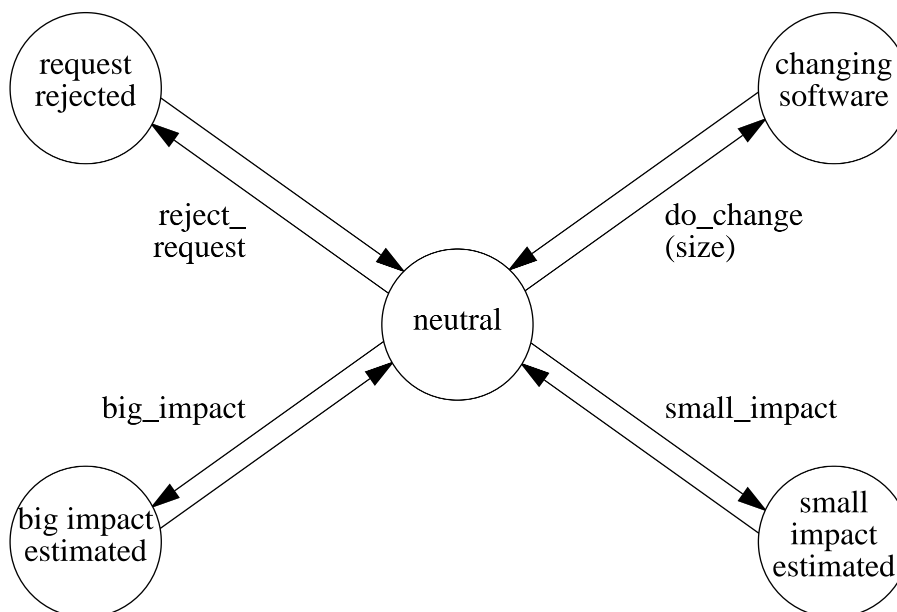
The external behaviours of the new and changed classes will be specified (figures 48 through 51). From then on, the order in which the export-operations can be called will be known.



w.r.t. WODAN this is subprocess s-142 and the state space is trap t -142

Figure 48. CABSecretary: STD of the external behaviour

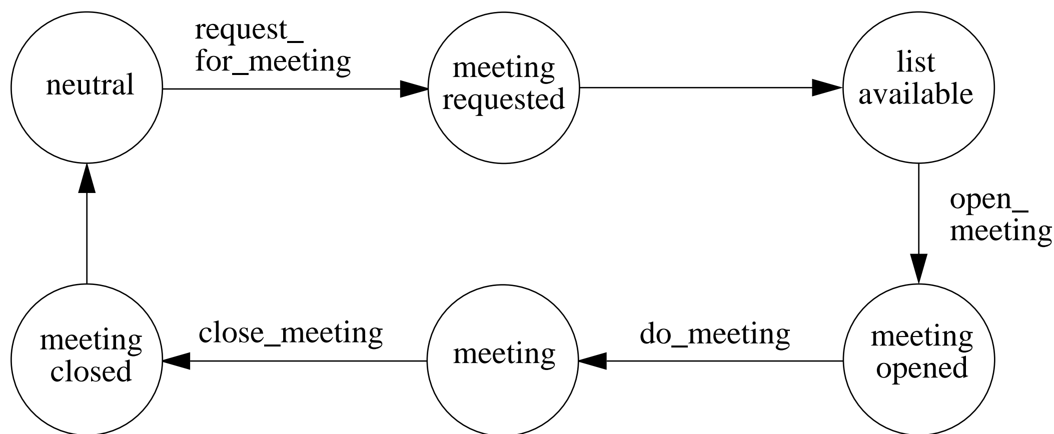
The transition labeled with *handle_change_request* replaces the "lower" part of the external behaviour of the class CAB. This transition has been parametrized with the parameter *request-id*, so actually there are as many transitions as there are requests. The reason for doing this is that by choosing the right subprocesses and traps for *int-handle_change_request* one can accomplish that more than one request at a time can be handled. This is necessary, otherwise everytime a request is handled one has to wait for the previous request being authorized or cancelled and there would be no profit of discussing more than one request in a meeting. It has also been parametrized with the parameter *size*, because the value of this parameter is needed in the internal behaviour of the operation to make the call to *do_change*.
Note that the operations *add_to_list* and *send_list* have been parametrized implicitly with a parameter *list-id*.



w.r.t. WODAN this is subprocess s-143 and the state space is trap t-143

Figure 49. Request: STD of the external behaviour

For every request (i.e request-id) an instance of the class Request exists, so for every request (i.e request-id) an STD, as given in Figure 49, exists. When making the transition to the state *changing software* the value of the parameter *size* is known, as it is passed through by *handle_change_request*, which makes the call to *do_change*.
Another possibility when modelling the class Request is to parametrize an export-operation *accept_request* with the parameter *size*, just like the *do_change* operation has been parametrized, instead of using *big_impact* and *small_impact*. Also the state *changing software* could be removed by labelling the transitions leading from the states *big impact estimated* and *small impact estimated* to the state *neutral* with *do_change*. This is possible as for every instance of the class Request *do_change* can only be called after *big_impact* or *small_impact* has been called. However, if one wishes to add another operation to the class, the external behaviour of Request as shown in Figure 49 is preferable.

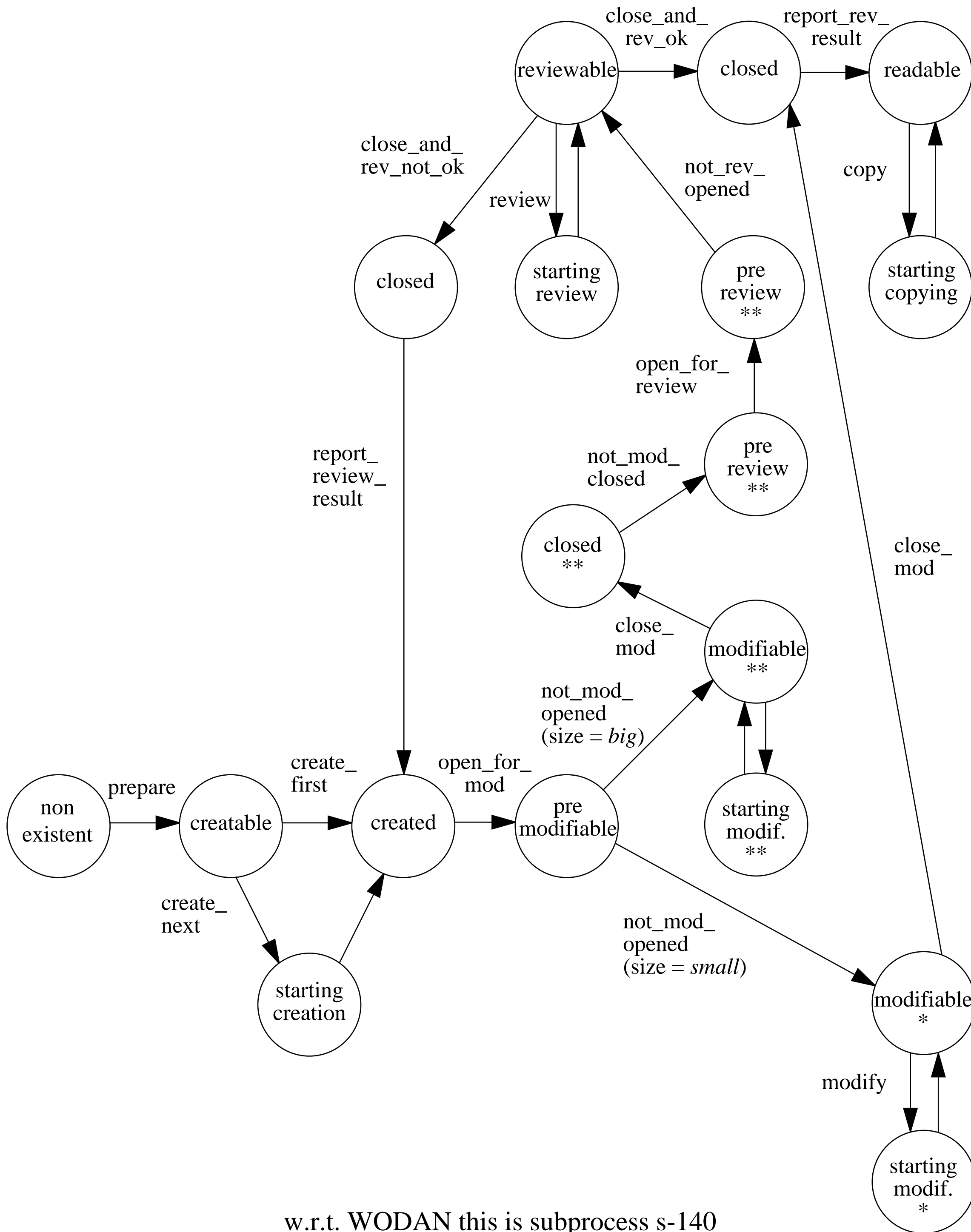


w.r.t. WODAN this is subprocess s-141 and the state space is trap t-141

Figure 50. NewCAB: STD of the external behaviour

The class NewCAB describes the behaviour of the Change Advisory Board in the new model. The external behaviour of NewCAB is similar to the "upper" part of the external behaviour of DepCAB, with the difference that the state *list available* has been added. Also the operation *request_for_change* has been replaced by the operation *request_for_meeting*. Consequently the class NewCAB will no longer handle the change-requests. As can be seen above, the new class CABSecretary will now perform this task.
There can only be one instance of the class NewCAB at a time, in contradistinction to the class DepCAB. Therefore meetings cannot be held simultaneously. However this will not be a problem, as we can assume that during a meeting a new list will not grow so full that a new meeting already has to be requested.

The class Design had to be changed to take into account the (estimated) size of the change.



w.r.t. WODAN this is subprocess s-140

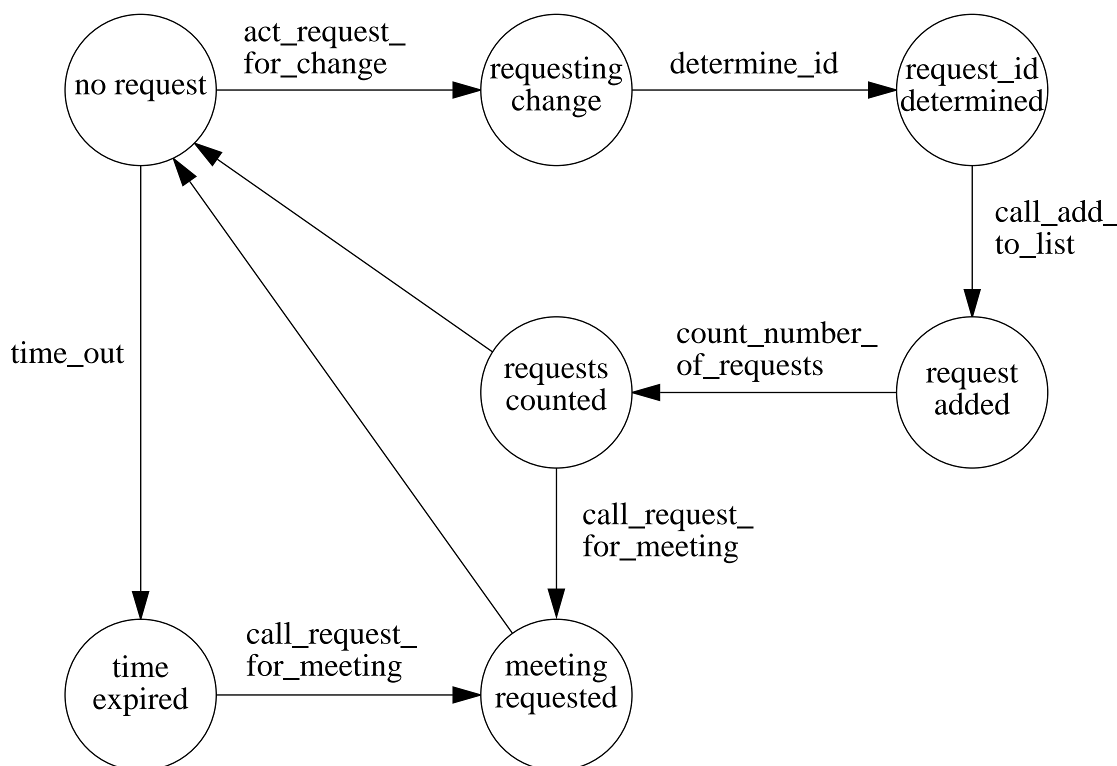
Figure 51. NewDesign: STD of the external behaviour

When the (estimated) size of the change is big (for instance when a new product has to be developed) NewDesign will follow the path from *pre-modifiable* to *reviewable* through the states marked by **, when the (estimated) size is small NewDesign will follow the path through the states marked with * (in the latter case the review will be skipped, but of course the code document will still be tested). This is done by parametrizing the transition labeled with *not_mod_opened* with a parameter *size* (possible values: *big* or *small*). The parameter *size* is passed through via *call_monitor* in *int-schedule_and_assign_tasks*. The STD of the internal be-

haviour of *int-monitor* will be shown in Figure 63. Figure 51 is very similar (but not equal) to another figure from Wulms, see [2, Figure 13]. Compared to the ISPW-6 example the path marked with * has been added.

4.2.2 Designing the new internal behaviours of the export-operations

After specifying the external behaviours of the classes, the internal behaviours of the operations can be specified. Of course only new and changed internal behaviours will be specified. The conventions used to specify the internal behaviours of the operations are the same as the conventions used in the old model.
The export-operations of the new class CABSecretary are *add_to_list*, *send_list* (these two operations have been parametrized implicitly with a parameter *list-id*), *request_for_change* and *handle_change_request*.



w.r.t WODAN this is subprocess s-144 and the state space is trap t-144

Figure 52. int-request_for_change

All requests are given a request-id, then the requests are put on a list. When the maximum number of requests is reached or the time has been expired a meeting has to be requested. Note that in the internal behaviour of *request_for_change* an operation (*add_to_list*) belonging to the same class as *request_for_change* is called. Note also that the internal behaviour of *request_for_change* has changed completely in comparison with the old model (see Figure 13.).

The internal behaviours of both *add_to_list* and *send_list* are relatively simple. Activation of the operations denotes execution of the operations. After that they will return to their neutral states.

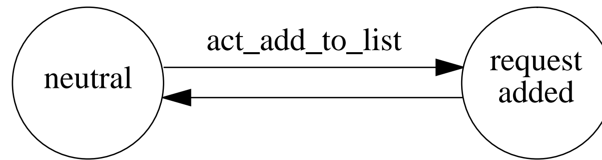


Figure 53. int-add_to_list

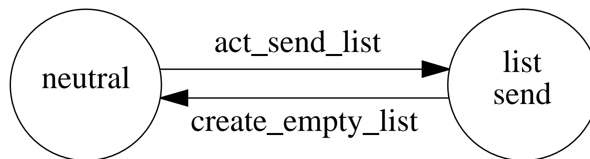


Figure 54. int-send_list

Whenever a list has to be send (to NewCAB), an empty list to put new requests on will be created too. This internal operation will also determine a new *list-id* for the empty list.

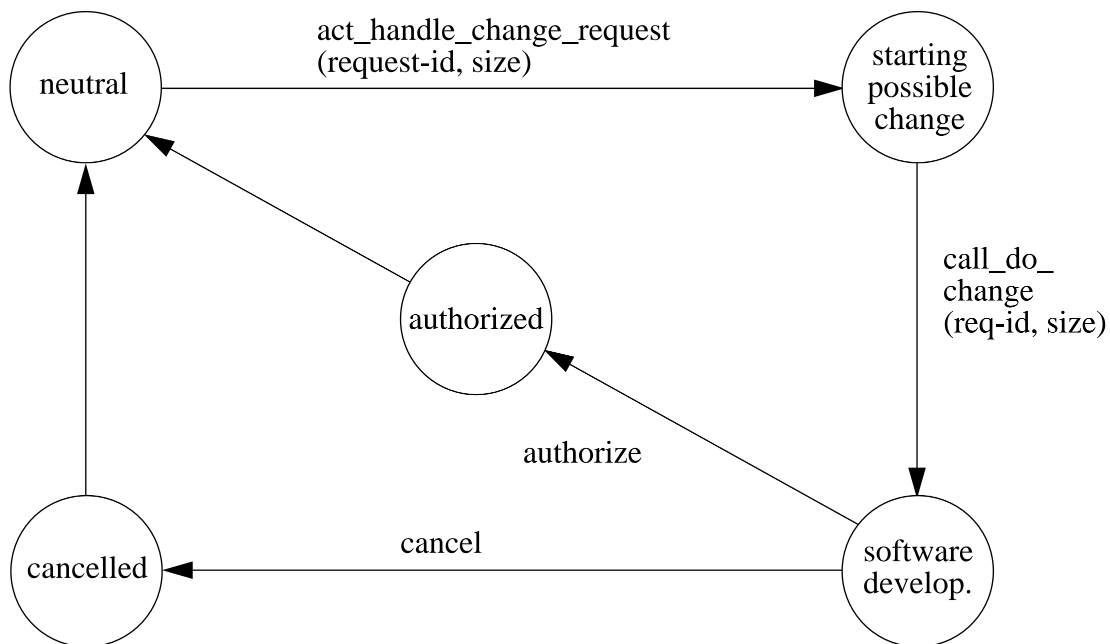


Figure 55. int-handle_change_request

In the old model the internal behaviour of *handle_change_request* is part of the external behaviour of the class CAB (see Figure 9). In the new model *handle_change_request* is an export-operation of the class CABSecretary, as it is just a formality to start the actual change after the decision has been taken in the meeting, although it could also be an export-operation of the class CAB.

The export-operations of the new class Request are *reject_request*, *small_impact*, *big_impact* (as mentioned before these last two operations could be joined and parametrized) and *do_change*.

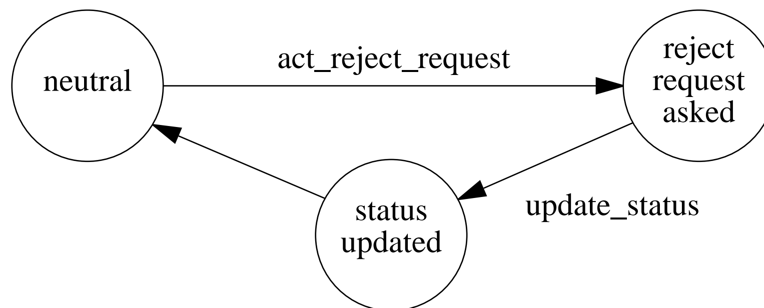


Figure 56. int-reject_request

In the internal behaviours of *big_impact* and *small_impact* the value of the parameter *size* is initialized within the call to *handle_change_request*. The parameter is needed there for the call to *do_change*.

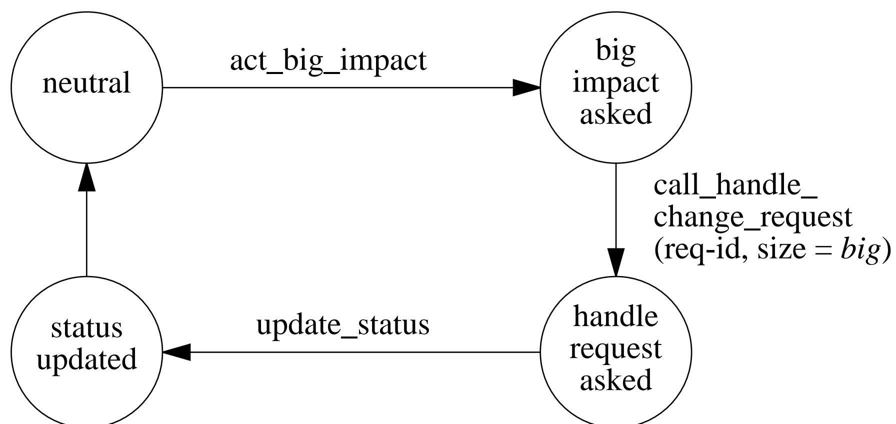


Figure 57. int-big_impact

Note that the value of *size* must be *big*.

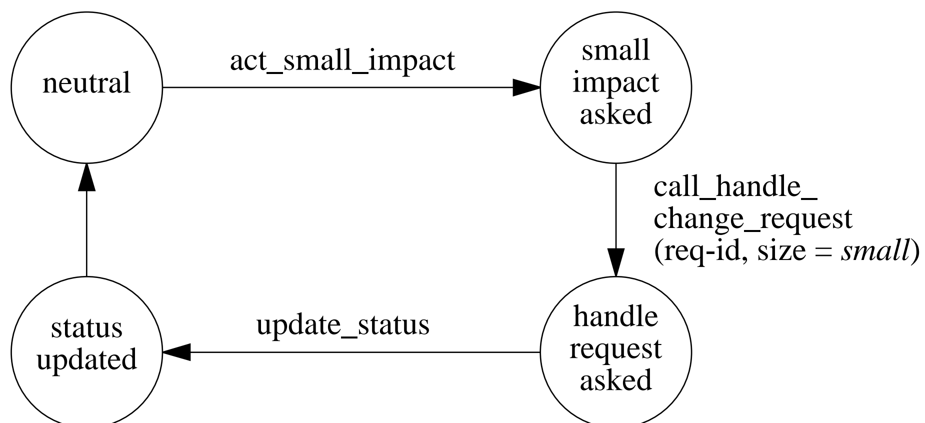


Figure 58. int-small_impact

Note that the value of *size* must be *small*.

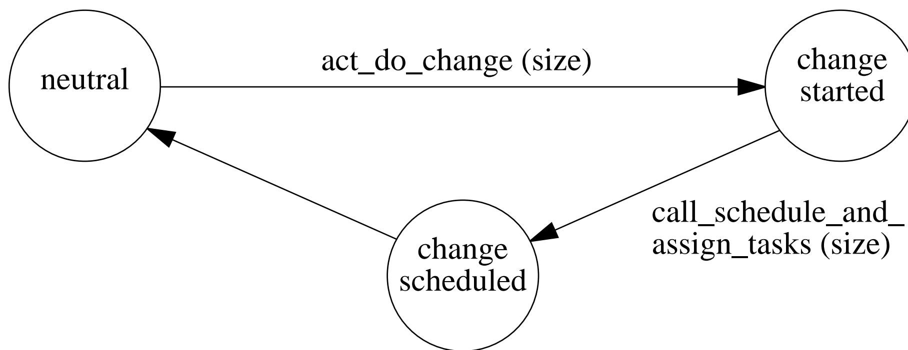


Figure 59. int-do_change

Note that the call to *schedule_and_assign_tasks* is parametrized with the parameter *size* (possible values: *big* or *small*). This information is needed inside *int-schedule_and_assign_tasks* for *call_monitor*. Note that *call_schedule_and_assign_task*s implicitly has been parametrized with the parameter *doc_name* too (see also [1]).

The export-operations of the changed class NewCAB are *request_for_meeting*, *open_meeting*, *do_meeting* and *close_meeting*.

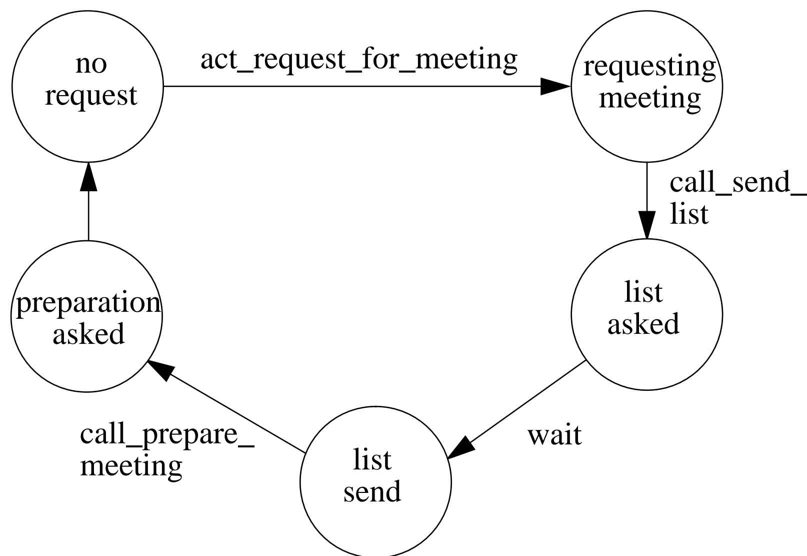
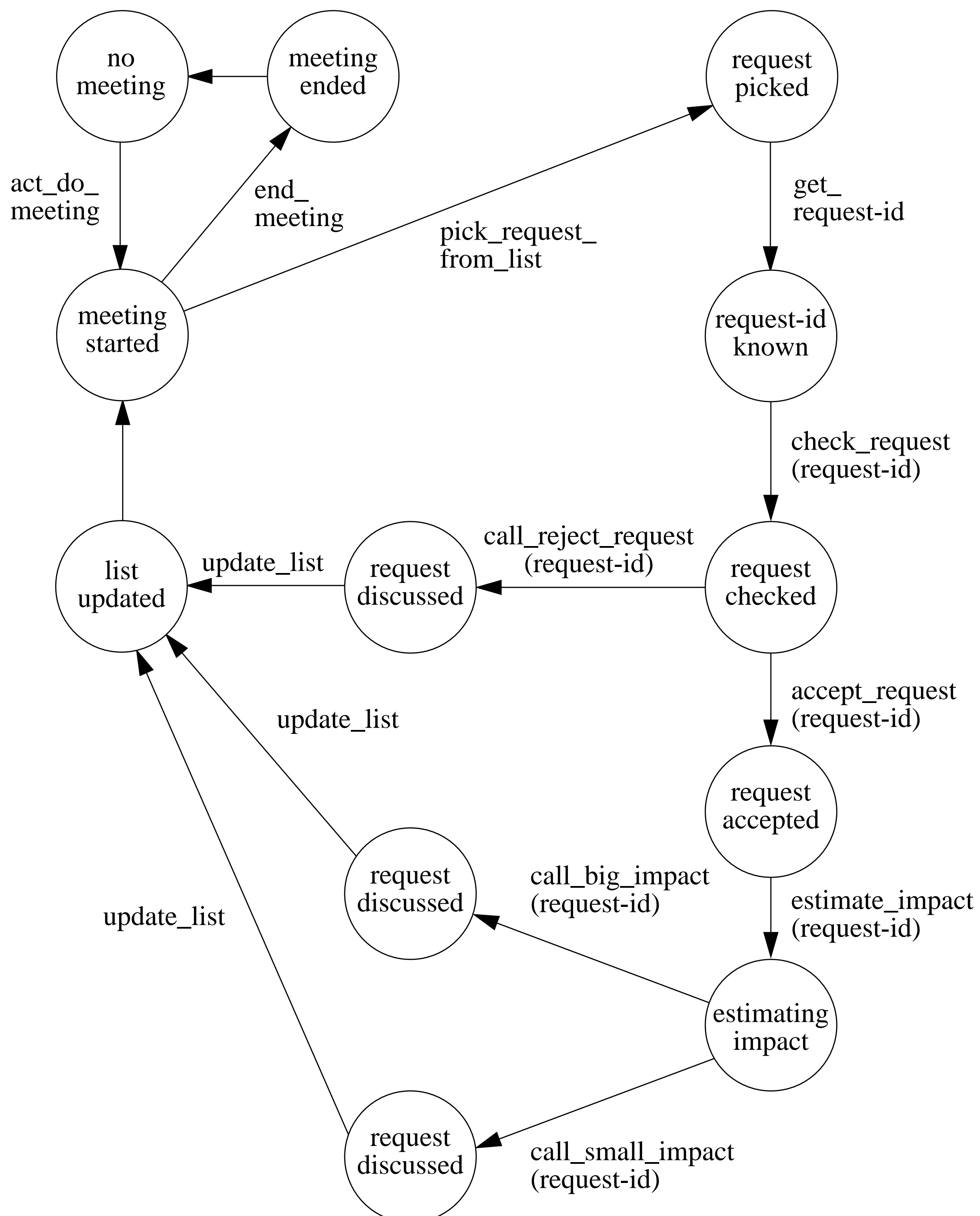


Figure 60. int-request_for_meeting

It is from the internal behaviour of *request_for_meeting* that *send_list* and after that *prepare_meeting* are called.
In the new model the behaviours of the export-operations *open_meeting* (Figure 14) and *close_meeting* (Figure 16) are the same as in the old model, as the use of a list of requests instead of a single request does not influence these behaviours. The export-operation *do_meeting* is influenced by this change and has to be changed (Figure 61).

In *do_meeting* a request is picked from the request-list and a decision is made about the request.



w.r.t. WODAN this is subprocess s-150 and the state space is trap t-150

Figure 61. int-do_meeting (new version)

After that the request-list is updated, which means the request actually will be removed from the request-list. If there are any requests left on the list, a new request wil be picked and the cycle starts all over again. If there are no requests left on the list the meeting will be ended.
Most internal operations as well as all calls are parametrized with the parameter *request-id* (taken from the request-list), because this *request-id* is needed to identify the right instance of the class Request.

Two export-operations of the class Projectmanager, that have to be changed due to the fact that there is a difference between big and small changes, are *schedule_and_assign_tasks* and *monitor*.

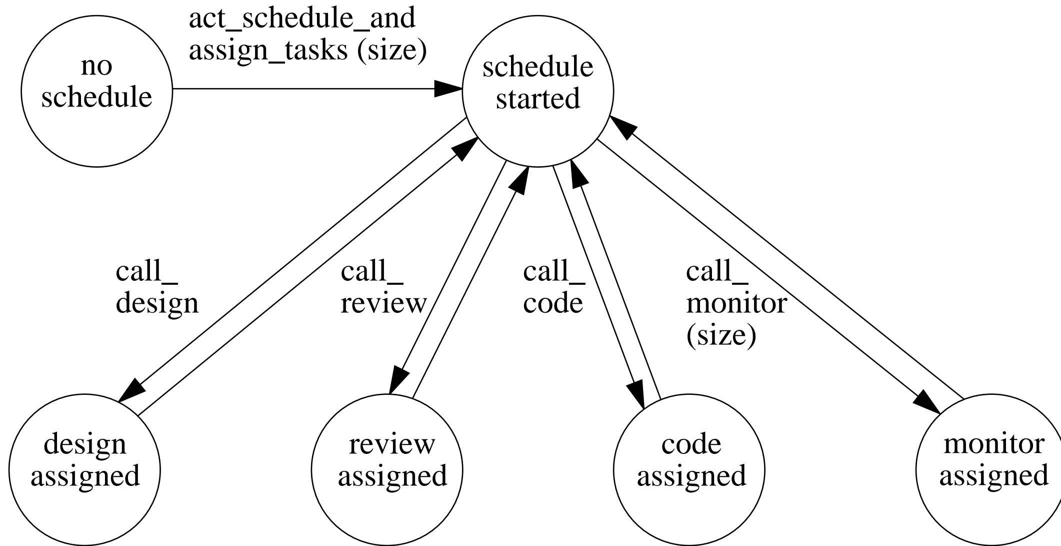
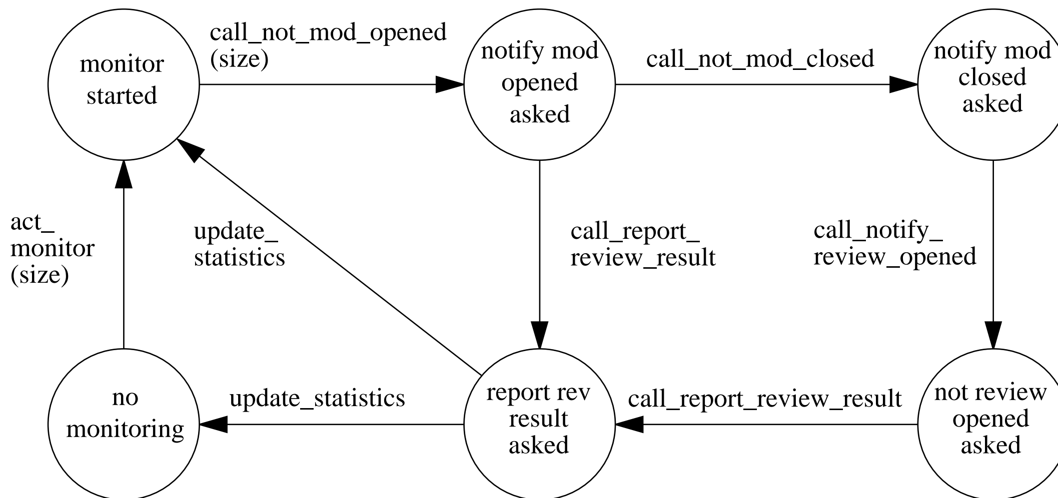


Figure 62. int-schedule_and_assign_tasks

Note that *call_design*, *call_review*, *call_code* and *call_monitor* all are parametrized with a document name *doc_name*. In addition *call_monitor* is parametrized with a parameter *size* (possible values: *big* or *small*), which is the (estimated) size of the change (and in *int_monitor* the call *not_mod_opened* is also parametrized with the parameter *size*). Note also that *review* will only be called in the case of a big change, as in the case of a small change the review will be skipped. The STD's of the internal behaviour of *int-design*, *int-review* and *int-code* can be found in [1, Figure 10], [1, Figure 11] and [2, Figure 34] respectively.



w.r.t. WODAN this is subprocess s-147

Figure 63. int-monitor

As can be seen above the STD of *int-monitor* has a short-cut from the state *notify mod opened asked* to the state *report rev result asked* because for small projects (i.e. changes) the exact intermediate result is not relevant. Note that this short-cut is an extra transition compared to the original ISPW-6 example. A figure similar to Figure 63 can be found in [2, Figure 15].

4.3 The communication in the new model

After the specification of the external and internal behaviours of the classes and operations, the communication between these behaviours has to be modelled. This communication is shown in five parts. Whenever (parts of the) comunication specification of the old model will be used, this will be mentioned. The standards used to describe the communication between the manager process and its employee processes are the same as the standards described in the previous chapter. When there is a deviation of these standards, a reason will be given.
The first part of the communication specification shows the communication between the manager process CABSecretary and its employee processes *int-add_to_list*, *int_send_list*, *int-request_for_change*, *int-handle_change_request*, *int-big_impact*, *int-small_impact* and *int-request_for_meeting*. The internal behaviours of the export-operations belonging to the class CABSecretary itself are *int-add_to_list*, *int_send_list*, *int-request_for_change* and *int-handle_change_request*. The internal behaviours of operations that call operations of the class CABSecretary are *int-big_impact*, *int-small_impact* and *int-request_for_meeting*.

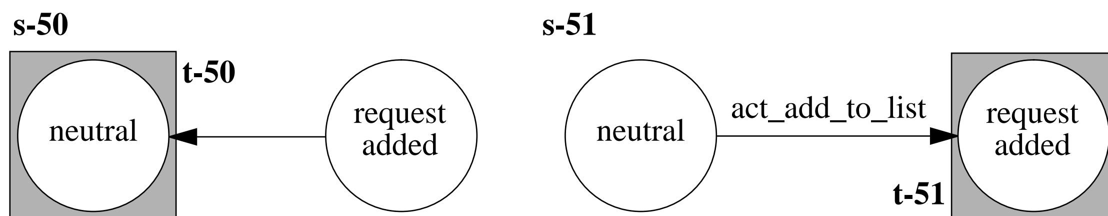


Figure 64. int-add_to_list's subprocesses and traps w.r.t. CABSecretary

CABSecretary (Figure 71) starts in its state *neutral*. There subprocess s-54 has been prescribed to *int-request_for_change* (Figure 65) If this behaviour has entered its trap t-54, which means that a previous request has been administrated or a time-out has occurred, then CABSecretary can make the transition to its state *request retrieved.* In that state subprocess s-56 is prescribed to *int-request_for_change* and the request can be administrated. Note that subprocess s-56 does not contain all states; the state *requests counted* has been omitted. However subprocess s-56 contains every state that can be reached in the state *request retrieved* (of CABSecretary). This can be seen as follows. The internal behaviour of *request_for_change* calls the export-operation *add_to_list* of the class CABSecretary itself. This operation has to be activated first before *int-request_for_change* can continue. However, *add_to_list* can only be activated in the neutral state (of CABSecretary). Consequently the state *requests counted* cannot be reached in the state *request retrieved* (of CABSecretary). Also the transition labeled with *time-out* has been omitted in subprocess s-56. By this means the operation is forced to add the incoming request to the list before a meeting can be requested due to time-out.
As soon as *int-request_for_change* enters its trap t-56, CABSecretary returns to its state *neutral*. There subprocess s-54 is prescribed to *int-request_for_change*. If this behaviour has entered its trap t-55, if moreover *int-add_to_list* is in its trap t-50, then CABSecretary will go to its state *request added*. In that state subprocesses s-51 and s-57 will be prescribed to *int-add_to_list* and *int-request_for_change* respectively and the request will be placed on the list. Again subprocess s-57 does not contain all states, but only the states that can be reached in the state *request added* (of CABSecretary). When *int-request_for_change* enters its trap t-57 and also *int-add_to_list* enters its trap t-51, CABSecretary will return to its neutral state.

Note that subprocess s-54 contains two traps, a large trap t-54 and a small trap t-55. Note also that a time-out can occur only in the subprocesses s-54 and s-57.

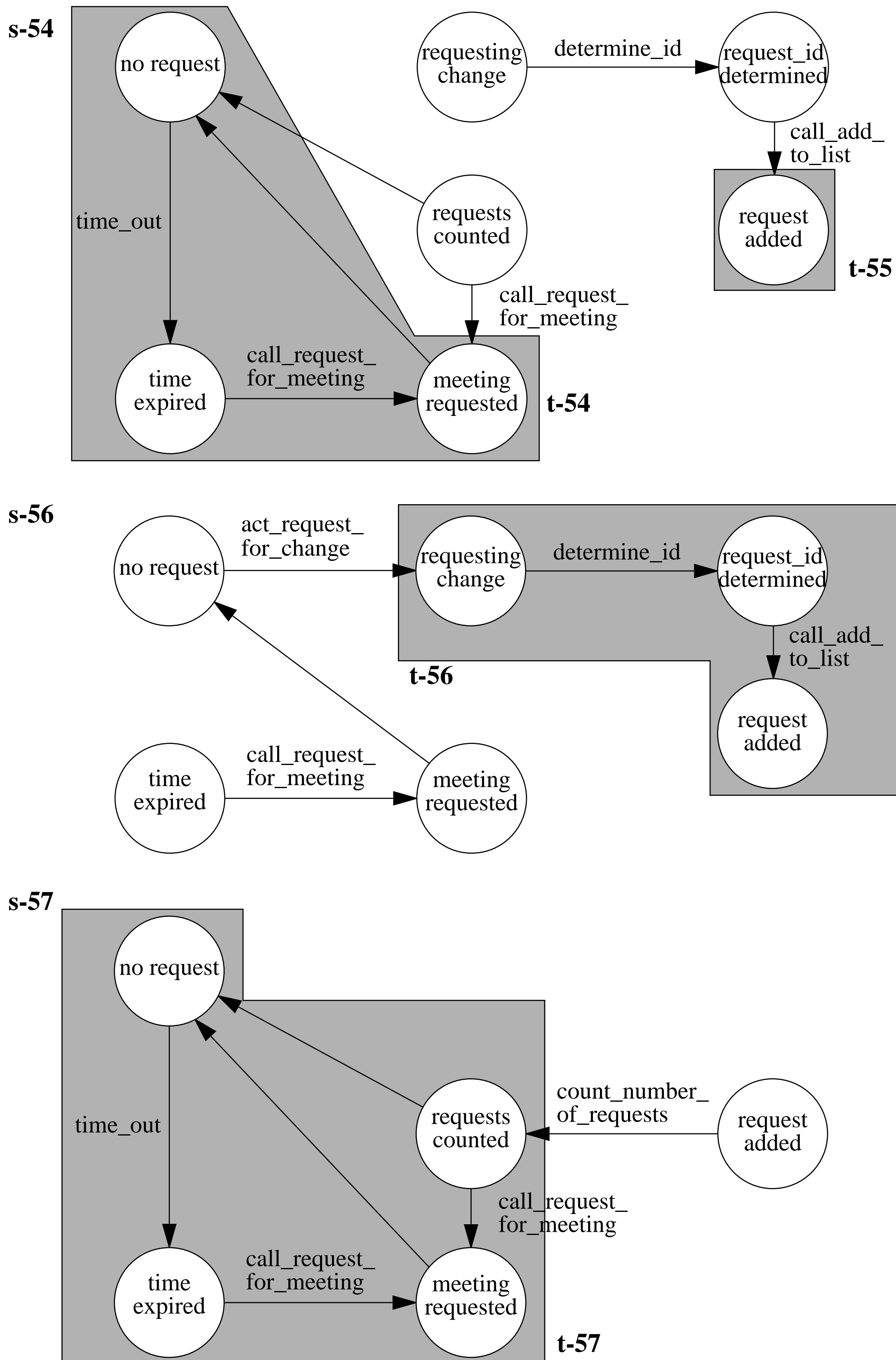


Figure 65. int-request_for_change's subprocesses and traps w.r.t. CABSecretary

Let us now continue with the manager process. CABSecretary is in its state *neutral*. If *int-handle_change_request* is in its trap t-58, which means a request can be handled, if moreover *int-big_impact* has entered its trap t-60 or *int_small_impact* has entered its trap t-62, then *handle_change_request* can be performed and the request will be handled. As soon as *int-handle_change_request* enters its trap t-59 and also *int-big_impact* enters its trap t-61 or *int_small_impact* enters its trap t-63, CABSecretary will return to its state *neutral* and new requests can be handled, administrated or placed on a list. Note that the order in which the operations are called is not completely arbitrary. A specific request has to be administrated before it can be handled.

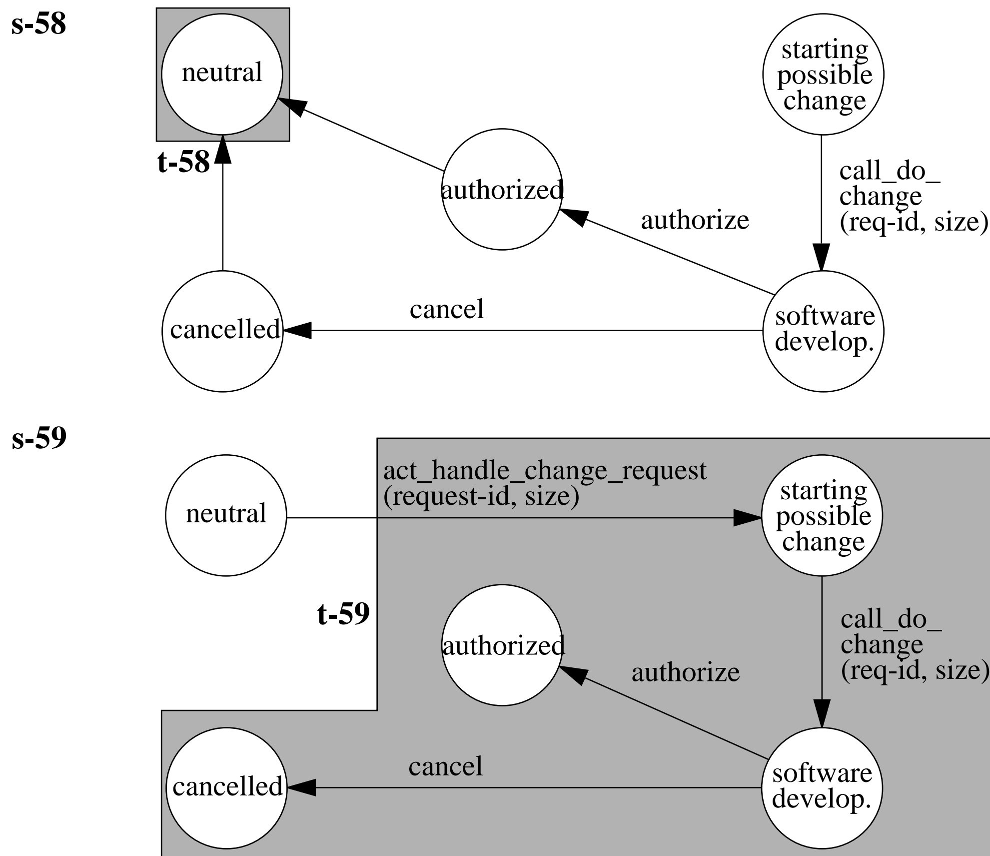


Figure 66. int-handle_change_request's subprocesses and traps w.r.t. CABSecretary

Note that for every request a *handle_change_request* operation and a partition of subprocesses of the internal behaviour of this operation exists as shown in Figure 66. So requests do not have to wait for each other to be authorized or cancelled.

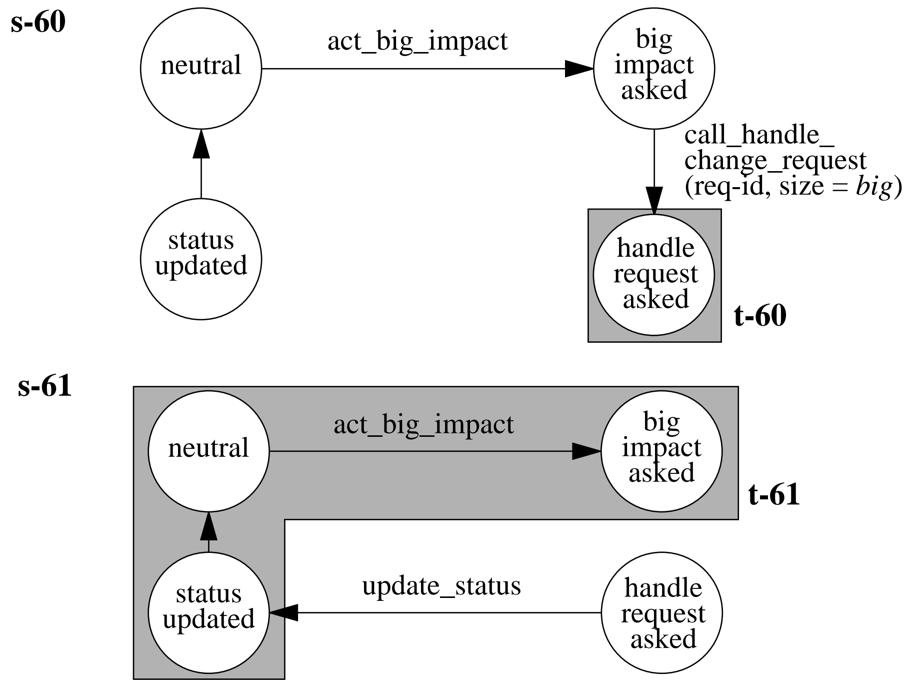


Figure 67. int-big_impact’s subprocesses and traps w.r.t. CABSecretary

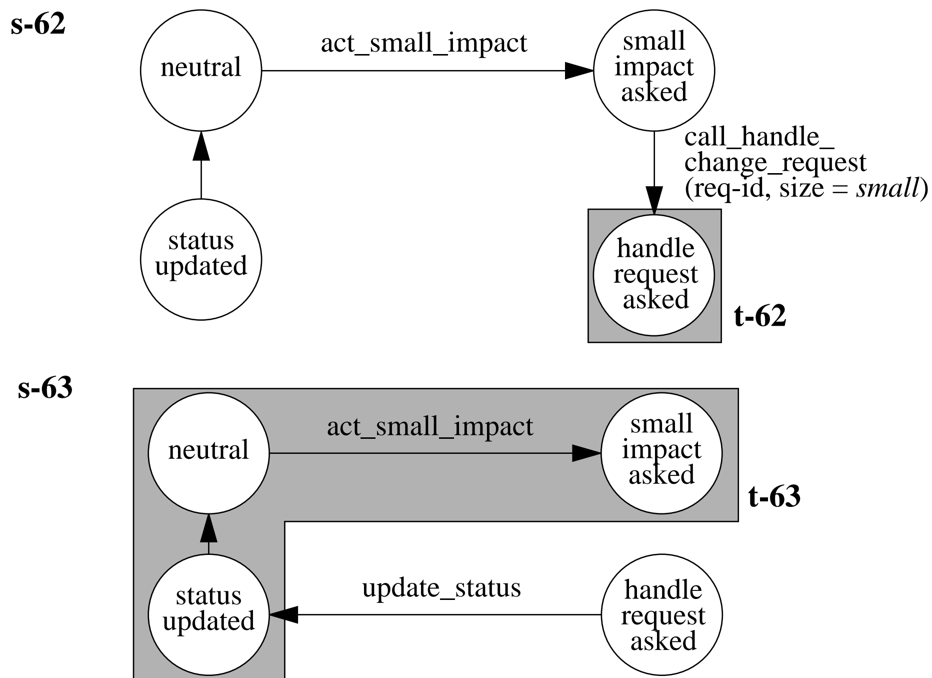


Figure 68. int-small_impact’s subprocesses and traps w.r.t. CABSecretary

CABSecretary can also execute the operation *send_list*. In order to activate this operation CAB-Secretary must be in its state *neutral*. There subprocesses s-64 and s-52 are prescribed to *int-request_for_meeting* and *int-send_list* respectively. If *int-request_for_meeting* has entered its trap t-64, if moreover *int-send_list* has entered its trap t-52, then CABSecretary will make the transition to its state *list send*. There the subprocesses s-65 and s-53 will be prescribed and the list will be send to NewCAB. As soon as *int-request_for_meeting* enters its trap t-65 and *int-send_list* enters its trap t-53, CABSecretary will return to its neutral state. There new requests can be handled and administrated.

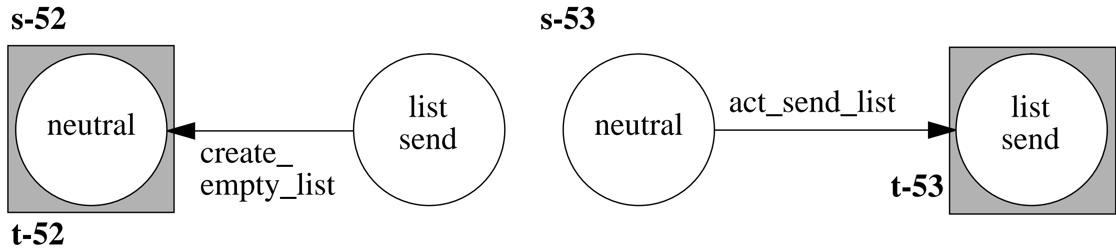


Figure 69. int-send_list's subprocesses and traps w.r.t. CABSecretary

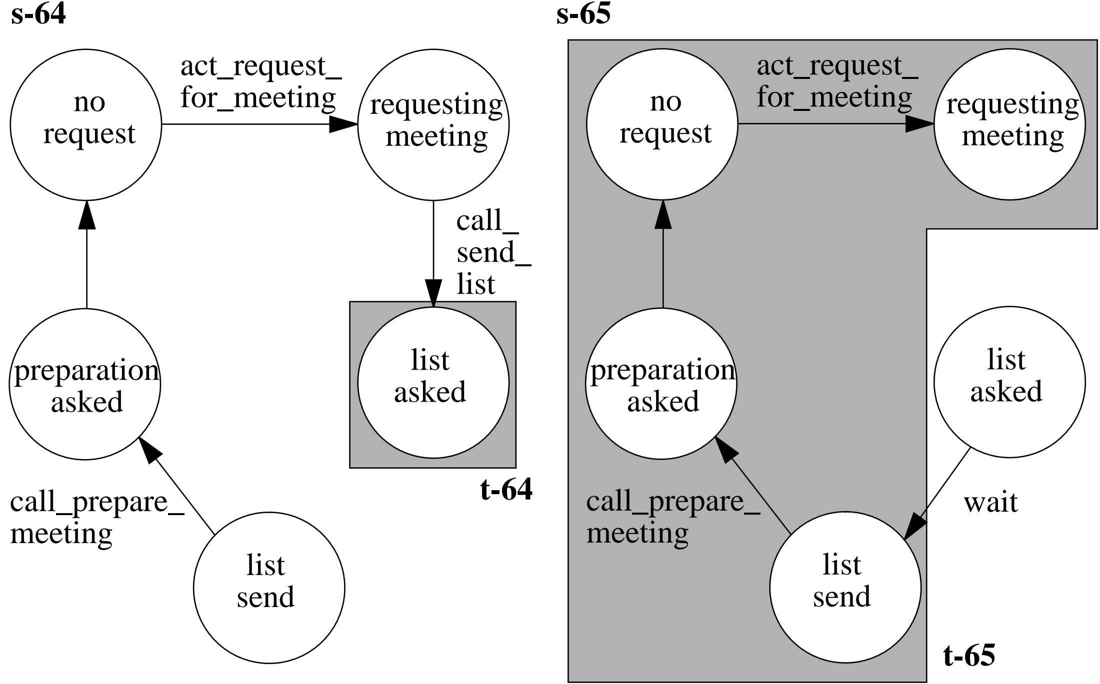


Figure 70. int-request_for_meeting's subprocesses and traps w.r.t. CABSecretary

Figure 71 shows the class CABSecretary as manager of *int-add_to_list*, *int-send_list*, *int-request_for_change*, *int-handle_change_request* (called operations), *int-big_impact*, *int_small_impact* and *int-request_for_change* (calling operations).

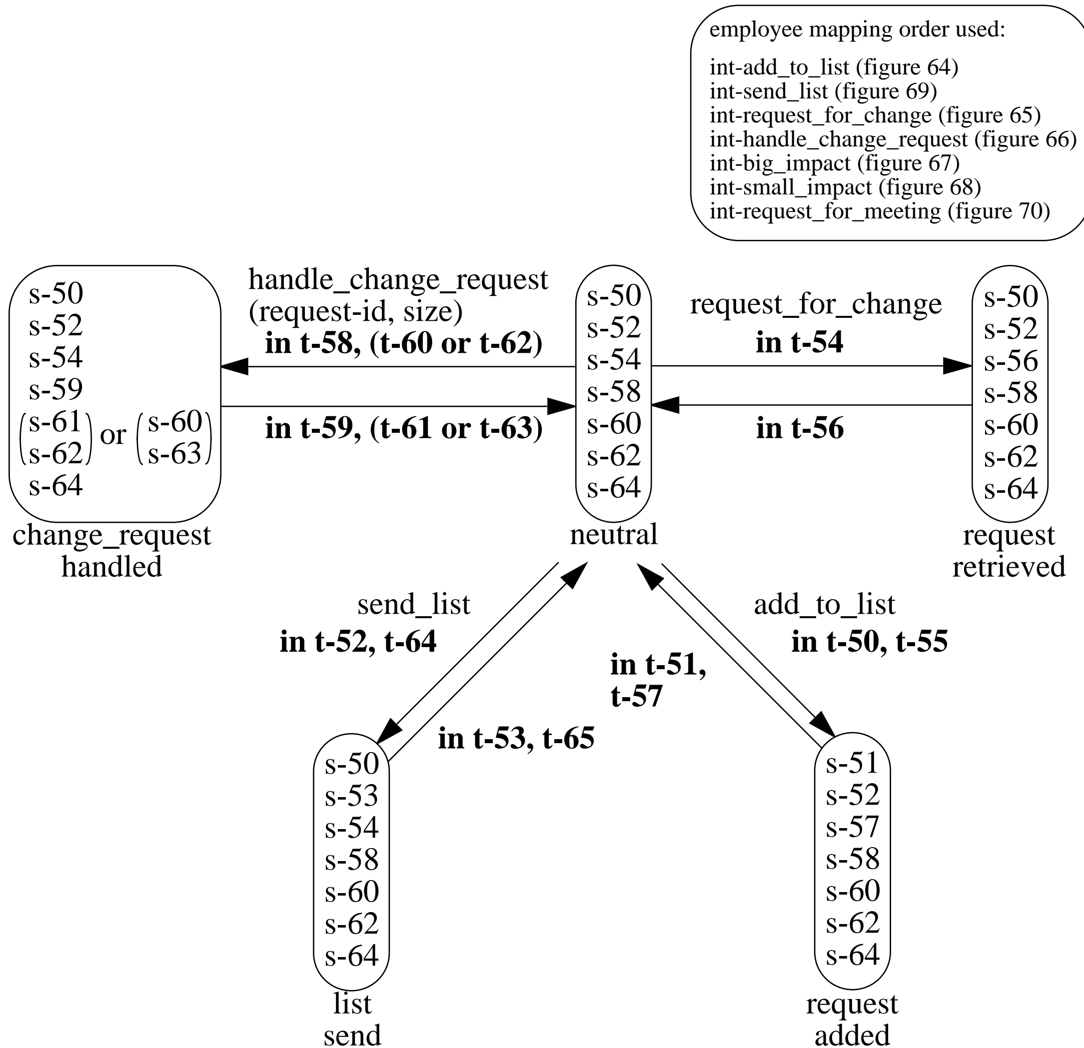


Figure 71. CABSecretary: viewed as manager of 7 employees

In the above figure the logical operator *or* is used to describe the conditions under which the operation *handle_change_request* can be executed. For now the above notation will be used. Details on the used notation will be given in Appendix B.

The second part of the communication specification shows the communication between the manager process Request and its employee processes *int-reject_request*, *int-big_impact*, *int_small_impact*, *int-do_change*, *int-handle_change_request* and *int-do_meeting*. The internal behaviours *int-reject_request*, *int-big_impact*, *int_small_impact* and *int-do_change* belong to the class Request itself. *Int-handle_change_request* and *int-do_meeting* are the internal behaviours of operations that call operations of the class Request.
Keep in mind that, as for every request (i.e. request-id) an instance of the class Request exists, the following will hold for every instance. Request starts in its state *neutral*. If *int-do_meeting* has entered its trap t-74, which means the CAB suggests the request should be rejected, if moreover *int-reject_request* is in its trap t-78, then Request can go to its state *request rejected* and the change-request will be rejected. However if *int-do_meeting* has entered its trap t-75, which means the CAB suggests the change will have a big impact on the software, and also *int-big_impact* is in its trap t-80, then Request can go to its state *big impact estimated* and the

change-request will be handled. Similarly, if *int-do_meeting* has entered its trap t-76, which means the CAB suggests the change will have a small impact on the software, if moreover *int-small_impact* is in its trap t-82, then Request can go to its state *small impact estimated* and the change-request will be handled. In all three states the subprocess s-77 is prescribed to *int-do_meeting*. Note that for every request only one of these three states can be reached, as a request cannot be rejected and accepted at the same time. Moreover a request cannot have both a big and a small impact on the software. When *int-do_meeting* enters its trap t-77 and also *int-reject_request* enters its trap t-79 or *int-big_impact* enters its trap t-81 or *int-small_impact* enters its trap t-83, Request returns to its state *neutral*.
If *int-handle_change_request* has entered its trap t-70, if moreover *int-do_change* has entered its trap t-84, then Request makes the transition to the state *changing software*. As mentioned before the value of the parameter *size* is known as it is passed through via *handle_change_request*. Request will return to its state *neutral* as soon as *int-handle_change_request* has entered its trap t-71 and *int-do_change* has entered its trap t-85. Note that, as we have mentioned before, the order in which the operations are called is not arbitrary, although it may seem so taking into account the interleaved character of the external behaviour of the class Request. When looking more carefully at the model, it is clear that before reaching the state *changing software* one of the operations *big_impact* or *small_impact* has to be executed, as these operations call *handle_change_request*, which in its turn calls *do_change*.

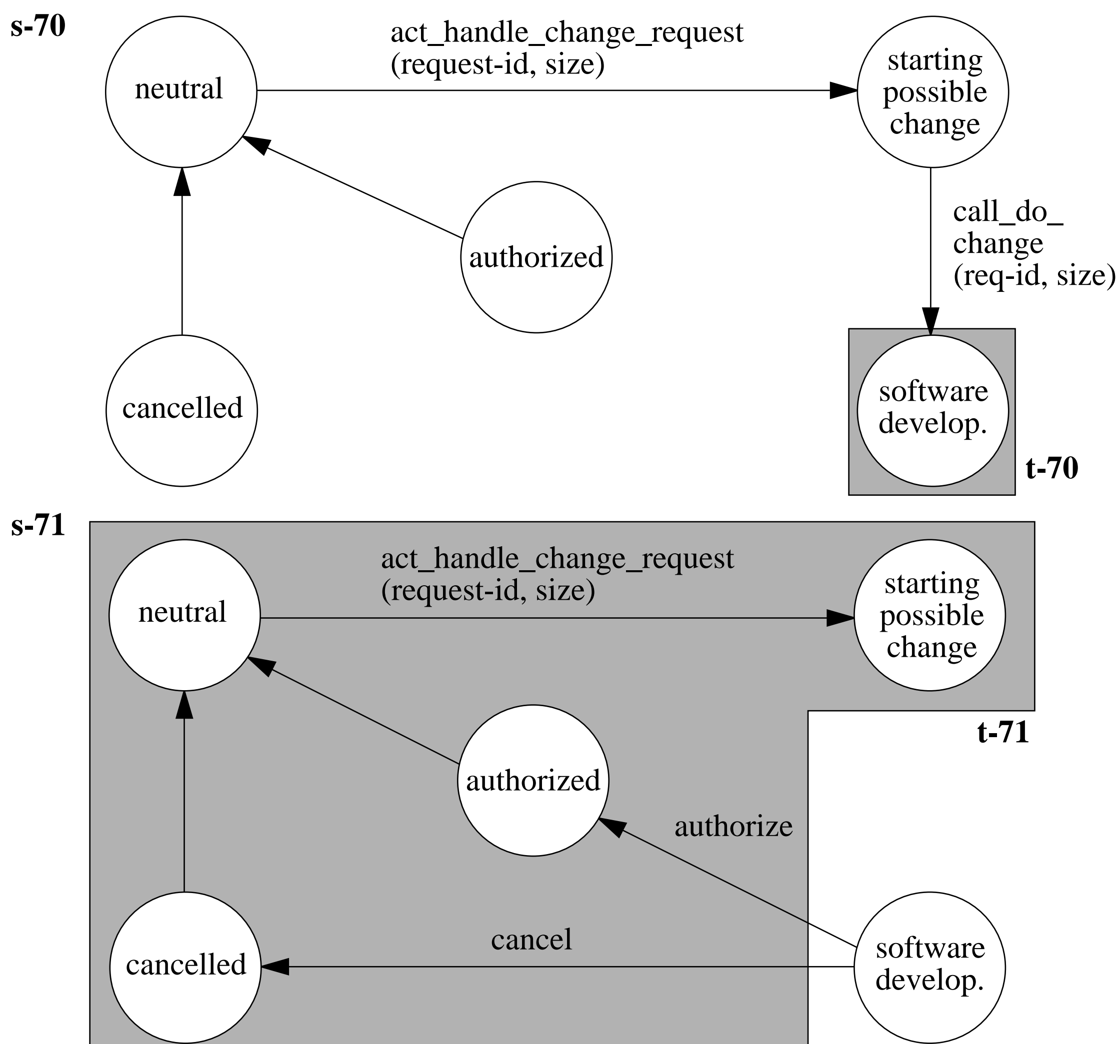


Figure 72. int-handle_change_request's subprocesses and traps w.r.t. Request

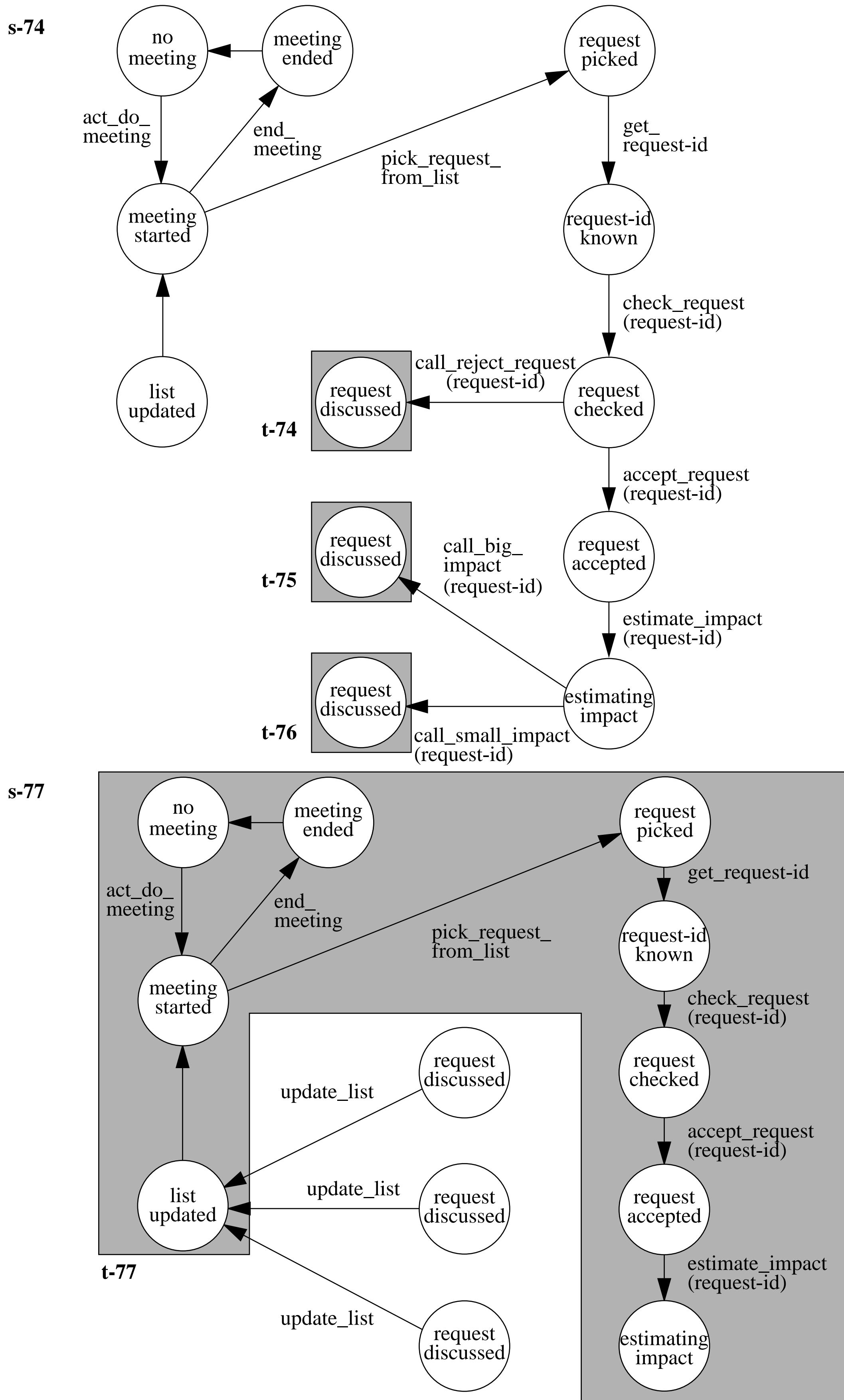


Figure 73. int-do_meeting's (new version) subprocesses and traps w.r.t. Request

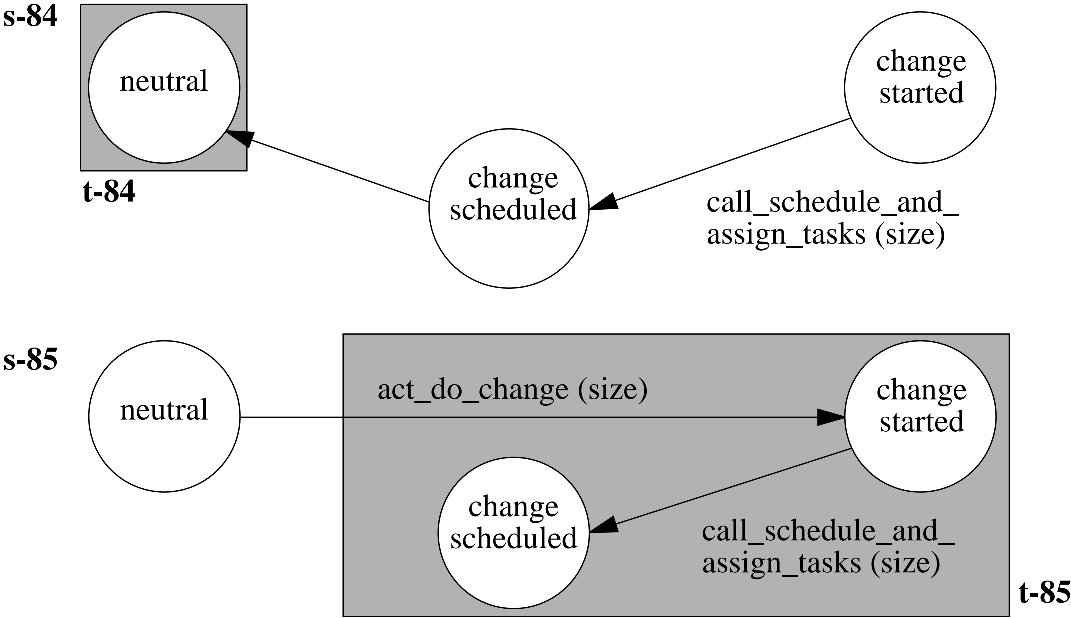


Figure 74. int-do_change's (new version) subprocesses and traps w.r.t. Request

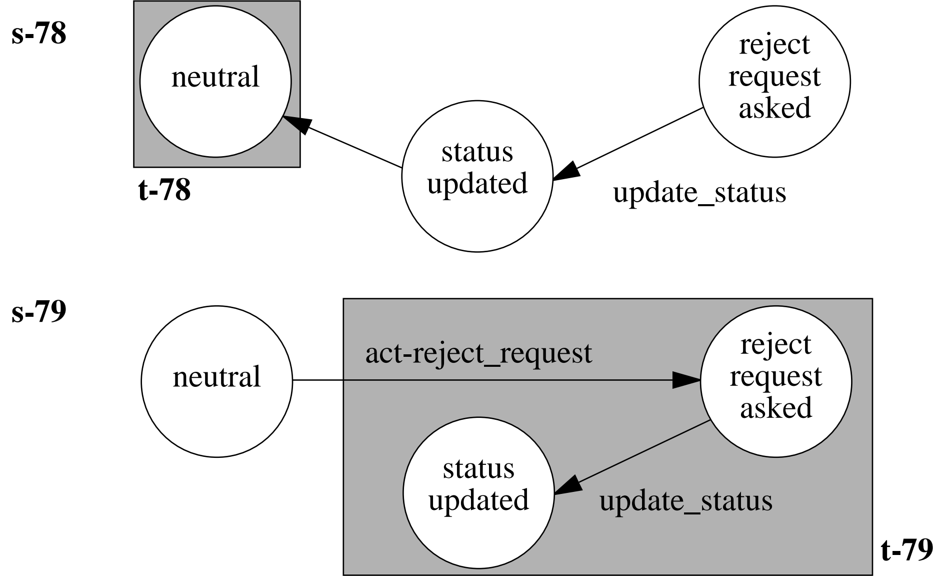


Figure 75. int-reject_request's subprocesses and traps w.r.t. Request

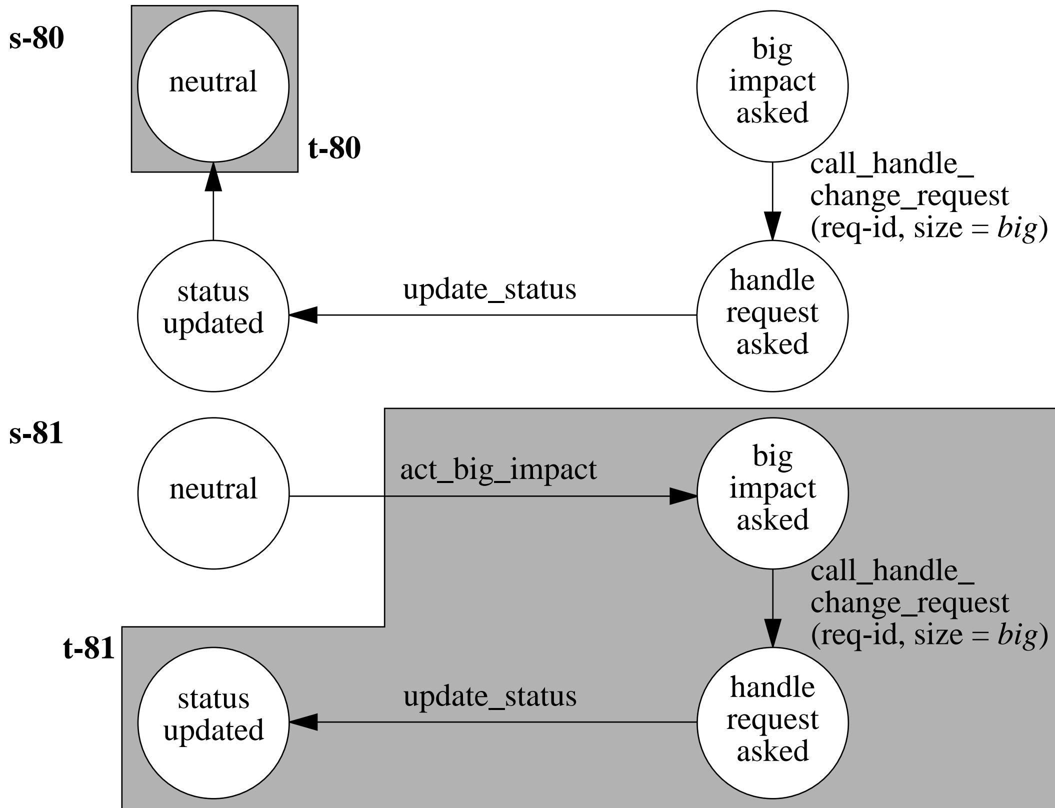


Figure 76. int-big_impact's subprocesses and traps w.r.t. Request

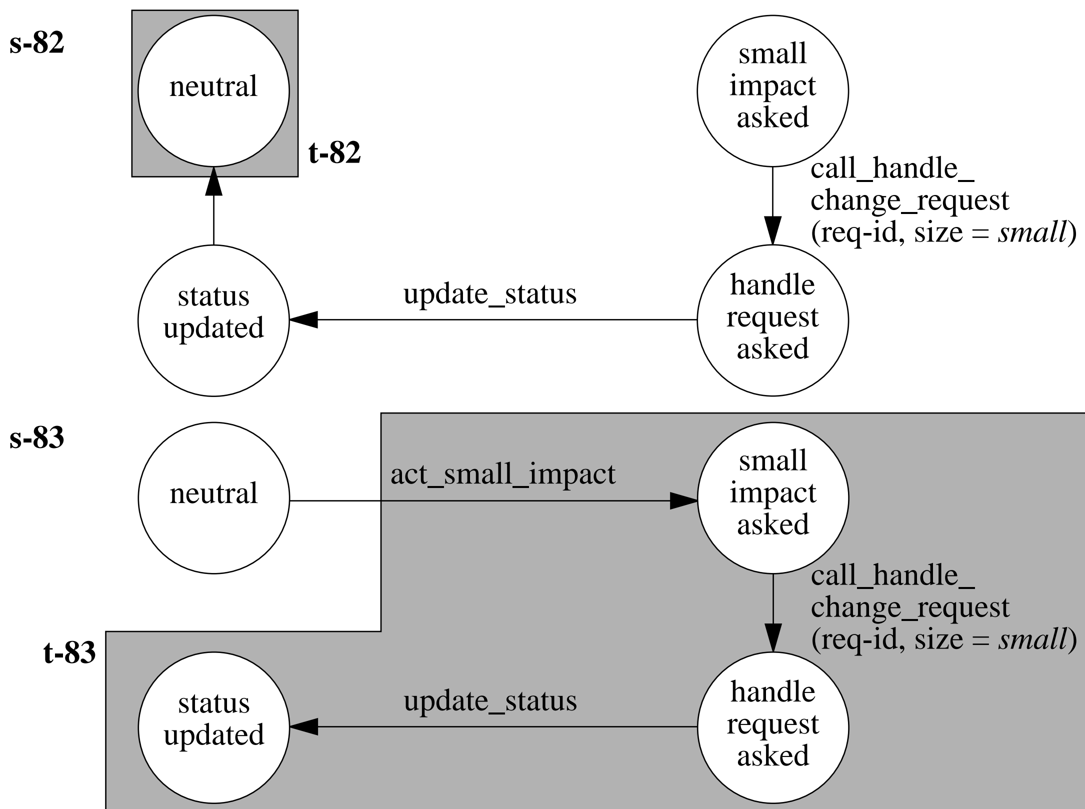


Figure 77. int-small_impact's subprocesses and traps w.r.t. Request

Figure 78 shows the class Request as manager of *int-reject_request*, *int-big_impact*, *int_small_impact*, *int-do_change* (called operations), *int-handle_change_request* and *int-do_meeting* (calling operations). As for every request-id an instance of the class Request exists, there will exist a manager process too for every request-id.

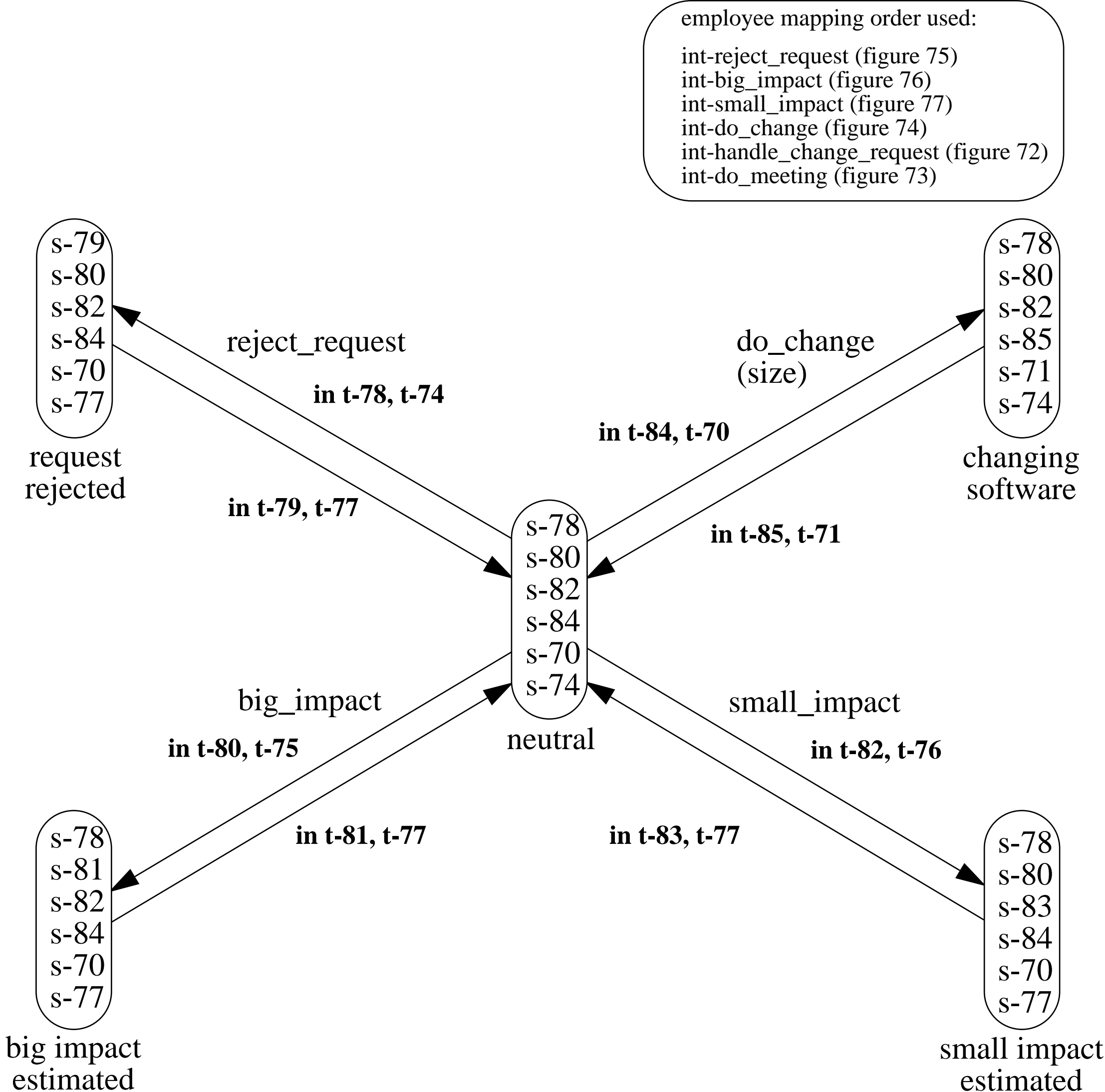


Figure 78. Request: viewed as manager of 6 employees

The third part of the communication specification shows the communication between the man-

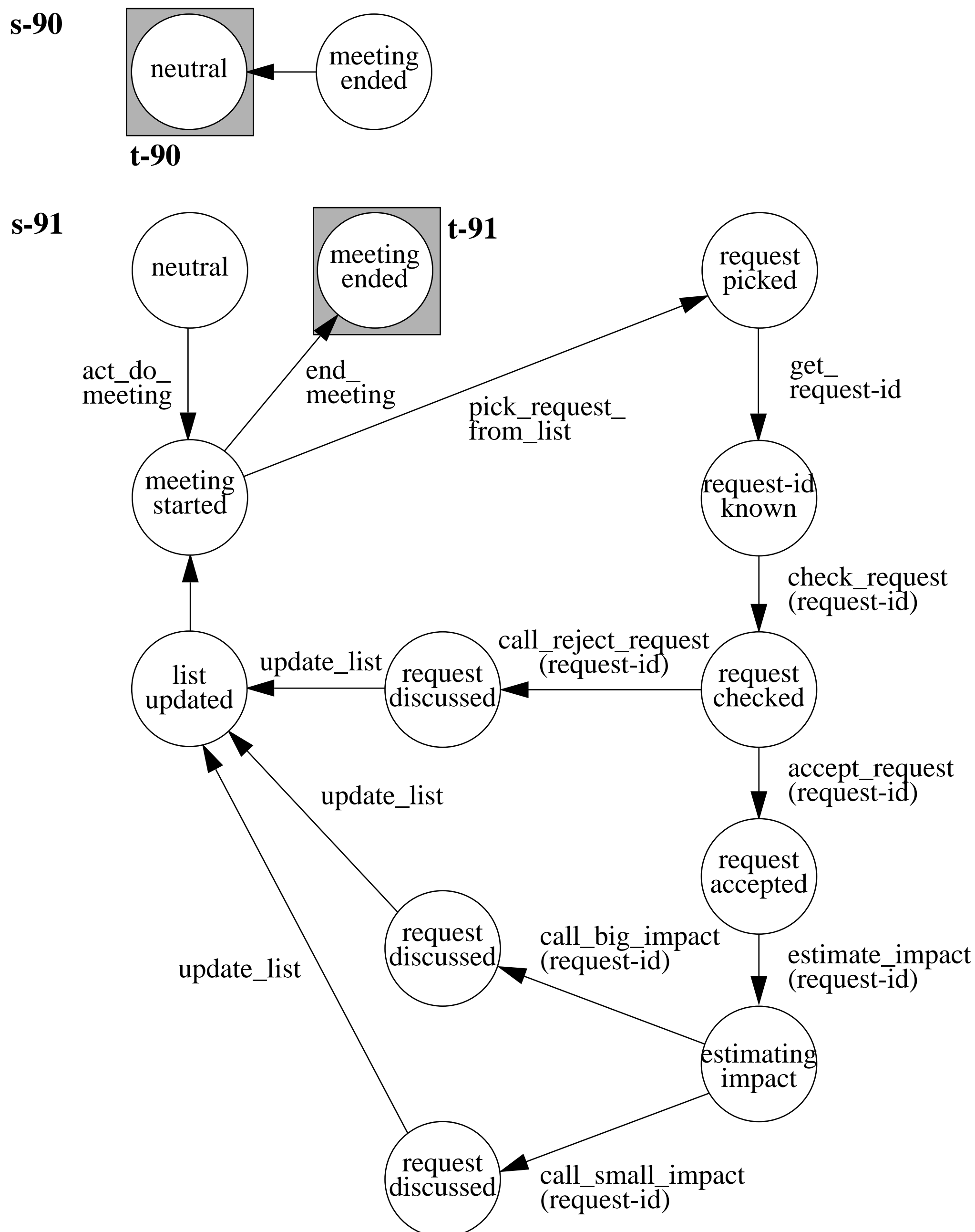


Figure 79. int-do_meeting's (new version) subprocesses and traps w.r.t. NewCAB

ager process NewCAB and its employee processes *int-request_for_meeting*, *int-open_meeting*, *int-do_meeting*, *int-close_meeting*, *int-request_for_change* and *int-prepare_meeting*. The internal behaviours of the export-operations belonging to the class NewCAB are *int-request_for_meeting*, *int-open_meeting*, *int-do_meeting* and *int-close_meeting*. The internal behaviours of operations that call operations of the class NewCAB are *int-request_for_change* and *int-prepare_meeting*. As *int-open_meeting*, *int-close_meeting* and *int-prepare_meeting* do not have to be changed to fit the new model, their subprocesses and traps do not have to be changed either. So the communation between NewCAB and these three employees is the same as the communation between DepCAB and these three employees. Therefore this part of the communication specification will not be repeated here.
Note that the trap-structure of *int-do_meeting* (see Figure 79) deviates from the standards de-

scribed in the previous chapter. *Do_meeting* still is a part of a bigger operation, that has been split into three parts. As these three parts cannot operate simultaneously the subprocesses of the internal behaviour of *do_meeting* have to be sequentialized. This is done by using small traps, because then the operation has to be finished before it can enter its trap and the manager thus makes the next transition only after. The trap-structures of *int-request_for_meeting* (Figure 80) and *int-request_for_change* (Figure 81) also deviate from the standards. This will be explained later.
NewCAB starts in its state *neutral*. When a meeting has been requested, the manager waits for *int-request_for_meeting* to be trapped in its trap t-92, which means a possible previous request has been dealt with, and for *int-request_for_change* to be trapped in its trap t-94, which means a meeting has been requested.

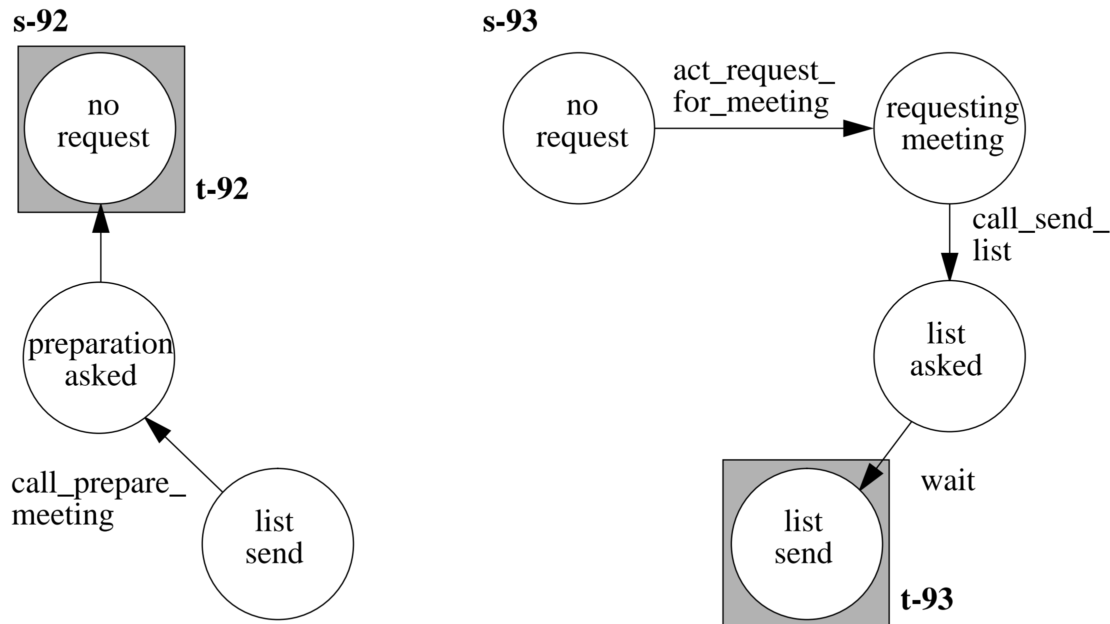


Figure 80. int-request_for_meeting's subprocesses and traps w.r.t. NewCAB

When these traps have been entered, NewCAB transits to the state *meeting requested* and prescribes subprocess s-93 to *int-request_for_meeting*. Also subprocess s-95 is prescribed to *int-request_for_change*. In this state the new requests cannot be placed on a list in order to prevent them from being placed on the already full list that (possibly) has not been sent yet. As soon as *int-request_for_meeting* has entered its trap t-93, which means *send_list* has been activated, and *int-request_for_change* has entered its trap t-95, the state *list available* can be entered. There subprocess s-92 is prescribed to *int-request_for_meeting*, thereby making it possible for ProjectManager to start actual preparations, and subprocess s-96 to *int-request_for_change*. Now CABSecretary can continue to administrate change-requests and place them on the list as far as NewCAB is concerned. Almost instantly *int-request_for_change* will enter its trap t-96. *Open_meeting* can be considered now and NewCAB will transit to the state *meeting opened*. There subprocess s-94 will be prescribed again to *int-request_for_change*. Note though that NewCAB cannot react to trap t-94 until its state *neutral*. However, as mentioned before, trap t-94 will usually be entered only in the neutral state, as we assumed that the list would not grow full during the meeting.
Let us now continue with the manager process. NewCAB is in its state *meeting opened*. When *int-do_meeting* is in its trap t-90, NewCAB can go to its state *meeting* and the requests can be discussed. If *int-do_meeting* has entered its trap t-91, which means the meeting has ended and results with respect to the requests have been established, then *close_meeting* is done and NewCAB can go to its state *meeting closed*. Eventually NewCAB will return to its state *neutral*.

As mentioned before the trap-structures of *int-request_for_meeting* (Figure 80) and *int-*

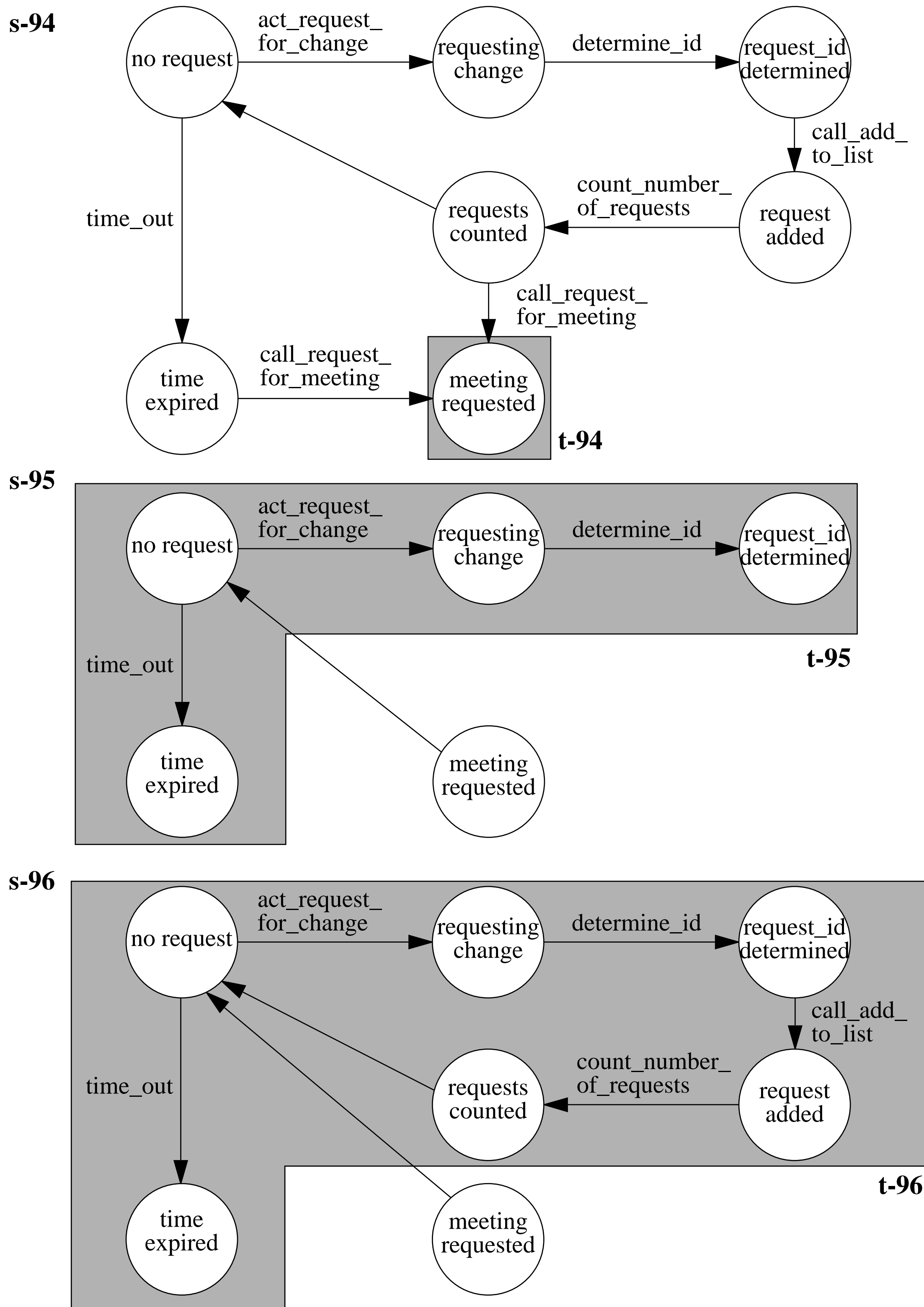


Figure 81. int-request_for_change's subprocesses and traps w.r.t. NewCAB

request_for_change (Figure 81) deviate from the standards. As *int-request_for_meeting* is a called operation one would expect subprocess s-93 to have a large trap. However in this case a small trap is used to assure that the call to *send_list* has been executed before entering the next state of NewCAB. As long as this is not assured (i.e. trap t-93 has not been entered) subprocess s-95 has to be prescribed to *int-request_for_change*. In subprocess s-95 the call to *add_list* has

been removed in order to prevent new requests from being placed on an already full list that (possibly) has not been sent yet. Only after *int-request_for_meeting* has entered its trap t-93, subprocess s-96 can be prescribed to *int-request_for_change* and requests can be put on a (new and empty) list again.
Figure 82 shows the class NewCAB as manager of *int-request_for_meeting*, *int-open_meeting*,

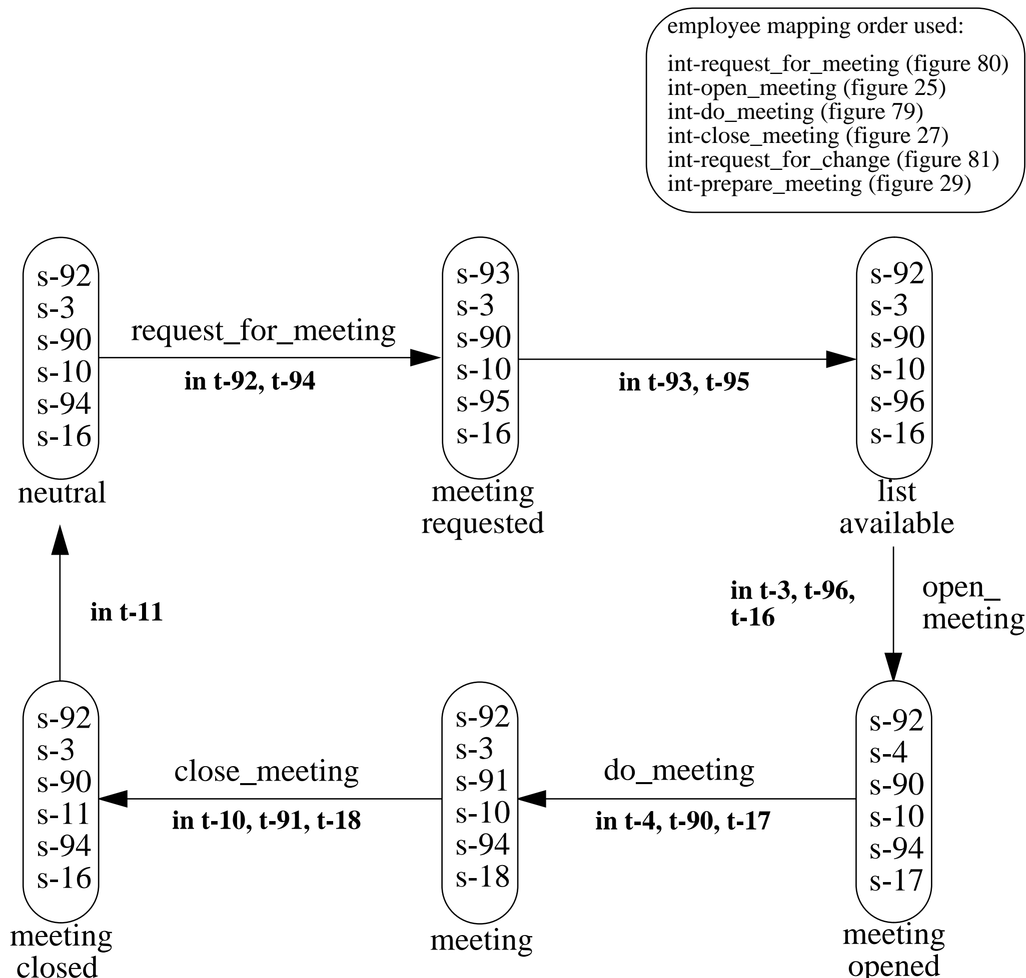


Figure 82. NewCAB: viewed as manager of 6 employees

int-do_meeting, *int-close_meeting* (called operations), *int-request_for_change* and *int-prepare_meeting* (calling operations).

The fourth part of the communication specification shows the communication between the manager process NewDesign and its employee process *int-monitor*, which is the internal behaviour of an operation calling export-operations of the class NewDesign. Subprocess s-100 is the start state of *int-monitor* for the first instance of Design; there is only one monitor per design document but there are many instances of Design for each design document: one instance for every separate version of the very same document (see [1] for a justification of this). In the subprocess s-100, *int-monitor* will be waiting until Design will start the modifications. In subprocess s-101 it will be waiting until Design has closed the modification and in subprocesses s-102 and s-103 it will be waiting until Design has started the review process and until it has reported the review result respectively. After this *int-monitor* will go back to subprocess s-100 (when the review result is *not_ok*) or to the neutral subprocess s-104 (when the review result is *ok*). Note that subprocess s-104 is also the starting state for the other instances of Design. Figures 83 and 84 can also be found in [2, Figure 16] and [2, Figure 23] respectively.

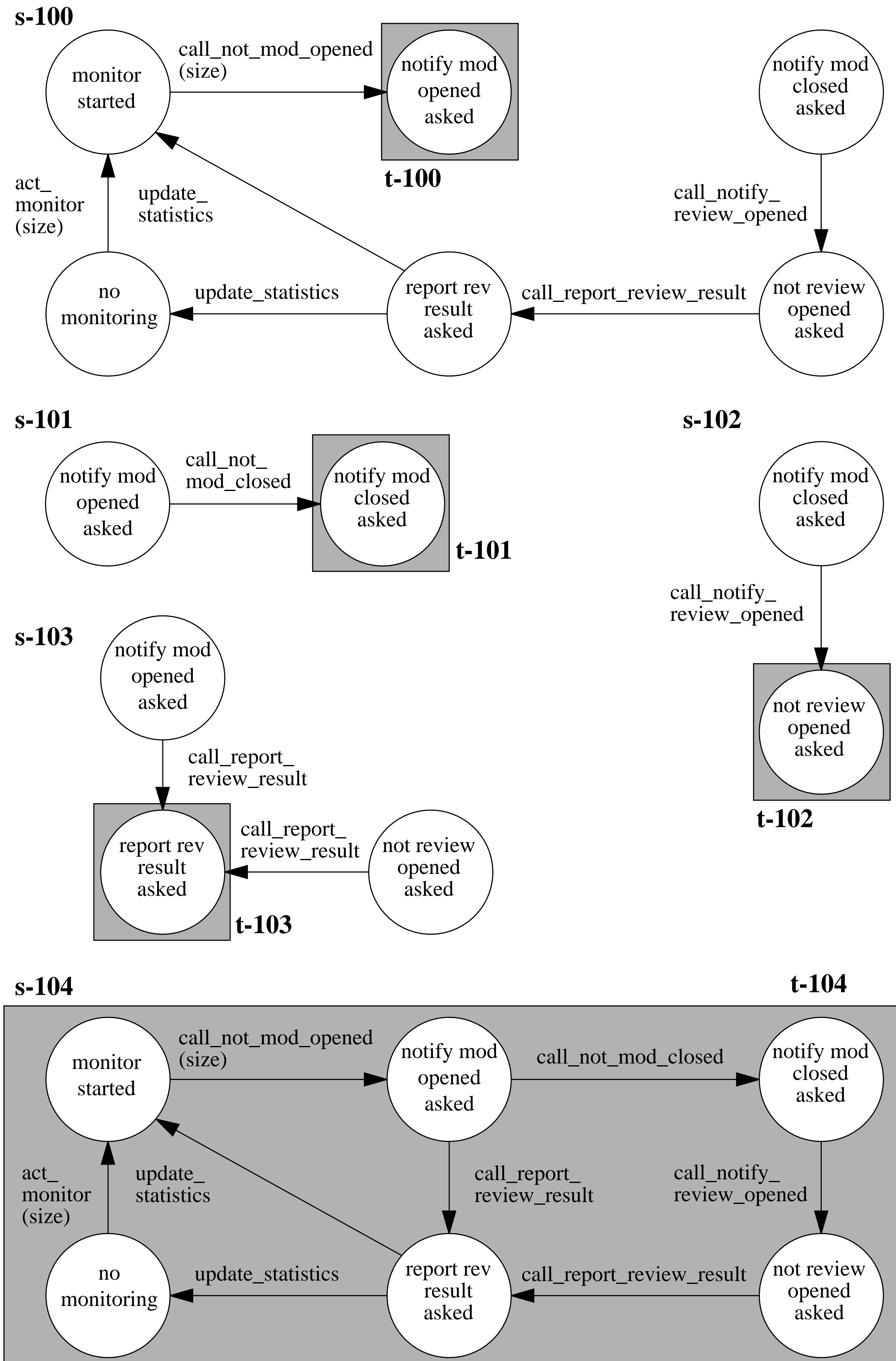


Figure 83. int-monitor’s subprocesses and traps w.r.t. NewDesign

Figure 84 shows the class NewDesign as manager of *int-monitor* (a calling operation).

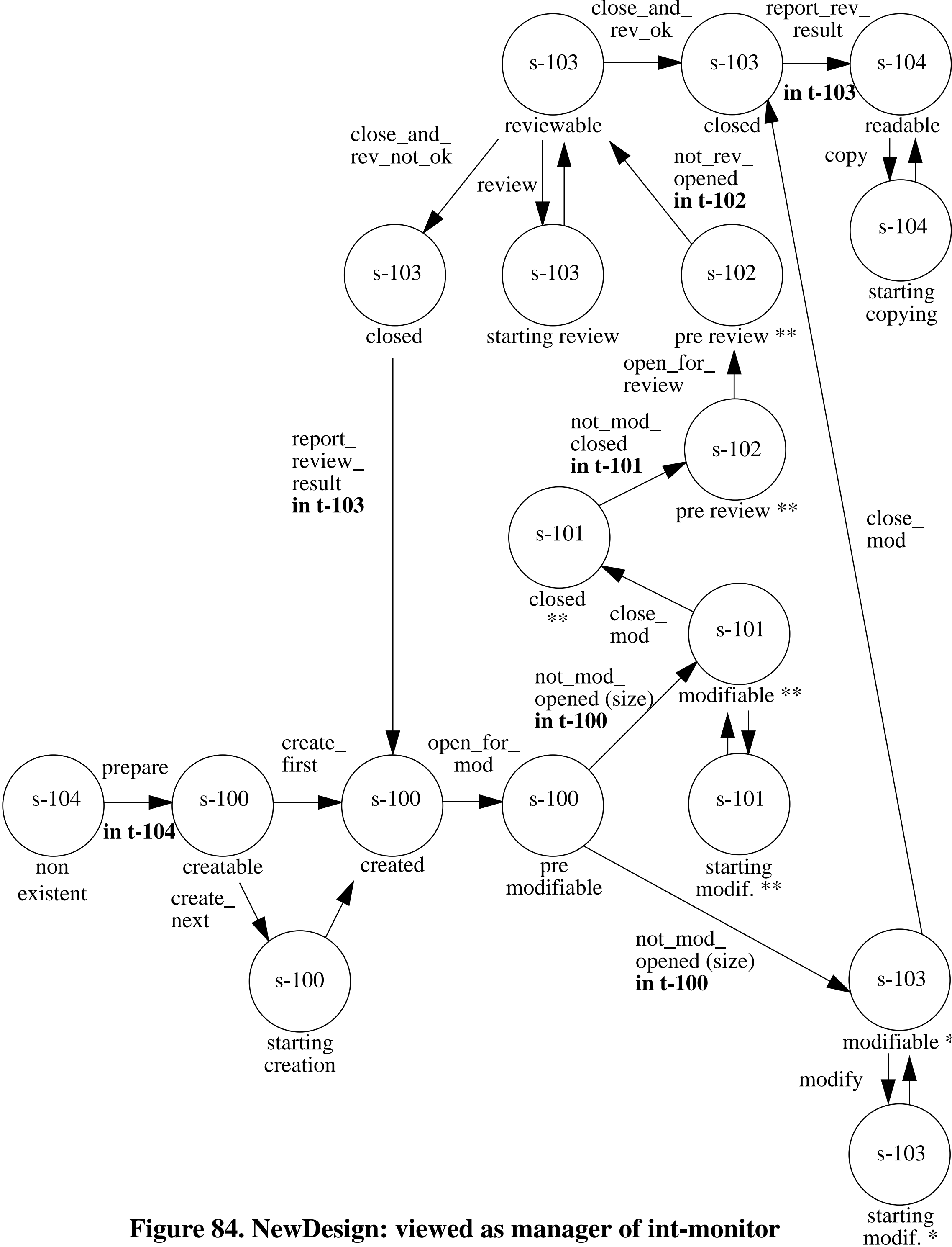


Figure 84. NewDesign: viewed as manager of int-monitor

The communication specification between the manager process ProjectManager and its employee processes also has to be remodelled, as ProjectManager no longer is a manager of *int-request_for_change*, but instead is a manager of *int-request_for_meeting* (which could be seen as an extension of *int-request_for_change* as used in the old model). This change has to be incorporated in the new model. Also *int-do_change* has slightly changed (one transition has been parametrized) and for the sake of completeness its subprocesses and traps will be given. The new, but very simular to the old subprocesses and traps of *int-schedule_and_assign_tasks* and *int-monitor* will not be given here. They are standard.
The fifth part of the communication specification shows the communication between the manager process ProjectManager and its employee processes *int-join_meeting*, *int-leave_meeting*, *int-check_agenda*, *int-receive_confirmation*, *int-prepare_meeting*, *int-request_for_meeting*, *int-open_meeting*, *int-close_meeting* and *int-do_change*. The internal behaviours of the export-operations belonging to the class ProjectManager are *int-join_meeting*, *int-leave_meeting*, *int-check_agenda*, *int-receive_confirmation* and *int-prepare_meeting*. The internal behaviours of operations that call export-operations of the class ProjectManager are *int-request_for_meeting*, *int-open_meeting*, *int-close_meeting* and *int-do_change*. Because the communication between the manager ProjectManager and its employees *int-join_meeting*, *int-leave_meeting*, *int-check_agenda*, *int-receive_confirmation*, *int-prepare_meeting*, *int-open_meeting* and *int-close_meeting* is the same as in the old model, these employees' subprocesses and traps will not be shown again. Also the subprocesses and traps of *int-schedule_and_assign_tasks* and *int-monitor* will not be given here, as the changes in the internal behaviours of these operations do not affect the subprocesses and traps w.r.t. ProjectManager. The subprocesses and traps of these operations will remain standard. So we restrict our attention to the communication between ProjectManager and the calling internal behaviours *int-request_for_meeting* and *int-do_change*.
ProjectManager (Figure 87) starts in its state *neutral*. If *int-request_for_meeting* has entered its trap t-105, which means a preparation has been asked, (and of course *int-prepare_meeting* is in its trap t-22) then *prepare_meeting* can be performed and ProjectManager can go to its state *starting preparations*. In that state subprocess s-106 is prescribed to *int-request_for_meeting*, so that a new request can be made as soon as possible. When ProjectManager is in its state *checking agenda* and *int-request_for_meeting* has entered its trap t-106 (and of course *int-check_agenda* has entered its trap t-21 and *int-prepare_meeting* has entered its trap t-23), ProjectManager can return to its state *neutral*. After that, when *int-do_change* has entered its trap t-107, ProjectManager can make the transition to the state *starting schedule*. As soon as *int-do_change* enters its trap t-108 ProjectManager will return to its neutral state. There other changes can be scheduled or new meetings can be prepared.

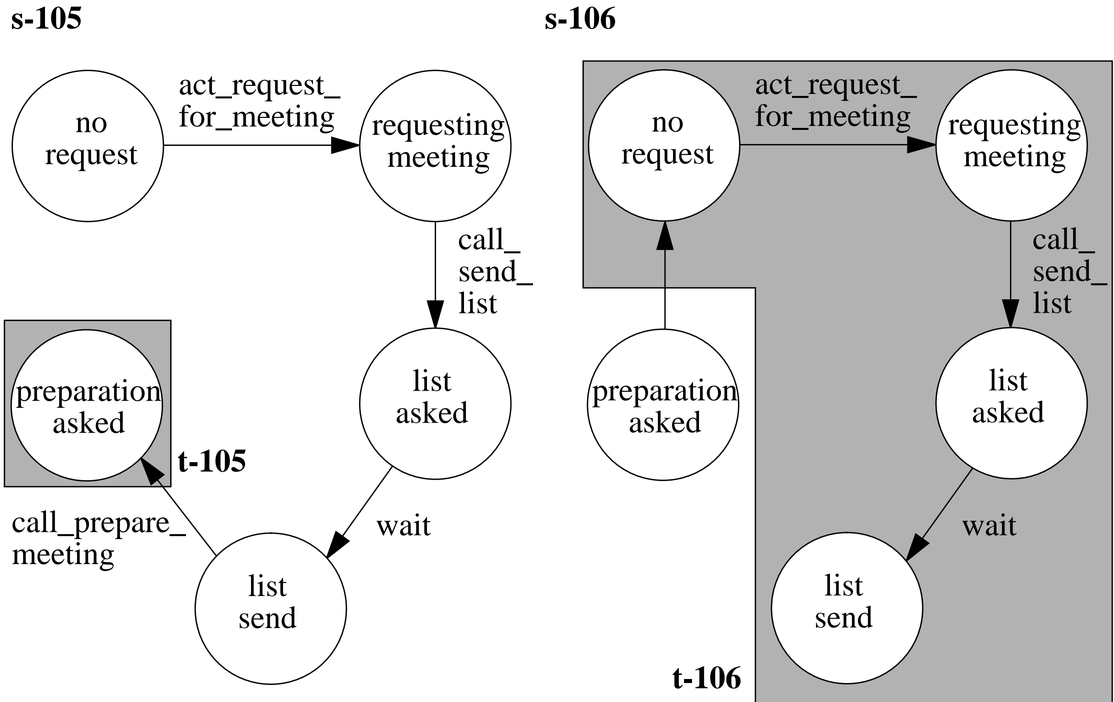


Figure 85. int-request_for_meeting's subprocesses and traps w.r.t. ProjectManager

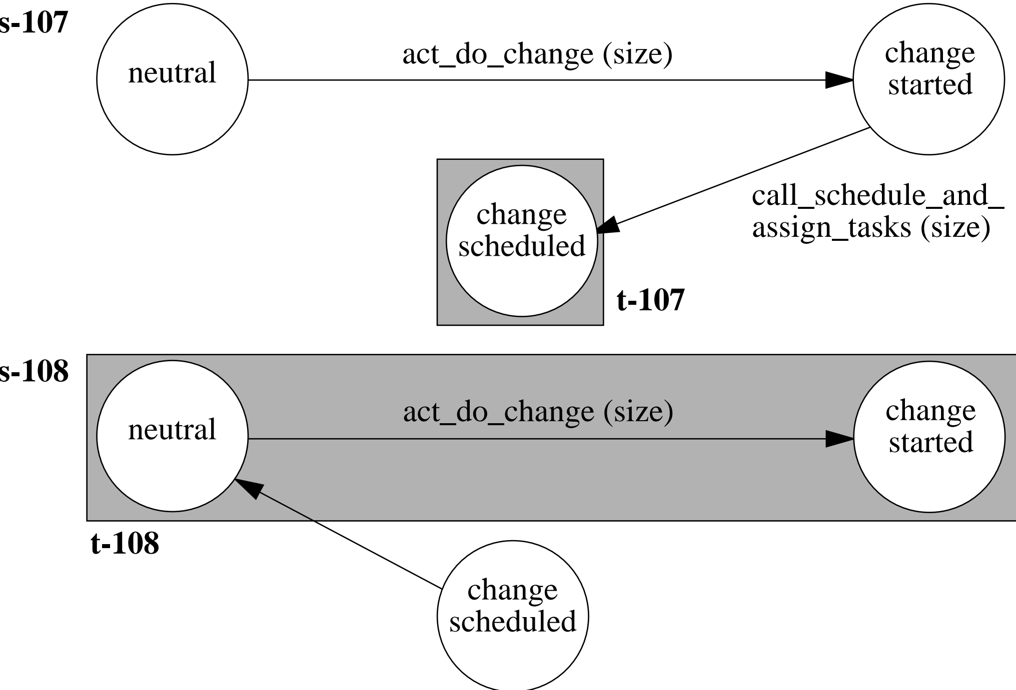


Figure 86. int-do_change's subprocesses and traps w.r.t. Projectmanager

Figure 87 shows the new version of Projectmanager as manager of *int-join_meeting*, *int-*

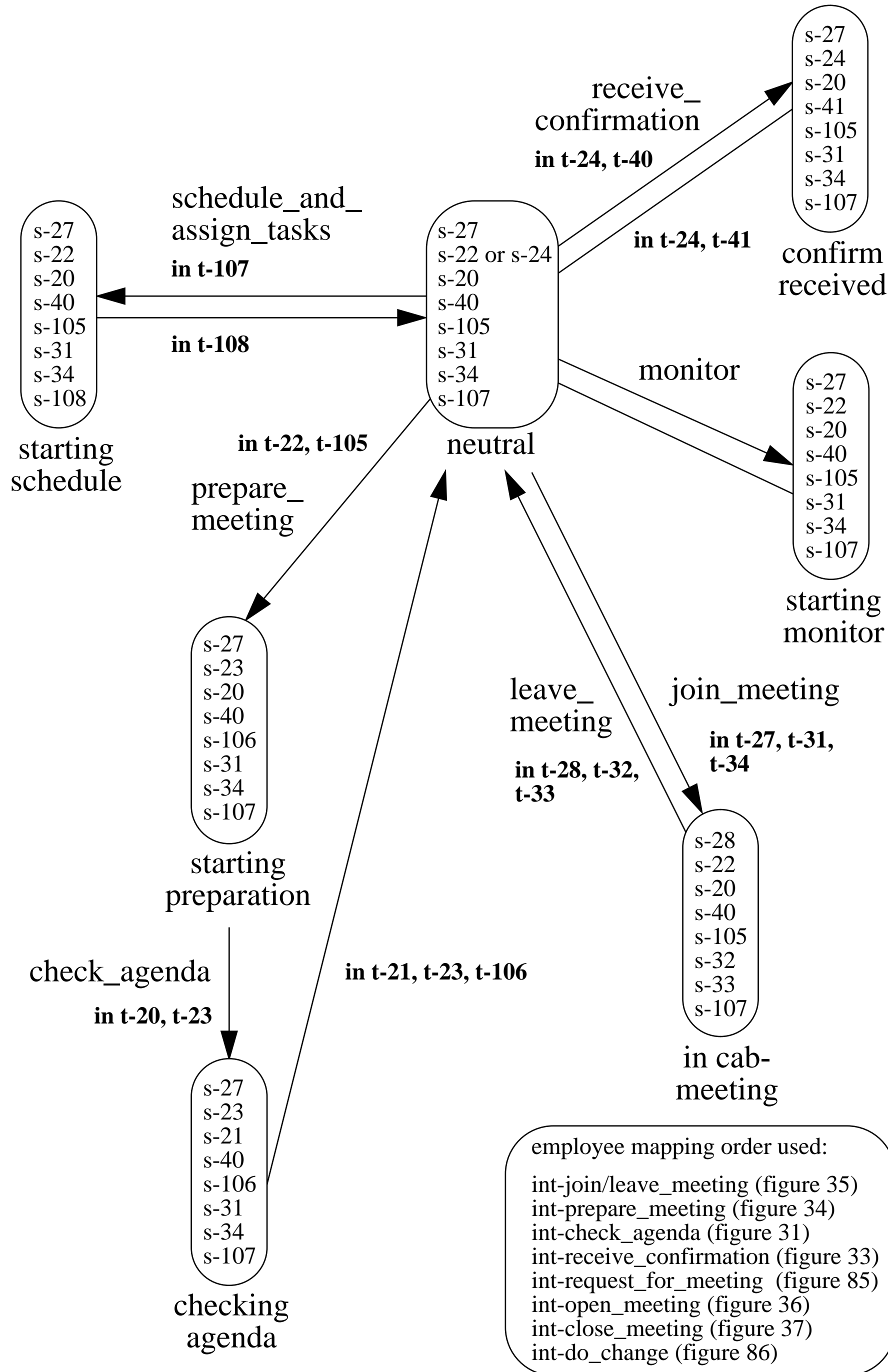


Figure 87. ProjectManager (new version): viewed as manager of 8 employees

leave_meeting, *int-check_agenda*, *int-prepare_meeting* (called operations), *int-request_for_meeting*, *int-open_meeting*, *int-close_meeting* and *int-do_change* (calling operations).
Note that as the external behaviours of DesignEngineer and QAEngineer and the internal behaviours of the operations of both classes remain unchanged, the subprocesses and traps of these operations w.r.t. these external behaviours remain unchanged as well. So in the new model the communication specification between the manager process DesignEngineer and its employees as well as the communication specification between the manager process QAEngineer and its employees is the same as in the old model. Which concludes our discussion of the communication of the new model.

5 WODAN, a method to describe change

5.1 Introduction

In Chapter 3 a simple model for the Change Management software process was developed. Later on, in Chapter 4, this model had been changed in order to meet new requirements. Of course this new model will have to be enacted at some point in time. However switching from one model to another is not a trivial thing to do. So we have to model the changing of the Change Management software process model as well. For this purpose an extra manager process, called WODAN, is introduced, which manages the change from the old model to the new model. WODAN, which stands for What Ought to be Done As Necessary, was first introduced in [2]. This section will just be a brief introduction into WODAN, more information can be found in [2, Chapter 3]. The actual change from the old model to the new model will be modelled in the next chapter.

In order to describe the way WODAN works, some terms have to be defined. The internal behaviour of an operation is called *an internal process* and the STD describing this behaviour is called *the internal process description*. Similarly the external behaviour of a class is called *an external process* and the corresponding STD is called *the external process description*.

When a software process model evolves, the behaviour of (at least a part of) the model has to be changed. The way this can be done is very similar to the way the behaviour of an internal process changes when it makes a transition from one subprocess to another. Consider a subprocess as a behaviour restriction, for a subprocess implies the restriction that the internal process may only behave in accordance to those states and actions imposed by the subprocess. Then the transition from one subprocess to another subprocess is the same kind of change as wanted for evolving software process models. This evolution of the software process model can be considered as being a transition from one evolution stage to another evolution stage. So there is an analogy between evolution stages and subprocesses.

One could change a software process model by viewing its external and internal process descriptions as evolution stages, and therefore as being subprocesses of some larger, not explicitly defined processes. These not explicitly designed external and internal processes combine the various behaviours (past, current and future) of the software process model during all possible evolution stages (EVS's) and will be called *anachronistic* external and internal processes.

WODAN will be the manager of all (not explicitly designed) anachronistic external and internal processes. WODAN is an (extra) manager process used to formalize the change of a software process model during enaction, in this case the change from the old Change Management software process model to the new and extended Change Management software process model. WODAN normally stays in the same state, just prescribing the current evolution stage, i.e. the current external and internal process descriptions. As mentioned before they are subprocesses of the anachronistic external and internal processes. When a change in the model has to be made, WODAN can go to a state which for example is called *changing the model*; when WODAN is in this state, the new external and internal process descriptions can be designed. Moreover, WODAN can also design new class descriptions when the static structure of the model has to be changed due to extra requirements (in our example: the classes CABSecretary and Request have been added). After the new model has been designed, WODAN will go to the state prescribing the new process descriptions. Sometimes this can be done in one step, sometimes intermediate steps are necessary.

Adding or removing processes within Socca is possible since the processes are subprocesses of the anachronistic processes. When a process E_1 has to be added, one could say that the anachronistic process E_A of which E_1 is a subprocess, already existed from the very beginning. However WODAN was prescribing a nearly empty subprocess of it before the process E_1 was

necessary. This nearly empty subprocess consists of a single state together with one transition from this state to itself. When WODAN prescibes the process E_1, E_A will transit from the single state of the nearly empty subprocess to any possible starting state of E_1. Also a process E_1 can be removed during evolution by prescribing a similar nearly empty subprocess of the anachronistic process E_A of which E_1 is a subprocess with respect to WODAN. In this case E_A will transit from any state of E_1 to the single state of the nearly empty subprocess. Such a nearly empty subprocess will be called the NULL process, or shorter NULL. This will also be used as a convention when designing WODAN to introduce new processes or to remove old processes; in the states of WODAN where the process did not exist yet or has been removed already, the NULL process will be prescribed.

5.2 Types of change

The changes made to a software process model to achieve evolution, can be split up in different types of change; they range from a relatively simple change to more complicated forms of change. The following two types of change [2] can be distinguished when changing the Change Management software process model (modelled in Chapter 6):

1. Add or remove processes and change the strategies and subprocesses of other processes. The following processes have been removed: MainCAB and DepCAB. The processes that have been added are: Request, *int-reject_request*, *int-small_impact*, *int-big_impact*, CAB-Secretary, *int-handle_change_request*, NewCAB and *int-request_for_meeting*.
2. Do not add or remove processes, only change strategies and subprocesses and add or remove states and transitions. In the new model states and transitions have been added to Design in order to create NewDesign. The same has been done to *int-request_for_change*, *int-do_meeting*, *int-monitor* and implicitly to all internal behaviours in which transitions have been parametrized (e.g. in *int-do_change*) in order to create the new versions. Also the subprocesses of these processes w.r.t. their managers have been changed. Strategies have been changed too. In the new model some of the internal behaviours will be managed by another manager (possibly a newly introduced process), e.g. *int-do_change* was managed by DepCAB and in the new model it will be managed by Request.

As mentioned above both types of change are used to model the change of the Change Management software process model.

5.3 Problems as a consequence of change

When changing the Change Management software process model some problems arise. Adding a parameter *size* does not cause a lot of problems. Most problems have to do with the fact that in the new model more than one request per meeting should be handled. This change affects a lot of processes, is responsible for the removal and addition of processes and therefore causes many poblems. The following problem is very common:

- A process is within EVS1 in a state without a corresponding state in the process prescribed within EVS2. This problem can arise for example with *int-request_for_change* and *int-do_meeting*.

In order to present a solution to this problem, the term *interrelated* has to be defined first:

- Let P_A be an anachronistic internal or external process with two subprocesses P_j and P_k, with P_j the internal or external process prescribed during EVS_j, P_k the internal or external process prescribed during EVS_k and k>j. So EVS_k is an evolution phase after EVS_j. Then the process P_k will be called *interrelated* with the process P_j.

- Let P_A, P_j and P_k denote the same processes as above. Furthermore, let X_1 be a state of P_A which existst in both subprocesses P_j and P_k of P_A. The state X_1 in P_k will be called *interrelated* with the state X_1 in P_j. So in fact, we will regard this same state as two different (but interrelated) states in the interrelated subprocesses.

Now the term *interrelated* has been defined. Again let P_A and P_j denote the same processes as above and let P_{j+1} be the subprocess of P_A prescribed during EVS_{j+1}. The problem to be solved is that the process P_A must go from EVS_j, where subprocess P_j of P_A is prescribed, to EVS_{j+1}, where subprocess P_{j+1} of P_A is prescribed. As long as there exist interrelated states in P_j and P_{j+1}, these states can be used as a trap w.r.t. WODAN. Only when P_j is in one of these states, P_A can transit to the next evolution stage. If there are no interrelated states in P_j and P_{j+1}, an intermediate subprocess $P_{j'}$ has to be created. Such an intermediate subprocess of P_A contains some states of the subprocess used in EVS_j and some states of the subprocess used in EVS_{j+1}. The interrelated states in P_j and $P_{j'}$ are used as a trap to the intermediate phase, the interrelated states in $P_{j'}$ and P_{j+1} are used as a trap to the next evolution stage. This general method will always work. Note that in this case WODAN will have to contain an intermediate state in which $P_{j'}$ is prescribed. However a new problem is introduced:

- Changing one process may conflict with changing another process. This can occur for instance when both the external behaviour of a class and an internal behaviour of an operation of that class are changed at the same time. The internal behaviour of the operation, when it has already entered its trap w.r.t. WODAN, could affect the external behaviour of the class in such a way that some states and possibly its trap w.r.t. WODAN could not be reached anymore. If for instance *int-do_meeting* and DepCAB would have been changed simultaneously (Figure 105 and Figure 93 would be prescribed respectively), it is possible that DepCAB could not leave its state *meeting opened* anymore and consequently could not reach its neutral state, which is its trap w.r.t. WODAN. The same problem arises when one would change *int-prepare_meeting* and DepCAB concurrently.

Therefore it is important to keep in mind that sometimes processes have to be changed in a specific order. This implies that WODAN sometimes will have to contain more than one intermediate state. Another problem is the following:

- When a process is removed (temporarily or permanently), its partial side-effects up to the actual removal have to be rolled back. Another way is to delay the removal by using a trap which only contains the states that can be reached without causing side-effects, e.g. a neutral state. This problem occurs for instance when DepCAB has to be removed; some requests still could not have been handled and some meetings could not have been finished yet. Therefore DepCAB can be removed only after these two problems have been coped with.

6 Changing the software process model using WODAN

6.1 A setup for WODAN

In order to model the change of the Change Management software process model, WODAN has to be designed. WODAN will manage the evolution from the old model (Chapter 3) to the new model (Chapter 4) via a state in which the new model is being designed. Some intermediate states have to be used to reach the final state. In our case WODAN will consist of six states.

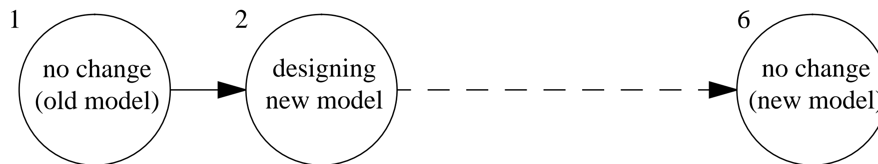


Figure 88. global external behaviour of WODAN

The six states of which WODAN consists are:

1 No change is made. The (old) model, described in Chapter 3, can be enacted as usual. This is EVS1.

2 In this state the new processes, described in Chapter 4, are being designed. Also the intermediate processes are being designed. These intermediate processes will be described in section 6.2. Everything still is enacting in the old way.

3 The new and the intermediate processes have been designed. As requests should not be received and handled in the old way anymore, MainCAB and DepCAB will be changed first together with the export-operation *request_for_change*. In order to finish what has been started, intermediate versions of MainCAB, DepCAB and *int-request_for_change* will be prescribed. Note that in this state *int-request_for_change* cannot be called. This implies that incoming requests cannot be received in this state of WODAN. However the traps are chosen in such a way that WODAN does not have to remain very long in this state.

4 In this state of WODAN MainCAB will be removed, as it has no function anymore. An intermediate version of CABSecretary called TempCABSecretary will take over the task of receiving the requests. Therefore TempCABSecretary will be the manager of an intermediate version of *int-request_for_change*. In this intermediate version a list of requests is made, but *int-prepare_meeting* cannot be called. So new meetings cannot be prepared yet. Also in this state of WODAN an intermediate version of DepCAB will be prescribed. Whenever a meeting has been started, i.e DepCAB has reached its state *meeting opened*, the meeting can be finished. In every other case the (old) request that would have caused a meeting will be placed on a list, just like the new requests. Note that both the old and the new requests are placed on the same list. In order to place the old requests on the list two things will happen. First of all, DepCAB will get a temporary export-operation *put_request_on_list*, which places the old request on a list. Also *int-prepare_meeting*, the operation that in fact coordinates the behaviour of DepCAB, will be changed in such a way that not only *do_meeting* and *close_meeting* can be called (in order to finish a meeting that has already been started), but also *put_request_on_list*. Second, every CABMember will get a temporary export-operation *cancel_meeting*, used when the preparations are being cancelled and the meeting had not been started yet but already had been confirmed. Only if every (old) request has been handled entirely or has been placed on a list, WODAN will transit to its next state.

5 In this state of WODAN the new model will be enacted almost entirely. However there are two operations of which the final versions cannot be prescribed yet: *do_meeting* and

prepare_meeting. To these operations intermediate versions are prescribed. These intermediate versions could not be prescribed in the previous state of WODAN, as they would conflict with the other changes.

6 The new model has been prescribed completely, also the final subprocesses of *int-do_meeting* and *int-prepare_meeting* have been prescribed here. Everything can enact in the new way now. This is EVS2.

6.2 Designing WODAN to manage the change

The first prescriptive step, from the old model to the first intermediate phase, is modelled in Figure 89. As mentioned before in the set-up only three processes are affected in this step.

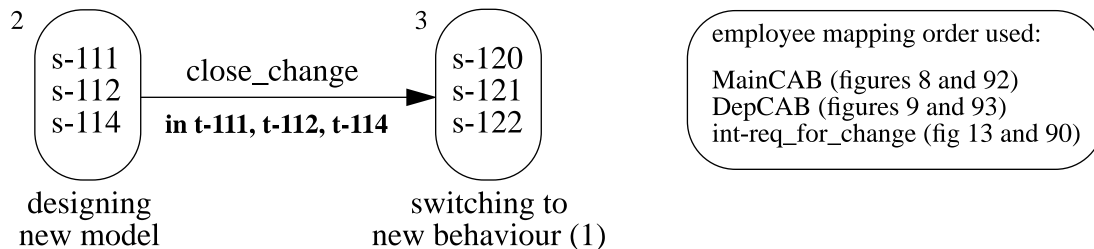
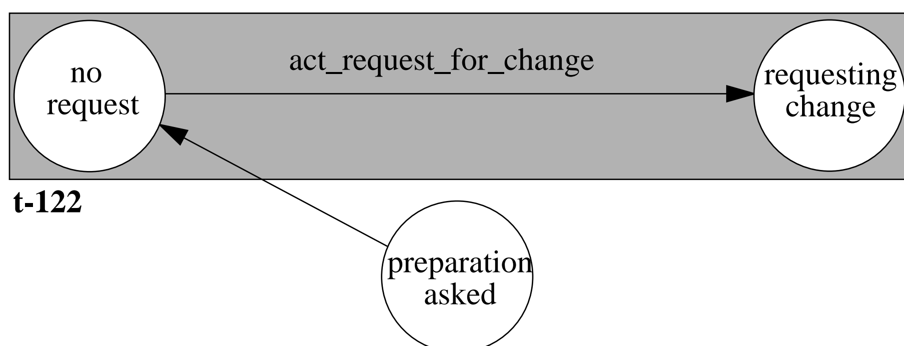


Figure 89. first prescriptive step of WODAN

As soon as the new model has been designed, WODAN will transit to its state *switching to new behaviour (1)*. Note that MainCAB, DepCAB and *int-request_for_change* were already waiting in their traps. WODAN will prescribe subprocesses s-120 and s-121 to MainCAB and DepCAB respectively, as MainCAB has to finish what it has started. Also subprocess s-122 will be prescribed to *int-request_for_change*, as the incoming requests should not be handled in the old way anymore. In fact incoming requests are not being handled at all in this state of WODAN, as the operation *request_for_change* cannot be called here, because the transitions labeled with *(dep-)request_for_change* have been removed from the external behaviours of MainCAB and DepCAB. However the traps necessary for WODAN to be entered in order to transit to its next state *switching to new behaviour (2)*, are entered almost instantly. Note that the state-action interpreter of MainCAB w.r.t. *int-request_for_change* has been changed. To all other processes that eventually have to be changed, the subprocesses used in the old model are still being prescribed.



w.r.t. WODAN this is subprocess s-122

Figure 90. the first intermediate phase of int-request_for_change

The subprocesses and traps of the first intermediate phase of *int-request_for_change* w.r.t. CABSecretary can be found on the next page.

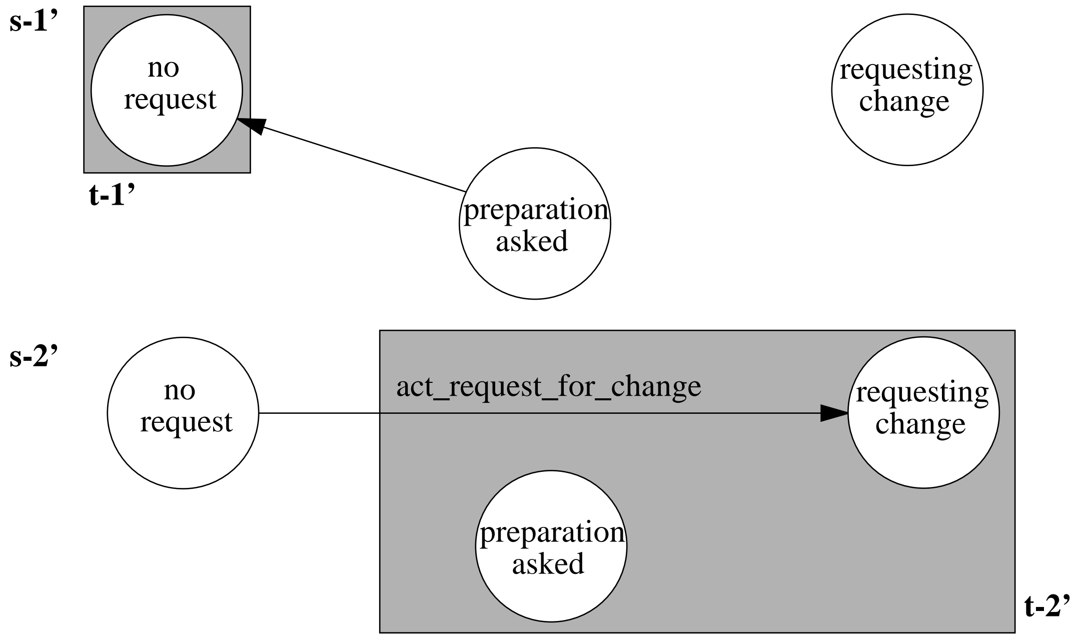
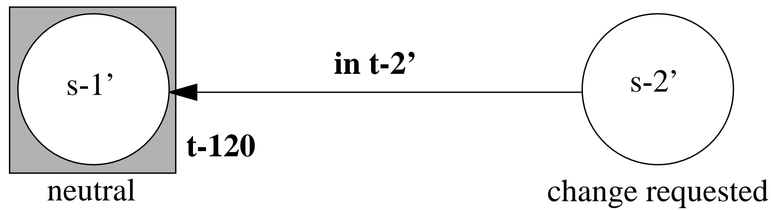
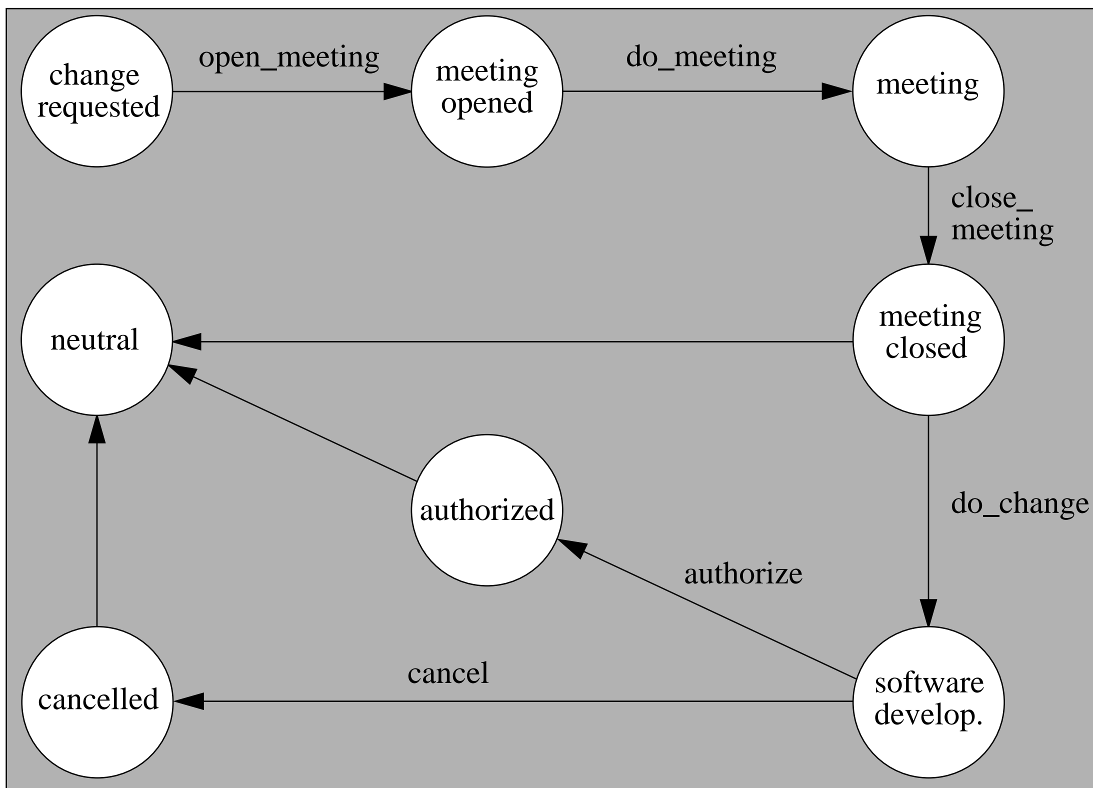


Figure 91. int-request_for_change's subprocesses and traps w.r.t. MainCAB



w.r.t. WODAN this is subprocess s-120

Figure 92. intermediate phase of MainCAB



w.r.t. WODAN this is subprocess s-121 and the state space is trap t-121

Figure 93. first intermediate phase of DepCAB

As the external behaviour of DepCAB has changed, its subprocesses and traps w.r.t. MainCAB have changed too. However they will not be given here, as they only have changed slightly.

The second prescriptive step, from the first intermediate phase to the second intermediate phase, is modelled in the following figure. The processes that are affected for the fist time this step are marked with *.

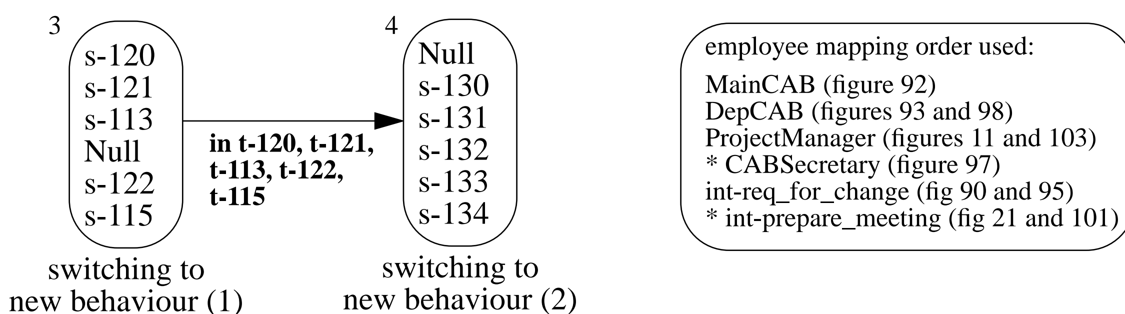
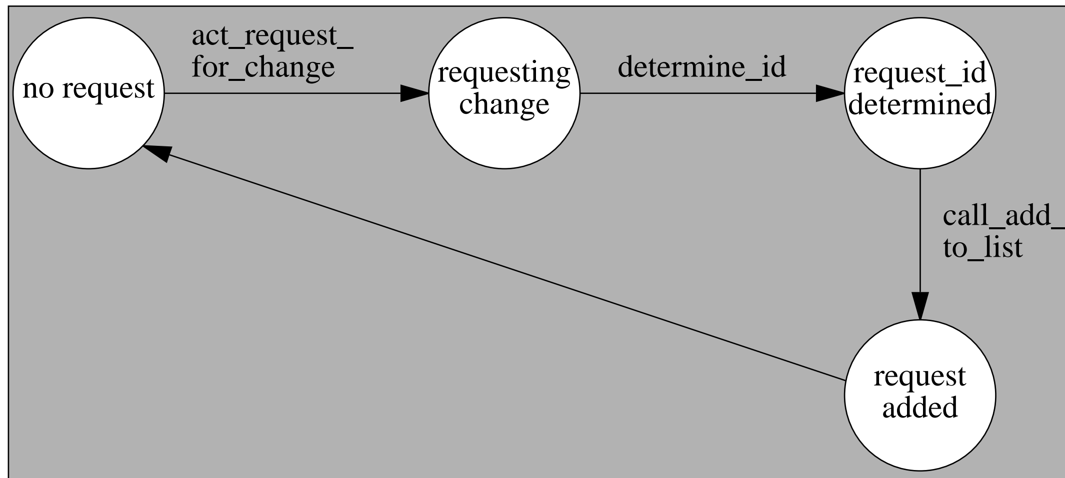


Figure 94. second prescriptive step of WODAN

As stated before, MainCAB will enter its trap t-120 almost immediately and also *int-request_for_change* will enter its trap t-122 (this trap contains the only states that have a inter-related state in the new model) almost instantly. DepCAB and *int-prepare_meeting* were already waiting in trap t-121 and t-115 respectively, as the state spaces of the behaviours are its traps. WODAN will then continue to go to its next state *switching to new behaviour (2)*. There TempCABSecretary, the intermediate version of CABSecretary, will be installed by prescribing

subprocess s-132 to it. Also subprocess s-133 will be prescribed to *int-request_for_change*, of which TempCABSecretary will be the manager. This way new requests will be administrated during switching. Just like in the new model the new requests will be put on a list, but *request_for_meeting* cannot be called yet (see Figure 95). Now it can be made clear why *int-request_for_change* could not be called in the previous state of WODAN. The reason is that MainCAB and TempCABSecretary are managing different intermediate versions of *int-request_for_change*, which cannot be prescribed at the same time, i.e. in the same state of WODAN. The difference is of course that the intermediate version of CABSecretary places the requests on a list and MainCAB does not.



w.r.t. WODAN this is subprocess s-133 and the state space is trap t-133

Figure 95. the second intermediate phase of int-request_for_change

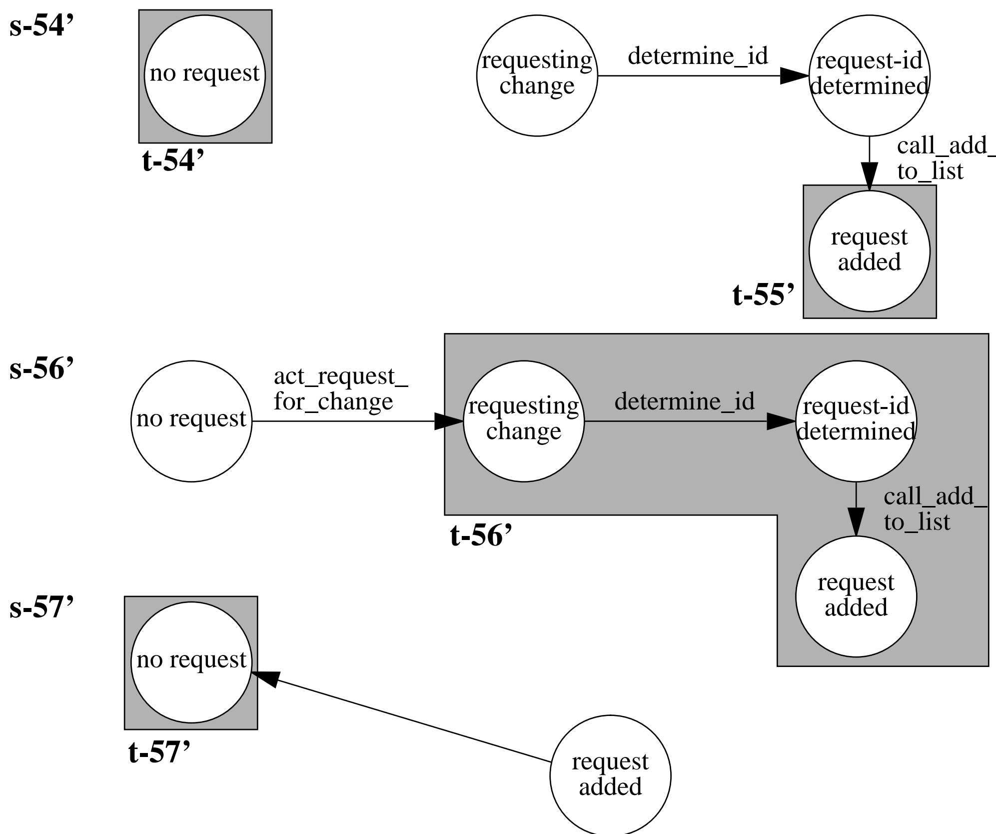
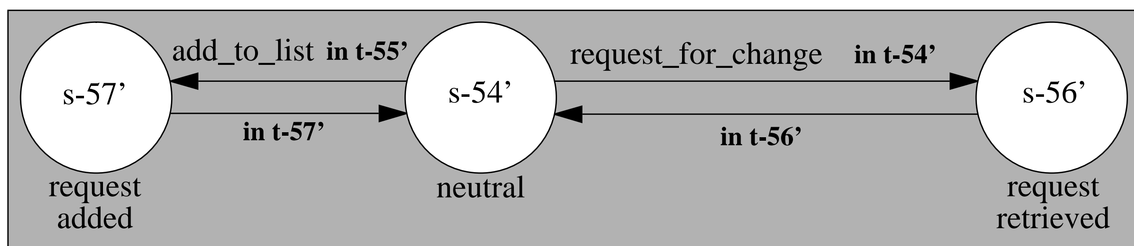


Figure 96. int-request_for_change's subprocesses and traps w.r.t. TempCABSecretary

Whenever the internal behaviour of an operation changes, the subprocesses and traps w.r.t. the managers have to be changed too. Also the managers will change as the state-action interpreters w.r.t. these operations have changed. This can be seen in figures 90, 91, 92 and 95, 96, 97. These changes are standard; just omit in the subprocesses all transitions and states that have been omitted in the internal behaviour of the operation. As this is a standard procedure, the subprocesses and traps of other internal behaviours, that will be changed, will be given only in an exceptional case.



w.r.t. WODAN this is subprocess s-132 and the state space is trap t -132

Figure 97. TempCABSecretary: viewed as manager of int-request_for_change

Compared with the new model TempCABSecretary has less states, only two export-operations (*add_to_list* and *request_for_change*) and a different state-action interpreter w.r.t. *int-request_for_change*.

Also in this state of WODAN the Null-process is prescribed to MainCAB, as it has no function anymore. Subprocess s-130 will be prescribed to (every) DepCAB, as the current meetings have to be finished before switching to NewCAB. Note that there could be activated more than one external behaviour of DepCAB, as for every change requested in the past the external behaviour of DepCAB had been activated in order to open a meeting. Normally these meetings would take place one by one after the previous meeting had been finished, assuming that in every meeting the same members have to be present. Now the requests that would have caused these meetings are placed on a list. To that aim DepCAB has been extended with a temporary export-operation *put_request_on_list* (Figure 99). Also the class CABMember has been extended with a temporary export-operation *cancel_meeting* (Figure 100), in order to cancel a meeting whenever it had already been confirmed but not been opened yet. It is from the internal behaviour of *prepare_meeting* (Figure 101) that these operations are called. Therefore in this state of WODAN subprocess s-134 will be prescribed to *int-prepare_meeting* (Figure 101). Also subprocess s-131 will prescribed to ProjectManager (Figure 103). To all other processes that eventually have to be changed, the subprocesses used in the old model are still being prescribed.

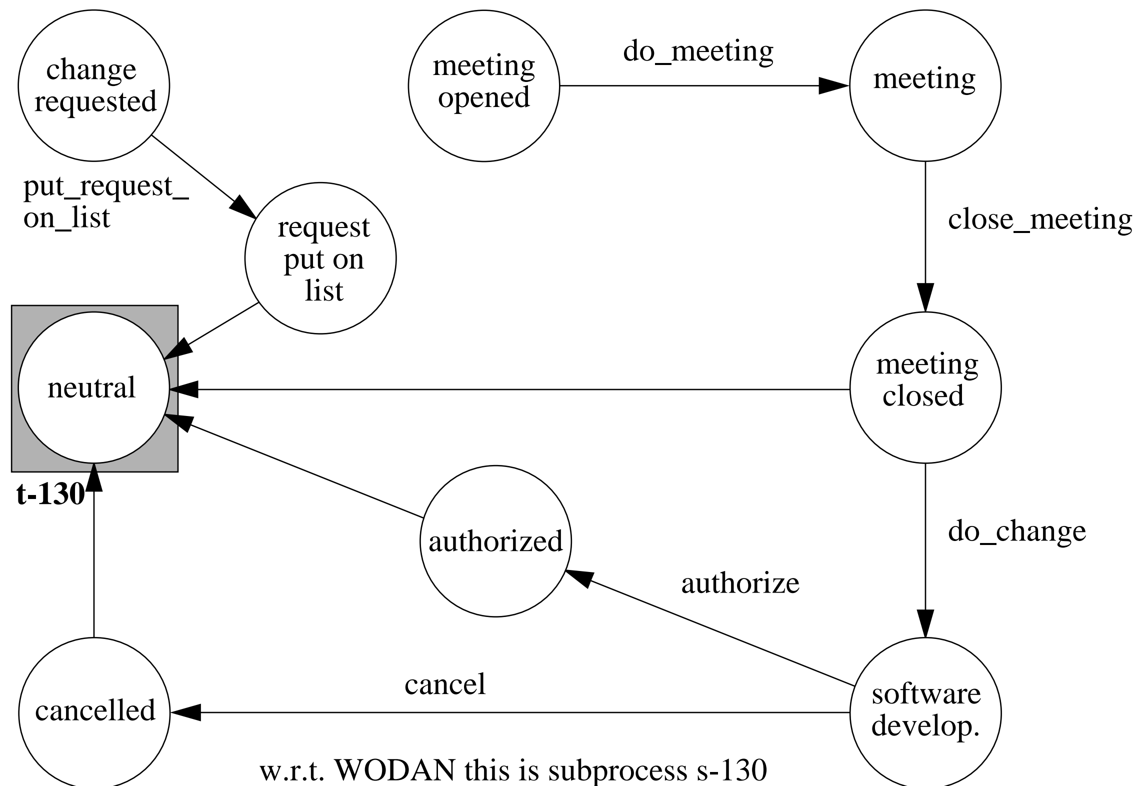


Figure 98. second intermediate phase of DepCAB

Note that Figure 98 is valid for every external behaviour of DepCAB that has been activated in the past. Note also that the transition labeled with *open_meeting* has (temporarily) been omitted, as in this phase of WODAN no new meetings can be opened. Only the meetings that had already been opened can be finished.

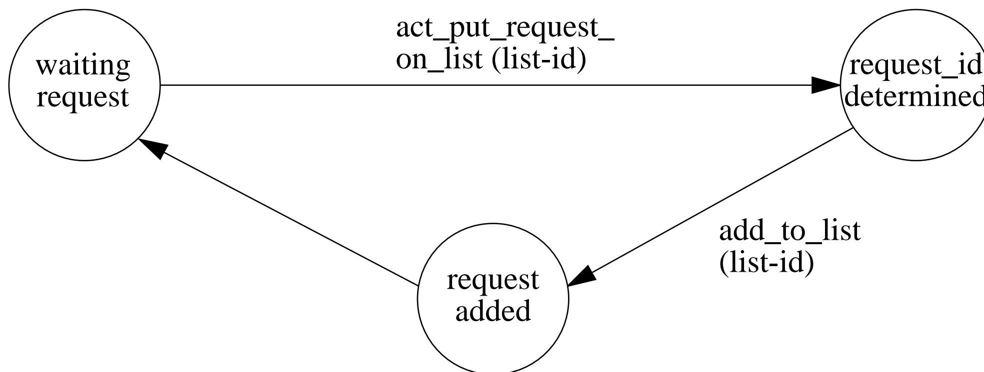


Figure 99. int-put_request_on_list: a temporary operation of the class CAB

The operation *put_request_on_list* places the request on a list, in order to handle all requests from this list at the same time, as soon as the new model will be prescribed. The list on which these requests are placed is the same list as on which the new requests are placed. Just like the other export-operations of DepCAB the operation *put_request_on_list* has been (implicitly) parametrized with the parameter *request-id*. The operation has also been parametrized explicitly with a parameter *list-id*, as all requests have to be placed on the same list. The subprocesses and traps *int-put_request_on_list* can be generated by applying the standards described in section 3.3.3 to it. They will not be given here.
As mentioned before also the class CABMember has been extended with a temporary export-operation: *cancel_meeting*.

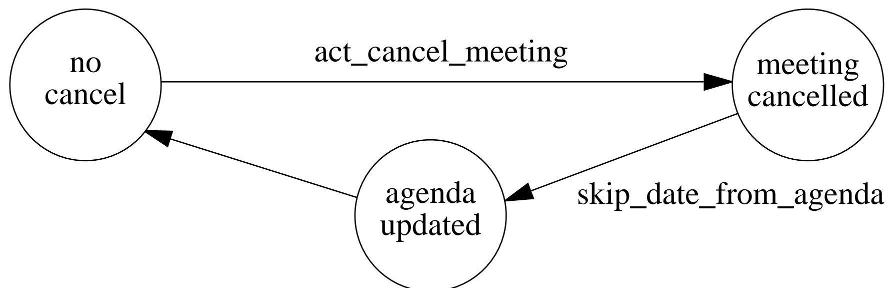


Figure 100. int-cancel_meeting: a temporary operation of the class CABMember

This operation will be called (from the temporary internal behaviour of *prepare_meeting*) whenever a meeting had already been confirmed but not been opened yet. The effect of the operation will neutralize the effect of *receive_confirmation*; the date of the meeting will be skipped from the agenda of the CABMember. The temporary operation *cancel_meeting* can be integrated into the model in the same way as *receive_confirmation*. Note that this operation, as it is an operation of the class CABMember, will be inherited by the classes ProjectManager, DesignEngineer, QAEngineer and UserRepresentative. Therefore the external behaviours of these classes will change. The altered external behaviour of ProjectManager can be found in Figure 103. The changed external behaviours of DesignEngineer, QAEngineer and UserRepresentative can be constructed in a similar way. They will not be given here. Also the subprocesses and traps of *int-cancel_meeting* will not be given here. They can be generated by applying the standards described in section 3.3.3 to it.

Now the changed internal behaviour of *prepare_meeting* will be given.

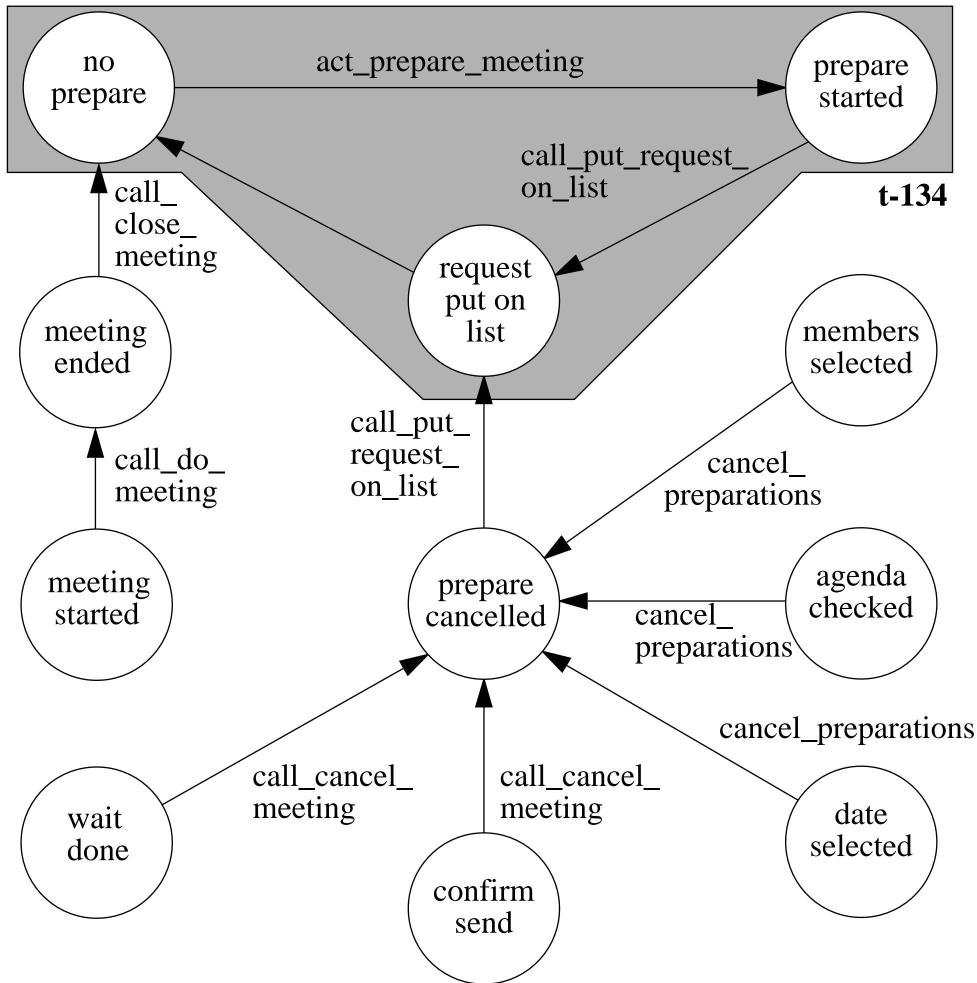


Figure 101. first intermediate phase of int-prepare_meeting

When the preparations have not already led to a meeting, the preparations will be cancelled. In case the preparations were finished, but the meeting had not been opened yet, the meeting will be cancelled. Of course meetings that had already been started can be finished. By this means most requests will be placed on a list instead of initiating a meeting, so the new model can be enacted as soon as possible. In this case *int-prepare_meeting*'s subprocesses and traps w.r.t. ProjectManager will be given (Figure 102), as they differ quite a lot from the original ones. Consequently the manager process ProjectManager changes a bit too (Figure 103).

The subprocesses and traps of *int-prepare_meeting* w.r.t. the other members can be generated

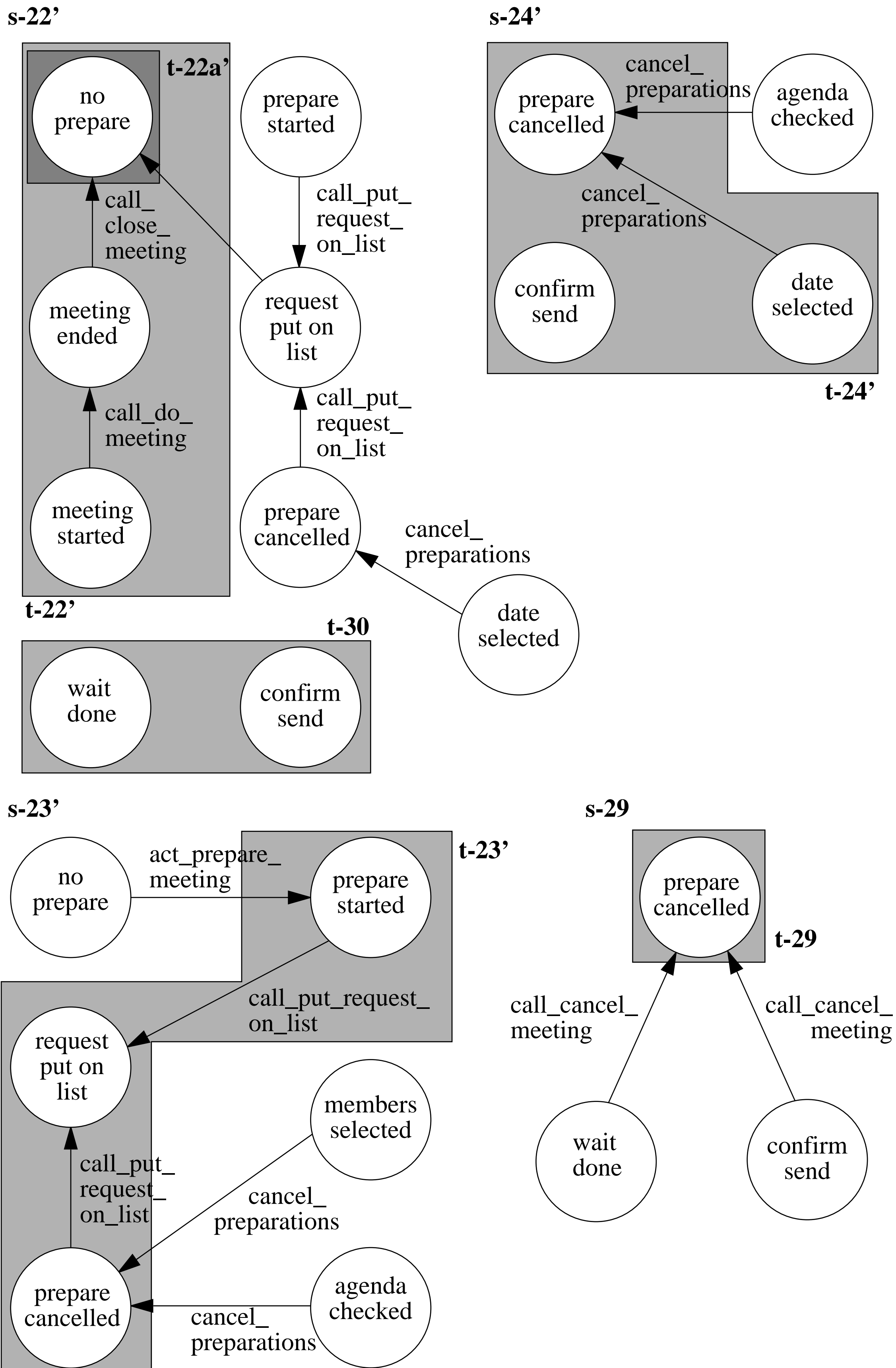


Figure 102. int-prepare_meeting's subprocesses and traps w.r.t. ProjectManager

by applying the standards described in section 3.3.3 to it. They will not be given here.

Now the changed external behaviour of ProjectManager will be given.

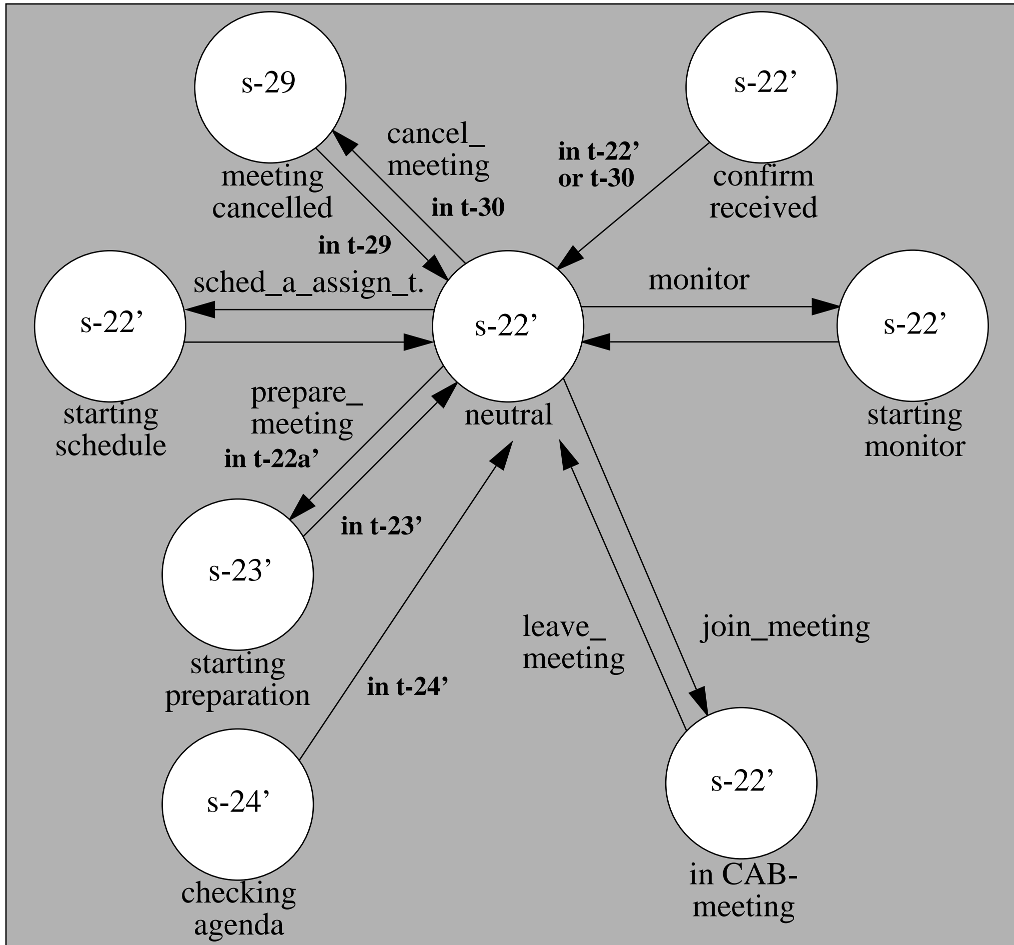


Figure 103. ProjectManager: viewed as manager of int-prepare_meeting

w.r.t. WODAN this is subprocess s-131 and its state space is trap t-131

The external behaviour of ProjectManager has been extended with a temporary export-operation *cancel_meeting*, also the transitions labeled with *check_agenda* and *receive_confirmation* have been removed from the external behaviour of ProjectManager, as these operations cannot be called in this state of WODAN. However the internal behaviours of these operations still exist and ProjectManager still manages them, as these operations could have been started, but not have been finished at the moment subprocess s-131 was prescribed to ProjectManager. The external behaviours of the other members (DesignEngineer, QAEngineer and UserRepresentative) can be constructed in a similar way.

Also ProjectManager's state-action interpreter w.r.t. *prepare_meeting* has changed. Note that in state *neutral* only subprocess s-22' (derived from subprocess s-22) can be prescribed, in contradistinction to both the old and the new model where subprocess s-22 or s-24 could be prescribed (Figure 39 and Figure 87). Subprocess s-24 was prescribed in state *neutral* only when ProjectManager still had to execute the operation *receive_confirmation*. However this operation cannot be called in this state of WODAN, so this will not occur. Note that when ProjectManager was prescribing subprocess s-24 (Figure 34) in its neutral state at the moment the first intermediate version of *int-prepare_meeting* will be prescribed, *int-prepare_meeting* must be in a state that is a part of trap t-24. As the state *date selected* is a part of this trap, this state has been included in subprocess s-22' and in trap t-24'.

The third prescriptive step, from the second intermediate phase to the third and last intermediate phase, is modelled in the following figure. The processes that are affected for the first time this step are marked with *.

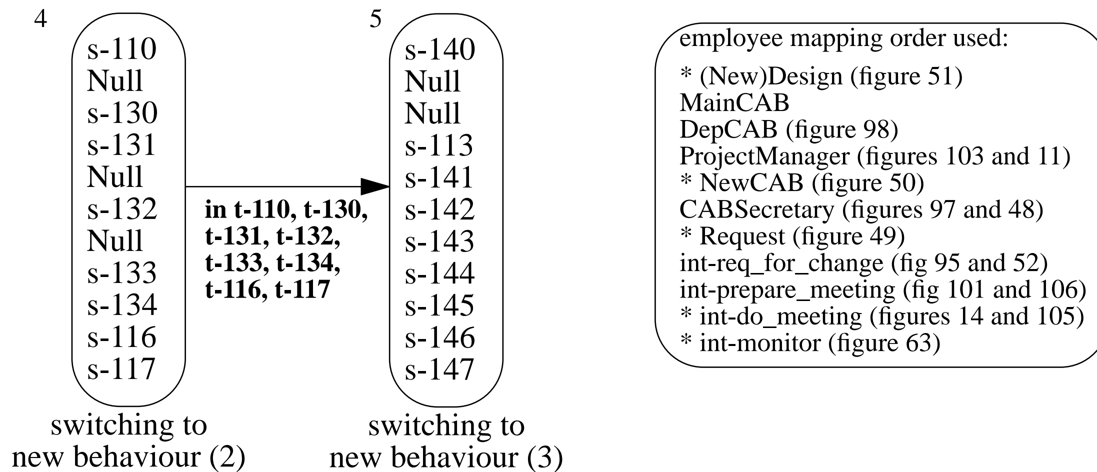


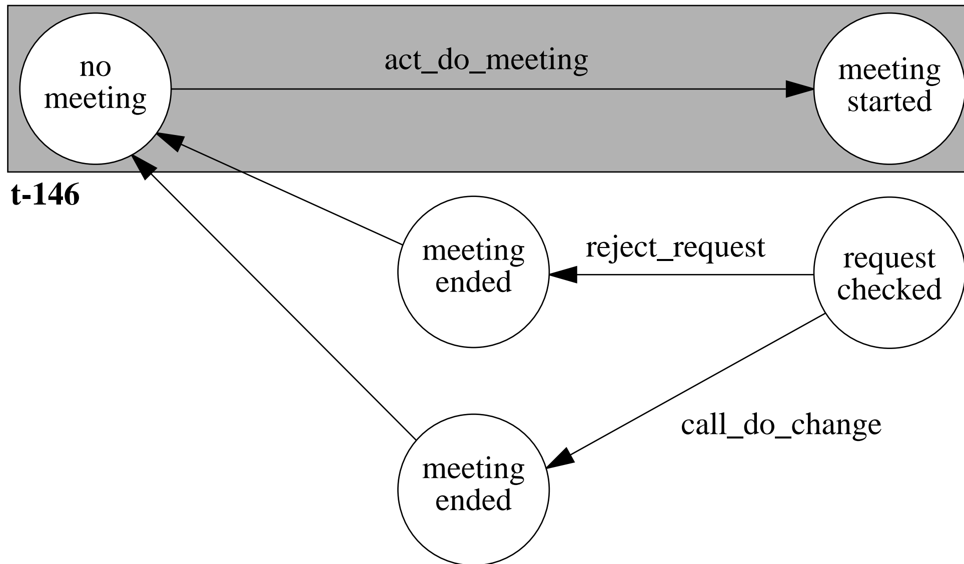
Figure 104. third prescriptive step of WODAN

So if (every) DepCAB has entered its trap t-130, if also ProjectManager has entered its trap t-131, if moreover TempCABSecretary has entered its trap t-132, if furthermore *int-request_for_change* has entered its trap t-133, and if (every) *int-prepare_meeting* has entered its trap t-134, then WODAN can and will transit to its next state *switching to new behaviour (3)*. Note that Design, *int-do_meeting* and *int-monitor* were already waiting in traps t-110, t-116 and t-117 respectively. Note that the state space of the external behaviour of Design (see [2, Figure 13]) is its trap w.r.t. WODAN, as no states have been removed from the external behaviour of Design, but only transitions and states have been added in order to model the external behaviour of NewDesign. For the same reasons the state space of *int-monitor* (old model, see [2, Figure 12]) is its trap w.r.t. WODAN. In this state of WODAN the changing from the old model to the new model has been completed almost entirely. Here the subprocesses s-113, s-140, s-141, s-142, s-143, s-144 and s-147 already will be prescribed to ProjectManager, NewDesign, New-CAB, CABSecretary, Request, *int-request_for_change* and *int-monitor* respectively. The Null-process is prescribed to DepCAB, as it has no function anymore. Every meeting has been finished and every (old) request has been placed on a list. Note that from now on meetings can be requested, as *request_for_meeting* can be called now from within subprocess s-144 of *int-request_for_change*.

Note that ProjectManager's trap t-131 (Figure 103) w.r.t. WODAN contains the state *meeting cancelled* and that ProjectManager's subprocess s-113 (Figure 11) does not. This is not in accordance with the SOCCA-conventions. However it is garantueed that when WODAN transits to its state *switching to new behaviour (3)*, ProjectManager will never be in its state *meeting cancelled*, as in that case *int-prepare_meeting* could not have entered its trap t-134 w.r.t. WODAN. This can be seen as follows. In order for ProjectManager to leave the state *meeting cancelled* before WODAN transits to its next state, *int-prepare_meeting* must have entered its trap t-29. This trap does not contain any state that is also contained by *int-prepare_meeting*'s trap t-134 w.r.t. WODAN (see Figure 101 and Figure 102). So only after ProjectManager's neutral state has been reached and subprocess s-22' has been prescribed to *int-prepare_meeting* again, *int-prepare_meeting*'s states that form trap t-134 can be reached.

Also in this state of WODAN subprocesses s-146 and s-145 are prescribed to (every) *int-do_meeting* and (every) *int-prepare_meeting* respectively. The final versions of these operations cannot be prescribed yet in this state of WODAN. If this had been done, the subprocesses used in the previous state of WODAN would have needed a trap as used in the subprocesses s-

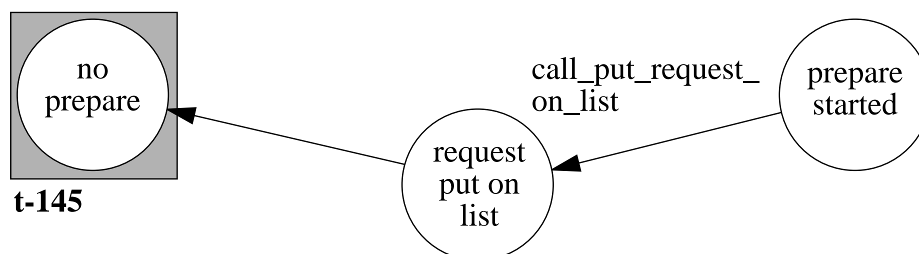
146 and s-145. However using such traps in the previous state would have caused problems. If for instance *int-do_meeting* had been trapped in its state *meeting started*, it could never reach its trap t-9 anymore. If at this moment DepCAB (Figure 93) would be in its state *meeting opened*, DepCAB could not reach its next state, as trap t-9 would never be entered, and therefore could never reach its state *neutral*, which is its trap w.r.t. WODAN.



w.r.t. WODAN this is subprocess s-146

Figure 105. intermediate phase of int-do_meeting

The subprocess of *int-prepare_meeting* used in the previous state has only one state (*no prepare*) that could be used as a trap to the final version of *int-prepare_meeting*. When trapped in this state it is possible that the call to *put_request_on_list* would not be carried out, because the transition labeled with *act-prepare_meeting* leading out of the trap was omitted. In that case DepCAB would never reach its state *request put on list* and therefore would also never reach its state *neutral* (this is one of the problems described in section 5.3). In both cases the final state of WODAN cannot be entered.



w.r.t. WODAN this subprocess s-145

Figure 106. second intermediate phase of int-prepare_meeting

Note that Figure 105 and Figure 106 are valid for every internal behaviour of *do_meeting* and *prepare_meeting* that has been activated in the past.

The final prescriptive step from the third and last intermediate phase to the new model is modelled in the following figure. In this step no processes will be affected for the first time. Only *int-do_meeting* and *int-prepare_meeting* will be affected here.

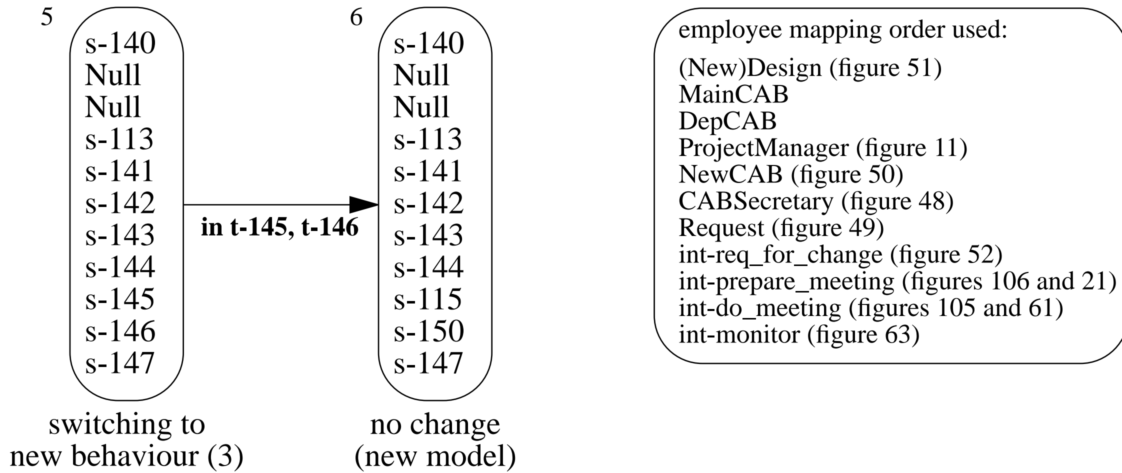


Figure 107. final prescriptive step of WODAN

In the fifth state of WODAN it is guaranteed that *int-do_meeting* is in its state *neutral* and that *int-prepare_meeting* is in one of its states *request put on list* or *no prepare*, otherwise DepCAB would have never reached its neutral state. This implies that (every) *int-do_meeting* already is in its trap t-146 and that (every) *int-prepare_meeting* already is in its trap t-145 or will enter this trap almost immediately (both traps contain the only states that have a interrelated state in the new model). So WODAN does not have to wait very long to enter its next state. In this last state the changing to the new model has been fully completed.

Figure 108 shows WODAN completely. A list of subprocesses w.r.t. WODAN is given in Appendix B.

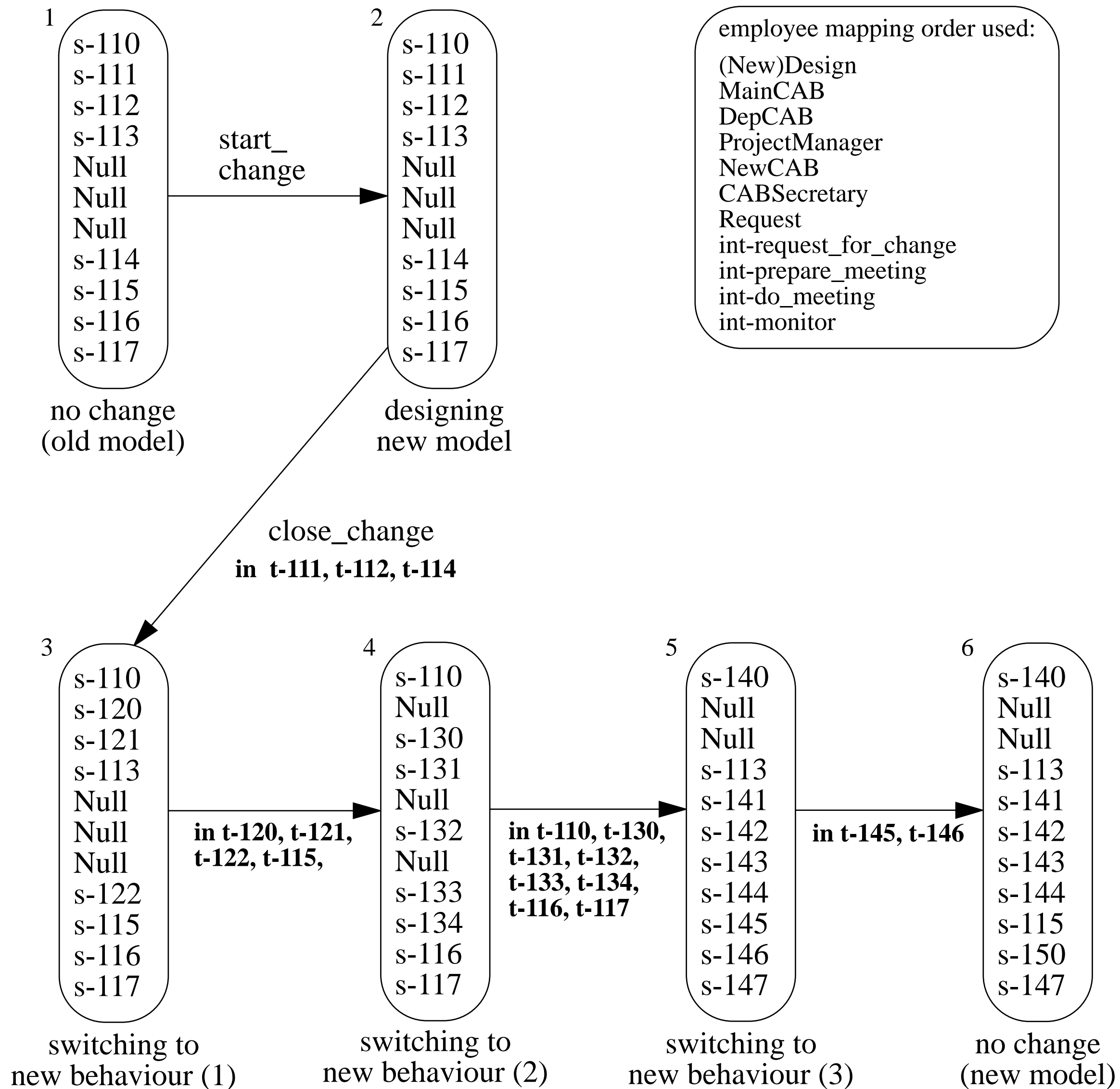


Figure 108. WODAN: viewed as manager of 11 employees

The internal behaviours of all new operations (*request_for_meeting*, *reject_request*, *big_impact*, *small_impact* and *handle_change_request*) are implicitly managed by WODAN. As soon as WODAN is in its fifth state or in state *no change (new model)*, which means the new model will be prescribed (almost) completely, the internal behaviours of the new operations will be prescribed too. In all other (previous) states the Null-process is prescribed to these operations. Every other internal or external behaviour has not been influenced by the change to the new model. The temporary operations *put_request_on_list* and *cancel_meeting* are also impicitly managed by WODAN. They only exist in the fourth state of WODAN, in every other state the Null-process is prescribed to these operations. Note also that every external behaviour of DepCAB should be trapped in its trap t-130, before the transition to the state *switching to new behaviour (3)* can be made. Also every internal behaviour of *do_meeting* and *prepare_meeting* has to be trapped in its trap t-146 and t-145 respectively, before the transition to the state *no change (new model)* can be made.

7 PMMS

7.1 Introduction

In Manchester the group of Warboys, together with some associates, has a long tradition in studying general business processes [9]. A characteristic feature of these processes is their ever changing nature. To describe this kind of change they developed PMMS, which stands for **P**rocess **M**odel for **M**anagement **S**upport. It is used for the description of general management support when designing, instantiating and enacting whatever process model, e.g a Socca-model. In PMMS evolutionary change is inherent, as so-called Terms of Reference (ToR) can be replaced by a new ToR. Every PMMS-model basically consists of four sequential components: managing, technology, logistics and administering. Dependent on the ToR extra sequential components are added. They are placed under the administering-component. Together these extra components are called the specific part of the PMMS-model. So a PMMS-model consists of the four basic components and a specific part consisting of the extra components.
The way the basic components are interconnected with each other is shown in Figure 109. The interconnections can be considered as, sometimes bidirectional, dataflows and the components as dataflow processes.

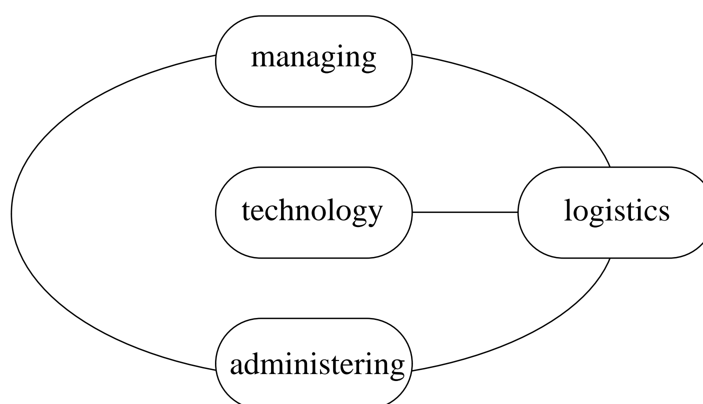


Figure 109. a basic PMMS-model and its four components

Every basic component has some special tasks. The managing-component generates the Terms of Reference (ToR), which can be seen as the verbal description of a model. A ToR tells what is to be expected of the model, not how the model should be constructed. The logistics-component then produces a setup for the model; a first, formal, but incomplete, description, which is supposed to answer to the ToR. After that the technology-component designs the corresponding methods in order to complete the formal description. The logistics-component instantiates these methods as processes and the administering-component enacts these processes.
Note that one could also decide to give the logistics-component only one task: the instantiation of the methods. In that case the technology-component will be asked by the logistics-component to produce the complete formal description of the model, including the setup. However, as in a PMMS-model no direct communication between the managing-component and the technology-component exists (see Figure 109), the managing-component then can influence the design process only indirectly, via the logistics-component. When describing the communication between the components (section 7.4) this will be more complicated. For this reason our logistics-component has the two tasks mentioned.
When designing a Socca-model the tasks of the logistics-component and the technology-component can be specified more exactly. The logistics-component in this case has to construct the class diagram of the model and the general relationships between the classes. These can be seen as the setup for the model. The task of the technology-component implies the description of the uses relationships between the classes, the external behaviours of the classes, the internal be-

haviours of the operations and the communication-structure. This can be seen as the completion of the formal model.
In [8] a small PMMS-model was transformed into a Paradigm-model. In the section 7.2 the Change Management (CM) models, described by means of Socca and Paradigm, will be transformed into a PMMS-model. After that, in sections 7.3 and 7.4, the management support for designing the models and managing the evolutionary change will be illustrated by presenting a full-blown Socca-model for the four basic components of the PMMS-model. As we will see the PMMS-structure provides an excellent setup for further refining and structuring WODAN. On the other hand, the Paradigm and Socca features indeed clarify many technical details, not explicitly specified by PMMS. So the work in [8] has been considerably extended from Paradigm towards whole Socca.

7.2 Change Management modelled by means of PMMS

The CM-models, described by means of Socca and Paradigm, will be transformed into a PMMS-model. First a PMMS-model, which describes the old CM-model (chapter 3), will be constructed. Naturally it will consist of the four basic components. Also, due to ToR_1, which denotes the old model, some extra components are added. These extra components are exactly the classes used to model Change Management. Note that only the classes, of which the external behaviour has been given explicitly in chapter 3, are used, as components, in the specific part of the PMMS-model. These classes are CAB, ProjectManager, DesignEngineer and UserRepresentative. Every other class presented in the class diagram (Figure 3), of which the external behaviour has not been given explicitly in chapter 3, is not displayed, as a component, in the specific part of the PMMS-model. Also there is a pool of Null-components, i.e. components that have ceased to exist or do not exist yet but will exist in the future.
ToR_2 denotes the new CM-model (chapter 4). Due to ToR_2 some other components will be added to the specific part of the PMMS-model: Request and CABSecretary. These components have not been used in the first CM-model, however they can be included into the PMMS-model. They will be in the pool of Null-components during the enactment of the first model.
The PMMS-model describing both CM-models is given in Figure 110. The dashed components and interconnections are added due to ToR_2. The interconnections between the extra components can be seen as the uses relationships between the classes.

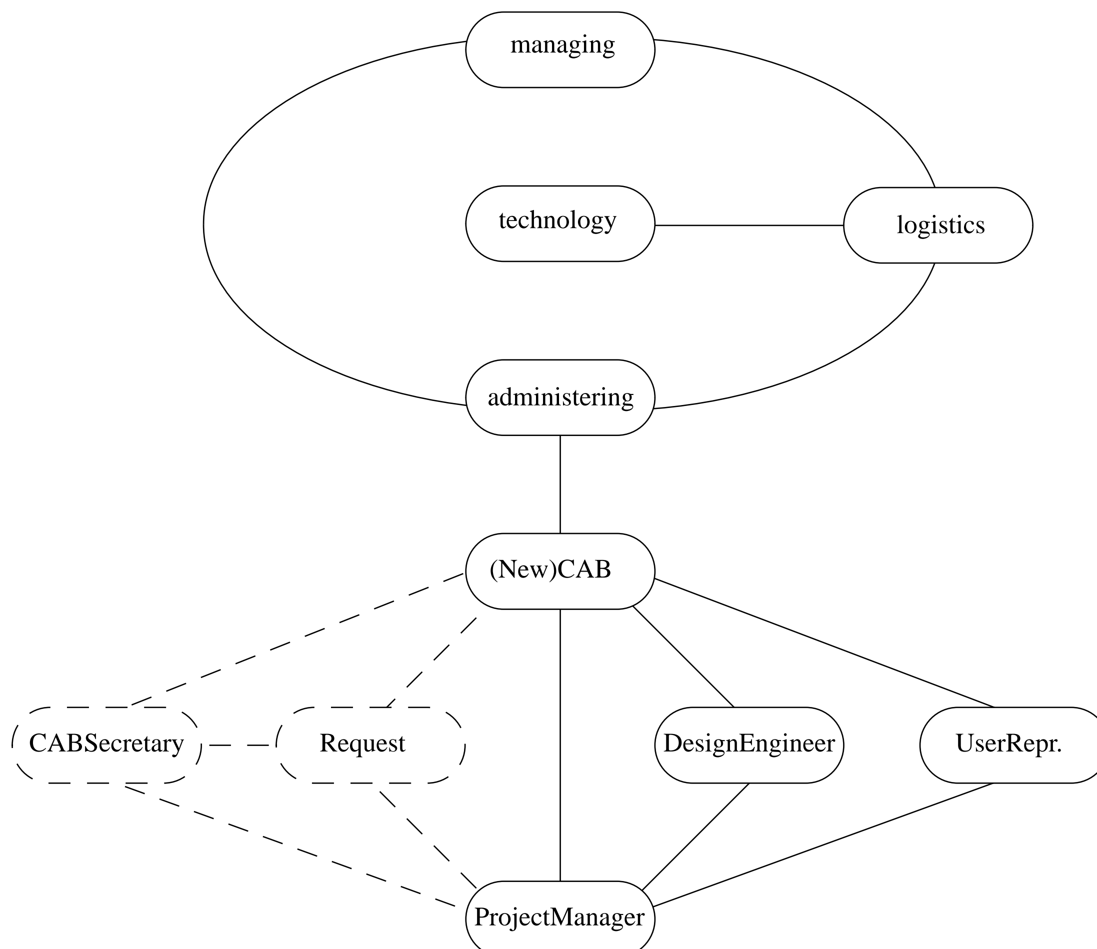


Figure 110. a PMMS-model describing Change Management

7.3 Behaviour of the basic components

The way the models are being designed and the way the change from ToR_1 to ToR_2 is being executed, as e.g. controlled by WODAN, is not shown explicitly in the previous figure. This is inherent to the PMMS-method, so to say. However it can be made more explicit by modelling the basic components in Socca. In that case first the data-perspective of the model has to be described. Therefore a class diagram has to be defined. Note that there are no IS-A or Part-Of relationships between the basic components, so the class diagram, presented in Figure 111, only shows the attributes and operations of these components.

managing	logistics	technology	administering
create_first_ToR schedule_design view_status create_next_ToR	generate_setup report_proposal modify_setup get_methods report_method_design implement_methods generate_change_setup report_interm_steps	make_methods report_methods modify_methods	enact_processes report_status

Figure 111. Class diagram: attributes and actions of the basic components

As a next step in describing the data perspective the uses relations between the components have to be given. They are shown in Figure 112. The corresponding import list is given in Figure 113.

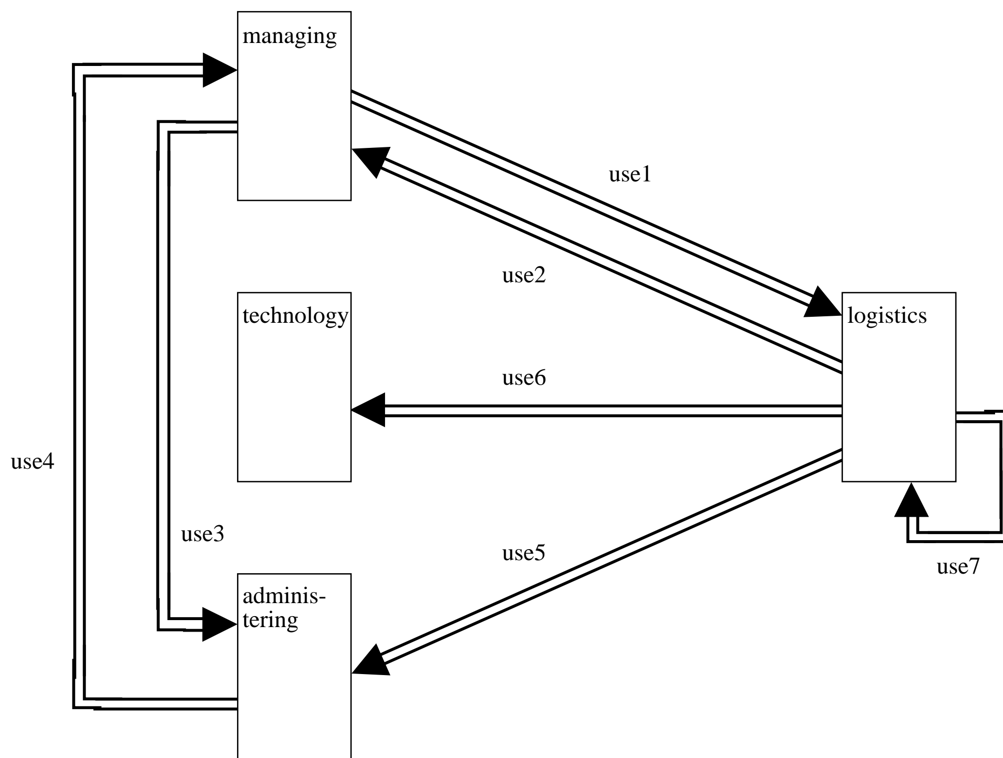


Figure 112. Import/export diagram

Note that the communication between the components is modelled solely by means of the uses relationship. The general relationships have been left out.

use1
generate_setup
report_proposal *
modify_setup
get_methods *
report_method_design *
generate_change_setup *
report_change_steps *
instantiate_methods *

use2
view_status
use3
report_status
use4
create_next_ToR *
use5
enact_processes

use6
make_methods
report_methods *
modify_methods
use7
get_methods

Figure 113. import list

Note that in *get_methods* can be imported by the managing-component and by the logistics-component itself.
Note also that, as indicated by the uses relations, in our model the operations marked with * will be called from elsewhere. On the other hand, one can easily imagine that these operations are spontaneously executed by the corresponding external behaviours. This then leads to a slightly different communication, which has not been worked out here.

Second the behaviour perspective of the model has to be described. Therefore the external behaviour of each of the basic components will be given, described by means of an STD as usual. The managing-component (Figure 114) will create a (first) ToR, which is the informal description of a software process model, and will supervise the design of the formal model describing the same software process. After the model has been enacted it will regularly view the status of the model. Whenever problems occur in the model, the managing-component will create a next ToR for that model. Note that for a model describing another software process, another (first) ToR has to be created.

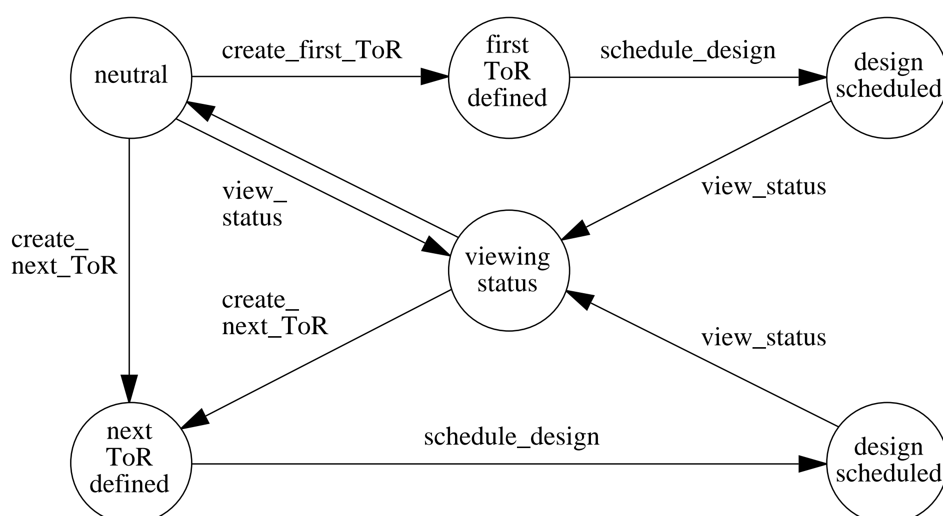


Figure 114. external behaviour of the managing-component

Note that there are three transitions labeled with *view_status*. The managing-component will transit from one of its states *design scheduled* to its state *viewing status* in order to check a model that has just been enacted. In that case *view_status* will be called from within the internal behaviour of *instantiate_methods* (Figure 123). In order to check a model that is already being operative for a while, the managing-component will transit, every now and then, from its state *neutral* to its state *viewing status*. This calling of *view_status* will not be modelled. Note that when the status is viewed before the first ToR has been created for a software process model, an empty status will be returned.

Note that there are also two transitions labeled with *create_next_ToR*. The managing-component will transit from its state *viewing status* to its state *next ToR defined* whenever the status report suggests to do so. In that case *create_next_ToR* will be called from within the internal behaviour of *report_status* (Figure 125). However the managing-component can also decide to create a next ToR without viewing the status first. In that case it will transit to its state *next ToR created* from its neutral state. Of course this can only be done whenever already a (first) ToR existed. This calling of *create_next_ToR* will not be modelled. Also the calling of *create_first_ToR* and *schedule_design* will not be modelled.

The logistics-component (Figure 115) will define a setup for the formal model, it will supervise the technology-component and it will play a part of the WODAN-role, i.e. design a setup for the change steps. All actions of the logistics-component will be called from within the internal behaviour of *schedule_design*. Note that the setup for the model could also be defined by the technology-component. However, as there exists no direct communication between the managing- and the technology-component, the communication will be more complicated in that case.

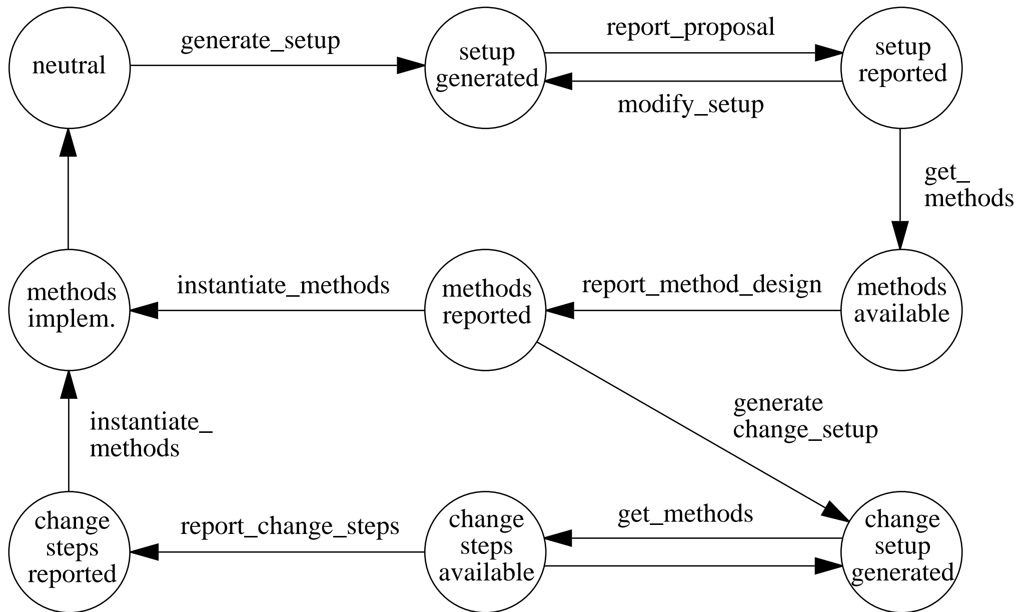


Figure 115. external behaviour of the logistics-component

The technology-component (Figure 116) will make, and if necessary will modify, all methods necessary for the model. All operations of the technology-component will be called from within the internal behaviour of *get_methods.*

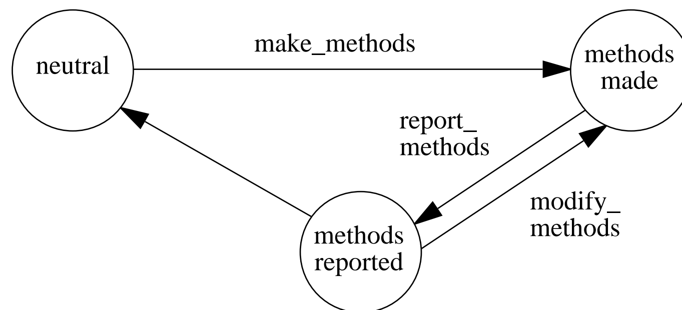


Figure 116. external behaviour of the technology-component

The administering-component (Figure 117) will enact all processes when the design has been completed. Whenever a new model has to be enacted, it will enact the intermediate models first. So this component plays another important part of the WODAN-role. Also the administering-component will regularly check the model, while it is running, to see whether problems have arisen.

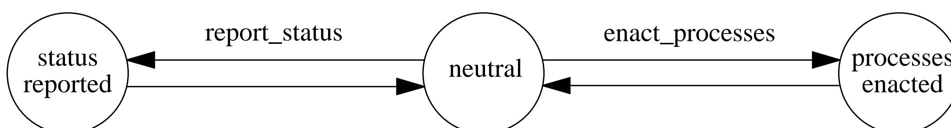


Figure 117. external behaviour of the administering-component

In order to clarify the behaviour of the basic components, the internal behaviours of some of the operations performed by these components will be given. This is the second step in describing the behaviour perspective. First the behaviour of the managing-component will be clarified. Therefore the internal behaviour of *schedule_design* and *view_status* will be given. The internal behaviour of *create_first_ToR* and *create_next_ToR* will not be given.

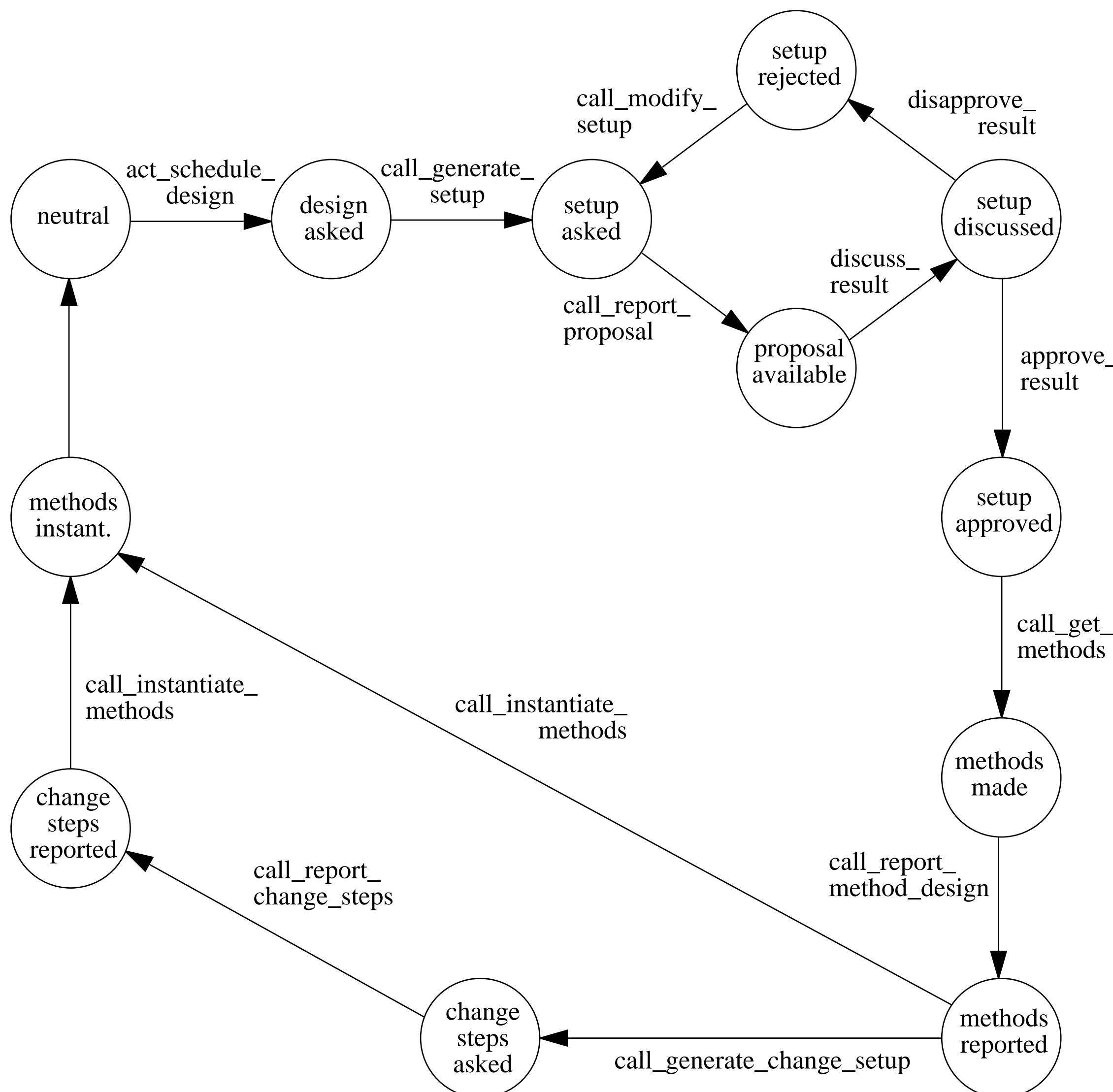


Figure 118. int-schedule_design

Note that there are two transitions labeled with *call_instantiate_methods*. The transition leading from the state *methods reported* to the state *methods instantiated* will be used only when the first model for a particular software process has to be instantiated, i.e. when in the external behaviour of the managing-component *schedule_design* is preceded by *create_first_ToR*. Otherwise, when in the external behaviour of the managing-component *schedule_design* is preceded by *create_next_ToR*, the other transition will be used.

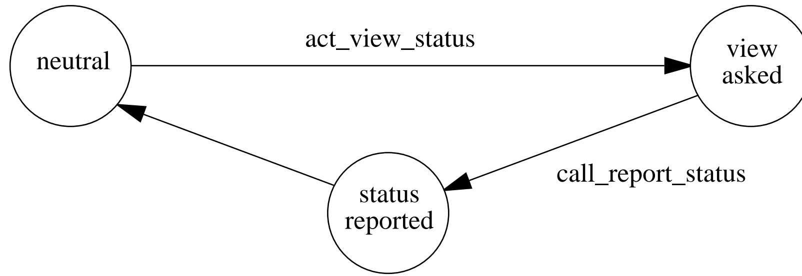


Figure 119. int-view_status

Second the internal behaviours of some operations performed by the logistics-component will be given: *generate_setup*, *generate_change_setup*, *get_methods* and *instantiate_methods*. In order not to complicate the model unnecessarily the internal behaviours of the other operations performed by the logistics-component will not be given here. Also they do not really contribute to the understanding of the way models are being designed and the way the change from ToR_1 to ToR_2 is being executed (WODAN).

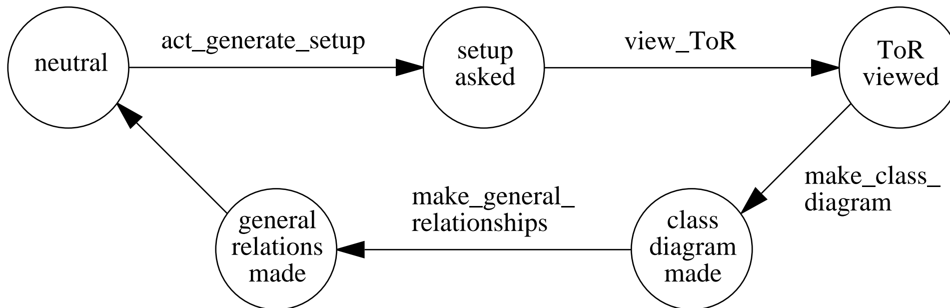


Figure 120. int-generate_setup

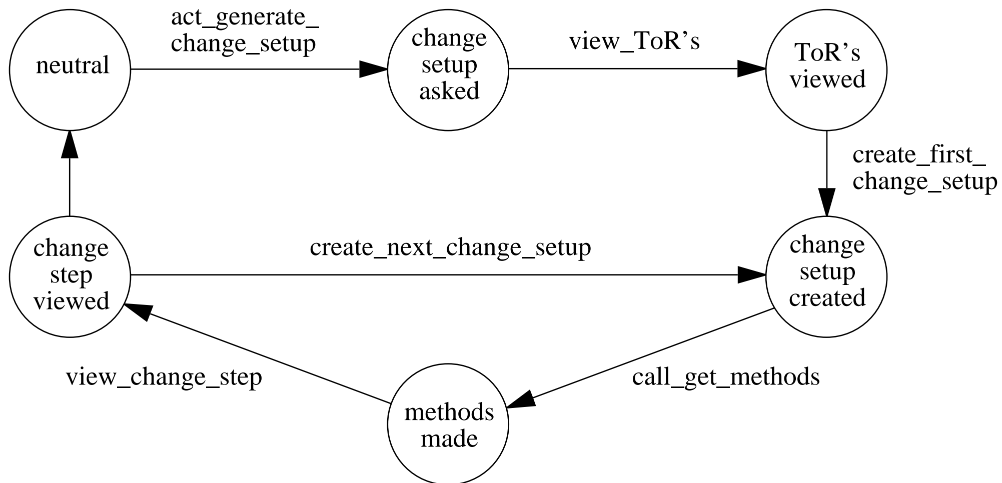


Figure 121. int-generate_change_setup

When within the internal behaviour of *generate_change_setup* the evolutionary change cannot be executed in only one change step, a next change step has to be developed. Consequently in that case a next change setup has to be generated first. Whether a next change step is necessary or not will be decided in the state *change step viewed*.

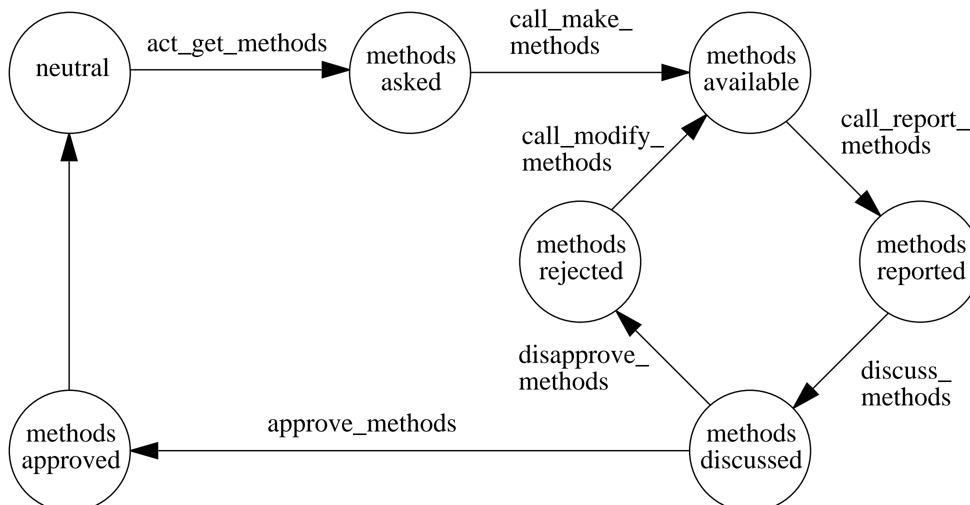


Figure 122. int-get_methods

As can be seen in the above figure, the logistics-component will decide whether the methods developed by the technology-component answer to the setup or not.

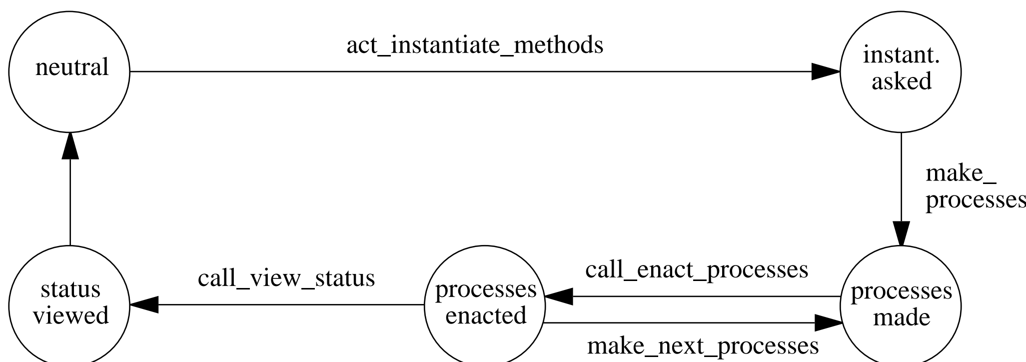


Figure 123. int-instantiate_methods

The internal behaviour of *instantiate_methods* contains the possibly repeated calls of *enact_processes*. So here the communication with the administering-component comes to light.

The only operation performed by the technology-component, of which the internal behaviour will be given, is *make_methods*.

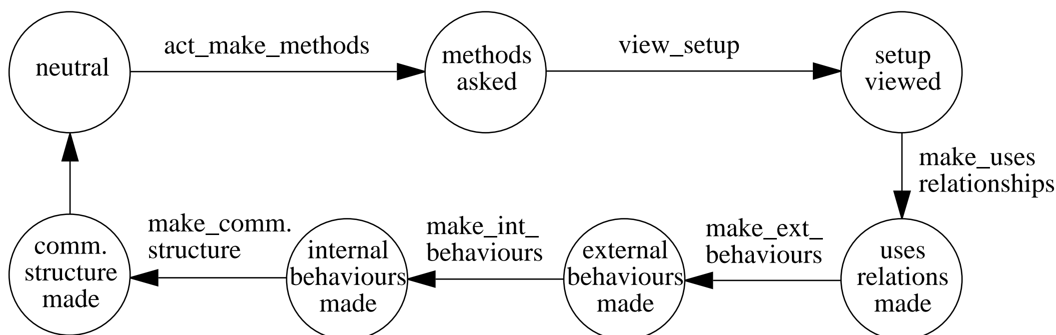


Figure 124. int-make_methods

Note that this operation will be called not only when a new model has to be designed, but also when the evolutionary transformation from one model to another has to be specified. In that case this action will change the external behaviours of the extra components (classes) and the internal

behaviours of operations of these components. In view of also making the communication structure make_methods will furthermore create subprocesses and traps, also w.r.t. WODAN. Moreover it will create temporary operations for the extra components, when necessary.

Finally the internal behaviour of two operations performed by the administering-component will be given. These operations are *report_status* and *enact_processes*.

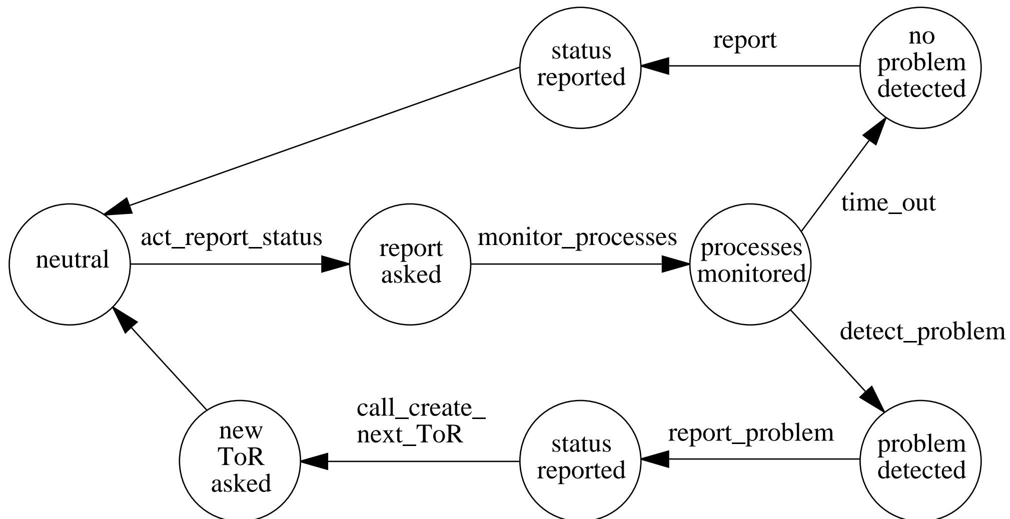


Figure 125. int-report_status

Note that this operation will not be called in between evolution phases, i.e. when intermediate processes are being prescribed. It will only be called to monitor the processes of a model representing an evolution phase. Whenever problems are detected, i.e. the model is insufficient for the actual situation, the managing-component will have to define a new ToR.

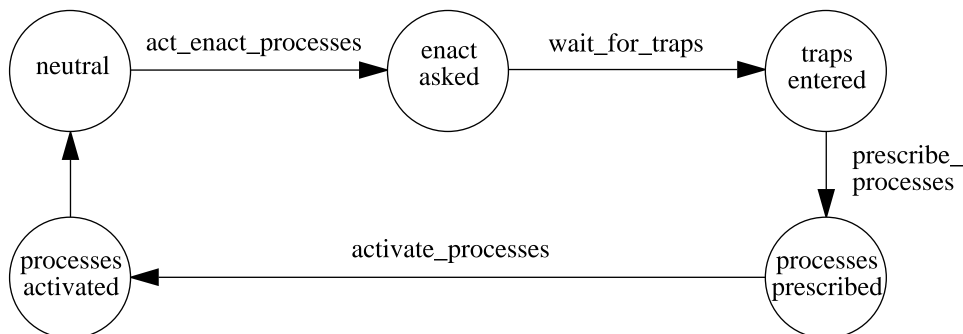
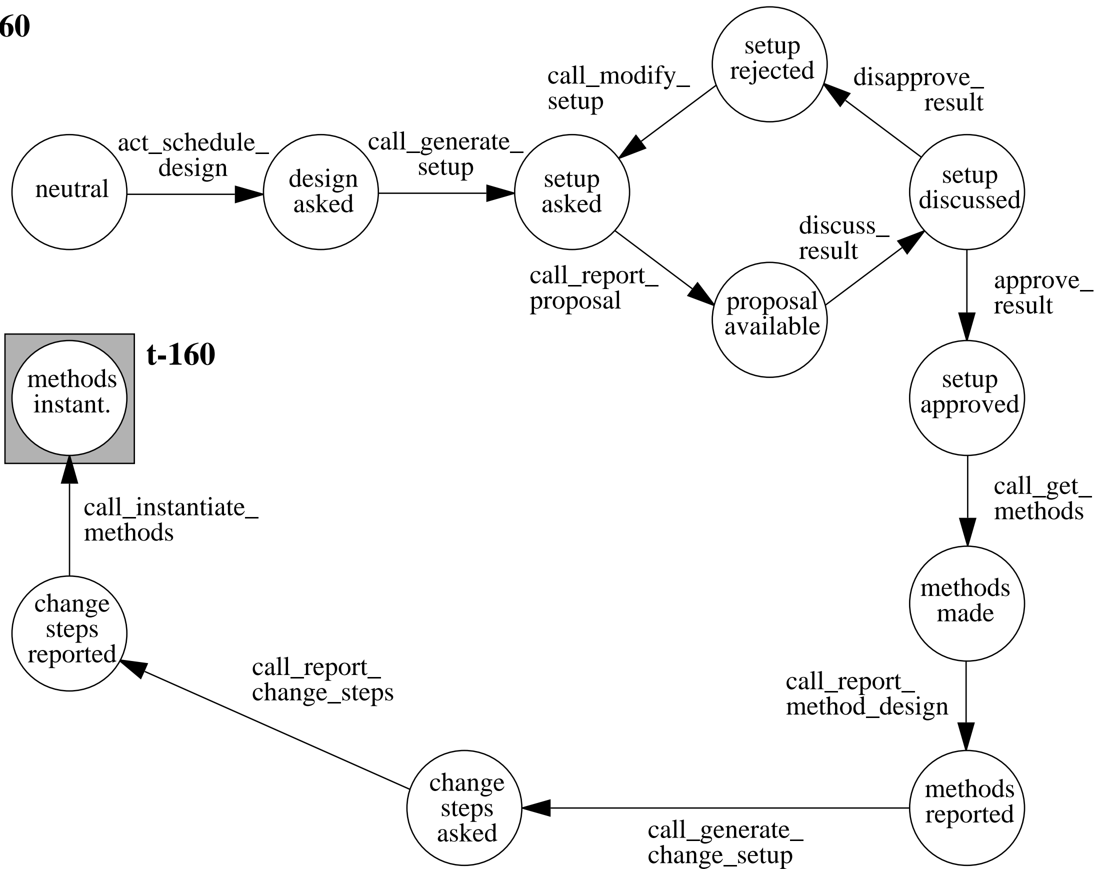


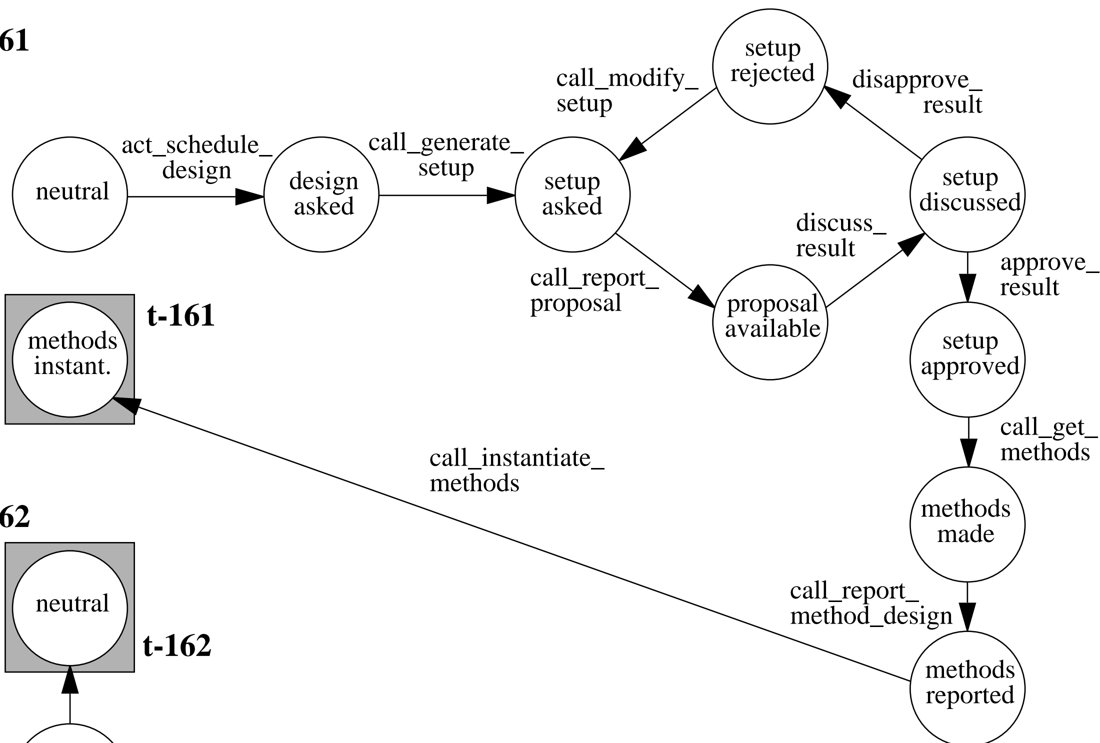
Figure 126. int-enact_processes

Int-enact_processes in fact represents (an important part of) WODAN. In *int-enact_processes* (new) processes are prescribed to both internal and external behaviours. These (new) processes can be viewed as subprocesses of anachronistic processes. In Socca however only external behaviours can prescribe subprocesses to internal behaviours. So this part of WODAN is not yet in accordance with the Socca-conventions (see [1]). For now the way this part of WODAN is modelled is sufficient. Remodelling WODAN in complete and detailed accordance with Socca is a topic of future research.

7.4 Communication between the components

The final step in describing the behaviour perspective of the model is specifying the communication between the behaviours. As usual within Socca it will be specified by means of Paradigm. The external behaviours of the basic components will act as the manager processes. The internal behaviours of operations performed by a component and also the internal behaviours of operations calling the operations of this component will be the employee processes. By specifying the communication between the behaviours the way models are designed and evolutionary change is conducted (WODAN) will be clarified. Note that the communication between a component and only a few of its employees, not all, will be specified here. Note also that the communication between the technology-component and its employees will not be specified at all, as this component is not very important in order to describe the way WODAN is organized. Furthermore the specification of this part of the communication is rather straigtforward.
For this part of the modelling it seems best to have a rather sequential, interleaved execution of some of the various behaviours, as some of the operations cannot be performed simultaneously. Therefore the trap-structures of these operations will deviate from the standards described in section 3.3.3., as mostly small traps are used.
First the communication between the managing-component and its employees *int-schedule_design*, *int-view_status*, *int-report_status* and *int-instantiate_methods* will be specified. The operations *create_first_ToR* and *create_next_ToR* have not been modelled, so the communication between the managing-component and these operations will not be specified. The internal behaviour of the operations *schedule_design* and *view_status* belong to the managing-component itself. *Int-report_status* and *int-instantiate_methods* are internal behaviours of operations calling operations of the managing-component.
The managing-component (Figure 131) starts in its neutral state. As soon as *create_first_ToR* is called, which has not been modelled, the managing-component will transit to its state *first ToR defined*. When the ToR has been defined and *int-schedule_design* has entered its trap t-162, the managing-component will make the transition to its state *design scheduled*. There subprocess s-161 will be prescribed to *int-schedule_design*. Now a setup, answering to the ToR, can be developed. It will be developed by the logistics-component. Note that in subprocess s-161 it is not possible to generate change steps. However, the model to be developed will be the first model and therefore the enactment of this model can be performed without any change step. Consequently it is not necessary to generate these change steps.





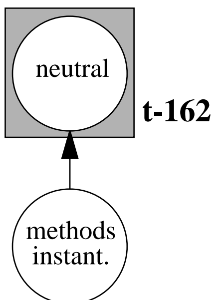


Figure 127. int-schedule_design's subprocesses and traps w.r.t. managing

If *int-schedule_design* has entered its trap t-161, which means the model has been instantiated, if furthermore *int-instantiate_methods* has entered its trap t-168, which denotes the new model has been enacted and the status of this enacted model is asked to be viewed, if moreover *int-view_status* has entered its trap t-166, which means the status can be viewed, and *int-report_status* has entered its trap t-165, then the managing-component will transit to its state *viewing status*. There subprocesses s-167, s-169 and s-162 are going to be prescribed to *int-view_status*, *int-instantiate_methods* and *int-schedule_design* respectively. Subprocess s-163 is prescribed to *int-report_status* in order to be able to react to a new status report. From there, dependent on which trap *int-report_status* will enter, the managing-component will either transit to its state *next ToR defined* or return to its neutral state. The neutral state will be reached only when *int-report_status* has entered its trap t-163, which means the model is functioning properly. The state *next ToR defined* will be reached only when *int-report_status* has entered its trap t-164, which means the model is not functioning properly. However to be able to reach one of these two states *int-view_status* must have entered its trap t-167 too and also *int-instantiate_methods* must have entered its trap t-169.

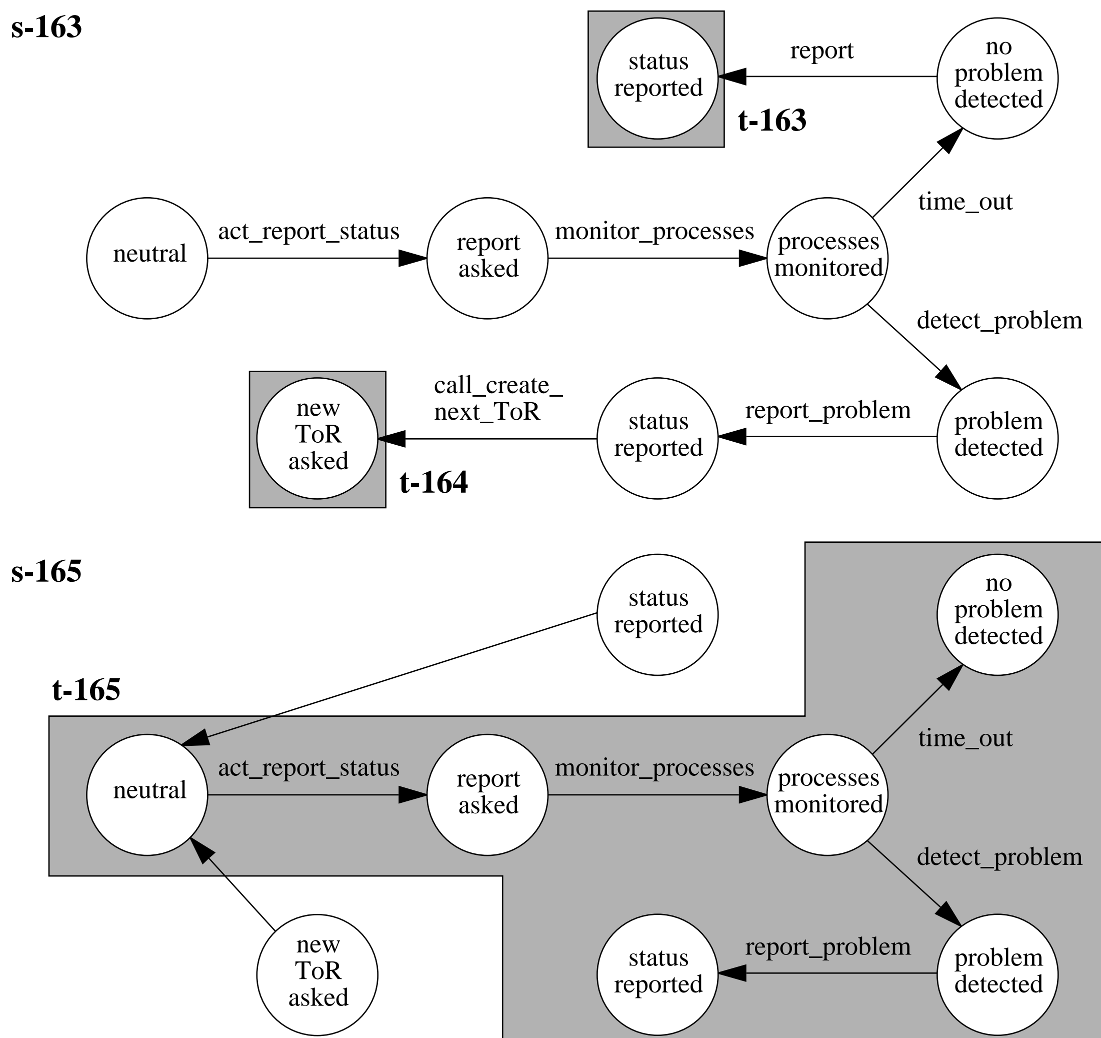


Figure 128. int-report_status's subprocesses and traps w.r.t. managing

In both states *next ToR defined* and *neutral* subprocesses s-166, s-168 and s-165 then will be prescribed to *int-view_status*, *int-instantiate_methods* and *int-report_status* respectively. In the state *next ToR defined* the managing-component will create a new ToR for the software process model, possibly based on the current ToR. When the new ToR has been created and *int-schedule_design* is in its trap t-162, the managing-component will transit to its state *design*

scheduled. In this state subprocess s-160 will be prescribed to *int-schedule_design*, thereby forcing the logistics-component to create not only a new setup for the model, but also a setup for the change steps necessary to be able to switch from the current model to that new model. As soon as *int-schedule_design* enters its trap t-160, which means the new model has been instantiated, *int-instantiate_methods* enters its trap t-168, which denotes the new model has been enacted and the status of the new enacted model is asked to be viewed, if furthermore *int-view_status* has entered its trap t-166, which means the status can be viewed, and *int-report_status* has entered its trap t-165, then the managing-component will transit to its state *viewing status* again. When this new model is also not functioning properly, the managing-component will again transit to its state *next ToR defined* and the whole cycle of scheduling the design and viewing the status starts all over again. Eventually however a properly functioning model will be developed and the managing-component will return to its state *neutral*.

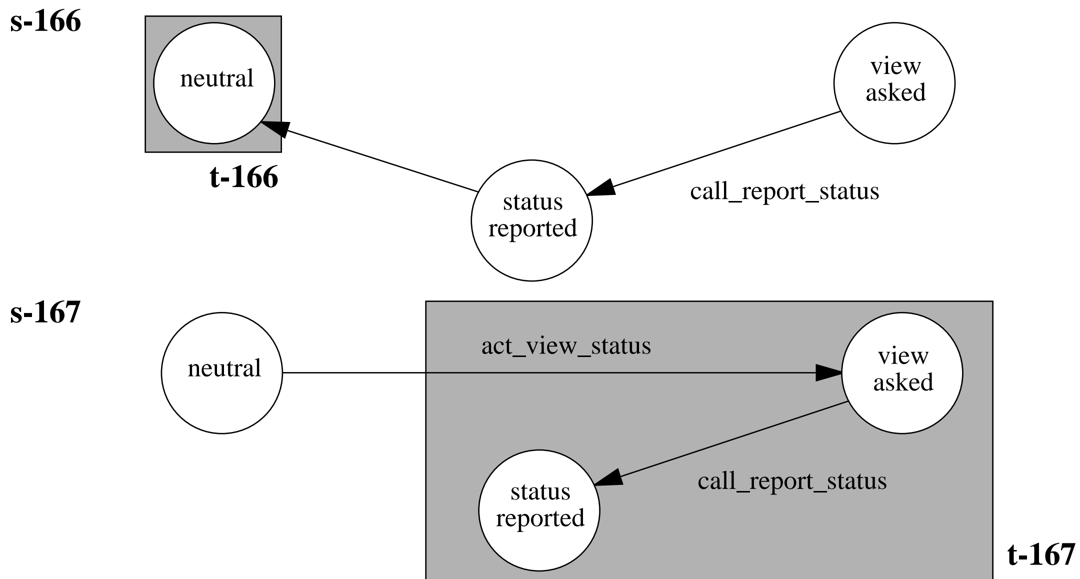


Figure 129. int-view_status's subprocesses and traps w.r.t. managing

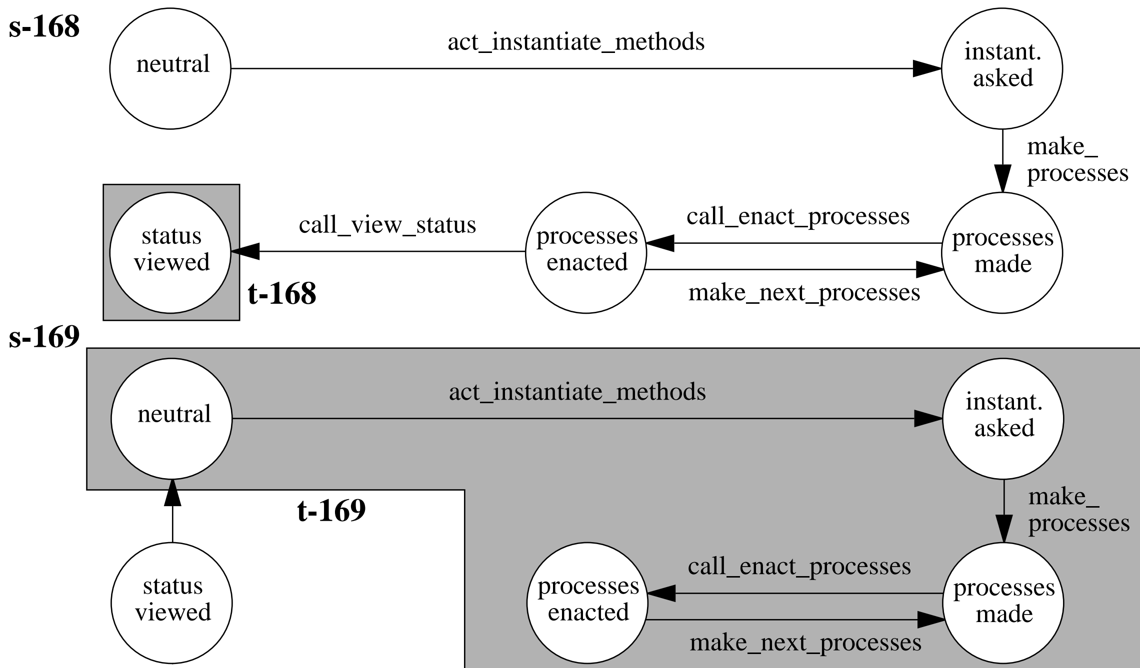


Figure 130. int-instantiate_methods's subprocesses and traps w.r.t. managing

The managing-component is in its neutral state again. Every now and then the status of the current software process model will be asked to be viewed again, in order to see whether problems have occured since the last time the model has been viewed. In that case *int-report_status* must be in its trap t-165, *int-view_status* must be in its trap t-166 and of course *view_status* must have been called, which has not been modelled. Only then the managing-component will make the transition to its state *viewing status*. Also in the neutral state the managing-component can decide, without viewing the status of the model first, to change the current model. In that case it will transit to its state *next ToR created*. Of course in state *neutral* the managing-component can also transit to its state *first ToR created* again in order to create a ToR for a model describing another software process.
Figure 131 shows the managing-component as manager of *int-schedule_design*, *int-view_status* (called operations), *int-report_status* and *int-instantiate_methods* (calling operations).

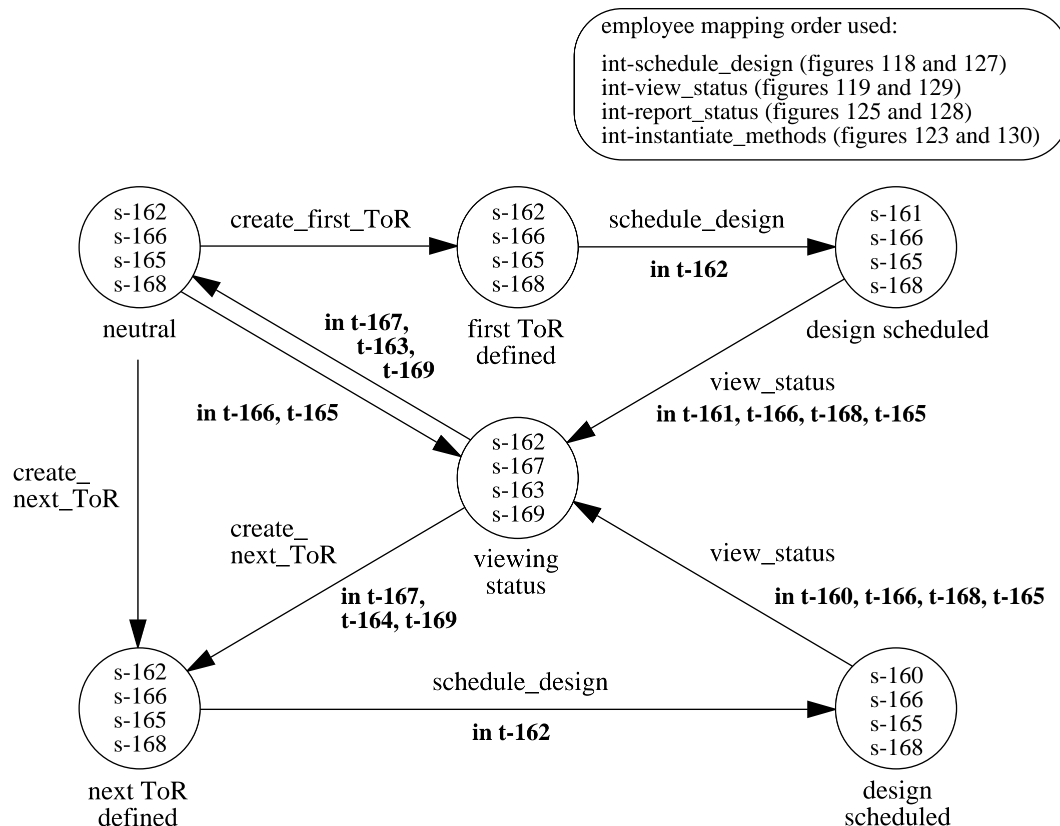


Figure 131. managing-component: viewed as manager of 4 employees

Second the communication specification between the logistics-component and its employees *int-schedule_design*, *int-instantiate_methods*, *int-generate_change_setup* and *int-get_methods* will be given. The operations *report_proposal*, *modify_setup*, *report_method_design* and *report_change_steps* have not been modelled, so the communication between the logistics-component and these operations will not be specified. Also the communication between the logistics-component and its employee *int-generate_setup* will not be specified, as it is not that important for our discussion. Furthermore the specification of this part of the communication is rather straigtforward.
The logistics-component (Figure 136) starts in its neutral state. There subprocess s-170 has been prescribed to *int-schedule_design*. If this behaviour has entered its trap t-170, which means a setup for a model has been asked, then the logistics-component will transit to its next state *setup generated*. In that state subprocess s-172 will be prescribed to *int-schedule_design* and a setup, which answers to the given ToR, will be generated. As soon as this setup has been generated

and *int-schedule_design* has entered its trap t-172, the logistics-component will report this setup to the managing-component and transit to its state *setup reported*. There subprocess s-170 will be prescribed again to *int-schedule_design*. When *int-schedule_design* again enters its trap t-170, which now means the setup has been disapproved of and a better setup is asked, the logistics-component will have to modify the setup and return to its state *setup generated*. As soon as the modification of the setup has been completed and *int-schedule_design* has entered its trap t-172 again, the logistics-component will report the modified setup to the managing-component and return to its state *result reported*. This cycle can be repeated several times.

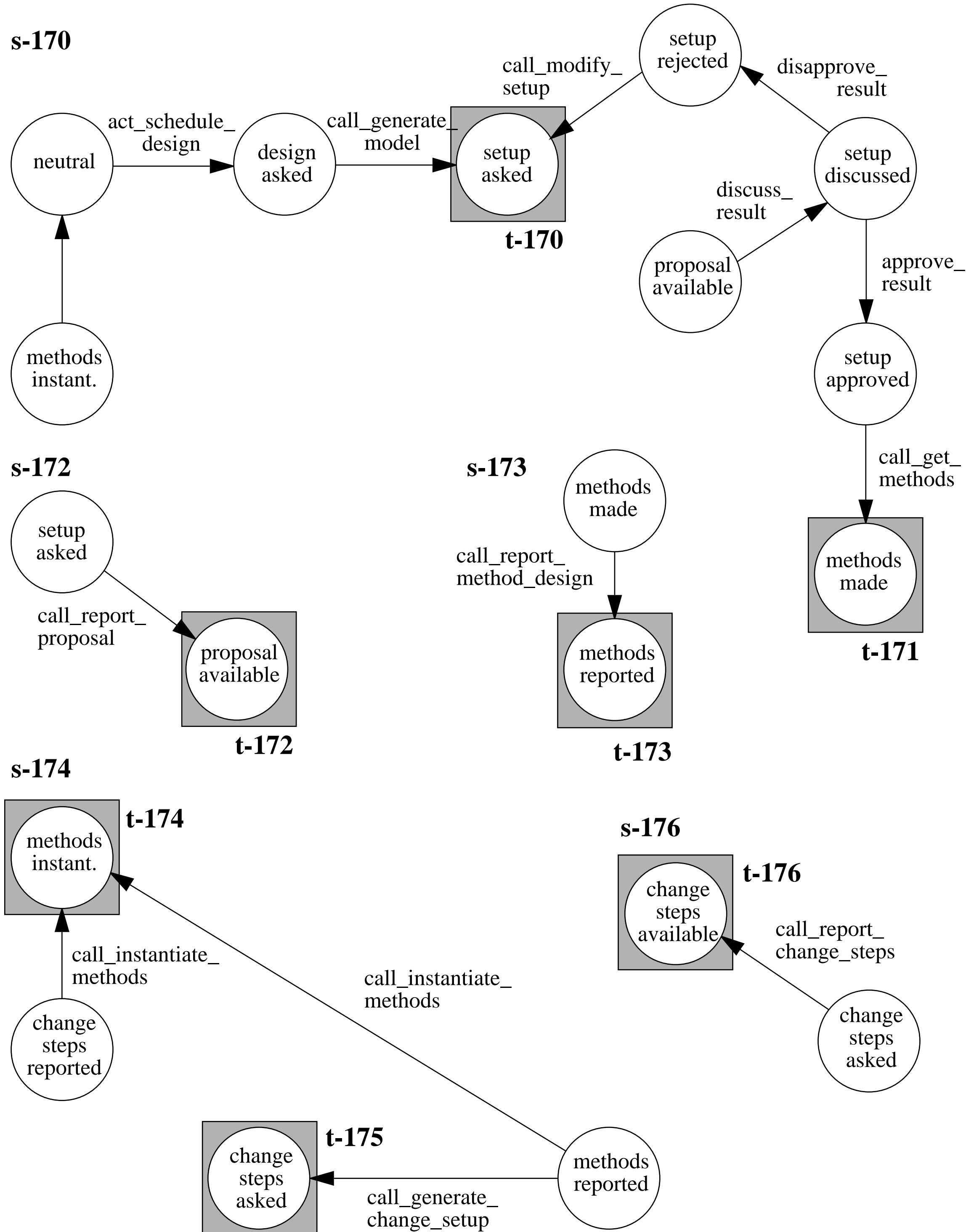


Figure 132. int-schedule_design's subprocesses and traps w.r.t. logistics

Eventually *int-schedule_design* will enter its trap t-171, which means the setup has been approved of and corresponding methods are being asked. If also *int-get_methods* is in its trap t-182, then the logistics-component will make the transition from its state *result reported* to its state *methods available*. There subprocesses s-173 and s-183 are going to be prescribed to *int-schedule_design* and *int-get_methods* respectively. The logistics-component will now order the technology-component to make the methods and also the logistics-component will supervise the development of these methods, i.e. approve or disapprove of them. In the latter case the logistics-component will order the technology-component to modify the methods.
If *int-get_methods* has entered its trap t-183, which means the methods have been developed and approved of, and *int-schedule_design* has entered its trap t-173, which means the methods are asked to be reported to the managing-component, then the logistics-component will transit to its next state *methods reported*. There subprocesses s-174 and s-182 will be prescribed to *int-schedule_design* and *int-get_methods* respectively. The methods will now be reported to the managing-component. Note however that the managing-component cannot overrule the decision of the logistics-component and disapprove of the methods.

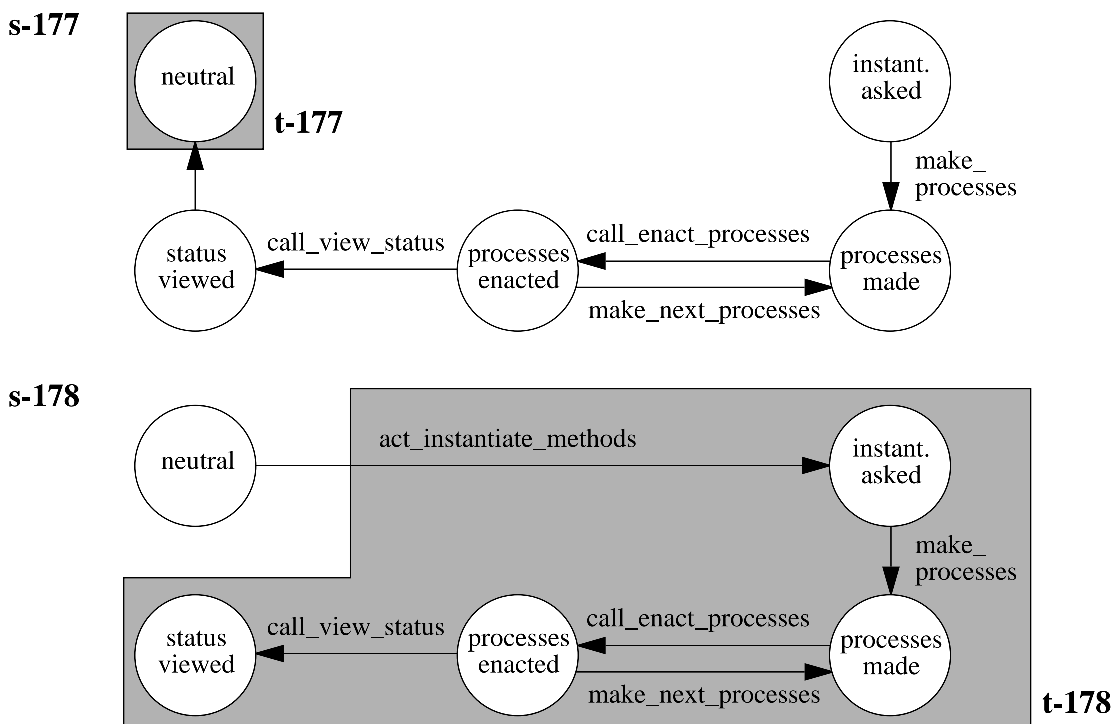


Figure 133. int-instantiate_methods's subprocesses and traps w.r.t. logistics

If *int-schedule_design* has entered its trap t-174, which means instantiation of the model is asked, if moreover *int-instantiate_methods* is in its trap t-177, which means instantiation of the model can be asked, then the logistics-component will transit to its state *methods instantiated*. Note that *int-schedule_design* will enter its trap t-174 only when the first model for a particular software process has to be instantiated, i.e. when in the external behaviour of the managing-component *schedule_design* is preceded by *create_first_ToR*.
Otherwise *int-schedule_design* enters its trap t-175, which means the evolutionary transformation from the current model to the just developed model has to be prepared first. Consequently the logistics-component is going to transit to its state *change setup generated*. Note that *int-generate_change_setup* was already waiting in its state *neutral*, which means trap t-179 had been entered in the past. In the state *change setup generated* subprocesses s-176 and s-181 will be prescribed to *int-generate_change_setup* and *int-schedule_design* respectively. Now a setup for the (first) change step will be produced. If trap t-180 has been entered by *int-*

generate_change_set_up, which means the setup for the change step has been generated and corresponding methods are being asked, if also *int-get_methods* has entered its trap t-182, which means the methods can be asked, then the logistics-component will get these methods and transit to its state *change steps available*. The logistics-component will order the technology-component to make the methods necessary for the change step and will approve or disapprove of the change step developed by the technology-component. After the change step has been approved of, i.e. when *int-get_methods* has entered its trap t-183, possibly a next change step has to be developed, as the evolutionary change cannot always be executed in one step. In that case *int-generate_change_setup* will enter its trap t-180 and the logistics-component will return to its state *change setup generated*. There subprocesses s-181 and s-182 will be prescribed again to *int-generate_change_setup* and *int-get_methods* respectively. A new change step can be developed now. This cycle will be repeated as many times as necessary in order to be able to complete the evolutionary transformation.
Eventually *int-generate_change_setup* will enter its trap t-179, which means enough change steps have been constructed by the technology-component to perform the evolutionary change. If more *int-get_methods* has entered its trap t-183, if furthermore *int-schedule_design* has entered its trap t-176, which means the change step(s) are asked to be reported to the managing-component, then the logistics-component will transit to its state *change steps reported*. There subprocess s-174 will be prescribed to *int-schedule_design*. Note that subprocess s-179 will still be prescribed to *int-generate_change_setup*. The change steps will now be reported to the managing-component. Note however that the managing-component cannot overrule the decision of the logistics-component and disapprove of the change steps.

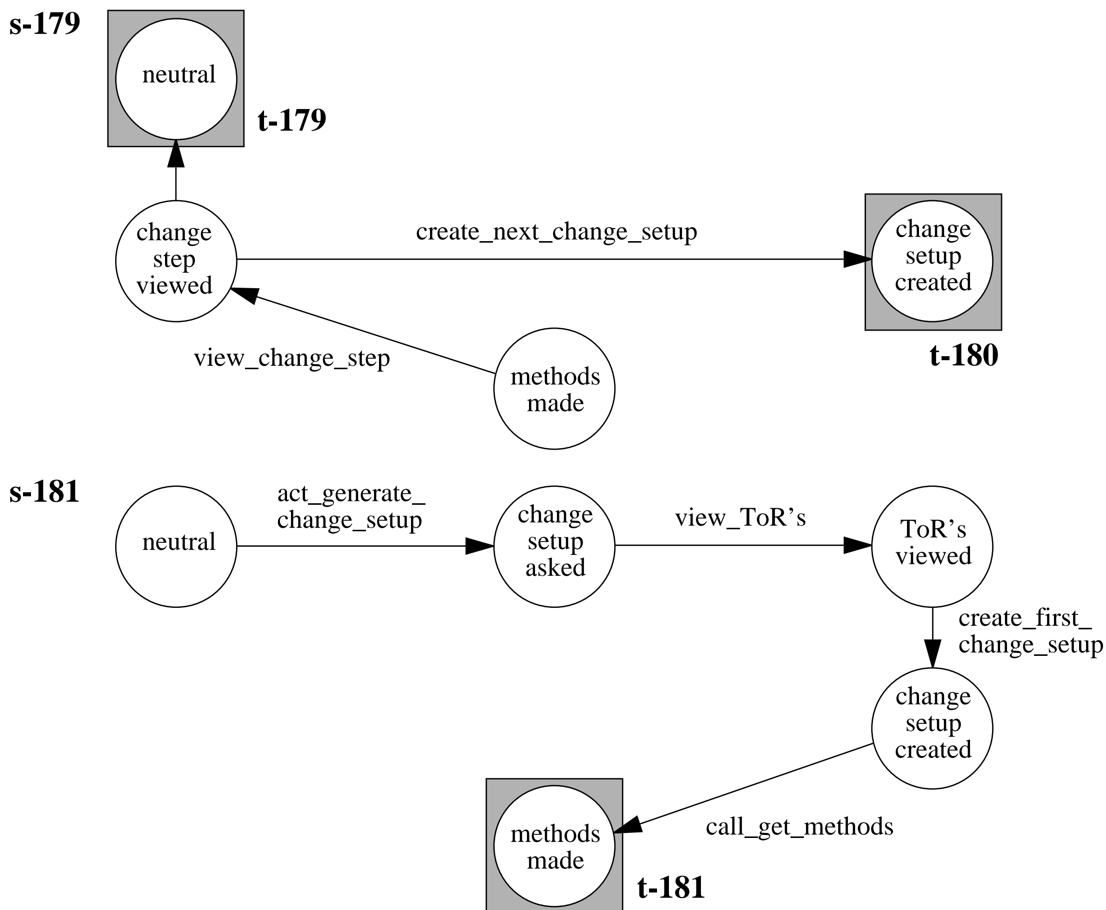


Figure 134. int-generate_change_setup's subprocesses and traps w.r.t. logistics

When *int-schedule_design* enters its trap t-174, which means instantiation of the model is asked, and if also *int-instantiate_methods* has entered its trap t-177, which means instantiation of the model can be asked, then the logistics-component will make the transition to its state *methods instantiated*. There subprocesses s-170 and s-178 will be prescribed to *int-schedule_design* and *int-instantiate_methods* respectively. In this state *enact_processes* will be called for every intermediate step and of course for the new model. Note that the logistic-component knows how many change steps have been generated and therefore knows how many times *enact_processes* has to be called. The conditions under which the processes can be enacted will not be checked by the logistics-component. They will be checked within *int-enact_processes* itself. If *int-instantiate_methods* has entered its trap t-178, then the logistics-component can return to its neutral state.

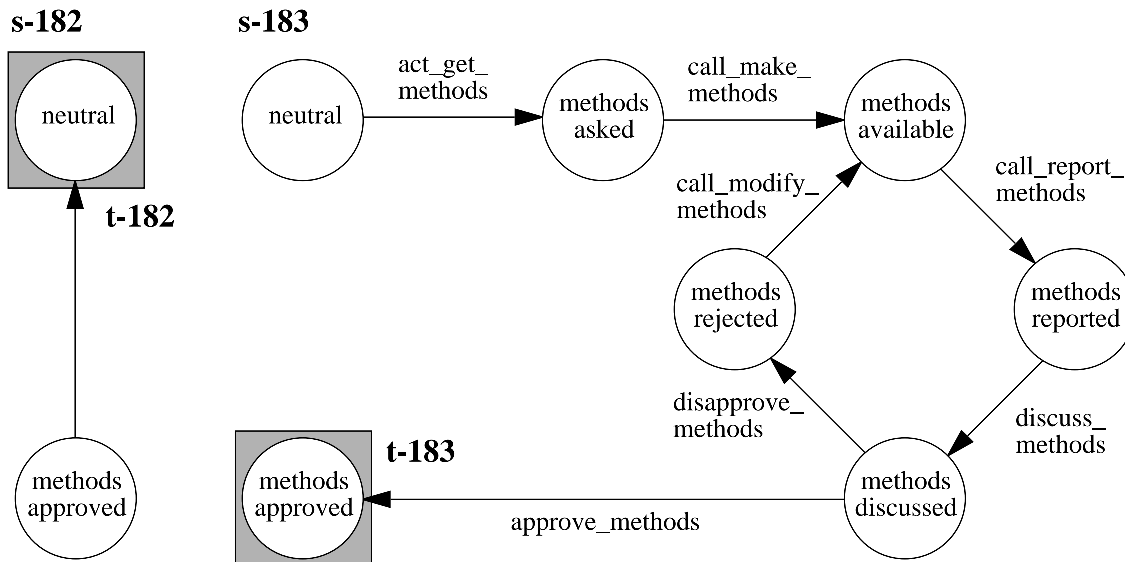


Figure 135. int-get_methods's subprocesses and traps w.r.t. logistics

Figure 136 shows the logistics-component as manager of *int-instantiate_methods*, *int-get_methods* (called operations), *int-schedule_design* (a calling operation) and *int-generate_change_setup* (both a called and a calling operation).

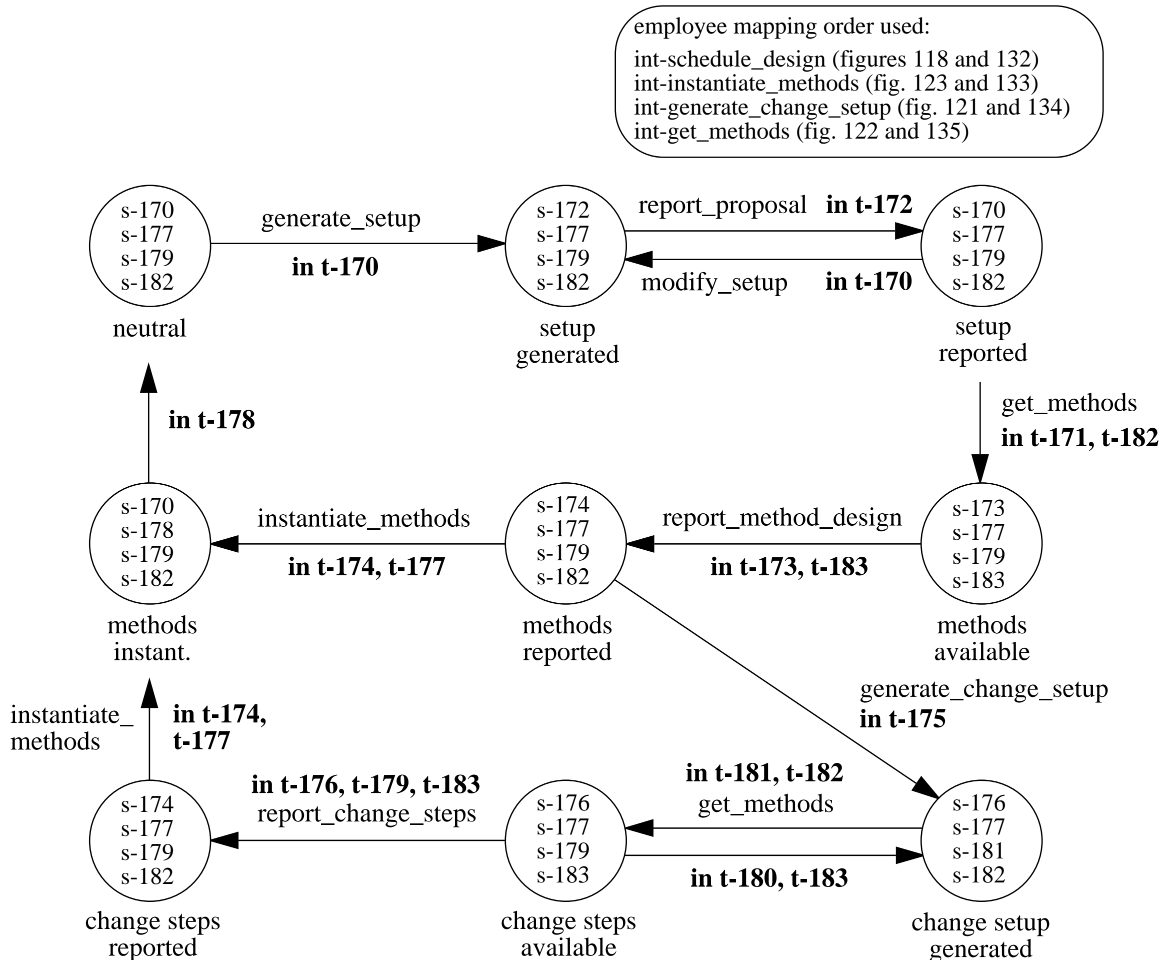


Figure 136. the logistics-component: viewed as manager of 4 employees

Finally the communication between the administering-component and its employees *int-view_status*, *int-report_status*, *int-enact_processes* and *int-instantiate_methods* will be specified. The internal behaviour of the operations *report_status* and *enact_processes* belong to the administering-component itself. The internal behaviours of operations calling the operations of the administering-component are *int-view_status* and *int-instantiate_methods*.
The administering-component (Figure 141) starts in its state *neutral*. If *int-instantiate_methods* has entered its trap t-192, which means the enactment of the just instantiated methods has been called, if further *int-enact_processes* has entered its trap t-190, which means a model can be enacted, then the administering-component will transit to its state *processes enacted*. There subprocesses s-193 and s-191 are going to be prescribed to *int-instantiate_methods* and *int-enact_processes* respectively. Now all (intermediate) processes of the same (intermediate) phase are being enacted. As soon as *int-enact_processes* enters its trap t-191, which means all (intermediate) processes have been enacted, and *int-instantiate_methods* enters its trap t-193, the administering-component will return to its neutral state. There subprocesses s-190 and s-192 will be prescribed to *int-enact_processes* and *int-instantiate_methods* respectively. This cycle will be repeated as many times as necessary in order to enact every change step from an old model to a new model. Note that in the internal behaviour of *instantiate_methods* it is known how many times *enact_processes* has to be called.

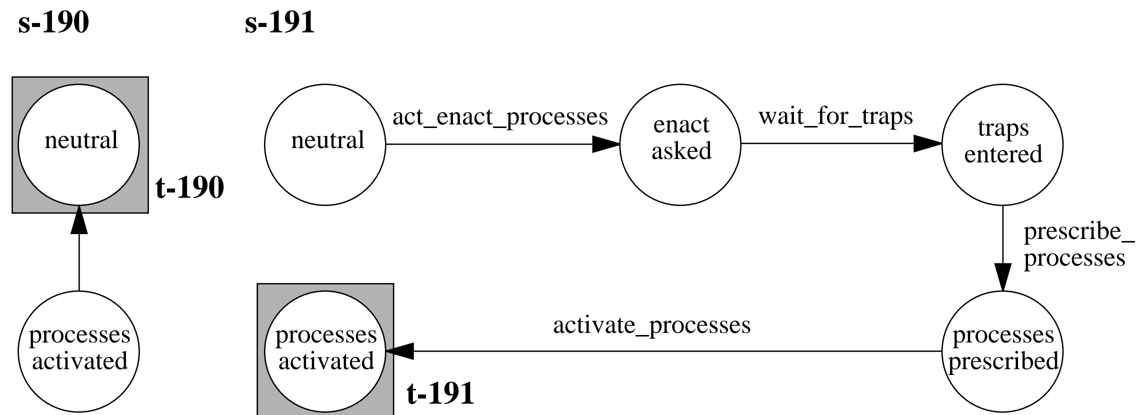


Figure 137. int-enact_processes's subprocesses and traps w.r.t. administering

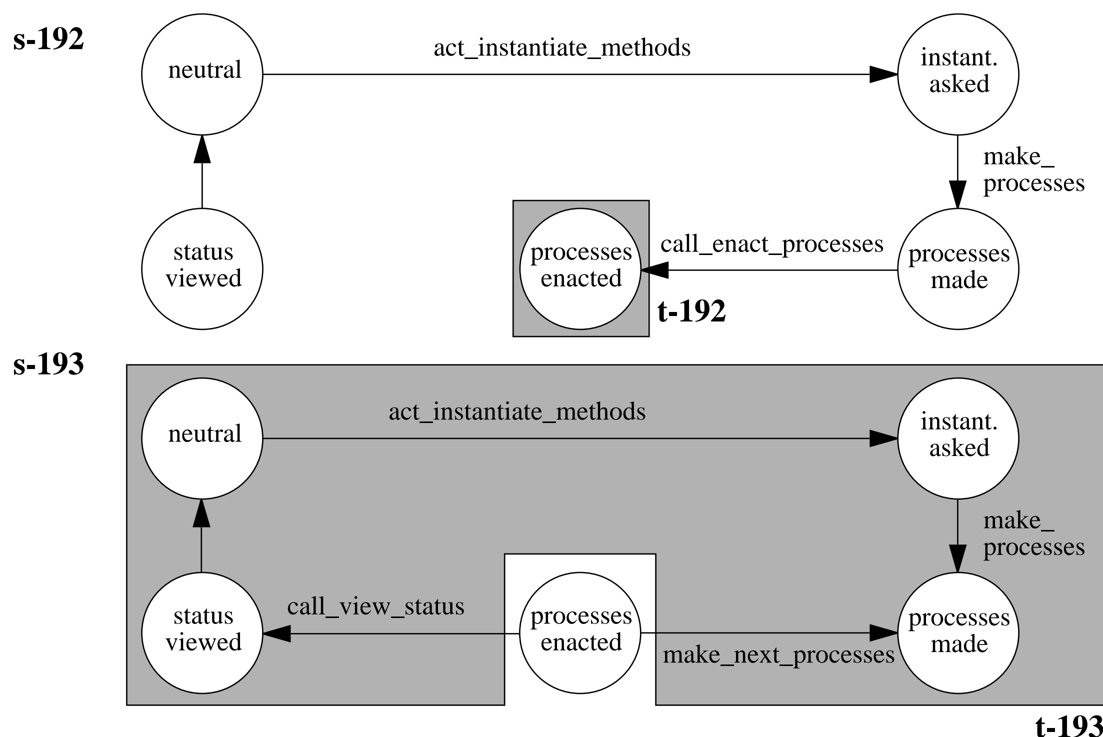
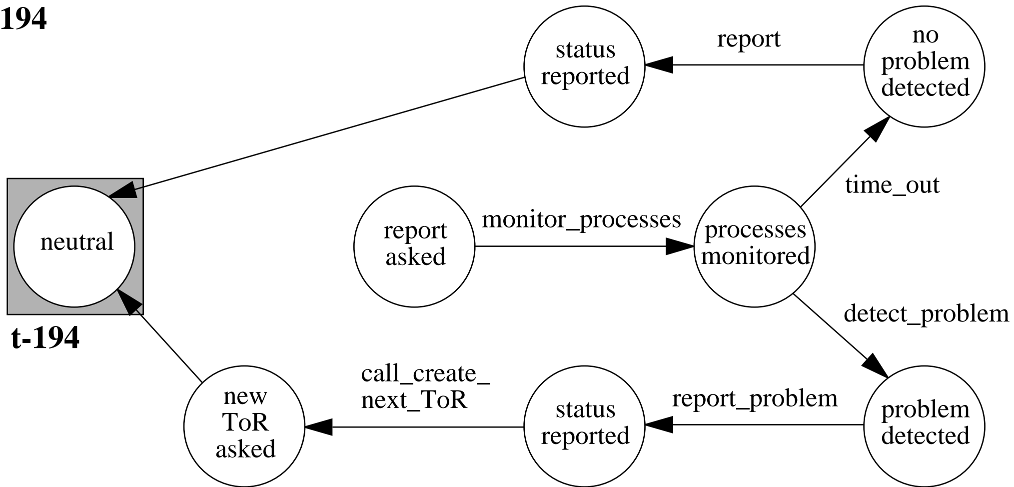


Figure 138. int-instantiate_methods's subprocesses and traps w.r.t. administering

The administering-component will make the transition from its state *neutral* to its state *status reported*, when *int-report_status* has entered its trap t-194, which means a status report can be given, and *int-view_status* has entered its trap t-196, which means a status report has been asked. In the state *status reported* subprocesses s-195 and s-197 will be prescribed to *int-report_status* and *int-view_status* respectively. Now the administering-component will start monitoring all running processes to check for possible problems. If *int-report_status* has entered its trap t-195, if moreover *int-view_status* has entered its trap t-197, then the administering-component will go back to its state *neutral*. There subprocesses s-194 and s-196 will be prescribed to *int-report_status* and *int-view_status* respectively.
Note that *view_status* can be called, from within the internal behaviour of *instantiate_methods*, before the new model has been enacted completely. In *int-view_status* therefore *report_status* can be called before the new model has been enacted completely. It might seem as if this is in-

consistent. However *report_status* can be executed only after the administering-component has reached its neutral state and this state can be reached only when the model has been enacted completely.



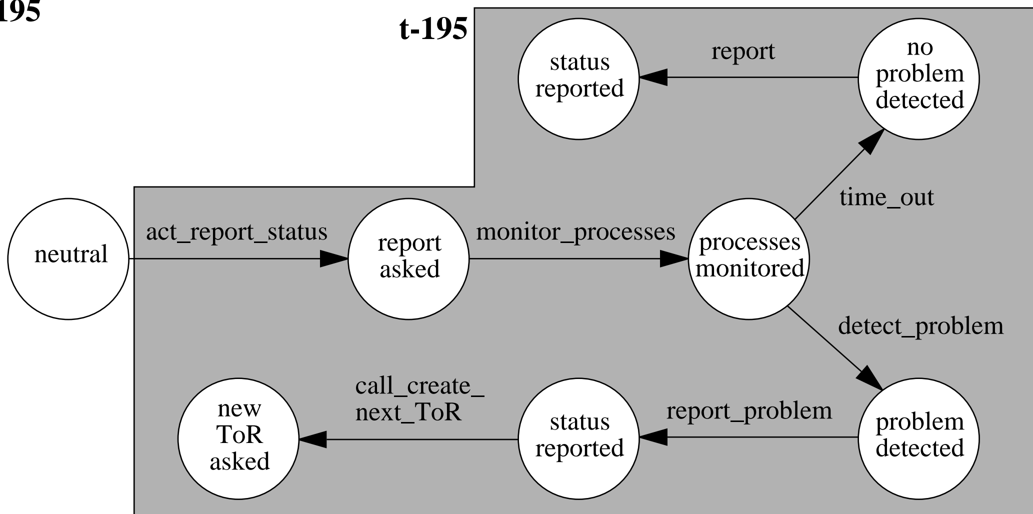


Figure 139. int-report_status’s subprocesses and traps w.r.t. administering

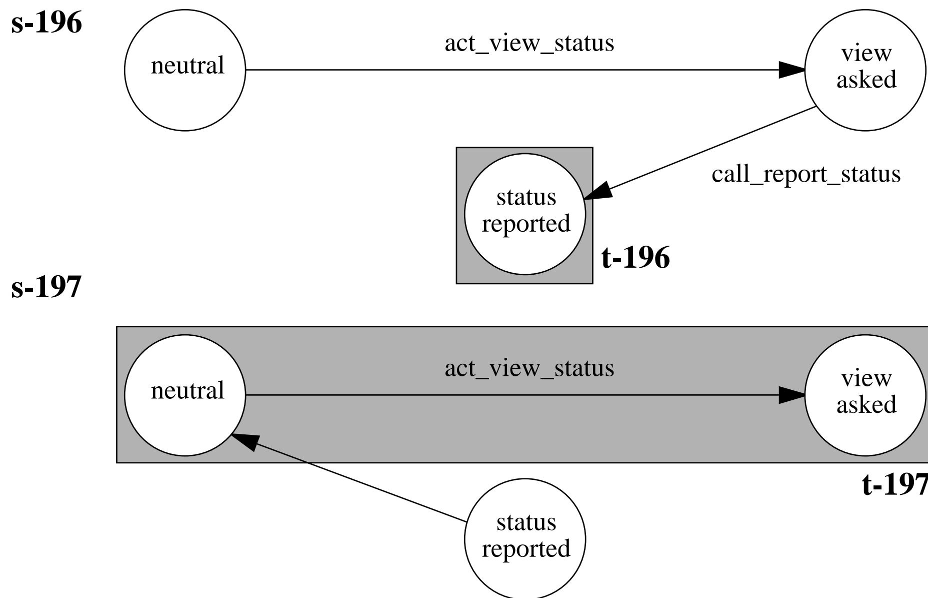


Figure 140. int-view_status's subprocesses and traps w.r.t. administering

Figure 141 shows the administering-component as manager of *int-enact_processes*, *int-report_status* (called operations), *int-instantiate_methods*, *int-view_status* (calling operations).

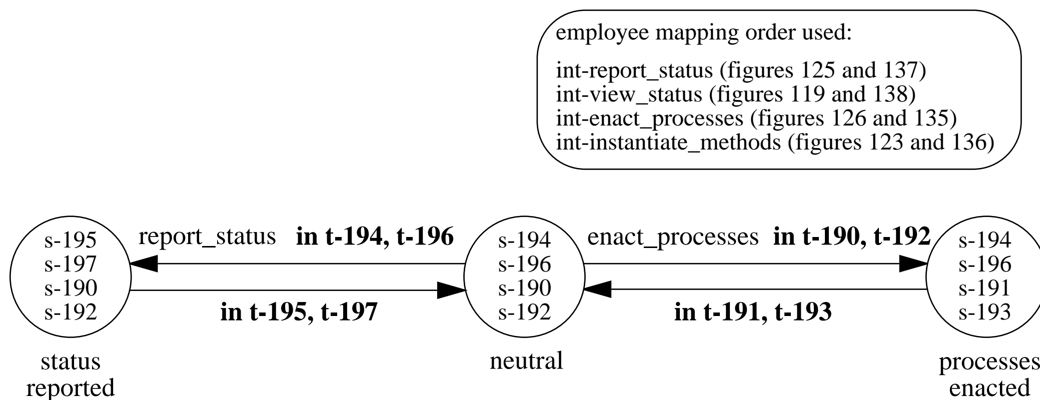


Figure 141. the administering-component: viewed as manager of 4 employees

As mentioned before the communication between the technology-component and its employees will not be specified here. Note that in a first model describing the (communication between the) components the logistics-component did not only create a setup for the change steps, but also the methods for the change steps. Consequently during the phase of designing these change steps there was no communication between the logistics- and the technology-component. Later we added this communication, as we decided that the technology-component had to create the methods for the change steps, just like it had to create the methods for the model. Therefore some external and internal behaviours had to change. Changing this model to answer to the new conditions was very easy. So we can say it is a flexible model, that can be adapted easily. Which concludes our discussion of the communication of the basic components in PMMS.

8 Conclusions and future research

In this thesis the evolution of a particular software processs has been described by means of WODAN and PMMS. This thesis shows that both WODAN and PMMS are suited to describe such an evolution. WODAN is especially suited when all details of the evolutionary transformation itself have to be given explicitly. PMMS does not show these details, but clarifies some issues on evolutionary change on a more global level by providing a well-structured approach in presenting the process steps of designing, instantiating and enacting. WODAN does not provide a structure for these issues. So in fact these methods can be thought of as being complementary.

First the WODAN approach has been used to describe evolutionary change. It provides a lot of insight in the actual evolutionary transformation of one model to another model. However the WODAN approach still has its shortcomings. In [2] guidelines were given on how to switch from one evolution stage to another. With these guidelines inconsistencies, which arise as a consequence of the change, can be solved or avoided. In [2] these guidelines have been applied to small models, which consist of only one process. This thesis extends this approach to larger models. While changing a (larger) model intermediate steps can occur. One can assume that the larger the model, the higher the number of intermediate steps. In our case three intermediate steps have been introduced. However the guidelines did not tell how (large) models could be changed best, i.e. how the number of intermediate steps could be minimized. A topic for furher research is to analyse this. It is very likely that guidelines or rules for handling the change of large models can be established, thereby optimizing the WODAN approach.

Another topic for further reseach is to change WODAN so that it will behave in accordance with the Socca-conventions (see [1]). WODAN now prescribes (new) processes to both internal and external behaviours. As mentioned before these (new) processes can be viewed as subprocesses of anachronistic processes. In Socca however only external behaviours can prescribe subprocesses to internal behaviours. So WODAN is not yet in accordance with the Socca-conventions. The solution to this problem is to ensure that WODAN no longer manages the internal behaviours directly. In that case the new subprocesses of the internal behaviours should be passed through somehow via the external behaviours, which actually are the only possible manager processes within the Socca approach.

Second in this thesis the evolutionary change has been described by means of PMMS, and also an attempt has been made to describe the basic components of the PMMS-model by means of Socca. By doing this the ideas behind PMMS were expressed explicitly. It is a topic of future research to specify these basic components more exactly, especially the administering-component, as *enact_processes* has not been worked out completely yet. Possibly some of the ideas from WODAN can be used, thereby creating a more complete method to describe evolutionary change.

9 References

1 Engels G., Groenewegen L.: *SOCCA: Specifications of Coordinated and Cooperative Activities.* Technical Report 94-10, University of Leiden, Department of Computer Science, February 1994.

2 Wulms A.: *Adaptive software process modelling with SOCCA and PARADIGM*. M.Sc. thesis, University of Leiden, Department of Computer Science, 1995.

3 Willemsen, R.: *TEMPO and SOCCA, concepts, modelling and comparison*. M.Sc. thesis, University of Leiden, Department of Computer Science, 1995.

4 Verkoren E.H.: *Een kwaliteitsaudit in IT-organisaties*. M.Sc. University of Leiden, Department of Computer Science, 1993.

5 Kellner M., Feiler P., Finkelstein A., Katayama T., Osterweil L., Penedo M., Rombach H.: *ISPW-6 Software Process Example*. In: Proc. of the 6th Int. Software Process Workshop: support for the software process. Japan, October 1991.

6 Penedo M., Finkelstein A., Futatsugi K., Ghezzi C., Kaiser G., Narawanaswamy K., Perry D.: *ISPW-9 Life-cycle (Sub) Process Demonstration Scenario*, March 1994.

7 Groenewegen L.: *Parallel Phenomena 1-14*. Technical Reports 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.

8 Groenewegen L.: *PMMS and Paradigm, Simple Banking Example*. Personal notes for a talk held in Manchester, 1995.

9 Snowdon R.: *An Example of Software Change*. In Derniame J.C. (ed.): *Software Process Technology.* Springer-Verlag, Lecture Notes in Computer Science 635, 1992.

Appendix A Simultaneous calls described in more detail

The operations *prepare_meeting* (figures 21, 34 and 40), *open_meeting* (figures 14 and 36) and *close_meeting* (figures 16 and 37) are operations that perform a call to all members of the board simultaneously. This is however a simplified representation of the realistic and more complicated behaviour. This more complicated behaviour will be given for the state *agenda_checked* (from the internal behaviour of *prepare_meeting*), which is reached after *call_check_agenda* has been performed. The method used will also hold for the states *confirm send* (also from *int-prepare_meeting*), *members joined* (from *int-open_meeting)* and *members left* (from *int-close_meeting*).
Suppose there are 3 members. Each of these three members can be either in a state in which the agenda has not been checked yet or in a state in which the agenda indeed has been checked. This means there must be 2^3 (=8) states to cover all possibilities. Note that whenever a state has been reached in which a particular member has checked its agenda, it is not possible to go to a state in which the same member has not checked its agenda yet. So there must be 3! (=6) different ways to go through the 8 states and reach the state *date selected*. This behaviour is shown in Figure 142. In this figure the state *agenda checked* has been replaced by a 3D-structure and there are 6 (= 3!) different ways, via the edges of that structure, to reach the state *date selected*. As mentioned before one can construct a similar STD to model *call_receive_confirmation*, *call_join_meeting* and *call_leave_meeting*. For n members a n-dimensional structure can be used.
Not only the internal behaviour of *int-prepare_meeting* changes in accordance with the refinement from Figure 142, also the subprocesses and traps w.r.t. a particular member change when replacing a part of *int-prepare_meeting* by the more exact representation of that part. Figure 143 shows the subprocesses and traps w.r.t. member 1. The upper plane of the 3D-structure contains all states, in which member1 has not checked its agenda yet. So this plane will be trap t-42a w.r.t. member1 and it will replace the original trap t-42 (Figure 40). The lower plane of the 3D-structure contains all states, in which member1 indeed has checked its agenda. So this plane and the state *date selected* will be trap t-44a w.r.t. member1 and will replace the original trap t-44. Also the subprocesses s-42 and s-44 will be replaced by the new subprocesses s-42a and s-44a. Of course these new subprocesses and traps also contain every other state the original simplified subprocesses and traps contain. The subprocesses and traps w.r.t. member2 (back and front planes) and member3 (left and right planes) can be constructed in a similar manner. Note that any of the three possible states *agenda checked (X,Y)* can only be reached when *int-prepare_meeting* has entered both trap t-44a w.r.t. memberX and trap t-44a w.r.t. memberY. From there it follows that the state *agenda checked (1,2,3)* can only be reached when *int-prepare_meeting* has entered trap t-44a w.r.t. every member. So only when every member has reacted to trap t-42a and consequently has prescribed subprocess s-44a, *int-prepare_meeting* can continue to go to the state *date selected*, as it is the internal behaviour of an operation of another class (Figure 40).
In the simplified representation it seemed that every member had to wait for all other members before returning to its neutral state, as trap t-44 could only be entered if state *date selected* had been reached. When looking at the trap-structure for the exact representation it is clear that this is not the case. Member1 can react to trap t-44a before state *date selected* has been reached and therefore the external behaviour of Member1 can return to its neutral state without waiting for the other members. Note that the order in which the members are called does not matter, as all n! orders are possible.

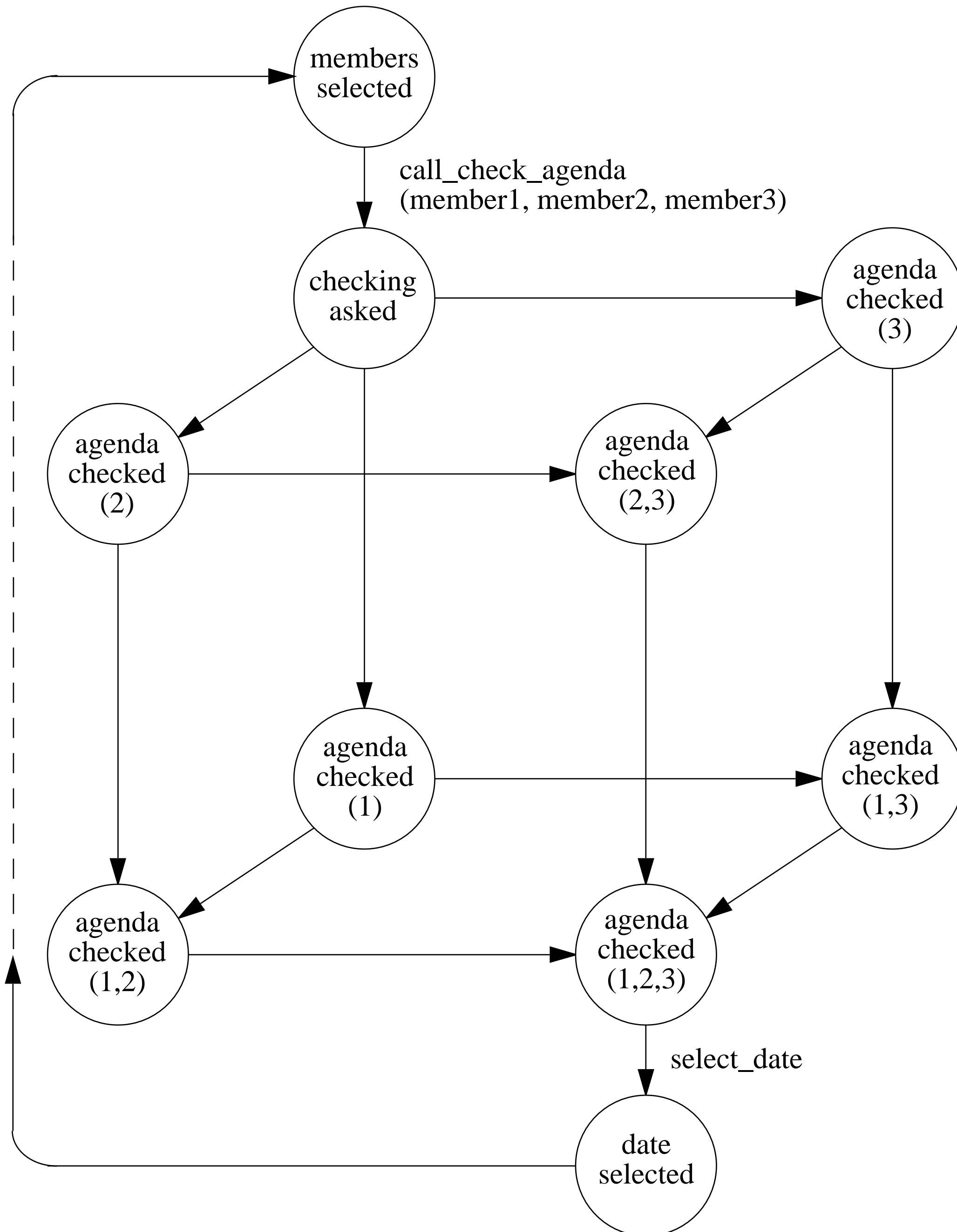
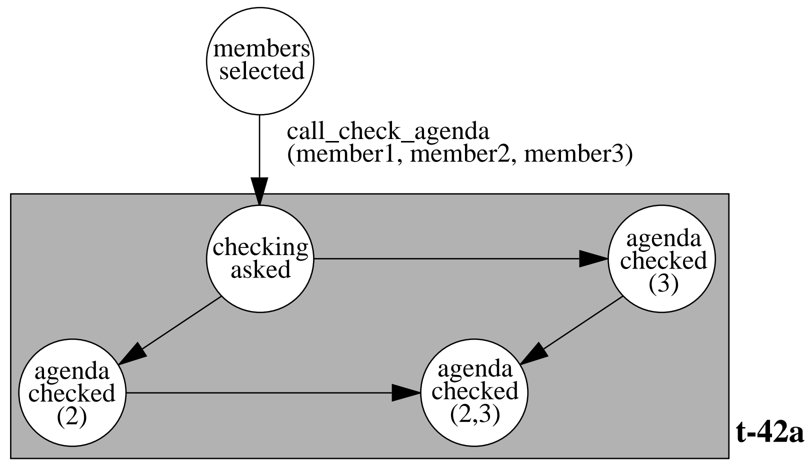
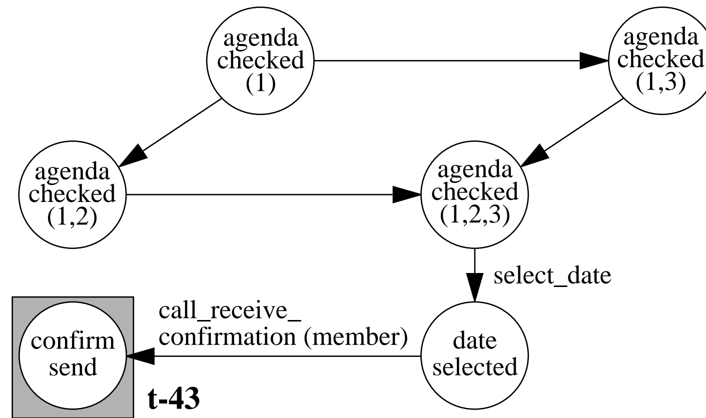


Figure 142. a part of int-prepare_meeting modelled in more detail

The dotted transition from the state *date selected* to the state *members selected* emphasizes that the rest of the operation remains as it is.

s-42a





s-44a

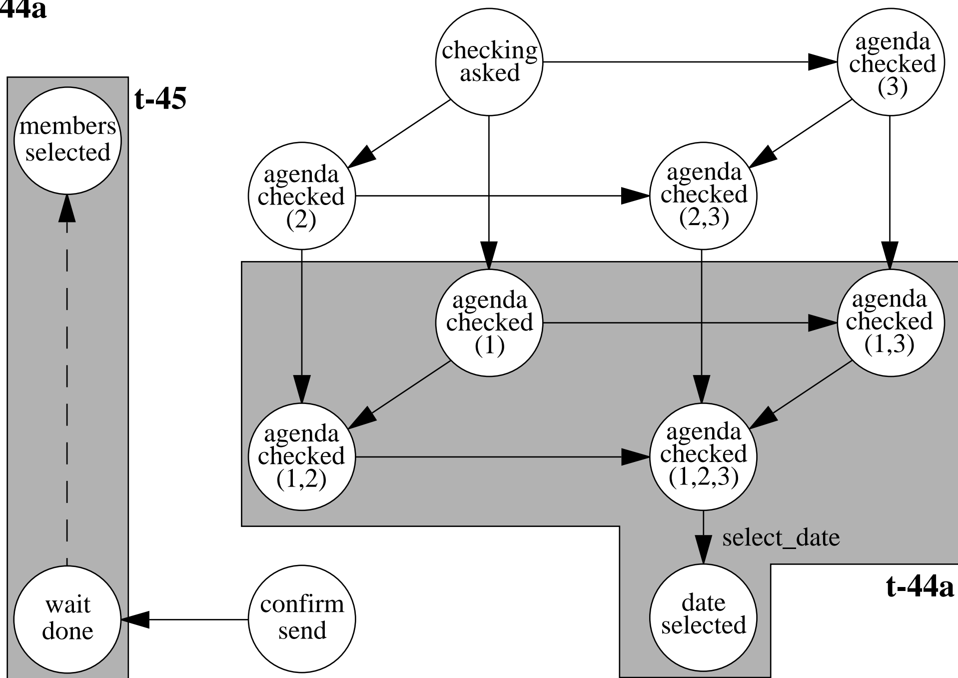


Figure 143. subprocesses and traps w.r.t. member1 modelled in more detail

Note that traps t-43 and t-45 remain unaltered by describing the state *agenda checked* in more detail. However they will change by describing the state *confirm send* in more detail.

Appendix B Logical transitions

As can be seen in Figure 71 logical expressions have been used to label transitions of the manager process CABSecretary. We call such transitions *logical transitions*. In Figure 144 the part of CABSecretary that uses the logical transitions is shown again. Logical transitions can be used to avoid multiple presences of the same state in the external behaviour of a class. These states are identical. A logical transition can occur only if a transition has been labeled with an operation containing a parameter. When, dependent on the value of this parameter in the label of the transition, different traps have to be entered and after that different subprocesses have to be prescribed, logical transitions can be used. In this case the parameter *size* causes a logical transition, as dependent on the value of this parameter to either *int-big_impact* or *int-small_impact* a different subprocess has to be prescribed.

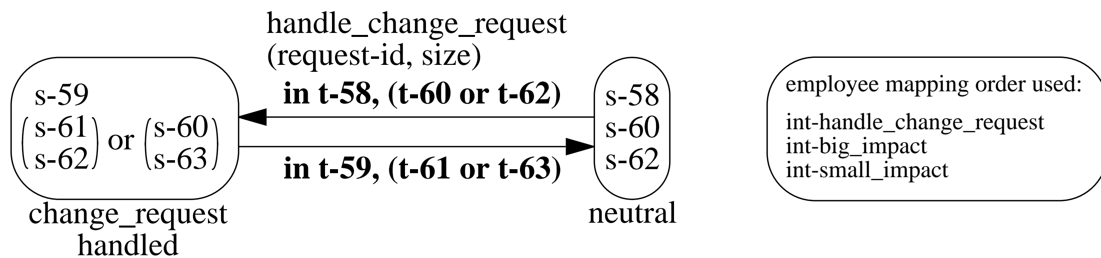


Figure 144. a part of the manager process CABSecretary using logical transitions

Figure 145 shows the same part of the manager process CABSecretary as shown in the above figure without using logical transitions. This figure is in accordance with the Socca-conventions. One can see that the only difference in the labels of both transitions is the value of the parameter *size*.

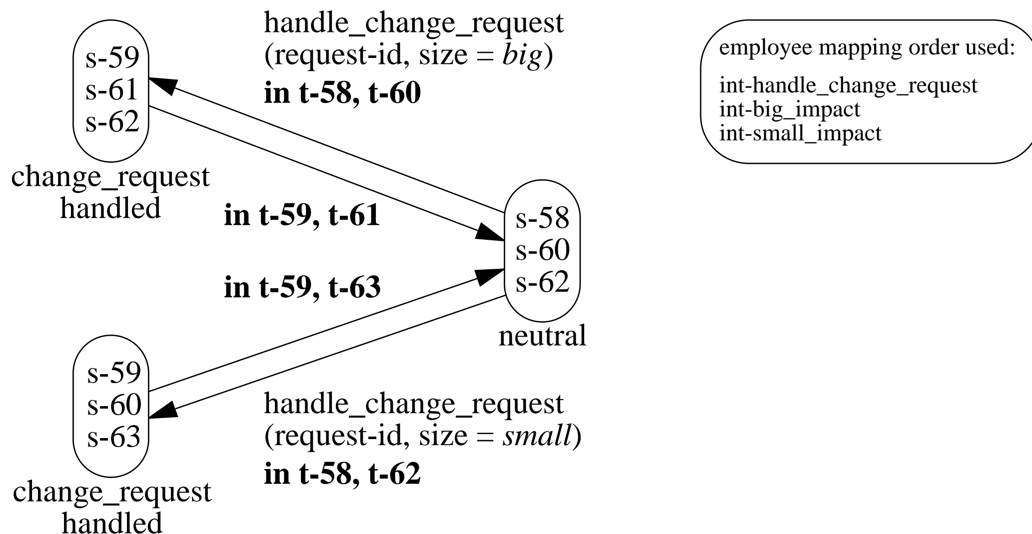


Figure 145. multiple transitions (and states) replacing logical transitions

Of course logical transitions could be avoided by replacing the operations *big_impact* and *small_impact* with a single operation *impact_estimated*, which is parametrized with a parameter *size*.

Appendix C List of subprocesses w.r.t. WODAN

subprocess	behaviour	figure
s-110	Design	[2, fig 13]
s-111	MainCAB	8
s-112	DepCAB	9
s-113	ProjectManager	11
s-114	int-request_for_change (old model)	13
s-115	int-prepare_meeting	21
s-116	int-do_meeting (old model)	15
s-117	int-monitor (old model)	[2, fig 12]
s-120	intermediate phase of MainCAB	92
s-121	first intermediate phase of DepCAB	93
s-122	first intermediate phase of int-req_for_change	90
s-130	second intermediate phase of DepCAB	98
s-131	intermediate phase of ProjectManager	103
s-132	intermediate phase of CABSecretary	97
s-133	second intermediate phase of int-req_for_change	95
s-134	first intermediate phase of int-prepare_meeting	101
s-140	NewDesign	51
s-141	NewCAB	50
s-142	CABSecretary	48
s-143	Request	49
s-144	int-request_for_change (new model)	52
s-145	second intermediate phase of int-prepare_meeting	106
s-146	intermediate phase of int-do_meeting	105
s-147	int-monitor (new model)	63
s-150	int-do_meeting (new model)	61

Table 1: Subprocesses w.r.t. WODAN

Appendix D List of figures

1. Class diagram: classes and IS-A and Part-Of relationships 14
2. Class diagram: attributes and operations 14
3. Class diagram: classes and general relationships 15
4. Import/export diagram 16
5. Import list 16
6. CAB: possible STD of the external behaviour 17
7. CAB (solution 1): the interleaved version 18
8. MainCAB (solution 3): Main-STD of the external behaviour 18
9. DepCAB (solution 3): Dep-STD of the external behaviour 19
10. UserRepresentative: STD of the external behaviour 19
11. ProjectManager: STD of the external behaviour 20
12. DesignEngineer: STD of the external behaviour 20
13. int-request_for_change 21
14. int-open_meeting 22
15. int-do_meeting 22
16. int-close_meeting 23
17. int-do_change 23
18. int-join/leave_meeting 23
19. int-check_agenda 24
20. int-receive_confirmation 24
21. int-prepare_meeting 25
22. int-request_for_change's subprocesses and traps w.r.t. MainCAB 27
23. DepCAB's subprocesses and traps w.r.t. MainCAB 28
24. MainCAB: viewed as manager of 2 employees 29
25. int-open_meeting's subprocesses and traps w.r.t. DepCAB 30
26. int-do_meeting's subprocesses and traps w.r.t. DepCAB 30
27. int-close_meeting's subprocesses and traps w.r.t DepCAB 31
28. int-do_change's subprocesses and traps w.r.t. DepCAB 31
29. int-prepare_meeting's subprocesses and traps w.r.t. DepCAB 32
30. DepCAB: viewed as manager of 5 employees 33
31. int-check_agenda's subprocesses and traps w.r.t. CABMember 34
32. int-request_for_change's subprocesses and traps w.r.t. ProjectManager 35
33. int-receive_confirmation's subprocesses and traps w.r.t CABMember 35
34. int-prepare_meeting's subprocesses and traps w.r.t. ProjectManager 36
35. int-join/leave_meeting's subprocesses and traps w.r.t. CABMember 37
36. int-open_meeting's subprocesses and traps w.r.t. CABMember 38
37. int-close_meeting's subprocesses and traps w.r.t. CABMember 38
38. int-do_change's subprocesses and traps w.r.t. Projectmanager 39
39. ProjectManager: viewed as manager of 8 employees 40
40. int-prepare_meeting's subprocesses and traps w.r.t. CABMember 42
41. DesignEngineer: viewed as manager of 6 employees 43
42. UserRepresentative: viewed as manager of 6 employees 44
43. Class diagram of the new model: classes and IS-A and Part-Of relationships 46
44. Class diagram of the new model: attributes and operations 47
45. Class diagram of the new model: classes and general relationships 48
46. Import/export diagram 49
47. Import list 49
48. CABSecretary: STD of the external behaviour 50

49. Request: STD of the external behaviour 50
50. NewCAB: STD of the external behaviour 51
51. NewDesign: STD of the external behaviour 52
52. int-request_for_change 53
53. int-add_to_list 54
54. int-send_list 54
55. int-handle_change_request 54
56. int-reject_request 55
57. int-big_impact 55
58. int-small_impact 55
59. int-do_change 56
60. int-request_for_meeting 56
61. int-do_meeting (new version) 57
62. int-schedule_and_assign_tasks 58
63. int-monitor 58
64. int-add_to_list’s subprocesses and traps w.r.t. CABSecretary 59
65. int-request_for_change’s subprocesses and traps w.r.t. CABSecretary 60
66. int-handle_change_request’s subprocesses and traps w.r.t. CABSecretary 61
67. int-big_impact’s subprocesses and traps w.r.t. CABSecretary 62
68. int-small_impact’s subprocesses and traps w.r.t. CABSecretary 62
69. int-send_list’s subprocesses and traps w.r.t. CABSecretary 63
70. int-request_for_meeting’s subprocesses and traps w.r.t. CABSecretary 63
71. CABSecretary: viewed as manager of 7 employees 64
72. int-handle_change_request’s subprocesses and traps w.r.t. Request 65
73. int-do_meeting’s (new version) subprocesses and traps w.r.t. Request 66
74. int-do_change’s (new version) subprocesses and traps w.r.t. Request 67
75. int-reject_request’s subprocesses and traps w.r.t. Request 67
76. int-big_impact’s subprocesses and traps w.r.t. Request 68
77. int-small_impact’s subprocesses and traps w.r.t. Request 68
78. Request: viewed as manager of 6 employees 69
79. int-do_meeting’s (new version) subprocesses and traps w.r.t. NewCAB 70
80. int-request_for_meeting’s subprocesses and traps w.r.t. NewCAB 71
81. int-request_for_change’s subprocesses and traps w.r.t. NewCAB 72
82. NewCAB: viewed as manager of 6 employees 73
83. int-monitor’s subprocesses and traps w.r.t. NewDesign 74
84. NewDesign: viewed as manager of int-monitor 75
85. int-request_for_meeting’s subprocesses and traps w.r.t. ProjectManager 77
86. int-do_change’s subprocesses and traps w.r.t. Projectmanager 77
87. ProjectManager (new version): viewed as manager of 8 employees 78
88. global external behaviour of WODAN 83
89. first prescriptive step of WODAN 84
90. the first intermediate phase of int-request_for_change 84
91. int-request_for_change’s subprocesses and traps w.r.t. MainCAB 85
92. intermediate phase of MainCAB 85
93. first intermediate phase of DepCAB 86
94. second prescriptive step of WODAN 86
95. the second intermediate phase of int-request_for_change 87
96. int-request_for_change’s subprocesses and traps w.r.t. TempCABSecretary 88
97. TempCABSecretary: viewed as manager of int-request_for_change 88
98. second intermediate phase of DepCAB 89

99. int-put_request_on_list: a temporary operation of the class CAB 90
100. int-cancel_meeting: a temporary operation of the class CABMember 90
101. first intermediate phase of int-prepare_meeting 91
102. int-prepare_meeting's subprocesses and traps w.r.t. ProjectManager 92
103. ProjectManager: viewed as manager of int-prepare_meeting 93
104. third prescriptive step of WODAN 94
105. intermediate phase of int-do_meeting 95
106. second intermediate phase of int-prepare_meeting 95
107. final prescriptive step of WODAN 96
108. WODAN: viewed as manager of 11 employees 97
109. a basic PMMS-model and its four components 99
110. a PMMS-model describing Change Management 101
111. Class diagram: attributes and actions of the basic components 101
112. Import/export diagram 102
113. import list 102
114. external behaviour of the managing-component 103
115. external behaviour of the logistics-component 104
116. external behaviour of the technology-component 104
117. external behaviour of the administering-component 104
118. int-schedule_design 105
119. int-view_status 106
120. int-generate_setup 106
121. int-generate_change_setup 106
122. int-get_methods 107
123. int-instantiate_methods 107
124. int-make_methods 107
125. int-report_status 108
126. int-enact_processes 108
127. int-schedule_design's subprocesses and traps w.r.t. managing 110
128. int-report_status's subprocesses and traps w.r.t. managing 111
129. int-view_status's subprocesses and traps w.r.t. managing 112
130. int-instantiate_methods's subprocesses and traps w.r.t. managing 112
131. managing-component: viewed as manager of 4 employees 113
132. int-schedule_design's subprocesses and traps w.r.t. logistics 114
133. int-instantiate_methods's subprocesses and traps w.r.t. logistics 115
134. int-generate_change_setup's subprocesses and traps w.r.t. logistics 116
135. int-get_methods's subprocesses and traps w.r.t. logistics 117
136. the logistics-component: viewed as manager of 4 employees 118
137. int-enact_processes's subprocesses and traps w.r.t. administering 119
138. int-instantiate_methods's subprocesses and traps w.r.t. administering 119
139. int-report_status's subprocesses and traps w.r.t. administering 120
140. int-view_status's subprocesses and traps w.r.t. administering 121
141. the administering-component: viewed as manager of 4 employees 121
142. a part of int-prepare_meeting modelled in more detail 128
143. subprocesses and traps w.r.t. member1 modelled in more detail 129
144. a part of the manager process CABSecretary using logical transitions 131
145. multiple transitions (and states) replacing logical transitions 131