Adaptive software process modelling with SOCCA and PARADIGM

by Alex Wulms

University of Leiden, Department of computer science

April 12, 1995

Supervision by Dr. L.P.J. Groenewegen

Abstract

One of the unsolved problems in the world of software process modelling is the question of formally incorporating change of a software process in the very model. To elucidate this problem and possibly solve it, the following approach has been chosen. In a standard case, the so-called ISPW-7 example, is change a part of the described problem situation. The purpose is that all, or many, existing software process modelling methodologies incorporate this change in their model for the ISPW-7 case. This hopefully leads to a better understanding of a general approach to model change.

In the ISPW-7 example has the process change been split up into two parts. One part addresses some form of permanent evolution of a process, this is called ‘process modification’. The other part addresses a temporary modification of the behaviour of the process, this is called ‘process exception’.

One of the existing software process modelling methodologies is SOCCA, which is currently still under development at the University of Leiden, department of Computer Science. This thesis concentrates on the topic of formally incorporating process change in the SOCCA methodology.

First, SOCCA has been extended with some new concepts to make it possible to model process change and secondly, this extended version of SOCCA has been applied to the ISPW-7 example. As it turns out, it is possible to model both the process modification part as the process exception part of the ISPW-7 example with aid of the new concepts introduced in this thesis.

Acknowledgement

With thanks to my parents and my friends for their moral support and with thanks to Luuk Groenewegen for the excellent supervision and the many discussions which lead to a deeper insight into the ins and outs of process change and evolution within SOCCA.

Table of contents

Abstract 2

Acknowledgement 2

Table of contents 3

1. Introduction 5

2. A short introduction into SOCCA and PARADIGM 7
- 2.1. A short introduction into PARADIGM 8

3. A way of viewing change 9
- 3.1. A condition for managing subprocesses of a subprocess. 10
- 3.2. Introducing the manager process WODAN. 10
- 3.3. Three types of change 11
- 3.4. The NULL process. 11

4. Using the new concepts to describe an evolution step 12
- 4.1. A general method to describe an evolution step 12
- 4.2. Exploiting similarities between processes to refine the evolution step description. 13
- 4.3. Possible inconsistencies as a consequence of process change. 14
  - 4.3.1. Consequences of process change for individual internal processes 14
  - 4.3.2. Consequences of process change for the cooperation control 15
- 4.4. Solving inconsistencies 18
  - 4.4.1. Solving problem P1: a subprocess of an internal process has less states 18
  - 4.4.2. Solving problem P2: the external process has less states 21
  - 4.4.3. Solving problem P3: the external process has reached a state to early. 22
- 4.5. Concluding remarks 25

5. An example of changing an enacting process model 27
- 5.1. Designing a new model 29
- 5.2. Starting the new model during the enactment of a software process. 32
- 5.3. Designing WODAN to manage the change. 33
- 5.4. Using an extra subprocess for int-monitor to avoid intermediate states 34
- 5.5. Losing some restrictions 35
- 5.6. Concluding remarks 36

6. Extending the model to cover more of the ISPW-6 example 38
- 6.1. Redesigning the class diagrams 38
- 6.2. Designing the external behaviours of the classes 41
- 6.3. Designing the internal behaviours of the export operations 43
- 6.4. Adding PARADIGM to model the communication. 45

6.5. Concluding remarks 55

7. Example 2: changing the model according to the ISPW-7 specification 58

7.1. Process modification: problem description 58
7.2. Designing the new model. 58
7.3. Introducing the ISPW-7 model. 61
7.3.1. Option 1: do not solve the inconsistency 61
7.3.2. Option 2: solve the inconsistency 64
7.4. Process exception: problem description. 68
7.5. Designing the exceptional model. 70
7.6. Starting the exceptional case 72
7.7. Concluding remarks 72

8. A very brief comparison of SOCCA with other paradigms 74

8.1. The notion of process variables applied to SOCCA 74
8.2. Comparing SOCCA with Document Flow Model (DFM) 75
8.3. Comparing SOCCA with SPADE 75

9. Conclusions and further research 77

10. References 78

Appendix A. List of figures 79

Chapter 1
Introduction

One of the unsolved problems in the world of software process modelling is the question of formally incorporating evolution of a software process in the very model. To this aim the standard software process modelling example as formulated in ISPW-6 [3], has been extended, among others, with change aspects. This has resulted in the ISPW-7 [4] example.

The extensions in the ISPW-7 example focusing on this so-called process change, have been split up into two parts. One part is intended to represent some form of permanent evolution of a process, this is called ‘process modification’ and it describes a permanent change in the behaviour of the process. The other part addresses a temporary modification of the behaviour of the process handling some exceptional circumstance, which has been called ‘process exception’; after the temporary modification, the process returns to its original form.

One way to model “normal” behaviour without any flavour of the above mentioned change, especially the behaviour of a software process model, is by means of dynamic descriptions. Dynamic descriptions which are used in this manner are in fact describing a specific form of change, as the current state of the object behaving according to the dynamic description, changes into another current state, which in turn changes into yet another current state, and so on. So a dynamic description of behaviour reflects some kind of change.

Within this thesis we have put this observation the other way round: it is possible to describe a change to something, like a software process model, with a dynamic description. Since a behaviour description is a dynamic description, the description of process change as mentioned in the paragraph above, which is a change in the behaviour of a software process model, can be considered as being a dynamic description (the change description) of another dynamic description (the behaviour description).

In PARADIGM, see [1], the concepts of processes, manager processes, employee processes, subprocesses, traps and state-action interpreters, have been used to describe a behaviour change as a consequence of the communication between the various processes: a manager process prescribes a subprocess to an employee process reflecting the behaviour of this employee before the communication; the employee may only behave according to the restrictions as imposed by this subprocess. After a while such an employee will enter a trap to another subprocess, which implies a communication to the manager, and from that moment on, the manager process may decide to grant permission to this employee to enter its new subprocess. This permission is the communication the employee is “waiting for” inside its trap. The effect of this communication then is the new behaviour of the employee after the communication as reflected by the new subprocess. So within PARADIGM communication has been used to describe a change of behaviour. By means of the communication between manager processes and their employee processes, this actually is a dynamic description of a dynamic description. Therefore it is useful to examine whether the PARADIGM approach can also describe evolution.

As PARADIGM is an integrated part of SOCCA and as SOCCA aims at modelling software processes, we will actually investigate whether the SOCCA approach can describe evolution too. This thesis gives the results of this investigation. In order to present these results, the thesis has been structured as follows: the following chapter consists of a short introduction into SOCCA and PARADIGM. In chapter 3 new concepts are introduced which are generalizations of the SOCCA concepts of external and internal processes. With aid of these new concepts, evolving software process models within SOCCA become something quite natural. In chapter 4 is shown how the new concepts introduced in chapter 3 can be used. Furthermore, chapter 4 shows some simplifications of the general approach, which problems can occur using these simplifications and some general solutions to solve those problems. Chapter 5 shows an example of the concepts developed in chapter 3 and in chapter 4. This example is based on a small part of the ISPW-6 example. In chapter 6 a larger part of the ISPW-6 example will be modelled using SOCCA and in chapter 7 is this model used to show more of the aspects developed in chapter 3 and in chapter 4. This chapter follows the process change parts from the ISPW-7 example. In chapter 8 a brief comparison between SOCCA and some other modelling paradigms will be given with respect to process change and finally in chapter 9, some conclusions and topics for further research are given.

Chapter 2
A short introduction into SOCCA and PARADIGM

In [2] a complete introduction of SOCCA has been given. Those readers familiar with SOCCA may skip this chapter.

SOCCA is a software process modelling methodology, still under development at the University of Leiden, department of Computer Science. A SOCCA model describes the software process from three different perspectives; the data perspective, the process perspective and the behaviour perspective. To achieve this, SOCCA consist of (parts of) several formalisms combined together to describe the software process models.

The following formalisms have been used to cover the different perspectives:

- The data perspective is described by means of object-oriented class diagram models, based on Extended Entity-Relationship (EER) concepts. One of the features of the classes is that they can have export operations. Such an export operation of one class can call export operations of another class. To this aim in SOCCA an extra relationship exists between the various classes; the so-called uses relationship. Therefore, the class models have been extended with an extra diagram to display this uses relationship. Such a diagram is called an import/export diagram of the model.
- The process perspective will be described with so-called Object Flow Diagrams (OFD), being an extension of data flow diagrams with operations derived from the class diagram perspective. So in an OFD not only the dataflow is given as in data flow diagrams, but also the flow of the operations will be shown. The integration of OFD’s into a SOCCA model is still a topic of research. They will not be discussed further in this thesis.
- The behaviour perspective is covered with State Transition Diagrams (STD’s) with PARADIGM on top of them. The various classes have export operations which can be called in some order. To describe the order in which the export operations can be called, STD’s are used. 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. Each such export operation has an internal behaviour; this internal behaviour actually achieves the task the corresponding export operation is supposed to perform. In SOCCA, STD’s have been used as well to describe these internal behaviours of the export operations. Since the cooperation between the external behaviour of a class and the internal behaviours of its export operations has to be coordinated, PARADIGM has been used in SOCCA to this aim. 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 behaviour of its export operations, but there should also be communication between the internal behaviour of the export operations from one class and the external behaviour of other classes. This communication is also modelled by means of PARADIGM. The STD’s describing the external behaviours are PARADIGM *manager processes* and the STD’s showing the internal behaviours of the export operations, are PARADIGM *employee processes*. In this way, the internal behaviour of an export operation can be controlled via *the state-action interpreter* of the external behaviour of a class. The italic terms will be described in the following section which gives a short introduction into PARADIGM.

2.1. A short introduction into PARADIGM

PARADIGM is a specification mechanism developed to model parallel processes. A PARADIGM model can be designed in the following manner:

- Describe the sequential behaviour of each process by means of a STD.
- Within each STD so-called *subprocesses* can be indicated. These subprocesses are subdiagrams of the STD and are used for the coordination with other processes. The set of subprocesses of one process is called *a partition of that process*.
- Within each subprocess certain sets of states, so-called *traps*, can be identified. By entering such a trap, an object indicates that it is ready to switch to another subprocess. The set of traps of a process is called *the trap structure of that process*. One important property of a trap is that, within a subprocess, there are no transitions leading from one of the states of a trap to another state outside the trap. So when a process has entered a trap, it is no longer possible for the process to leave this trap as long as the same subprocess restriction remains valid.
- There is also an extra STD called the *manager process*. This process coordinates the behaviour of the various objects. The objects of which the behaviour is controlled by a manager process, are called the *employee processes* of that manager process. Depending on the state it is in, the manager process prescribes a subprocess to each of its employees; an 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; when one of the employees enters a trap to another subprocess, the manager process may follow a corresponding transition to another state where it can prescribe the subprocess the employee wants to enter. Note that the manager **can** prescribe the new subprocess to its employee but that it is not obligated to do this. The manager may postpone prescribing the new subprocess to its employee as long as it wants to. The mapping of the states of a manager process to the subprocesses it prescribes to its various employees and the mapping of the transitions that a manager can make to the traps of its employees, is called *the state-action interpreter* of the manager process with respect to the partition and trap structure of its employee processes. So the state-action interpreter of the manager process labels each state of that manager with the subprocesses it prescribes in such a state to its employee processes and it labels the transitions with those traps that enable this transition to be selected.

One individual process may be the employee of more manager processes. When this is the case, such a process will have a separate partition and trap structure with respect to each of its managers. The behaviour of that process then will be controlled by all of its manager processes together; at a given time instant, the process will be restricted to the intersection of the various subprocesses prescribed by its various manager processes.

Note that this is only a very informal introduction into PARADIGM. The sequential behaviour of one process is in fact not fully determined by its STD. In addition to the STD a strategy determines which transition in a state will be taken if several possibilities exist. Moreover, a sojourn mechanism determines how long the process remains in each state. The strategy and the sojourn mechanism can depend on the history of the process and on its current state. Formally spoken, the sequential behaviour of a process in PARADIGM is described by means of a so-called decision process from the operations research field. However, the most important features in SOCCA are formed by the STD which visualizes much of the behaviour of the process and in addition to that a strategy, often informally described only, which tells what transitions will be selected from a certain state when there are more possibilities.

Chapter 3
A way of viewing change

This chapter starts with the definition of some terms. As stated in the previous chapter, in SOCCA the various PARADIGM concepts have been used to manage the internal behaviour of the operations from a class via the state-action interpreters of the external behaviour of the classes. In the rest of this thesis, the internal behaviour of an operation is called *an internal process* and the corresponding STD is called *the internal process description*. Similarly the external behaviour of a class is called *an external process* and the STD describing this behaviour is called *the external process description*.

As also described in the previous chapter, the external process descriptions are PARADIGM manager processes and the internal process descriptions are the employees of these manager processes. During enaction of a software process model an internal process is restricted to one subprocess at a time (with respect to one partition of the internal process). Intuitively such a restriction to a subprocess is a behaviour restriction for this internal process; the internal process may only behave accordingly to those states and actions imposed by the subprocess. After remaining a while in the same subprocess an internal process will enter a trap towards another subprocess and at some time instant the permission to enter this new subprocess will be granted by the manager. As soon as the internal process receives this permission it will start behaving according to the new subprocess it has entered and it will remain behaving like this for a while. This type of behaviour change of an internal process is a direct consequence of the PARADIGM communication between the internal process and its manager process; the internal process changes from one behaviour restriction to another behaviour restriction as a consequence of this communication.

When analysing this total behaviour of an internal process it can be seen that the transition from one subprocess to another subprocess is the same kind of change as wanted for evolving software process models; when a software process model has to evolve, the behaviour of some part of the model has to be changed just like the behaviour of an internal process changes when it makes a transition from one subprocess to another subprocess. This evolution of the software process model can be considered as being a transition from one evolution stage (say evolution stage 1, EVS1) to another evolution stage (say evolution stage 2, EVS2). A conclusion drawn from this observation is that it might be possible to change a software process model by viewing its internal and external process descriptions as being subprocesses of some larger processes which have not explicitly been designed but which do exist. Furthermore it might be possible to view the total state space of the process descriptions from the SOCCA model up to now (which is in EVS1) as being a trap from the current process description to the newly designed process description (which forms the SOCCA model in EVS2).

For example, when one of the components has to be changed the new STD of the component, which does not yet exist, can be designed and after this design has been finished it can be activated by switching from the subprocess corresponding with the old STD to the subprocess corresponding with the new STD. This transition then corresponds with a transition from EVS1 to EVS2. Such a transition from one evolution stage to another evolution stage will be called an *evolution step*.

As the not explicitly designed external and internal processes combine the various behaviours of the software process model during all possible evolution stages, they will be called *anachronistic* external and internal processes in this thesis.

Note that viewing the process descriptions as being subprocesses means that we get a deeper management hierarchy, in which a process can be both an employee process and a manager process at the same time. We get the situation that a subprocess (of an anachronistic external process) is managing the subprocesses of a subprocess (of an anachronistic internal process).

For example, let E_1 be an external process, I_1 an internal process and let $I_1S_1, ... ,I_1S_n$ be the subprocesses of I_1 with respect to E_1. This means that in the original SOCCA approach, the process E_1 is a manager of the subprocesses $I_1S_1, ... ,I_1S_n$ of process I_1. However, when E_1 and I_1 are considered being subprocesses of larger, not explicitly designed, anachronistic processes (say E_A and I_A respectively), then one can say that the **sub**process E_1 is a manager of the subprocesses $I_1S_1, ... ,I_1S_n$ of the **sub**process I_1.

3.1. A condition for managing subprocesses of a subprocess

The notion of managing subprocesses of a subprocess is possible because a subprocess is a decision process by itself [1] and a decision process can have subprocesses [1]. Although it is possible to manage the subprocesses of a subprocess, a problem may arise; suppose that the above mentioned anachronistic internal process I_A has another subprocess I_2 and suppose that E_1 is still managing the subprocesses $I_1S_1, ... ,I_1S_n$ of I_1 (as in the previous section). Further suppose that for some reason the subprocess I_2 is prescribed in stead of I_1, for example when EVS2 has started in stead of EVS1. Then it can be no longer guaranteed that all states in the subprocesses $I_1S_1, ... ,I_1S_n$ can be reached or do exist as some of the states and transitions of $I_1S_1, ... ,I_1S_n$ may no longer exist in the subprocess I_2. Moreover, when the subprocess I_1 of which the processes $I_1S_1, ... ,I_1S_n$ are subprocesses, is not prescribed, one could say that the subprocess I_1 and its subprocesses $I_1S_1, ... ,I_1S_n$ temporarily do not exist. Therefore, it can not be allowed that a process E_1 is managing the subprocesses $I_1S_1, ... ,I_1S_n$ of a subprocess I_1 which is currently not prescribed by its manager process M1. Thus, to avoid these kind of problems, we need the following extra condition when viewing the external and internal processes as being subprocesses themselves:

When the subprocesses $I_1S_1, ... ,I_1S_n$ of a process I_1, which in turn is the subprocess of a process I_A, are being managed by a process E_1, then the process I_1 must be prescribed by the manager process M1 of the process I_A, with M1 being the manager process of I_A with respect to the partition of which I_1 is a subprocess of I_A.

3.2. Introducing the manager process WODAN

To formalize the change of a software process model during enaction, it is useful to introduce an extra manager process. This extra manager process will be called WODAN, which stands for What Ought to be Done As Necessary. The manager process WODAN is a manager of all (not explicitly designed) anachronistic external and internal processes and normally it stays in the same state, just prescribing the (explicitly designed) external and internal process descriptions. When a change 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. After the new model has been designed, WODAN can go to a next state prescribing the new process descriptions.

3.3. Three types of change

The changes made to a software process model to achieve evolution, can be split up into three different types of change; they range from a relatively simple change to more complicated forms of change. The following three types of change can be distinguished:

1) Do not change the state space, only change the strategies and the subprocesses and possibly add or remove transitions.
2) Do not add or remove processes, only change the strategies and subprocesses and add or remove states and transitions.
3) Add or remove processes and change the strategies, etc. of other processes.

3.4. The NULL process

Adding or removing processes within SOCCA, as in change type 3, is possible since the processes we are looking at, are only 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 one state together with one transition from this state to itself. In the same manner, 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.

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 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.

Chapter 4
Using the new concepts to describe an evolution step

In this chapter we will use the new concepts introduced in the previous chapter to describe the way in which an evolution step has to be performed. First we will give a very general method which will always work and then we will refine this method by making use of the notion that most times there will be many similarities between the processes before and after an evolution step. This however is a non-trivial notion which can lead to severe inconsistencies in the enactment state of the model when it is applied inaccurately. Thus we will also identify when such a refinement will fail and give some solutions to solve those inconsistencies in such a way that the refinements still can be used.

4.1. A general method to describe an evolution step

When an evolution step has to be made, the enacted SOCCA model has to make a transition from one evolution stage, for example EVS1, to another evolution stage, for example EVS2. During EVS1 a set of external and internal processes, reflecting the behaviour of some real life processes, will be prescribed by WODAN and during EVS2 another set of processes, reflecting the new behaviour of those real life processes, will be prescribed. Let for example P_1 be an internal or external process reflecting the behaviour of a real life process during EVS1 and let P_2 reflect the new behaviour of that same real life process during EVS2. Then, the processes P_1 and P_2 will both be subprocesses of the same anachronistic process P_a and during the evolution step, a transition from P_1 to P_2 has to be made.

The process P_a will be in one of the states of its subprocess P_1 at the moment that the evolution step has to be made and it will be in one of the states of its subprocess P_2 when the evolution step is finished. Thus the problem which has to be solved is that the process P_a must go from one of the states of its subprocess P_1 to one of the states of its subprocess P_2. As the subprocesses P_1 and P_2 may have no mutual states in P_a, this can be a real problem. This problem can be solved by using an intermediate, temporary, subprocess P_t of P_a which describes how the transition from P_1 to P_2 has to be made. This subprocess P_t has all states of P_1, a non-empty subset of the states of P_2 and possibly some extra states. Furthermore, it has for each state taken from P_1 a path leading from that state to one, or more, of the states of P_2. The trap from P_1 to P_t can consist of all states of P_1, making it possible to start the evolution step at any moment, regardless of the state of the subprocess P_1 during EVS1. The trap from P_t to P_2 can consist of those states of P_t which are taken from P_2, thus the evolution step can be finished as soon as the temporary process is in one of the states which actually reflect some part of the behaviour of the modelled process during EVS2.

When using such a temporary process P_t, process evolution can be modelled as follows: during EVS1, WODAN prescribes the subprocess P_1 of P_a. When process evolution is necessary, WODAN defines the new model, including the temporary phase and possibly a new EER model. WODAN will continue prescribing process P_1 while the design phase is active. When the design phase has been finished, WODAN will prescribe P_t, which reflects the behaviour during the evolution step, and as soon as P_t enters one of its traps, WODAN can prescribe P_2, thereby actually starting EVS2. An example of this approach is shown in figure 1.

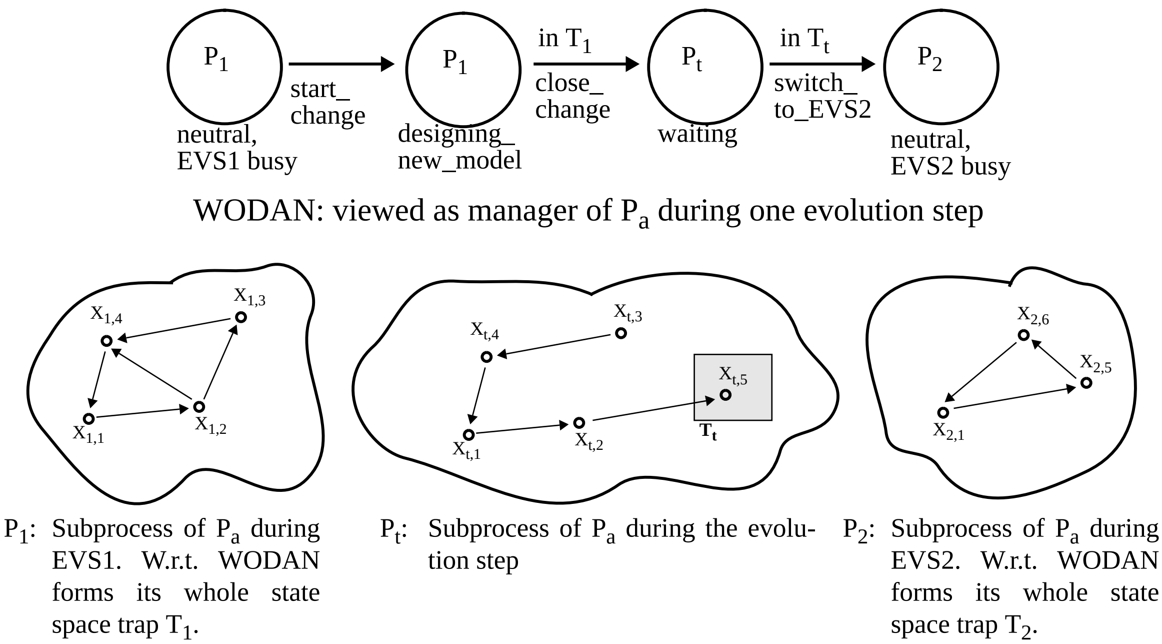


Figure 1. A general example of an evolution step

It is very likely that during one evolution step, more processes will change and that also the communication between the various processes will change. The communication may change due to extra requirements. This can be modelled similar to the above sketched evolution step. However, in stead of using one temporary process P_t for the change description of one anachronistic process P_a, one or more temporary processes $P^j_{t_i}$ have to be designed for each anachronistic process P^j_a of which the behaviour changes from EVS1 to EVS2. In this case, more temporary subprocesses may be necessary per anachronistic process that changes because of the change in the communication structure. Such a communication structure change makes the evolution step more complicated; it may be necessary that the evolution step has to be executed in a phased manner to get the new communication between the various processes appropriate. Thus, in such a case, WODAN will first prescribe the first version of the temporary processes, wait until the appropriate temporary processes reach their traps, prescribe new temporary processes, and so on until the whole model is in a state which reflects some behaviour of EVS2 and from that moment on, the subprocesses P^j_2 can be prescribed to the anachronistic processes P^j_a.

Note that further categorisation of the various problems as a consequence of process change is a topic of future research.

4.2. Exploiting similarities between processes to refine the evolution step description

The general method described in the previous section makes no use of any information available about the specific problem of making one transition step. The temporary process P_t has one separate state for each state of the process P_1 before the evolution step and some extra states taken from a non-empty subset of the states of the process P_2 which is prescribed after the evolution step. However, the processes P_1 and P_2 both represent the behaviour of the same real life process, which likely only changes slightly during the evolution step. Thus, there will be many similarities between the processes P_1 and P_2. For example, there can be a large overlap in the state space of P_1 and P_2, many transitions may be the same and there may be even many similarities between the trap and partition structure of P_1 and of P_2 when P_1 and P_2 are internal processes.

For example, when making a change of type 1 as described in section 3.3, the processes P_1 and P_2 will have exactly the same state space. In such a case, it might be possible to make the transition from EVS1 to EVS2 immediately, without using a temporary process P_t, as the process P_a can stay in the same state when making the evolution step. Also, when a temporary process P_t is necessary, this process P_t can be designed much simpler; all states of P_t can be taken from P_1 and the additional states which are normally taken from P_2 are now not necessary as they already exist in P_t. Thus, also the paths leading from the states taken from P_1 to the states taken from P_2 are in such a case not necessary.

In the following sections, we will examine whether it is generally possible to exploit these kind of similarities between the processes in the various evolution stages to simplify the evolution steps. We will also examine which kind of problems can arise with exploiting these similarities and sketch some ways to solve these problems in a formal manner with the aid of WODAN.

The study of the problems which may arise will be split up in two parts.

In the first part, an analysis at the level of the internal processes will be made; this discussion brings forward what inconsistencies may arise for the employee processes.

In the second part, the transition will be analysed at the more global level of the external processes. Since an external process in its role of manager process is responsible for the cooperation between the internal processes that form its employees, this part of the discussion underlines the possible inconsistencies in the cooperation control of the employees.

All inconsistencies will be related to the three types of change indicated in section 3.3. However, before starting with this survey, some terminology has to be introduced 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 EVSj, P_k the internal or external process prescribed during EVSk and k=j+1. So EVSk is the evolution phase next to and after EVSj. Then the process P_k will be called *corresponding* with the process P_j.
- Let P_A, P_j and P_k denote the same processes as above. Furthermore, let X_l be a state of P_A which exists in both subprocesses P_j and P_k of P_A. Then the state X_l in P_k will be called *corresponding* with the state X_l in P_j. So in fact, we will regard this same state as two different (but corresponding) states in the two corresponding subprocesses.

4.3. Possible inconsistencies as a consequence of process change

In this section, the consequences of switching from EVS1 to EVS2 will be examined to detect possible inconsistencies. This will be done in two steps; first we will examine the consequences of the process change for the individual internal processes and then we will examine the consequences of the process change for the cooperation between the internal and external processes.

4.3.1. Consequences of process change for individual internal processes

In this subsection, the consequences of switching from EVS1 to EVS2 will be examined for the internal processes. We assume that only one process changes during the transition from EVS1 to EVS2. When more processes change, the cases below will hold for each individual process. The internal process under consideration will be called I_1 with subprocesses I_1S_1, ... ,I_1S_n during EVS1 and the corresponding internal process during EVS2 will be called I_2 with subprocesses I_2S_1, ... ,I_2S_m. Furthermore we assume that the manager E_1 of I_1, and later of I_2, prescribes I_1S_i during EVS1 and I_2S_j during EVS2, and that E_1 remains in the same state during the transition from EVS1 to EVS2. So this means that the state-action interpreter φ also changes during the switching from EVS1 to EVS2, even if E_1 itself remains unchanged. Note that in this section we will not take the cooperation between the various processes into consideration. Thus, in this subsection is it not relevant whether the trap of I_2S_j differs from the trap of I_1S_i. The consequences of changing traps will be analysed in subsection 4.3.2.

This situation is shown in figure 2.

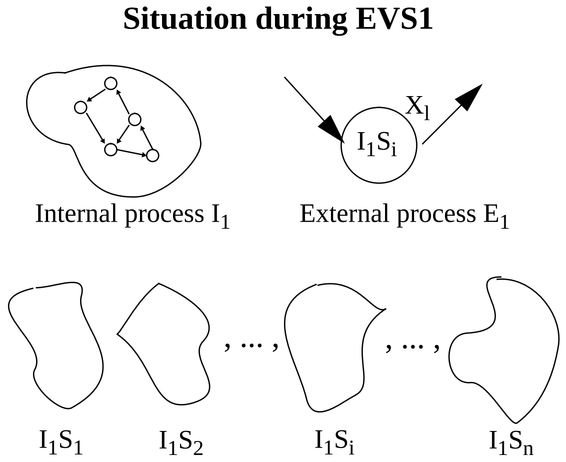


Figure 2. The general situation considered in this part

As only one subprocess of an internal process is active at the same time, we will consider the possible differences between the subprocess I_1S_i and the subprocess I_2S_j. There can be many differences between these two subprocesses. However, each difference will be a combination of one or more of the following four cases:

a1) I_2S_j equals I_1S_i; this is the trivial case. As $I_2S_j = I_1S_i$ there is no change of process. Even $\varphi(.)$ can remain unchanged.

a2) I_2S_j does not have all states from I_1S_i; in this case the transition from EVS1 to EVS2 can not be made as long as I_1S_i is in a state that does not exist in I_2S_j. This is problem P1 and some solutions to it will be given in section 4.4. One of these solutions is very easy to apply, but can not be used always. The other solutions are more complicated but they can be used always.

a3) I_2S_j does not have all transitions from I_1S_i; if I_2S_j is designed properly this does not cause any problem, the only consequence of missing transitions is that I_2S_j's behaviour differs from I_1S_i's behaviour.

a4) I_2S_j has some states or transitions that do not exist in I_1S_i; this either does not cause any problem. The only consequence of this case is that I_2S_j can not be in one of these extra states at the moment that the moment that the evolution step starts.

Thus, we have found only one problem, called problem P1, which occurs in case a2.

Note that all four cases can occur which each type of change as mentioned in section 3.3. This is the case as we are looking at subprocesses of an internal process; even for the change of type 1, where no states are removed from or added to an internal process, a **sub**process can have extra or less states since the subprocesses may always change.

4.3.2. Consequences of process change for the cooperation control

After the survey at the subprocess level in the previous subsection, we will examine the cooperation between the internal processes via an external process in this subsection. It is assumed that the external process can make the transition from E_1 to E_2 when the evolution step from EVS1 to EVS2 has to be made. When the change is of type 1, this assumption will always hold. However, when the change is of type 2 or type 3 it might be possible that the external process E_2 misses some states which exist in the external process E_1. This problem, which will be called problem P2, is similar to the problem P1 mentioned above. It can be solved with one of the solutions shown in section 4.4.2 when it occurs. Furthermore, it is assumed that only one external process changes. When more external processes change, the cases found below apply to each external process individually.

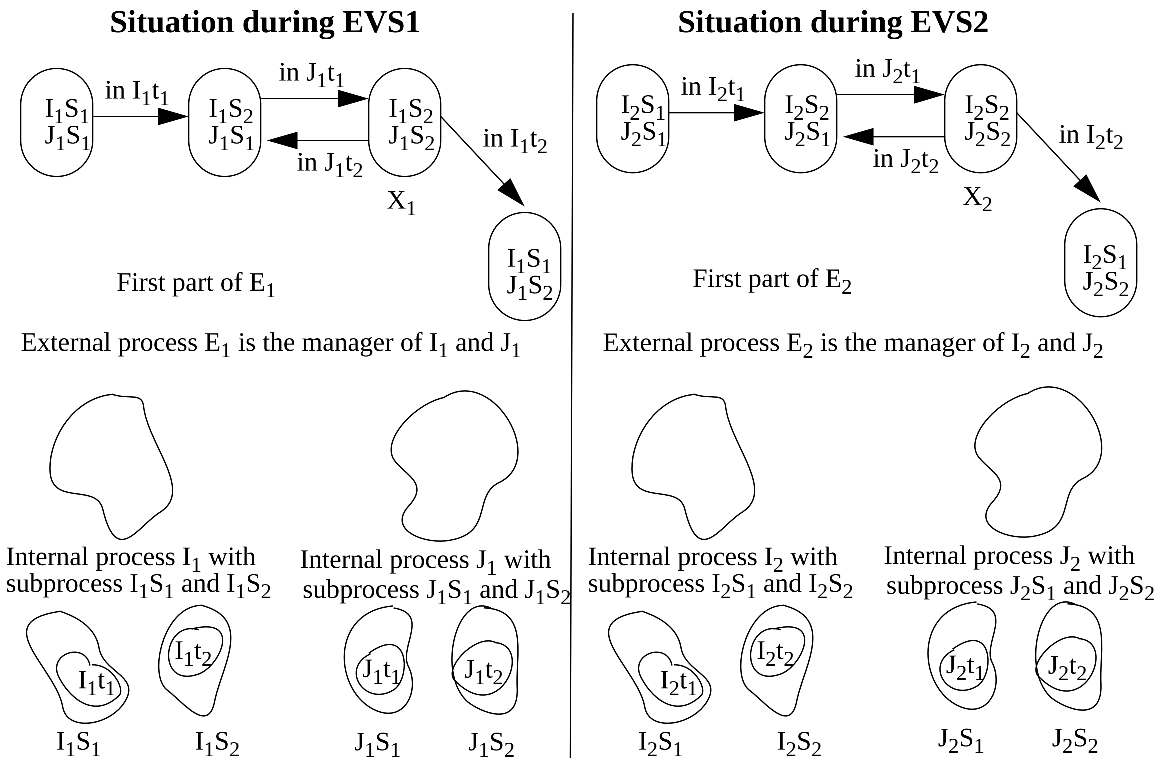
For the cooperation between the external process and the internal processes that form its employees, only the state-action interpreter of the external process and the partition and trap structure of its employees are relevant so it will be only examined how the transition from EVS1 to EVS2 influences these. The external process in EVS1 will be called E_1 and in EVS2 it will be called E_2. Furthermore, the state in which E_1 is before the transition, will be called X_1 and the corresponding state in E_2 will be called X_2 and the employees of E_1 will be called I_1 and J_1 and the corresponding employees of E_2 will be called I_2 and J_2 respectively. The following two cases can be distinguished:

b1) The state X_1 in which E_1 is at the moment that the evolution step from EVS1 to EVS2 starts, can only be reached via a path within E_1 which is equal to the path leading to the corresponding state X_2 within E_2. This means that E_1 and E_2 have an overlapping part from the start state up to the states X_1 and X_2 respectively, thus E_1 and E_2 have the same states, transitions, strategy and state-action interpreter from their respective start state up to and including the states X_1 and X_2 respectively. In this case, after the evolution step E_2 will be in a state in which it would have arrived at exactly the same manner as when E_2 had been active since the very beginning of the enactment of the model. Since E_2 would have arrived in this state in the same manner if it had been active from the beginning, there will be no difference between E_2's history as it is after the evolution step and E_2's history when E_2 would have been prescribed from the very beginning. Therefore, the evolution step from EVS1 to EVS2 can be made at once in this case. An example of this situation is shown in figure 3.

    Note that in this case it is not relevant whether I_2 and J_2 have new or modified subprocesses during EVS2; the manager E_1, and later E_2, does not prescribe these new subprocesses in this part of the model so these subprocesses can not influence the behaviour here.

b2) This is the opposite of case b1, thus the state X_1 in which E_1 is at the moment that the evolution step from EVS1 to EVS2 has to start, can be reached via a path within E_1 which is different from the path to the corresponding state X_2 in E_2. In this case, the external process will have arrived in the state X_2 after the transition from EVS1 to EVS2 via a path it would not have followed when E_2 had been active since the start of the enactment of the model. As the path that E_2 would have followed to arrive in the state X_2 determines the global behaviour of its employees (via the subprocesses E_2 prescribes to its employees) and as it is also influenced by the behaviour of the employees (via the traps E_2 has to wait for before it can make the transition to a next state), it is possible that E_2's state and history is not consistent with the state and history of E_2's employees. One of the main problems that can occur, which will be called problem P3, is that E_2 should not have arrived in the state X_2 yet, because one of its employees has not yet reached a trap which E_2 was required to wait for in the past. Some solutions to problem P3 are shown in section 4.4.3.

    An example of this problem is shown in figure 4. The manager E_1 is in state X_1, waiting for I_1S_1 to enter trap I_1t_1 when the transition to EVS2 is made. The manager, now called E_2, is still in the with X_1 corresponding state X_2 after this transition. However, according



Note that I_1S_1 in fact is the same subprocess as I_2S_1. Otherwise, the state-action interpreter of E_2 would differ from the state-action interpreter of E_1. The same applies for the combinations (I_1S_2, I_2S_2), (J_1S_1, J_2S_1) and (J_1S_2, J_2S_2).

Figure 3. Situation b1: the first part of E1 and E2 is the same

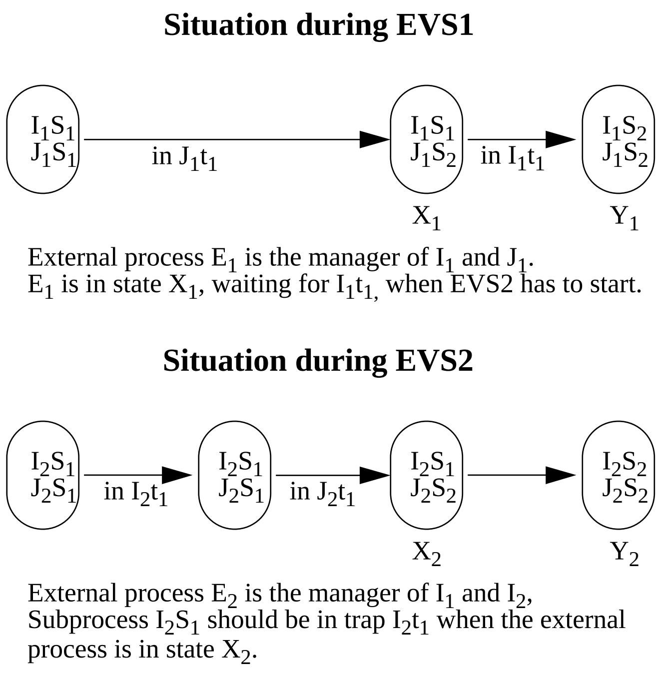


Figure 4. The manager is different during EVS2

to the new model is I_2S_1, corresponding with I_1S_1, already in trap I_2t_1 when the manager is in state X_2. Thus, the manager makes the assumption that it can make the transition to state Y_2, were it will prescribe I_2S_2 to the employee I_2. This, however, may not happen as the trap I_2t_1 had not been entered when the transition from EVS1 to EVS2 was made. Section 7.3 shows an example which is very similar to this one.

Note that both case b1 and b2 can occur with all three types of change.

4.4. Solving inconsistencies

In the previous section, three different problems have been mentioned that can arise at the moment that the evolution step from EVS1 to EVS2 has to be made. These three problems are:

P1 The subprocess I_1S_i of internal process I_1 is within EVS1 in a state X_1 without a corresponding state X_2 in the subprocess I_2S_j which will be prescribed within EVS2 in stead of the subprocess I_1S_i. This problem can occur with all three types of change.

P2 The external process E_1 is within EVS1 in a state X_1 without a corresponding state X_2 in the with E_1 corresponding external process E_2 within EVS2. This problem can only occur with change type 2 and change type 3.

P3 The process P is an external process and therefore a manager of some internal processes. It will arrive within EVS2 in a state X, which it could not have reached yet within EVS2, since one of its employees has not reached a trap for which the manager should have waited in the past. This problem will mostly arise with change type 3, when a new process is added to the model.

In the following subsections, solutions to these problems will be suggested. Some of these solutions have also been used in the examples described in this thesis. The problem P1 arises in section 5.2, the problem P2 in section 7.6 and the problem P3 can be found in section 7.3.

4.4.1. Solving problem P1: a subprocess of an internal process has less states

Let I_1 be the internal process during EVS1, I_1S_i the subprocess prescribed by the manager, X_1 the state in which I_1S_i is before the transition and let E_1 be the manager of I_1 with respect to the partition of which I_1S_i is a subprocess. Furthermore, let I_2 be the corresponding internal process during EVS2, I_2S_j the corresponding subprocess prescribed by the manager, X_2 a state corresponding with X_1 and let E_2 be the manager of I_2 with respect to the partition of which I_2S_j is a subprocess. This situation is shown in figure 5.

Note that the state X_2 is no part of I_2S_j. However, it can be a state of I_2. If there is no state X_2 in I_2 at all, a temporary version of the internal process can be defined which has a state corresponding to state X_1 in I_1. This temporary version of the internal process will be called I_t, the state corresponding to state X_1 will be called X_t and the subprocess corresponding with I_2S_j will be called I_tS_j. Furthermore, the manager of this process will be called E_t.

Note that X_t, I_t and E_t are not always necessary to solve the inconsistency. Whether they are needed or not, depends on the chosen solution. They will be used in the second solution down here. The processes named E_t in the solution S1 have nothing to do with the E_t as in the solution S2. The name E_t is only used to indicate that it is a temporary process.

The problem P1 can be solved in the following manners:

S1) Let X_{ek} be the set of states of E_1 in which the problem occurs. Thus, E_1 prescribes in each state of X_{ek} a subprocesses I_1S_i while E_2 prescribes from its corresponding state a subprocess I_2S_j which misses one or more states occurring in I_1S_i. It is possible to define a temporary external process E_t with the aid of the states X_{ek}. How this process will be designed is described below. When E_t has been designed, it is possible for WODAN to manage the transition from EVS1 to EVS2. To do this, it will consider all states of E_1 together being a trap from E_1 to the process that forms the successor of E_1 during the evolution of the software process model. This successor will now be the external process E_t

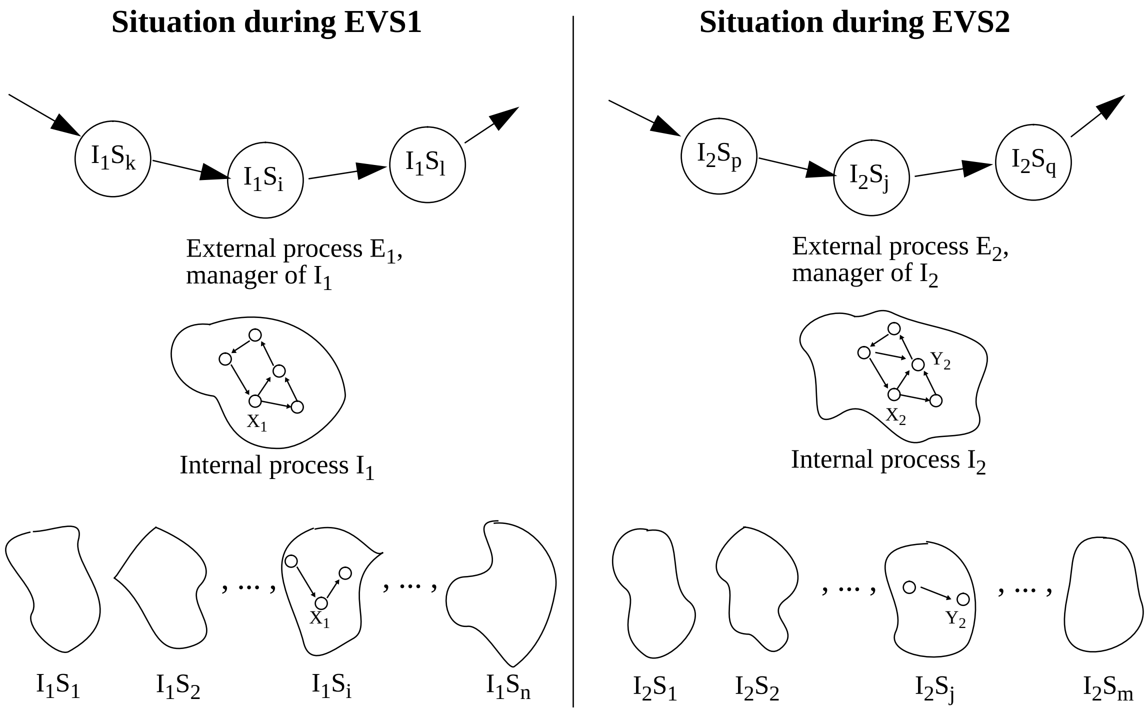


Figure 5. Problem situation P1

and the successor of E_t will be E_2.

The temporary external process E_t will be designed analogous to the external process E_1. However, there will be two differences between E_1 and E_t. The first difference is that E_t has a smaller trap with respect to WODAN; the trap of E_1 consists of all states of E_1 while the trap from E_t to E_2 consists of all states of E_t except for the states of X_{ek}, which lead to the problem. In this way, WODAN is forced to wait with completing the transition from EVS1 to EVS2 until E_t has left the states which form a problem. Secondly, all transitions leading out of the trap of E_t must be removed, so E_t can no longer enter a state of X_{ek} after it has entered its trap to E_2. This is required to make sure that E_t remains trapped in the states from where the transition to E_2 can be made safely. This solution has been used in the sections 5.3 and 7.6. An example of this solution can be found in figure 6.

Note that this solution can not be used when $X_{ek} = X_{E_1}$, thus when X_{ek} consists of all states of E_1 as E_t will have an empty trap in this last case.

S2) In the second solution, a temporary version of the internal process I_2 will be used, it will be called I_t. This temporary process I_t will be designed analogous to I_2. However, it will have some extra states and transitions: all states that are no part of I_2 but that are in I_1 will be part of I_t together with transitions leading from those extra states to the states that are also part of I_2. The extra states that exist in I_t-I_2 will be called X_{tk}. In the same way, the subprocesses of I_t must be designed analogous to the subprocesses of I_2 with eventually extra states from I_1. In the description of the rest of this solution, the example given in the introduction of this section will be considered. Thus we have a subprocess I_1S_i of I_1, a subprocess I_2S_j of I_2 and a state X_1 in I_1S_i with no corresponding state in I_2S_j. Following the first part of this solution, we now also have a temporary process I_t that resembles I_2. As, in the example, I_2 has the same states as I_1, no extra state X_t will exist in I_t. The process I_t now needs a subprocess I_tS_j which consists of I_2S_j together with an extra state corresponding with the state X_1 in I_1 and a transition from that state to one of the other states of I_tS_j. As the state X_1 in I_1 has a corresponding state X_2 in I_2 (and in I_t), this state X_2 will be cho-

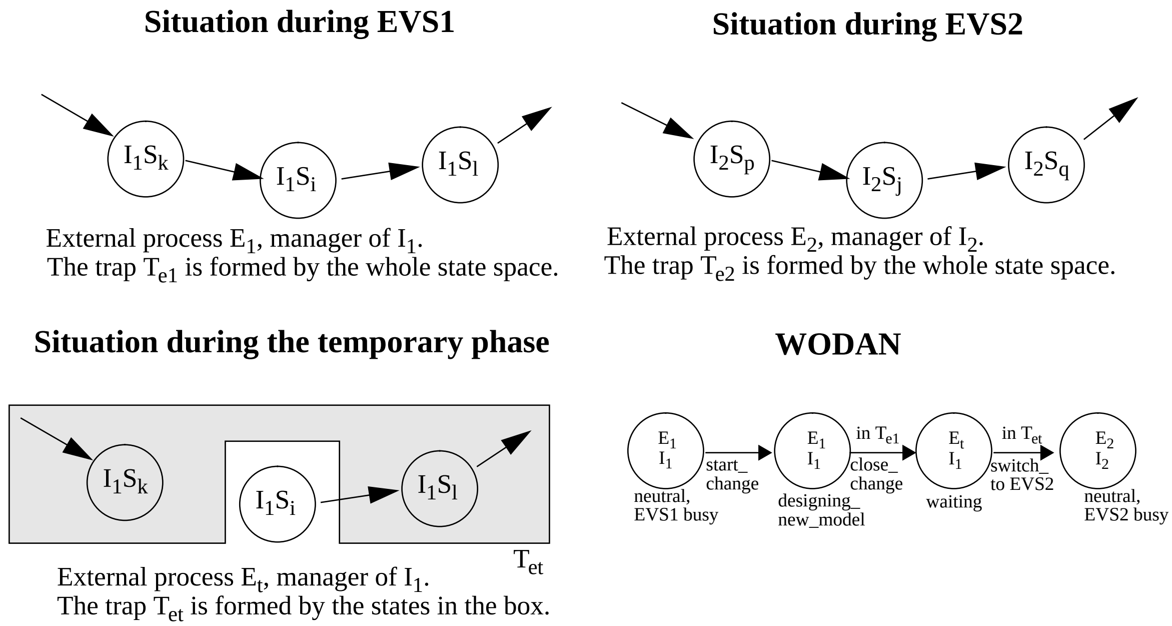


Figure 6. Example of solution S1

sen to complement I_tS_j. Generally spoken is it possible to design the subprocesses I_tS_j for all combinations of subprocesses I_1S_i and I_2S_j that can lead to problems when switching from EVS1 to EVS2. After these subprocesses have been designed, a temporal manager E_t can be made. This process E_t can be designed analogous to the process E_2. However, there will be one difference: in the states where E_2 prescribes a subprocess I_2S_j, the process E_t will prescribe a subprocess I_tS_j. Just as in solution S1, the set of these special states of E_t will be called X_{ek} and the trap from E_t to E_2 will consist of all states of E_t except for the states of X_{ek}. When this all has been designed, the transition from EVS1 to EVS2 will elapse as follows: as soon as the transition has to be made, WODAN will prescribe E_t. Since E_t resembles E_2, this means that in fact the new behaviour will start from the very beginning. However, as long as one of the subprocesses is still in a state that only exists during EVS1, this subprocess will keep behaving according to the EVS1 behaviour. After a while, such a state will be left and from that moment on the subprocess will have the EVS2 behaviour. As soon as the external behaviour enters a safe state (a state in which no problems can arise), WODAN can prescribe the external process E_2, thereby making the EVS2 behaviour definitive. An example of this solution is shown in figure 7. This solution is not used any further in the examples in chapter 5 or chapter 7. However, the following variant of it, called solution S3, is used in section 5.4.

S3) As mentioned above, is this solution a small variant of solution S2. It follows solution S2 until the point where E_t should be designed. In stead of designing a temporal process E_t, the process E_2 will be modified slightly: in all states where E_2 should prescribe a subprocess I_2S_j, E_2 can now choose between prescribing the subprocesses I_2S_j and I_tS_j. The strategy from E_2 will be adapted to make this decision; there will be an extra statement like: when it is possible to prescribe both I_2S_j and I_tS_j, the subprocess I_tS_j should only be prescribed when the internal process under consideration is in a state that only exists in I_tS_j. In this way, the temporary subprocesses will only be used just after the transition from EVS1 to EVS2, as only immediately after this transition, the internal process under consideration can be in such a state. When using this variant of the solution, the subprocesses I_tS_j are part of the same partition as the subprocesses I_2S_j; they are both subprocesses of the very internal process I_t and the slightly modified manager E_2 is the manager of I_t with respect to this partition.

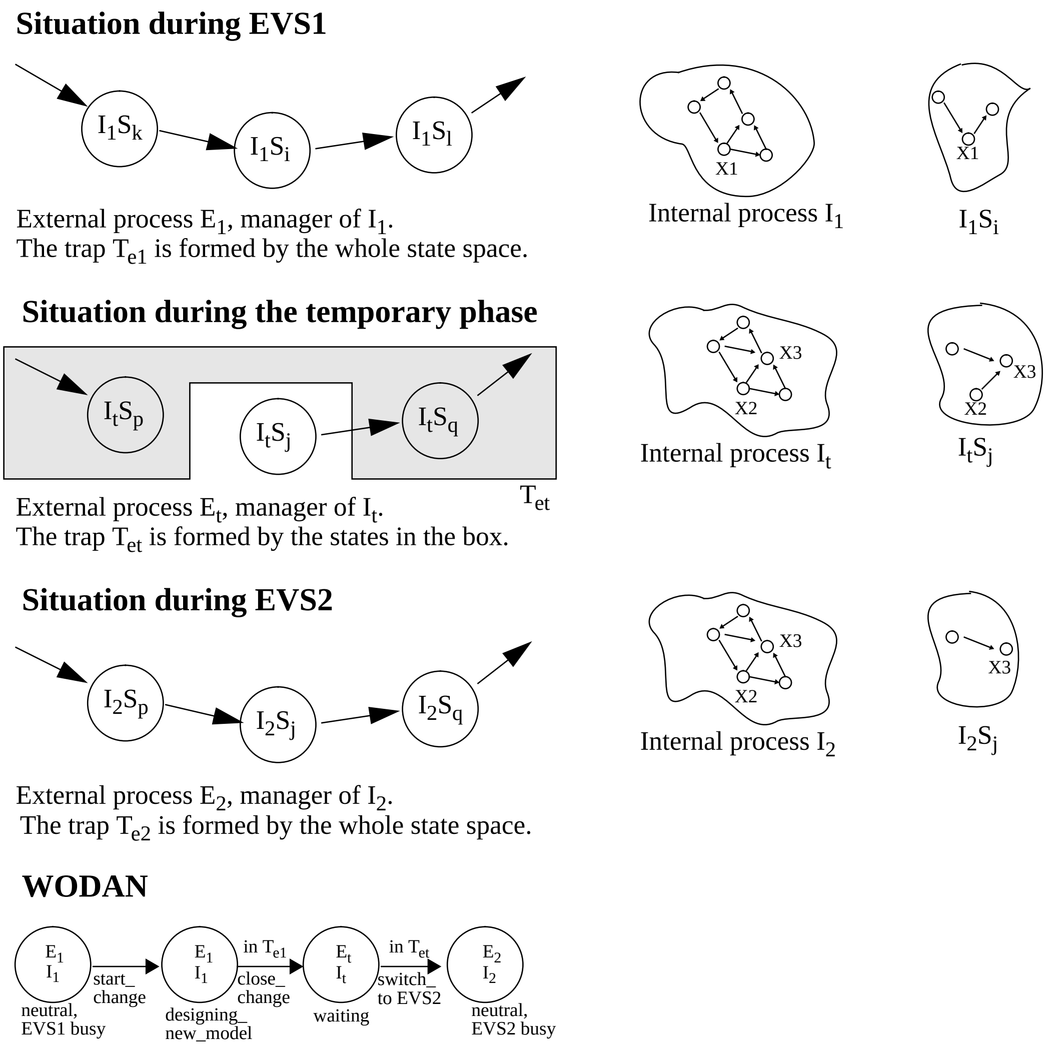


Figure 7. Example of solution S2

4.4.2. Solving problem P2: the external process has less states

Let E_1 be the external process in EVS1, E_2 the corresponding external process in EVS2 and let X_j be the states of the external process that do exist in E_1 but that have no corresponding states in E_2. Just as in the case of the internal processes, two possible solutions can be given:

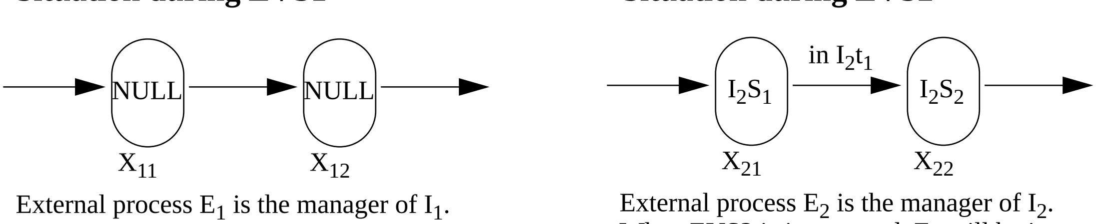
S4) Make a temporary version E_t of the external process. This E_t is designed analogous to E_1. However, there are two differences: the first difference is that E_t has a smaller trap then E_1; the trap of E_t exists of all states except for the states X_j. The other difference is that all transitions which would lead out of the trap E_t, must be removed from the process E_t to assure that E_t will stay in its trap as soon as it has been entered. WODAN can now prescribe E_t as soon as the transition from EVS1 to EVS2 has to be made and as soon as E_t enters its trap, WODAN can prescribe E_2 to finish the transition from EVS1 to EVS2. A combination of this solution with solution S1 has been used in section 7.6.

S5) Also in this solution, a temporal version E_t of the external process will be used. This temporal version however, is designed analogous to E_2; the STD of E_t exists of the STD of E_2 together with the states X_j and transitions leading from these states X_j to the states that are taken from X_2. The trap of E_t consists of all states taken from E_2. As in the previous solution, WODAN can prescribe E_t as soon as the transition from EVS1 to EVS2 has to be made and it can prescribe E_2 to finish the evolution step as soon as E_t has entered its trap towards E_2.

The intuitively difference between solution S4 and solution S5, is that in solution S4 the model will switch to the EVS2 behaviour only after a state that exists both in the EVS1 and EVS2 behaviour, has been entered. It will always follow its natural path (according to the EVS1 behaviour) to arrive in such a state. In solution S5 however, the EVS1 behaviour can be aborted as soon as possible; as soon as the model wants to leave the state which only exists in the EVS1 case, it can travel via the extra transitions of E_t to a state of the EVS2 behaviour. In this case, the model does not have to follow the complete path through the EVS1 behaviour as in the case of solution S4.

4.4.3. Solving problem P3: the external process has reached a state to early

Since an external process manages the subprocesses of some internal process, an external process can only make the transition from a state X_1 to a state X_2 after the appropriate subprocess has reached the trap that corresponds with this transition. A consequence of this behaviour of the external process is, that its employees must have passed through various subprocesses and have reached various traps, before the external process can arrive in a state X_j. For example, let the internal process I_2 with subprocesses I_2S_j be an employee of the external process E_2. Furthermore, let state X_{21} be a state of E_2 in which E_2 prescribes the subprocess I_2S_1 and in which E_2 has to wait until I_2S_1 has reached its trap I_2t_1 to the subprocess I_2S_2. As soon as I_2S_1 has entered this trap, E_2 may follow the transition towards state X_{22}, thereby prescribing I_2S_2 as soon as the state X_{22} has been entered. When the external process E_2 with its employees is active from the start of the enactment of the model on, the model will be in a consistent state when the external process E_2 has entered the state X_{22}; the internal processes which are employees of E_2 have previously reached the traps to the subprocesses which they are currently restricted to by E_2. However, suppose that E_2 and its employees will be activated for the first time when EVS2 starts and that during EVS1, the external process E_1 will be used in stead of E_2. Let us consider the case in which the evolution step to EVS2 has to be made at the moment that E_1 is in the state X_{12}, which corresponds to the state X_{22} of E_2. In this case E_2 will prescribe subprocess I_2S_2 to the internal process I_2 regardless of I_2 has reached its trap to I_2S_2 or not. When I_2 had not reached the trap to I_2S_2, the model will not be in a consistent state, since a subprocess of an internal process is prescribed at a moment that this is not yet allowed. Such a situation is shown in figure 8.



External process E_1 is the manager of I_1. On the moment of the transition to EVS2 is E_1 in the state X_{12}.

External process E_2 is the manager of I_2. When EVS2 is just started, E_2 will be in state X_{22}. However, this may not be allowed as it is not assured that I_2 had already entered the trap I_2t_1 from I_2S_1 to I_2S_2.

Note that the external process E_2 is corresponding to the external process E_1 and the internal process I_2 is corresponding to the internal process I_1.

Figure 8. Example of problem P3

Note that this problem will only arise when the trap structure of I_2 is really different from that of I_1, with I_1 being the corresponding internal process during EVS1. This will mostly happen when the internal process I_2 is an entirely new process; since during EVS1 the NULL process will have been prescribed, the internal process will have had no history at all and therefore it will not likely be in the state where it should have been at the moment of entering EVS2. However, since this problem also can arise in other cases, the internal process during EVS1 still will be referred to as I_1 and not as NULL.

The above given problem can be solved in one of the following manners:

S6) Let X_{2k} be the set of states of E_2 in which the problem can arise. Furthermore, let X_{1k} be the set of corresponding states of E_1. Design a temporary external process E_t which is designed analogous to E_1. There will be two differences between E_1 and E_t. The first difference is that the trap of E_1 consists of all states of E_1 while E_t's trap consists of all states of E_t except for the states of the set of states X_{tk} corresponding with the set of states X_{1k} of E_1. The second difference is that all transitions leading out of the trap of E_t have to be removed. WODAN can then prescribe E_t as soon as the transition from EVS1 to EVS2 has to be made and it can finish the transition by prescribing E_2 as soon as E_t has entered its trap.

Note that this solution can only be used when the states of X_{tk} of E_t don't form a trap by themselves, otherwise the transition to EVS2 can never be finished when E_t has entered a state X_{tk}. This solution has been used in section 7.3.1. An example of this solution is shown in figure 9.

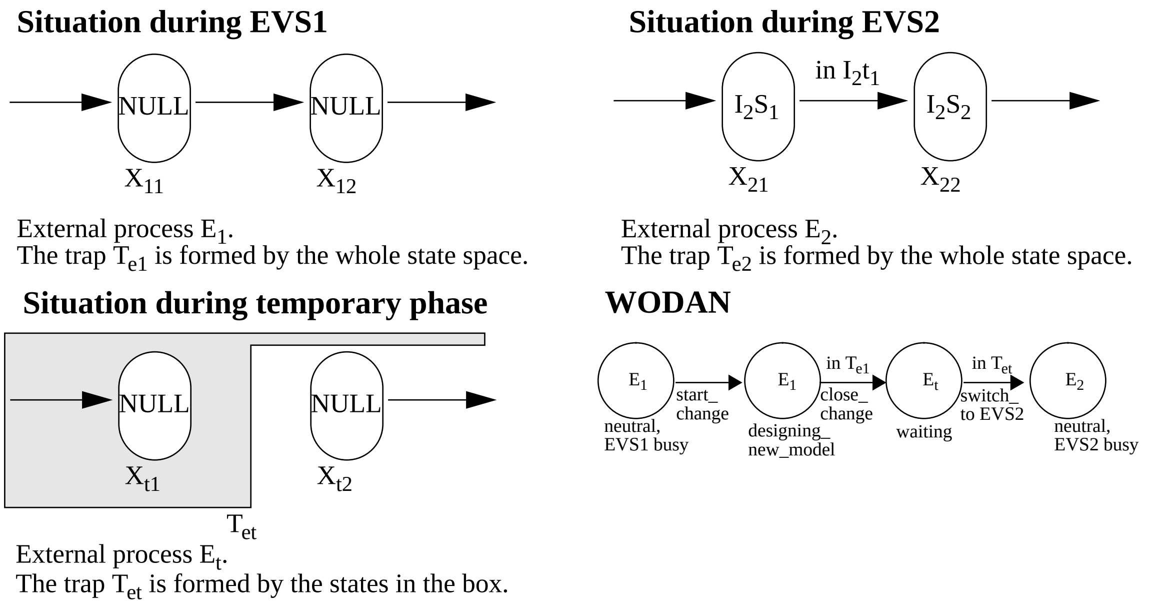


Figure 9. Example of solution S6

S7) First analyse the specification of the software process during EVS1 and during EVS2 and decide whether it is allowed to have an inconsistency during the first moment that EVS2 is active. In the following step, a temporary external process E_t can be made, which consists of some kind of mixture of the external processes E_1 and E_2; this E_t will partially have the behaviour of E_2 but it will wait for the trap of I_2 in another state then E_2 would have done. In which state it will wait depends on where the inconsistency may exist according to the analyses in the first step and where it is not allowed. In this way, when the external process E_t is in some states, it will permit the inconsistency to exist and when E_t is in some other states, it will prohibit the inconsistency to exists. The states in which the inconsistency is not allowed, can form the trap from E_t to E_2. Just as in the previous solutions, WODAN can first prescribe E_t to start the transition from EVS1 to EVS2 and it can prescribe E_2 to finish the transition as soon as E_t has entered its trap. An example of such a solution is shown in figure 10.

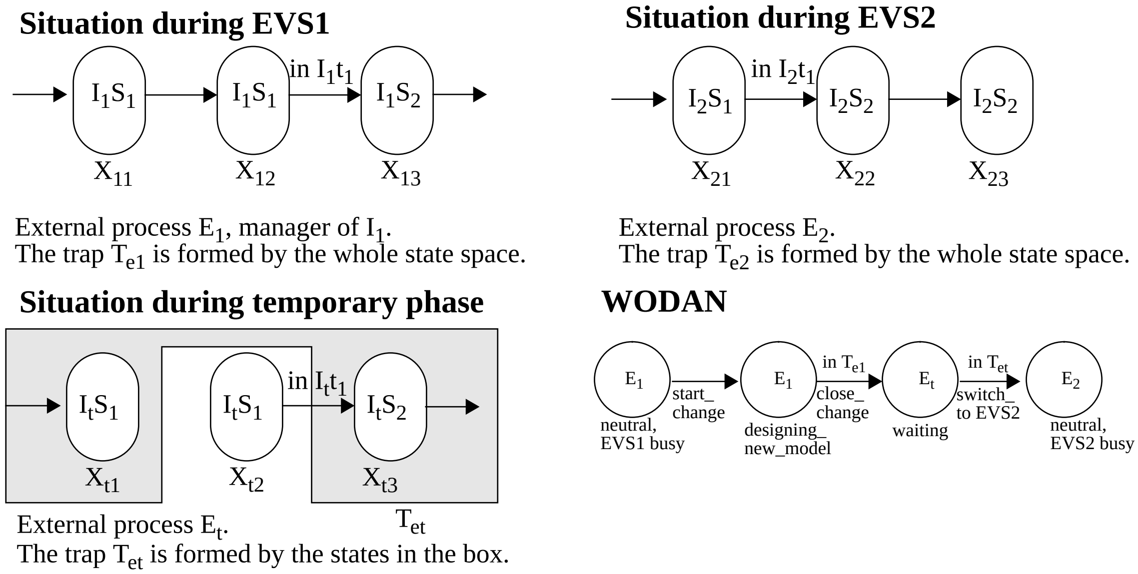


Figure 10. Example of solution S7

S8) This is the most complex solution. In the previous two solutions, the transition was postponed until the external process was in a state were no inconsistencies can arise (solution 1) or the inconsistencies were allowed to exist for a while during the first moment of EVS2 (solution 2). In this solution however, the transition from EVS1 to EVS2 will be made immediately, with the restriction that, when the inconsistency arises on entering EVS2, the behaviour of E_2 and its employees will be “rolled back” to a state/subprocess which is normally reached earlier during the enactment of the model. To get this effect, again a temporary external process E_t will be necessary. The internal process I which causes the inconsistency will also need a temporary version I_t with subprocesses I_tS_j. Since using this solution is a matter of high qualified engineering for each individual change of each individual model, the way to solve this problem will be sketched only roughly here. A more detailed example can be found in section 7.3.2, where a solution of the ISPW-7 example is given.

Let E_1, X_{12}, E_2, X_{21}, X_{22}, I_2, I_2S_j, I_2S_1 and I_2S_2 be as in the introduction of this section. Then E_t can be designed analogous to E_2 but with the following differences: E_t has some extra state X_{ta} labelled ‘aborting process’, ‘turning back operations’ or whatever, with a transition from the state X_{t2} (corresponding with the state X_{22}) leading to this state X_{ta}. This transition can be labelled ‘abort process’ or something alike. Furthermore, a second temporary process E_u can be designed which consists at least of the state X_{ua} corresponding with X_{ta} of E_t and a state X_{u1} corresponding with state X_{21} of E_2. The process E_t can have two traps: one consisting of all states in which no inconsistencies can arise and the other one consists of the state X_{ta}. The first one is a trap towards the subprocess E_2 while the second one is a trap towards the subprocess E_u. The trap of E_u can consists of the state X_{u1}. Furthermore, the temporary internal process I_t and its subprocess I_tS_2 can have an extra transition in which it calls the export operation ‘abort process’ of the manager E_t and after this transition it will arrive in a trap to the previous subprocess I_tS_1. WODAN can now first prescribe the process E_t; when E_t is in a consistent state, it will arrive in the trap to E_2. In the other case, the internal process can be forced to follow the ‘call abort’ transition by prescribing an appropriate subprocess to it, and as soon as the internal behaviour has done this call, the manager process E_u can be prescribed by WODAN. The manager E_u can now manage the behaviour of the internal process until it arrives in a state that is completely consistent with the EVS2 behaviour and as soon as E_u has entered such a state, WODAN can prescribe E_2, thereby completing the transition to EVS2. An example of this scenario is shown in figure 11.

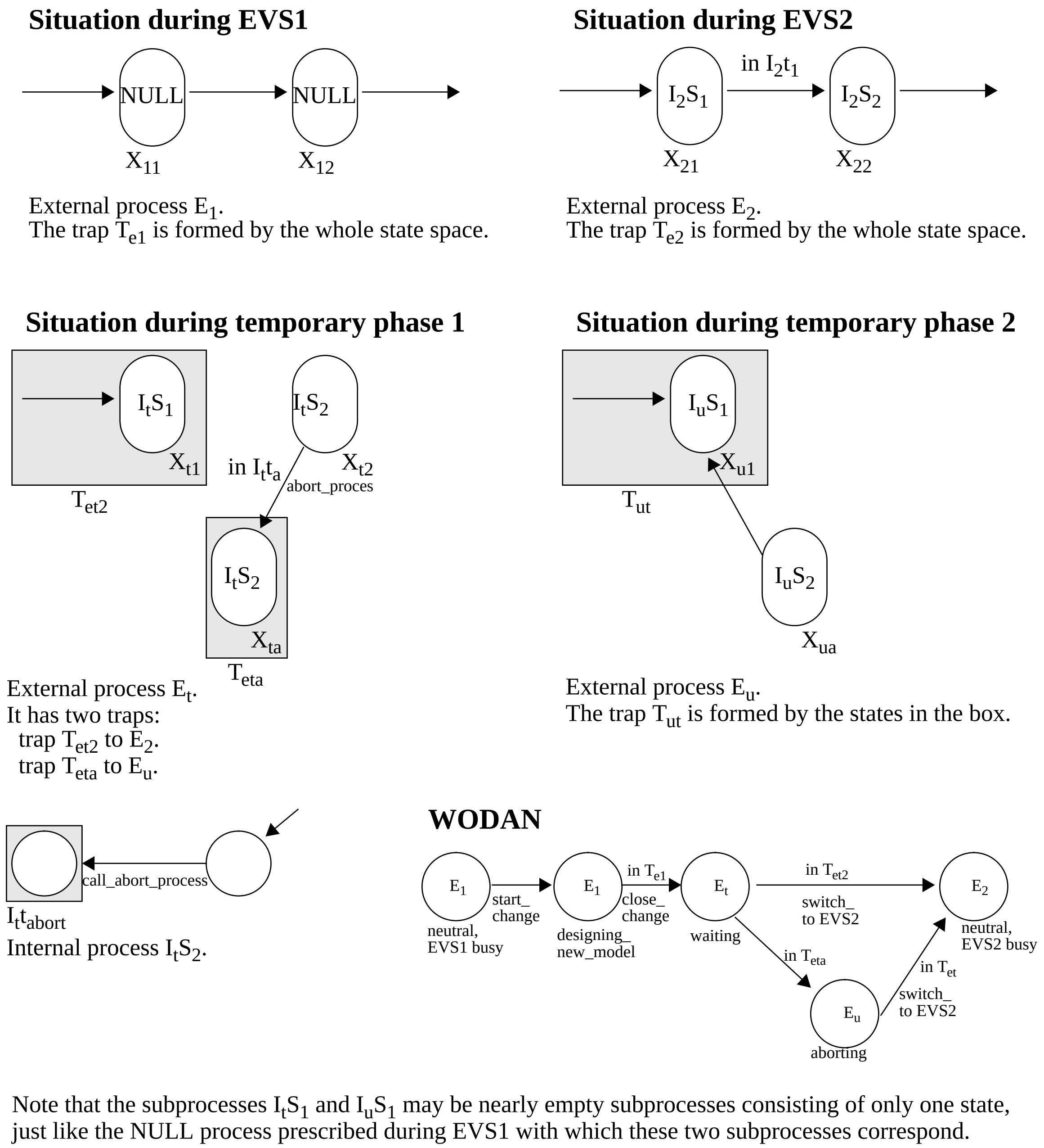


Figure 11. Example of solution S8

Note that this scenario is just a skeleton to sketch the process of rolling back the behaviour of a model during evolution to solve inconsistencies. In real life, many variants on this skeleton can exist. Even the example in section 7.3.2 differs a bit from this scenario.

4.5. Concluding remarks

In this chapter we have seen how the new concepts of WODAN and anachronistic external and internal processes can be used to describe process evolution. In the first section, a very general method to describe process evolution has been shown. This method can always be used but it will lead to unnecessary complex evolution descriptions in many cases. In the following sections, we have made use of the fact that many similarities between the processes used before the evolution step and the processes used after the evolution step may exist. This notion leads to simpler process evolution descriptions. However, there can be many problems when using such simplifications. The rest of this chapter contained some general solutions to solve the most important problems which can arise. These solutions range from very simple ones to very complex ones. The last solution, solution S8 shown in section 4.4.3., is even that complicated that it is merely an example of using the general method shown in section 4.1. then an example of exploiting the similarities in the model before and after the evolution step.

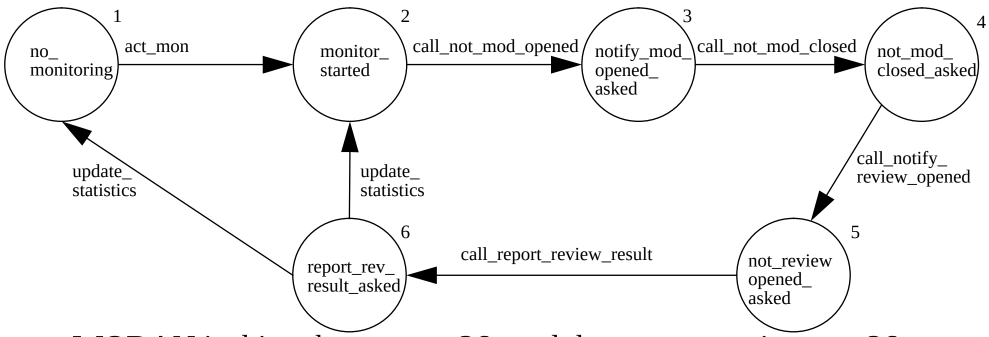
Chapter 5
An example of changing an enacting process model

In this chapter, a concrete example of process evolution is shown which makes use of the concepts developed in the previous chapters. In this first example, a dynamic change of type 1 will be made. This means that only the strategy, the transitions and the subprocesses may change. There will be no change in the state space of the processes or in the number of processes the model consists of.

The reason to make an example of change type 1, is to examine whether such a strong restriction is useful. As it turns out, it is possible to change a model with the type 1 change but it makes the model unnecessary complex. To avoid this complexity, the type 1 change will be redefined such that it will be a weaker restriction.

When making the wanted change to the model during enactment, the problem P1 -the subprocess of an internal process has less states- will arise and this problem will be solved with both solution S1 and solution S3.

The model that is going to be modified is an extension of the example in [2], which is an example of using SOCCA to model (a small part of) the ISPW-6 case. In that example the central class is the class *Design*, which is the model for the process of designing a document. In the extension, which is the start model for the example in this chapter, a monitor process *int-monitor1* has been introduced that monitors the progress of *Design*. *Design* has also been modified to support this monitor process. The STD of *int-monitor1* and of *Design* can be found in figure 12 and in figure 13 respectively.

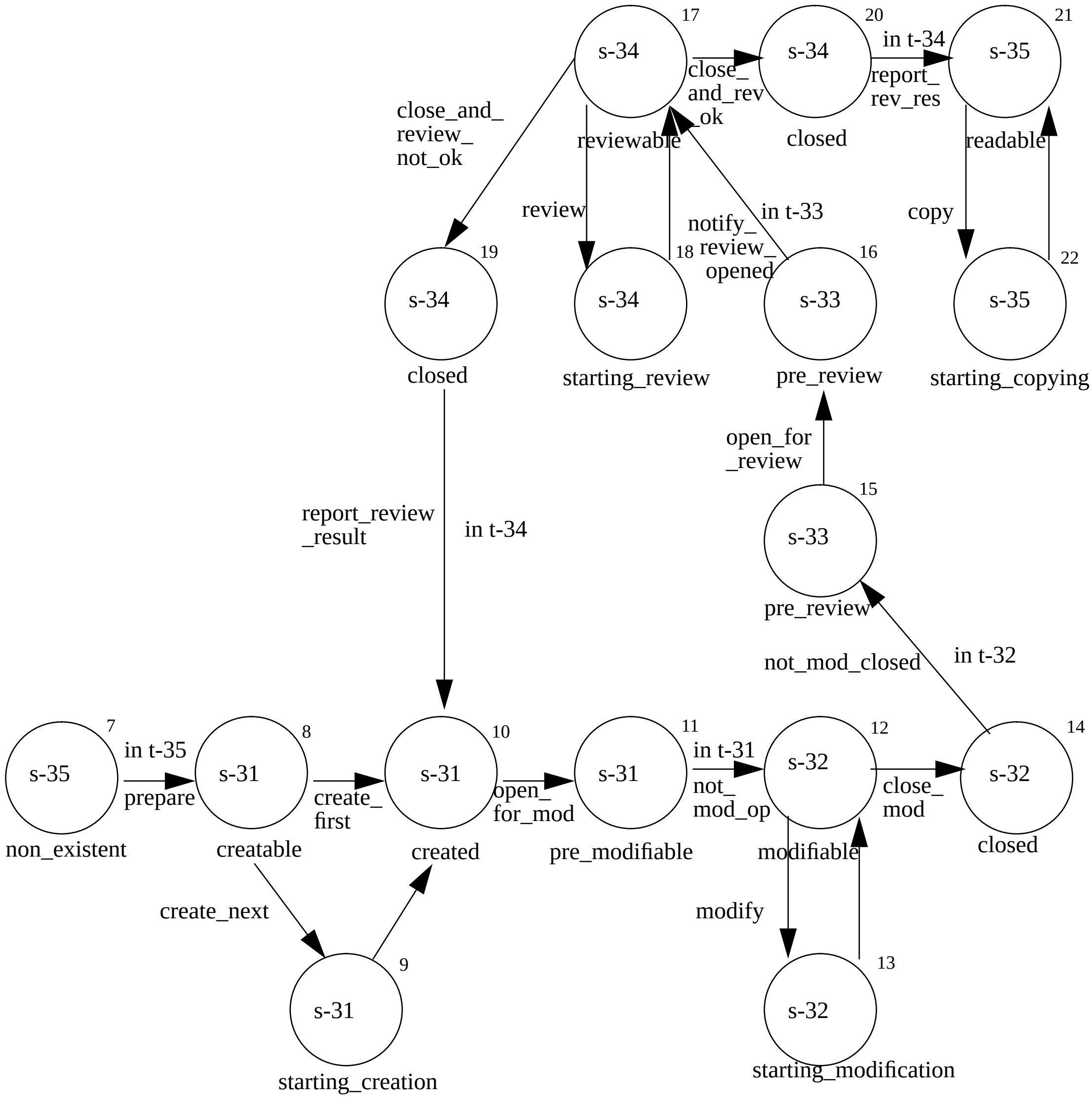


W.r.t. to WODAN is this subprocess s-36a and the state space is trap t-36a

Figure 12. Int-monitor1: STD of the internal behaviour

Int-monitor1 has been designed to follow every phase of *Design*: *Design* must send a notify to the monitor when the modification has been opened and when it has been closed and it must also notify the monitor when review has been opened and finally *Design* must report the review result to the monitor to give the monitor the opportunity to update the statistics (one of the requirements of the ISPW-6 case, see [3], section 2.6.3).

As can be seen in figure 13 the STD of *Design* (which is an external process and therefore a manager process of some internal processes) is in fact just a subprocess with respect to WODAN. In the normal case, when the software process is being enacted and no change has to be made to it, this subprocess will continually be prescribed by WODAN so this notion of an external process being only a subprocess does not influence the behaviour in normal case.



W.r.t. to WODAN is this subprocess s-36 and the state space is trap t-36

Figure 13. Design: only viewed as manager of int_monitor1

When the behaviour of *Design* has to be changed for any reason, a new STD for *Design* can be made and at the appropriate time instant this new STD can be prescribed by WODAN in stead of the actual one.

To monitor the behaviour of *Design*, the monitor needs some subprocesses, which can be found in figure 14.

Subprocess s-31 is the start state of *int-monitor1* 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 [2] for a justification of this). In the subprocess s-31, *int-monitor1* will be waiting until *Design* will start the modifications. In subprocess s-32 it will be waiting until *Design* has closed the modification and in s-33 and s-34 it will be waiting until *Design* has started the review process and until it has reported the review result respectively. After this *int-monitor1* will go back to subprocess s-31 (when the review result is *not_ok*) or to the neutral subprocess s-35 (when the review result is *ok*). Note that s-35 is also the starting state for the other instances of *Design*.

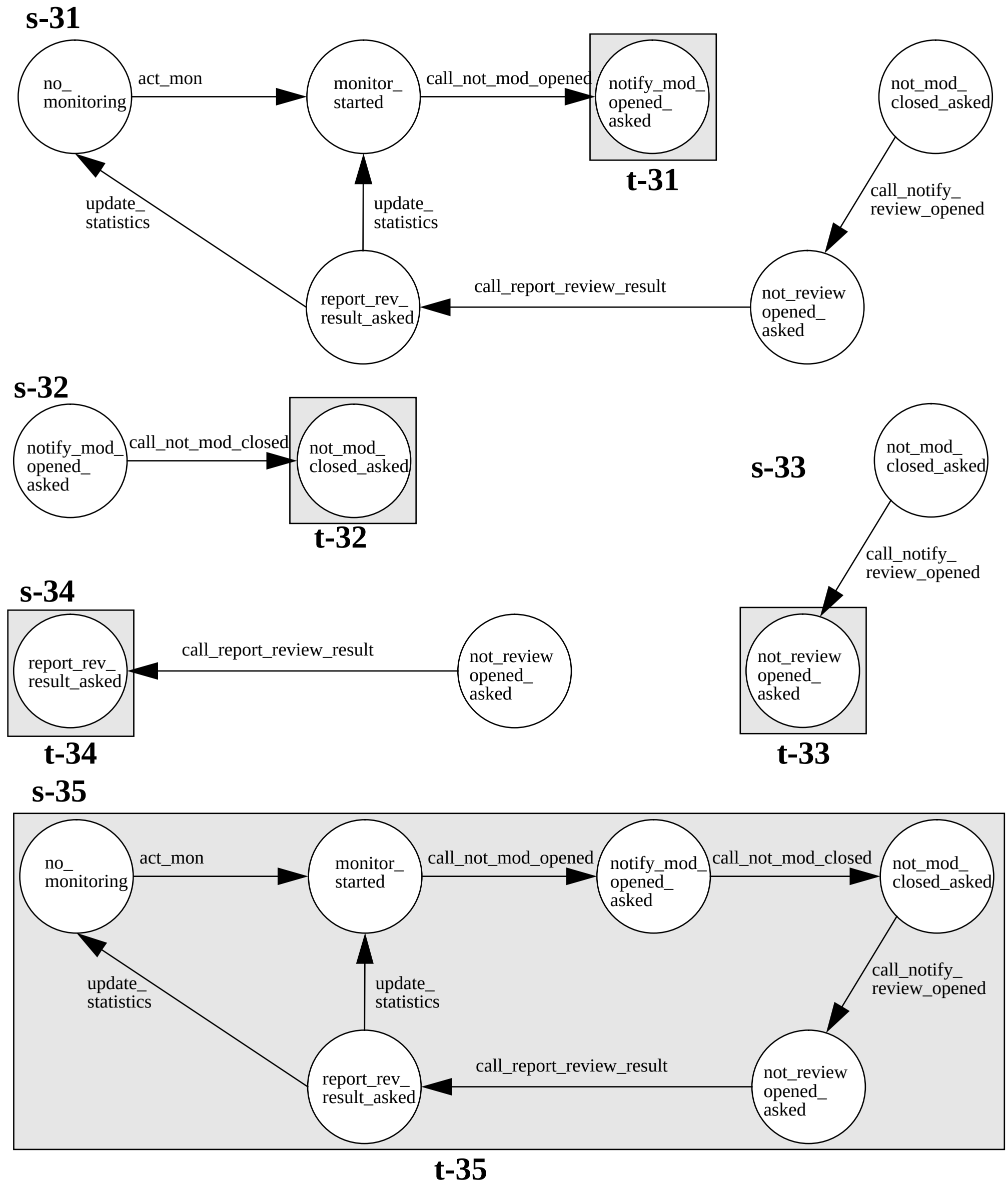


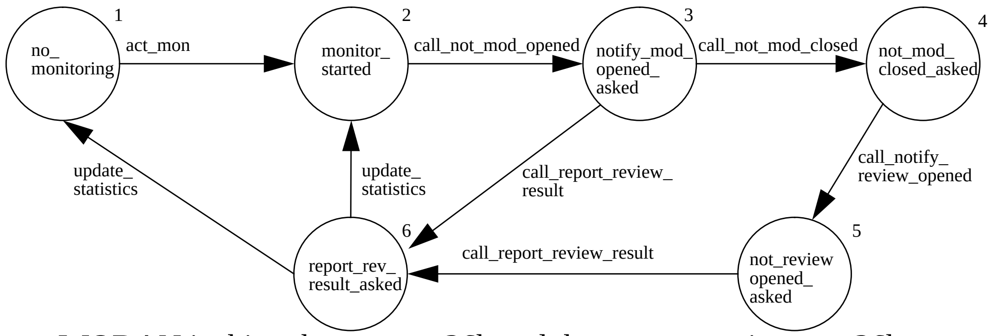
Figure 14. Int-monitor1: subprocesses and traps w.r.t. Design

5.1. Designing a new model

Suppose that the monitor gets a short-cut from *notify_mod_opened_asked* to *report_review_result* because for small projects the exact intermediate result is not relevant, then the monitor and its subprocesses can be designed as in figure 15 and in figure 16 respectively.

When comparing int-monitor1 with int-monitor2, the following notions can be found:

- The subprocesses of int-monitor2 are numbered the same as the subprocesses of int-monitor1. This is done to show the correspondence between the subprocesses of int-monitor1 and int-monitor2; subprocesses with the same number in both models correspond with each other.
- In the new model, subprocess s-34 has an extra state 3 and a transition from state 3 to 6.
- In the new model, subprocess s-35 has an extra transition from state 3 to 6.
- Int-monitor 2 has an extra subprocess s-39. This subprocess will be used in section 5.5, until there it can be ignored.



W.r.t. to WODAN is this subprocess s-36b and the state space is trap t-36b

Figure 15. Int-monitor2: new STD of the internal behaviour

As long as the current version of *Design* is being kept as a manager of monitor, it is possible to switch from *int-monitor1* to *int-monitor2* when enacting without introducing inconsistencies. Thus in this special case, we are not obliged to use a new version of *Design* when switching to the new evolution stage. However, as long as we keep using the old version of *Design*, *Int-monitor2* will show the same behaviour as *Int-monitor1*. This follows from the notions below:

- As, with respect to *Design*, trap t-31 is only a trap from subprocess s-31 to s-32 the newly introduced transition from state 3 (t-31) to state 6 (in s-34) will not be used, so the behaviour remains the same.
- With respect to *Design* s-34 can only be reached from s-33 so the newly introduced state 3 in s-34 can not be reached and because of this it will not affect the behaviour.
- Subprocess s-35 (the neutral subprocess) means that all states from the monitor can be reached so introducing an extra transition here does not effectively influence the behaviour.

As mentioned above, the design process also has to be changed to achieve the new result. This new design process will be called *Design2*. Its STD is shown in figure 17. *Design2* can prescribe different subprocesses in some states. Which subprocess will be chosen depends on the strategy:

Str-1 Determine in state 11 whether it is a small project or a big project. When it is a small project follow the path 12, (13), 15, 17 prescribing subprocess s-34 in all states. Otherwise follow the path 12, (13), 14, 15, 16, 17 prescribing subprocess s-32, (s-32), s-32, s-33, s-33 and s-34 respectively.

Another strategy which can be used is the following one:

Str-2 Determine in every state in which two different subprocesses can be prescribed whether it is a small project or a large project. When it is a small project subprocess s-34 has to be prescribed and the transition to the following state has to be taken following the path 12, (13), 15, 17. Otherwise subprocess s-32 or s-33 has to be prescribed (depending on the state) and the transition to the following state has to be taken following the path 12, (13), 14, 15, 16, 17.

When using strategy str-2, one of the consistency problems as mentioned in the previous chapter will arise. As it is one of the purposes of this example to clarify these problems with their solutions, the strategy str-2 will be worked out in the following sections.

Note that in this special case, the consistency problem could have been avoided by using strategy str-1 in stead of strategy str-2.

Note also that in fact the new process descriptions of *Design* and of *int_monitor* are only new subprocesses of some anachronistic processes. The transition from the old subprocess *int_monitor1* to the new subprocess *int_monitor2* and from the old subprocess *Design* to the new subprocess *Design2* is made when WODAN prescribes the subprocesses corresponding

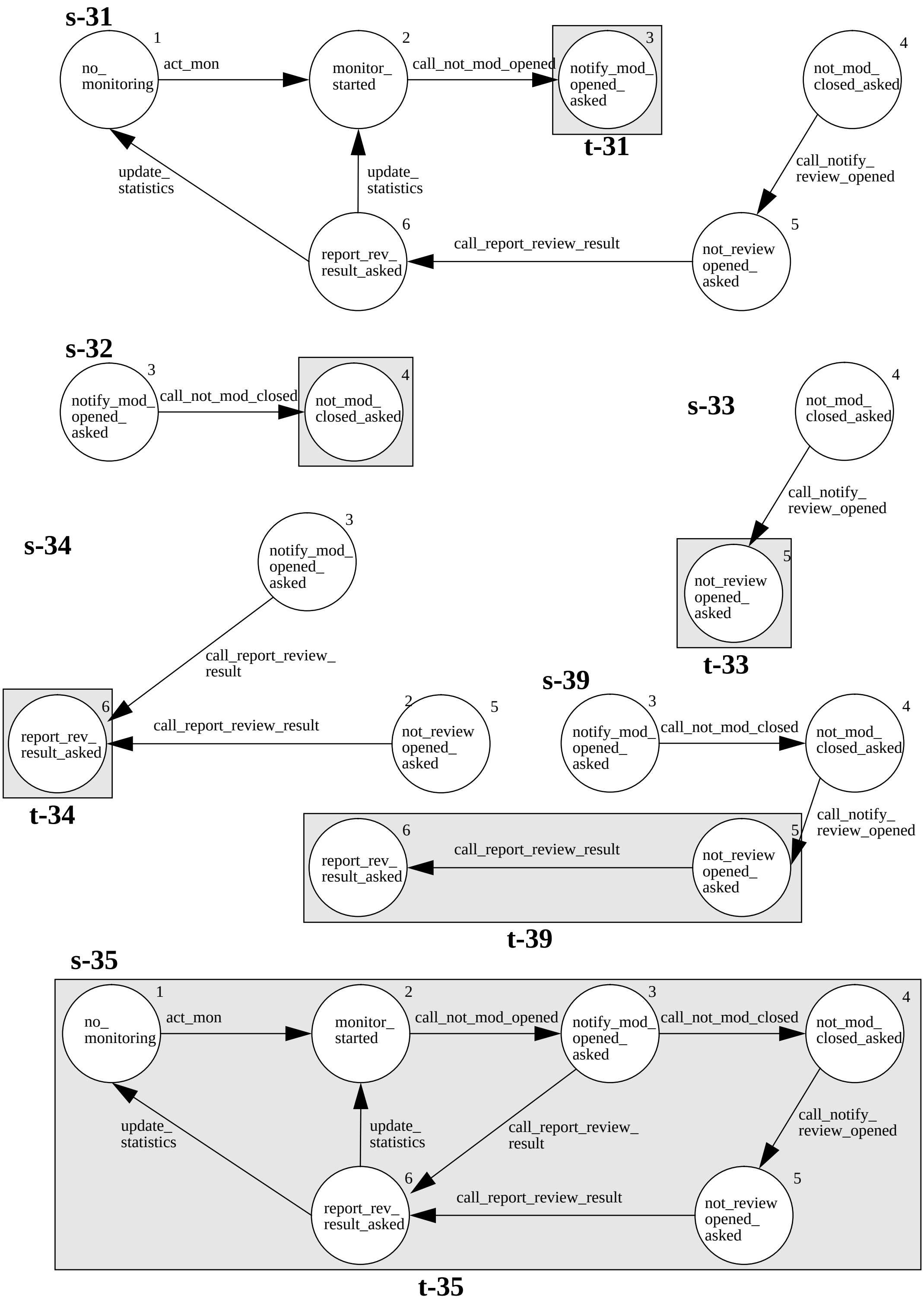
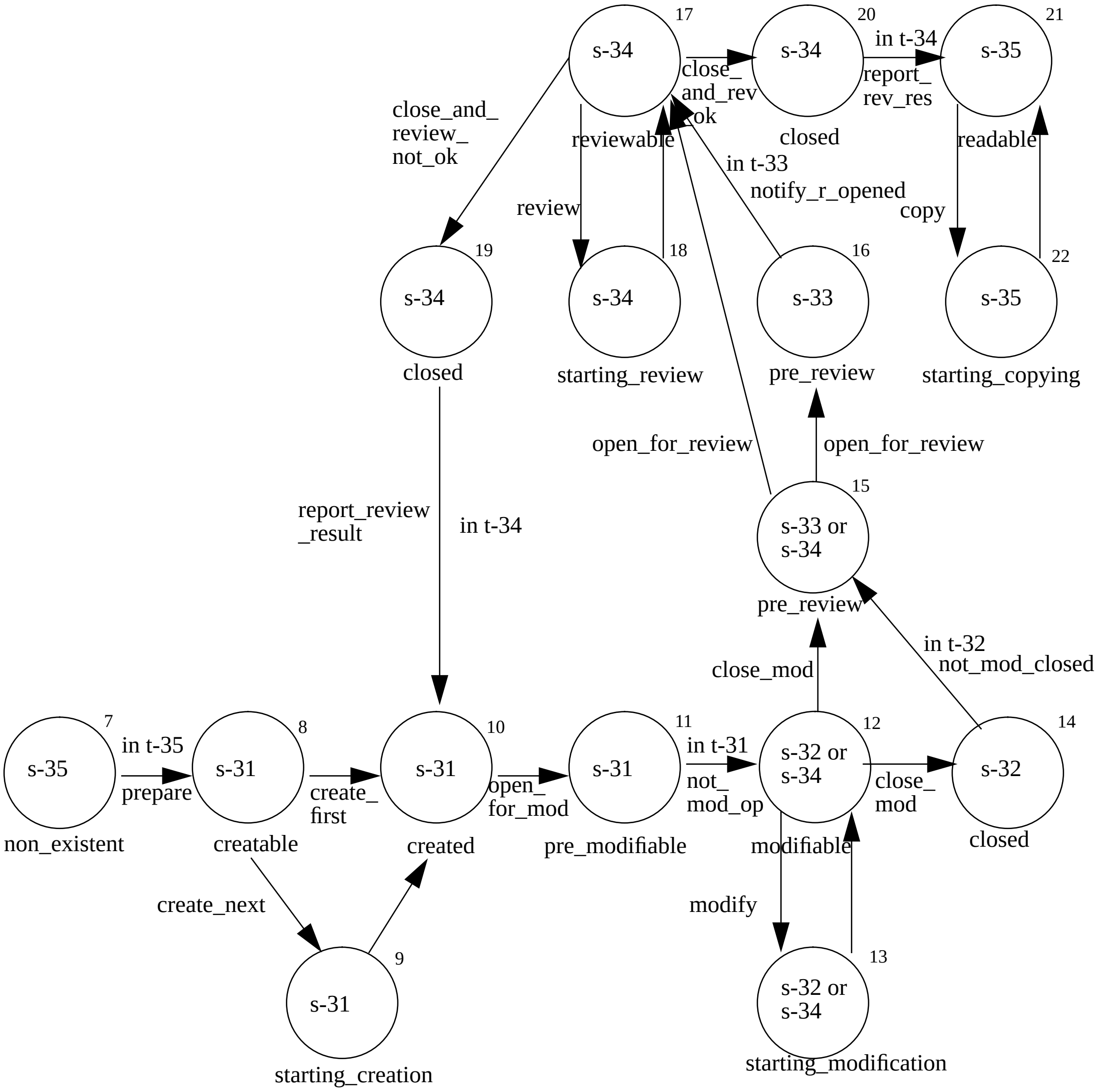


Figure 16. Int-monitor2: subprocesses and traps w.r.t. Design

with the new STD's.

Despite the remark above that we are not obliged to use a new version of *Design* when switching to the new evolution stage, it is still necessary to design WODAN in such a way that the transition from *int-monitor1* to *int-monitor2* is made simultaneously with the transition from *Design* to *Design2*, just as has been pointed out in section 3.1. This is the case since *Design2* can prescribe certain subprocesses of *Int-monitor2* which are no subprocess of *Int-monitor1*.

Thus in this case, it is possible to prescribe *Design* together with *Int-monitor2* (in accord-



W.r.t. to WODAN is this subprocess s-38 and the state space is trap t-38

Figure 17. Design2: only viewed as manager of int_monitor2

ance with the remark on page 30, but this is not very useful) but it is not possible to prescribe *Design2* together with *Int-monitor1*.

5.2. Starting the new model during the enactment of a software process

In the previous section, we have designed new models to model the changed behaviour of the monitor operation and the class *Design*. In this section, we will model the behaviour of the evolution step to switch from the old behaviour of the total model to the new behaviour of the total model. As only the behaviour of *Design* and of *Int-monitor* changes, we will only take these two behaviours into consideration.

In comparing the original model with the new model the following observations can be made:

A) When the actual instance of *Design* models designing a huge project the behaviour remains the same so the new processes can be started immediately.
B) When the monitor is in s-31 or s-34 no inconsistencies will be introduced.

C) When it is a small project and the monitor is in s-32 or s-33 a problem can arise: when switching to the new model, *Design2* will prescribe s-34 because of it is strategy str-2 (determine in every state which project type it is and prescribe the right subprocess according to the project type). When at this moment the monitor is in state 4 (which is possible in both subprocesses) it can not leave this state any more since state 4 and the transitions out of state 4 are no part of s-34. This is an example of problem P1 as mentioned in chapter 4.

The problem introduced in observation C can be solved in 3 manners:

- Introduce the new method only when the monitor is not in state 4, so when it is not in subprocess s-32 or s-33. This means that the new behaviour must be introduced in some phases: say to design that the new behaviour has started but that it has to wait with the new strategy until it is in a safe state. This solution, which in fact is solution S1 of the previous chapter, can be found in section 5.3..
- Make an extra subprocess s-39 which consists of the states from the subprocesses that would introduce inconsistencies and change the strategy of *Design2* to be the following one (str-3): when switching to the new process description while prescribing s-32 or s-33 and according to the new strategy s-34 would be necessary, then prescribe s-39 in stead of s-34. This method can be found in section 5.4. and it is an example of solution S3.
- Use another strategy in which the problem does not arise (like strategy str-1). This solution has not been mentioned in chapter 4 since it is not a solution to the general problem that the subprocess which will be prescribed during EVS2, misses one of the states of the corresponding subprocess during EVS1. However, it will always be a good advise to examine the model carefully to find out whether it is possible to avoid the inconsistencies by using a differently designed model which exploits specific properties of the real life process which has to be modelled and of the models which have been used in the original evolution stage. This approach is similar to the one used in the special solutions shown in chapter 4. The special solutions shown there exploited a specific property of the models before and after the evolution step; they all made use of the fact that many similarities exist between the STD's before and the STD's after the evolution step. The special solution shown here makes use of another specific property; it makes use of the fact that the total behaviour of the model is not only determined by the STD's but also by a strategy.

5.3. Designing WODAN to manage the change

When choosing for solution S1, WODAN must consist of 4 states:

- There is no change made, the whole process can be enacted at a normal way.
- The new processes are being designed. The process still has to be enacted at the old way.
- The new processes have been designed. The intermediate phase of the design process can be started.
- Design has reached a safe state, the final subprocess of design can be prescribed and everything can enact at the new way.

The intermediate subprocess of design, which will be called *TempDesign*, and WODAN are shown in figure 19 and in figure 18 respectively.

TempDesign is a manager of *int-monitor1*. As has been mentioned in solution S1, it is designed analogous to *Design* with the trap of *TempDesign* consisting of the states in which no problem will arise and with the transitions which would lead out of this trap, removed from the STD of *TempDesign*.

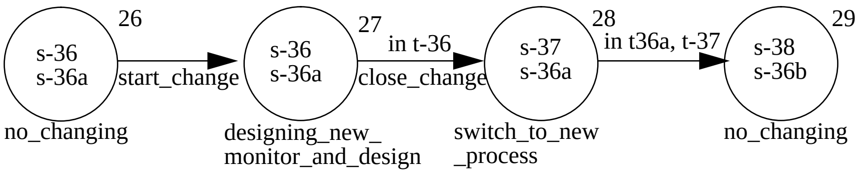


Figure 18. WODAN: switch to int-monitor2 and Design2 via TempDesign.

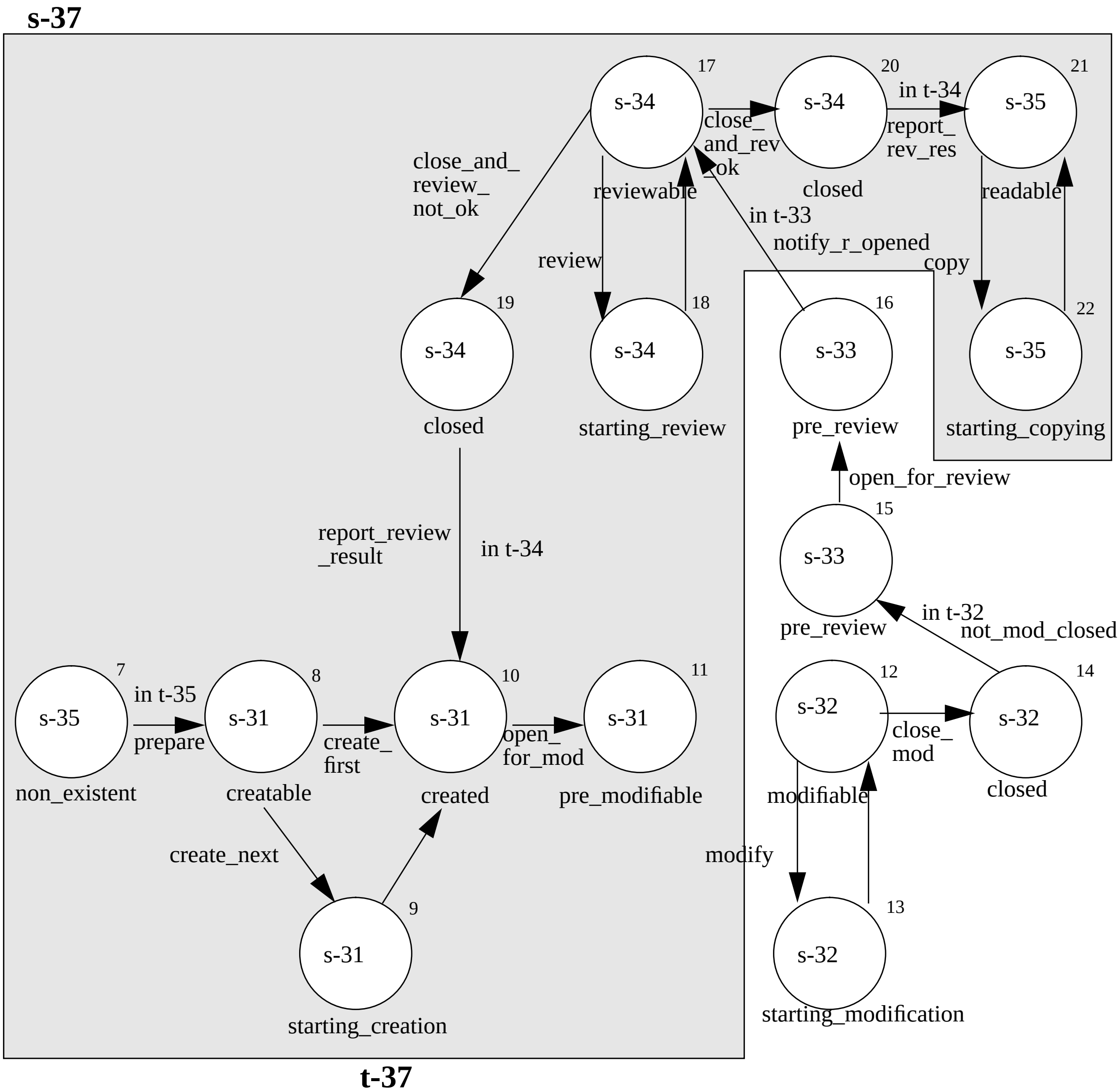


Figure 19. TempDesign: subprocess for transition to the new design

5.4. Using an extra subprocess for int-monitor to avoid intermediate states

When choosing for solution S3, WODAN only has to consist of 3 states since there is no intermediate phase for the design process. The design process, called *Design3*, has to manage everything at the right manner with the aid of the extra subprocess s-39 and strategy str-3.

Design3 and WODAN to control the change are as displayed in figure 21 and in figure 20 respectively.

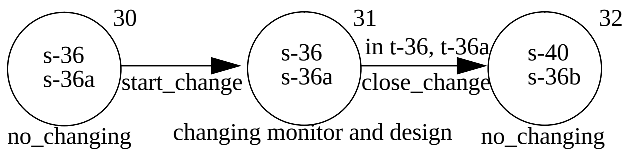
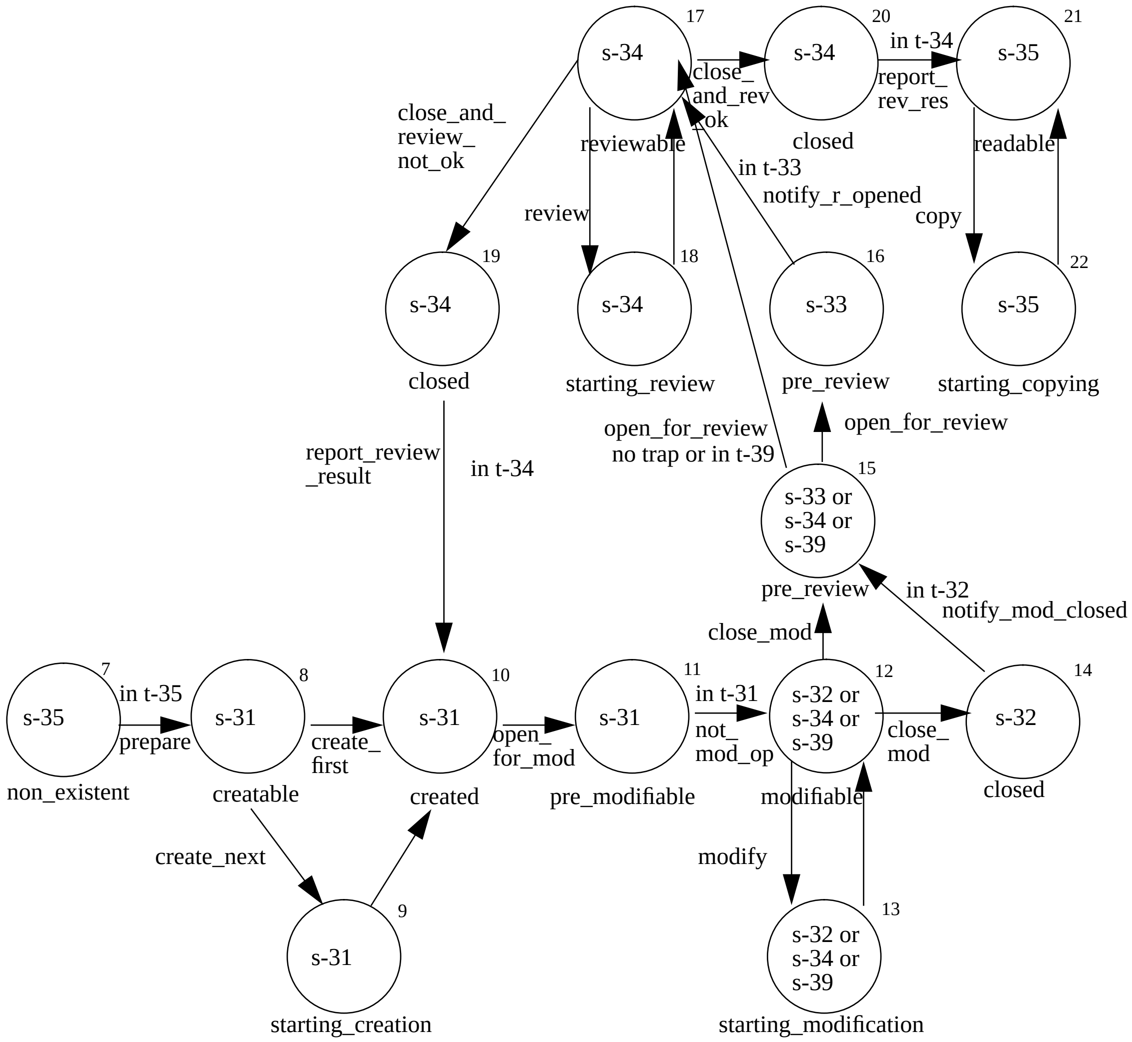


Figure 20. WODAN: switch to monitor2 and Design3.



W.r.t. to WODAN is this subprocess s-40 and the state space is trap t-40

Figure 21. Design3: only viewed as manager of int_monitor2

5.5. Losing some restrictions

As can be seen from this example, restriction 1 (only change strategies, transitions and subprocesses) is too strong to make it possible to design models for the external processes which are easy to interpret:

When there are no extra states (in accordance with restriction 1) the manager must examine its strategy to determine which subprocess should be prescribed in some states. This makes the model very complicated to interpret since one can no longer determine from the state of the manager process in which subprocess the employee process is. To find this out one should know all about the followed strategy and eventually the followed history up to now. Thus, it would be useful when the manager has some extra states to prescribe the new subprocesses introduced for the new behaviour restrictions.

Therefore, the first restriction will be redefined to the following one:

restriction 1') Do not change the state space of the internal processes, only change the strategies, transitions and subprocesses. When necessary, do introduce extra states for the external processes to prescribe the new subprocesses in a clear manner.

When applying this new restriction 1' to the example one only has to change the manager process (*Design*) and the introduction of this manager (via WODAN). In the new design process description there will be enough states to prescribe only one subprocess per state. This will also simplify the strategy:

Str-4 Determine in state 11 whether the current is project small or large. When it is a small project the path 23, (24), 25, 17 has to be followed, prescribing the right subprocesses. Otherwise the (old) path 12, (13), 14, 15, 16, 17 has to be followed, prescribing the right subprocesses.

In fact this strategy is analogously to strategy str-1 in section 5.1., the only difference is that following the right path and prescribing the right subprocesses is now explicitly forced by the states of the STD of *Design*, while in the other case it was implicitly forced by the strategy.

As this new version of *Design* has been extended with some extra states and transitions, it will be called *ExtendedDesign*. When switching to *ExtendedDesign*, no inconsistencies will be introduced since *ExtendedDesign* can not prescribe s-34 when it was prescribing s-32 or s-33 according to the old behaviour. It now first has to follow the path corresponding to the old path in subprocess s-36, since it can only decide in state 11 to follow the new path according to the new strategy.

Therefore, introducing this change of the software process model only requires 3 states for WODAN. WODAN and *ExtendedDesign* are shown in figure 22 and in figure 23 respectively.

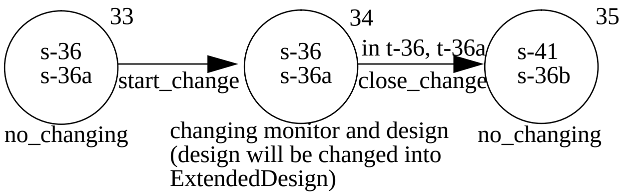
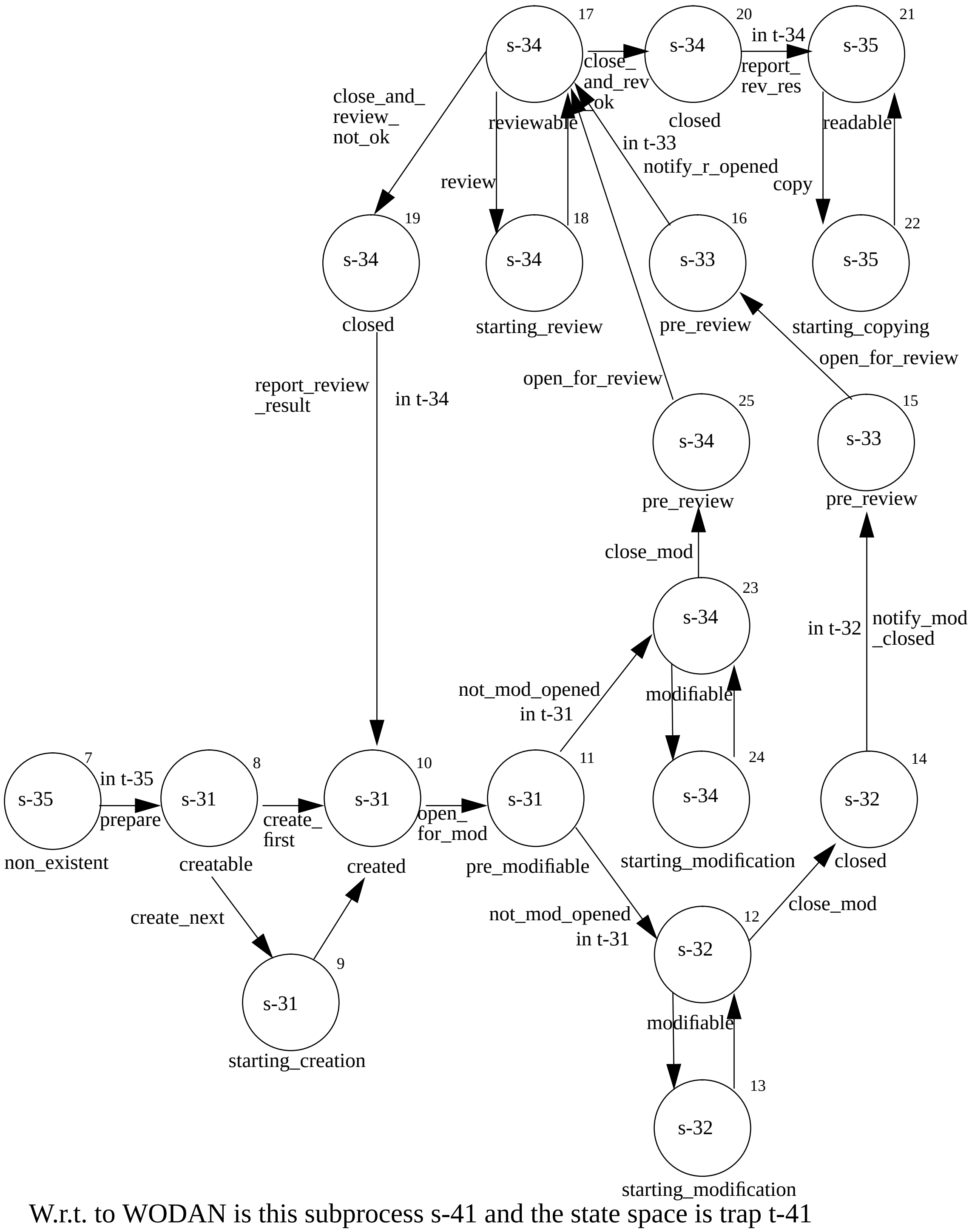


Figure 22. WODAN: switch to ExtendedDesign and int-monitor2.

5.6. Concluding remarks

- Although it is possible to change the behaviour of an enacting process with the restriction that the state space is not to be changed at all, this introduces some problems:
  - There is a high probability that problem P1 arises as in some states a new subprocess will be prescribed which contains other states and transitions then the subprocess that was prescribed according to the old strategy.
  - It is hard to interpret the exact state of the process as in one state more subprocesses can be prescribed. To interpret the exact state one should be aware of the followed strategy and the history up to then.

  The first problem can be solved with the solutions mentioned in chapter 4. The second problem however, can not be solved in such a manner. To avoid this problem, the first restriction has been made less stronger to allow extra states for the managers.
- The problems mentioned in the previous chapter can sometimes be avoided by carefully extending the model. In my personal opinion, such a carefully designed extension should be used whenever possible as it seems to be better to avoid inconsistencies then to solve them afterwards.
- In practice, it may be possible to exploit special properties of the real life processes and the



W.r.t. to WODAN is this subprocess s-41 and the state space is trap t-41

Figure 23. ExtendedDesign: only viewed as manager of int_monitor2

models which have been used before and after the evolution, to simplify the evolution step. Thus, one can not only make use of the similarities between the STD's before and after the evolution steps as has been shown in chapter 4 but one can also exploit other properties like the strategy which is being used.

Chapter 6
Extending the model to cover more of the ISPW-6 example

In the next chapter, the process change part of the ISPW-7 example will be worked out in SOCCA. However, before we can start with this, the SOCCA example, which is based on the ISPW-6 case, must be extended somewhat because of the following reason. The change that has been proposed in the ISPW-7 example concerns the cooperation between *DesignDocument* and (modify) *Code*. As *Code* has not yet been included in the current model, this has to be done first.

We will continue with the last version of the model as presented in the previous chapter; for designing a document, the model of *ExtendedDesign* (figure 23) will be used and *int-monitor2* (figure 15) will be used to model the process of monitoring the progress of *ExtendedDesig*n.

Since *ExtendedDesign* is a model for designing a document, it will be called *Design* again in the sequel of this thesis and likewise, *int-monitor2* will be referred to as *int-monitor* from now on. Furthermore, the SOCCA model as it is at this moment, will be called ‘the original SOCCA model’ or the ‘original SOCCA example’ and the model introduced in this chapter will be called ‘the current model’ or ‘the current SOCCA example’.

In this chapter, the SOCCA approach will be followed as far as possible; not only the behavioural aspects of the processes will be modelled with the PARADIGM part of SOCCA, but also the data aspects will be modelled by means of the EER based class diagrams and the SOCCA extensions to these. However, the process perspective will be ignored as the use of object flow diagrams and the integration of these into SOCCA has currently not been worked out completely.

6.1. Redesigning the class diagrams

First a new class diagram has to be defined, as in addition to the human agents also automated tools are necessary. For example a compiler to compile a code source document into a code object document. This class diagram is given in figure 24.

Together with this class hierarchy specification the attributes and operations of the various classes have to be defined. They are presented in figure 25.

In this class diagram two extra classes have been defined compared with the original SOCCA example:

- *Compiler*: this is an automated tool for compiling source code into object code. It will be discussed in detail further down.
- *ProjectManager*: this is the project manager with several tasks, like scheduling and assigning tasks and monitoring the progress of the design and review process. The behavioural aspects of the monitor process have been worked out in chapter 5. However, as chapter 5 was only meant to show the basic aspects of dynamic process change within SOCCA, the other SOCCA aspects of the monitor were not discussed there. Therefore, these other SOCCA aspects of the monitor will be discussed here.

  The project manager’s task ‘scheduling and assigning tasks’ is called *assign_and_schedule_tasks* and its behaviour will also be discussed here.

Furthermore, some operations are moved from the superclass *DesignDocument* to the subclass

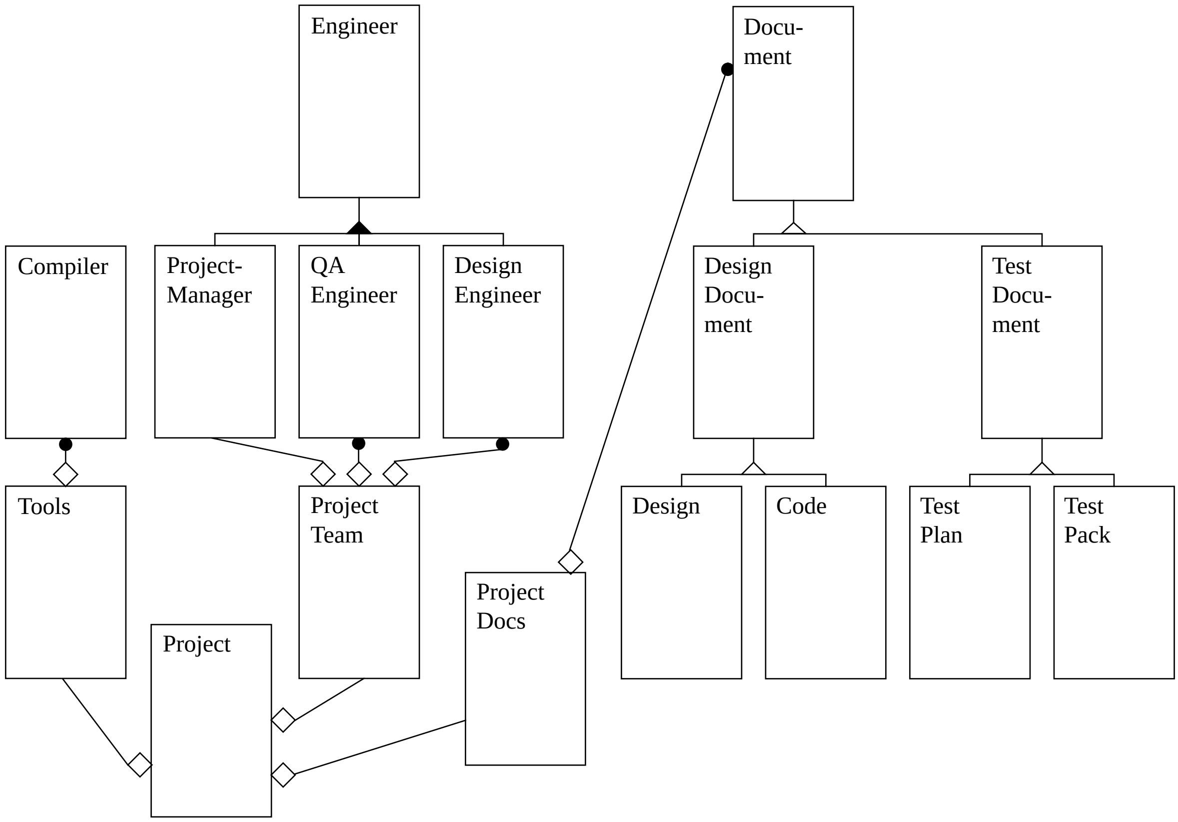


Figure 24. Class diagram: classes and is-a and part-of relationships

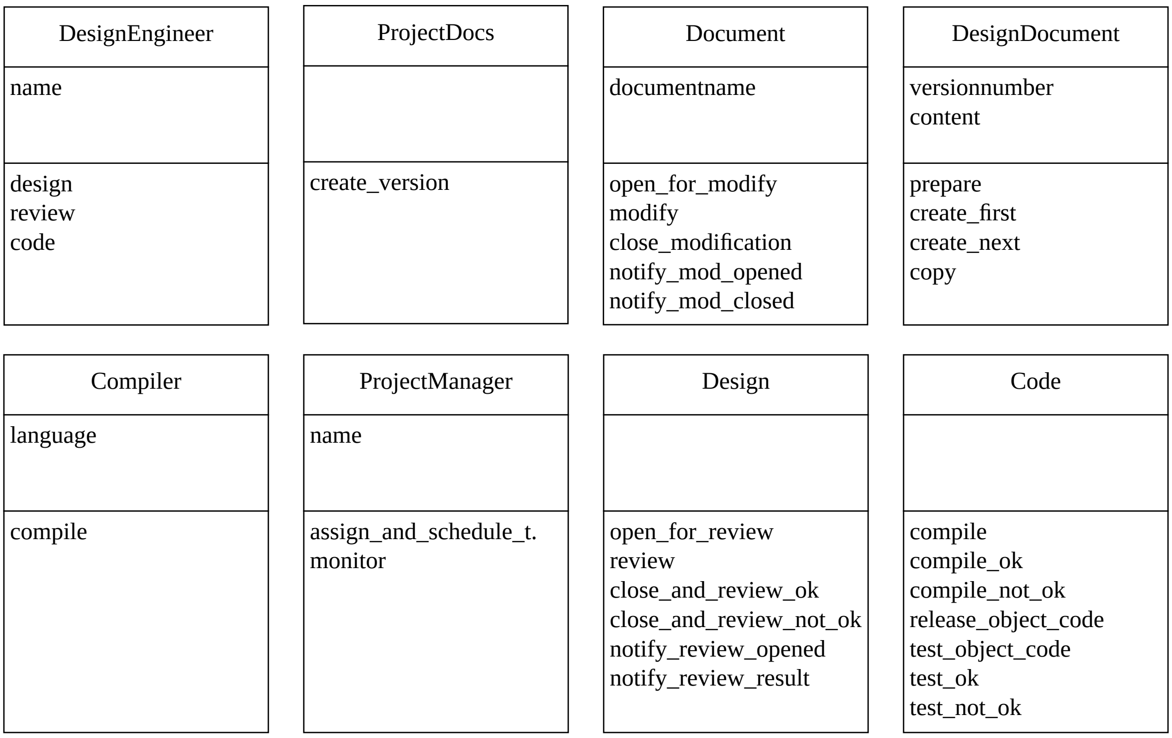


Figure 25. Class diagram: attributes and operations

Design, as these operations are only necessary in *DesignDocument's* subclass *Design* and not in *DesignDocument's* subclass *Code*. A last change in the class diagram is that the part-of relation ship between *ProjectDocs* and the various documents has been moved from the subclasses *Design*, *Code*, *TestPlan* and *TestPack* to their ancestor class *Document*. This has been done to show more explicitly that *ProjectDocs* is constituted of (many) *Document*s.

In the following step the general relationships between the classes have to be defined. For the sake of completeness not only the relationships between the new classes are shown but also the other relationships from the current SOCCA example. The general relationships are shown in figure 26.

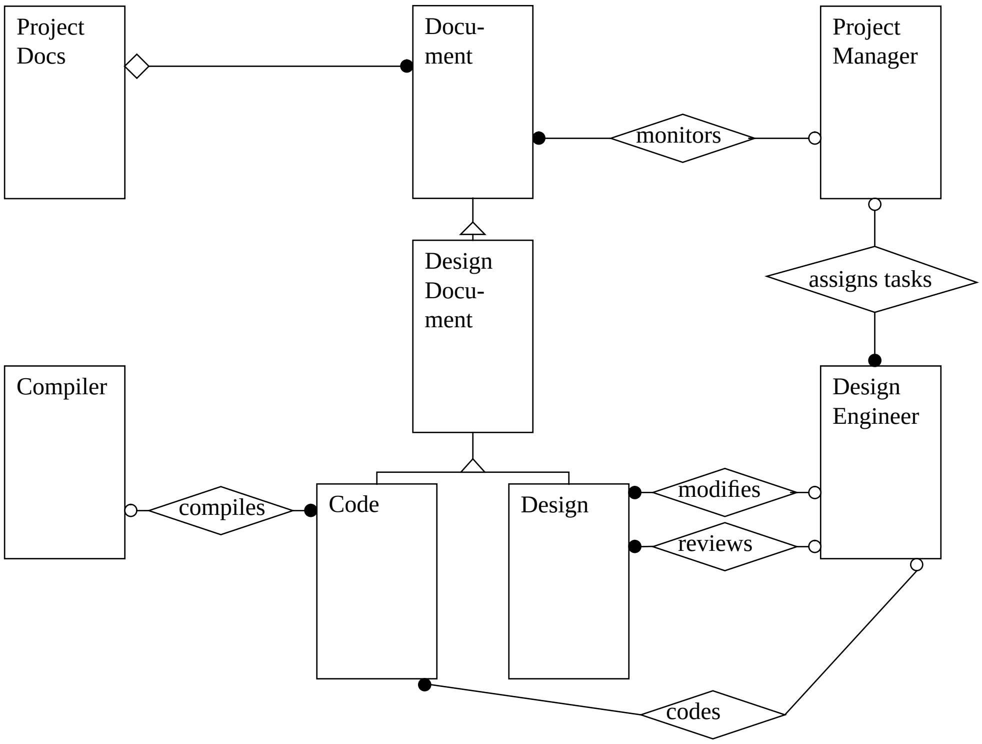


Figure 26. Class diagram: classes and general relationships

Note that in the SOCCA example the superclass *DesignDocument* has been defined. Both the classes *Code* and *Design* inherit operations and attributes of this superclass. The class *Code* represents a code document and likewise, the class *Design* represents a design document. In the discussion in this theses, terminology is used to address both code documents and design documents. The term 'design document' means an actual instance of the class *Design* and not an instance of the superclass *DesignDocument*.

As a last step in the EER part of the SOCCA specification, the uses relationship is given (figure 27) together with the import list (figure 28). Note that in the import list some export operations are parametrized with the parameter *doc_name*. Each such parametrized operation in fact stands for *n* separate export operations where *n* is the number of different document names.

One of the requirements of the ISPW-6 case is that *Code* has to wait with releasing the object code for the test phase until the design document has been approved. The behaviour of *Code* therefore depends on the behaviour of *Design*, this dependency is modelled via the internal behaviour of *release_object_code*. As can be seen in figure 56 on page 55, which shows the traps and subprocesses of *int-release_object_code* with respect to *Design*, *int-release_object_code* waits in its starting state until *Design* has been approved. Since it is waiting in its starting state, there is no transition to this state which could be associated with an export operation of *Design*. This means that the dependency of the behaviour of *Code* on the behaviour of *Design* is not modelled via an explicit export operation of *Design* but via an implicit dependency. Therefore this uses relationship (uses11) is shown with a dashed arrow in stead of a solid one in the import/export diagram. Note that this is a deviation of the original

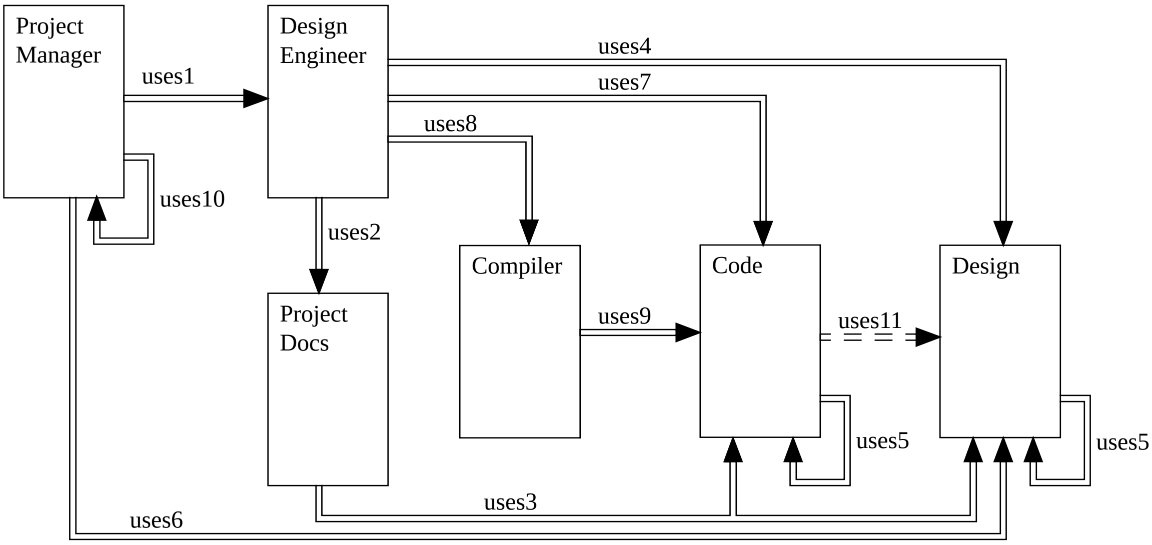


Figure 27. Import/export diagram

use1
- design(doc_name)
- review(doc_name)
- code(doc_name)

use2
- create_version(doc_name)

use3
- create_first
- create_next

use4
- open_for_modification
- modify
- close_modification
- open_for_review
- review
- close_and_review_ok
- close_and_review_not_ok

uses5
- copy
- prepare

uses6
- notify_modification_opened
- notify_modification_closed
- notify_review_opened
- report_review_result

uses7
- open_for_modification
- modify
- close_modification

uses8
- compile(doc_name)

uses9
- compile_ok
- compile_not_ok

uses10
- monitor(doc_name)

uses11
- waiting for design approved (no explicit export oper.)

Figure 28. Import list

SOCCA approach. We will come back to this further on in the thesis were we discuss the communication between *Codes* export operation *release_object_code* and the external behaviour of the class *Design*.

6.2. Designing the external behaviours of the classes

The next step consists of modelling the external behaviours of the classes. From then on, the order in which the export operations can be called will be known.

Modifying the code has to be carried out by a design engineer. The export operation *code* of *DesignEngineer* can be parametrized, just like the export operations *design* and *review* in the original SOCCA example. The operations can be called in any order, so adding the *code* operation to the external behaviour of *DesignEngineer* is straightforward. The new STD will be as in figure 29. Entering the state *starting code* can be viewed as starting the modify code activity.

The behaviour of the class *ProjectDocs* is also extended because now not only new versions

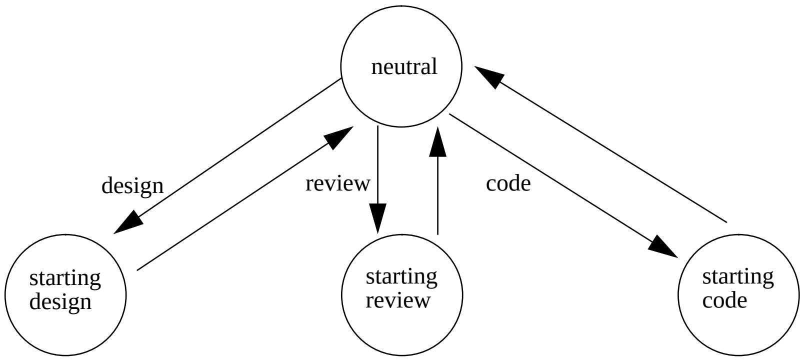


Figure 29. DesignEngineer: STD of the external behaviour

of design documents have to be created but also new versions of code documents. The new behaviour of *ProjectDocs* is shown in figure 30. This model has two transitions with the same

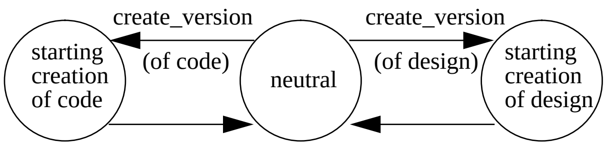


Figure 30. ProjectDocs: STD of the external behaviour

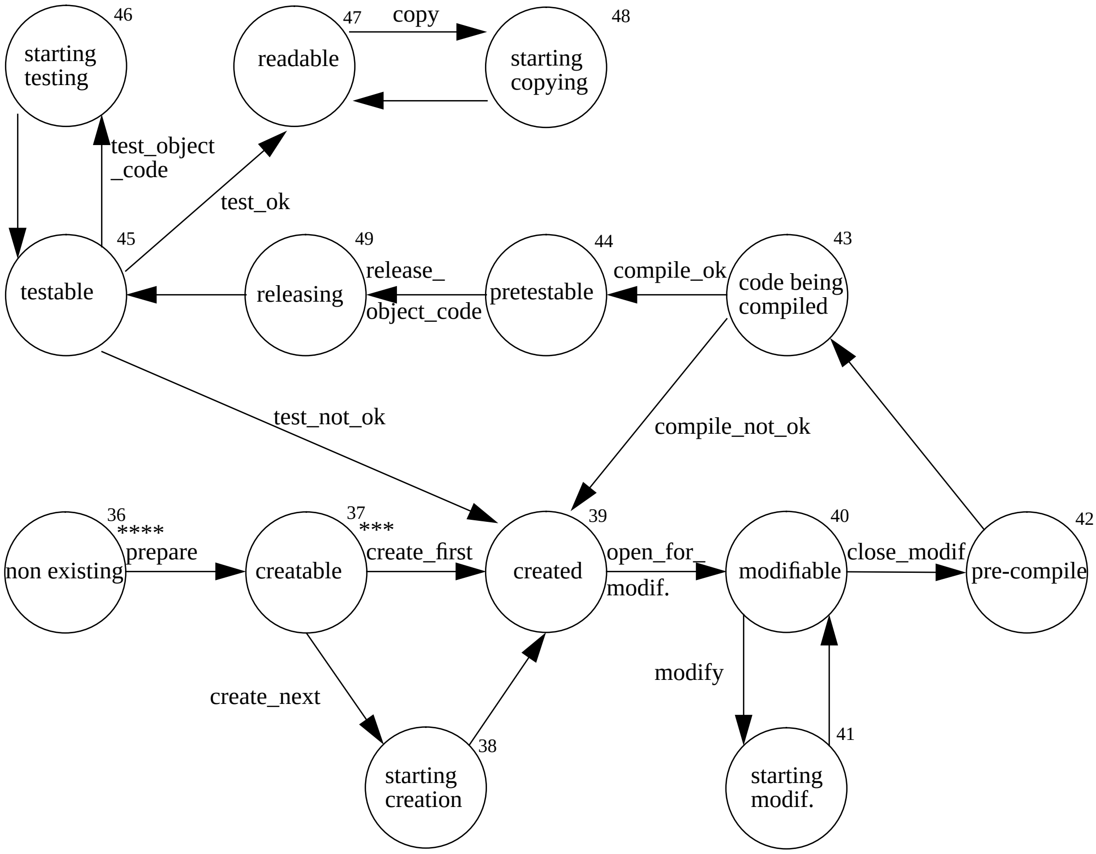
label *create_version*. This is necessary because of the trap structure of its employees *int-design* and *int-code*. One transition will be followed when *int-code* enters the appropriate trap and the other transition will be followed when *int-design* enters the appropriate trap. See section 6.4 for the details.

The third class, *Design*, remains unmodified as the extensions to the original SOCCA model do not influence the behaviour of this class.

Modifying the code document has been modelled by means of a separate class. This class is called *Code* and it is the fourth class in the current SOCCA example. Just like in the case of *Design* there will be one instance of *Code*, representing exactly one version of a code document (a source code and eventually the associated object code). The STD for *Code* will is displayed in figure 31. Some of its operations are inherited from its superclasses *DesignDocument* and *Document*. Other operations are specific for the class *Code* itself. The STD of *Code* is designed analogously to the STD of *Design*; the first part models creating the new version of the document and modifying it and the last part models the process of reading and copying it. The only real difference is in the middle part; for a design document the model has to reflect the behaviour of reviewing the design, while for the code document the behaviour of compiling and testing the code document has to be modelled. The part of testing the document will not be worked out further in this example.

Note that *compile* automatically creates an object code document when the compile result is *compile_ok*. Note also that the ISPW-6 requirement that it must be possible to have multiple object codes with one version of a source code, is not supported by this model. To support this requirement, a separate class for the object code document should have been defined, whose behaviour depends on the behaviour of the source code document. At this moment *Code* is one class, representing both the behaviour of a source code document and the only one object code document associated with it.

The fifth class is the class *ProjectManager*. In this example the export operations *monitor* and *schedule_and_assign_tasks* will be used. The export operation *monitor* is parametrized with the document name of the document which has to be monitored and it is called from the internal behaviour of the export operation *schedule_and_assign_tasks*. This means that the export operation *monitor* is an example of an export operation which is imported in another



W.r.t. to WODAN is this subprocess s-81 and the state space is trap t-81

Figure 31. Code: STD of the external behaviour

operation of the very same manager process. The STD of *ProjectManager* is given in figure 32.

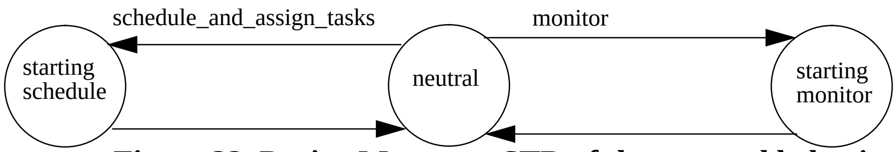


Figure 32. ProjectManager: STD of the external behaviour

The sixth and last class to be modelled is the newly introduced *Compiler* class with export operation *compile*. This export operation can be parametrized with a document name, just like the operations *design*, *review* and *code* of *DesignEngineer*, *create_version* of *ProjectDocs* and *monitor* of *ProjectManager*. The STD of *Compiler* is given in figure 33.

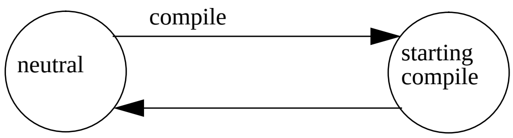
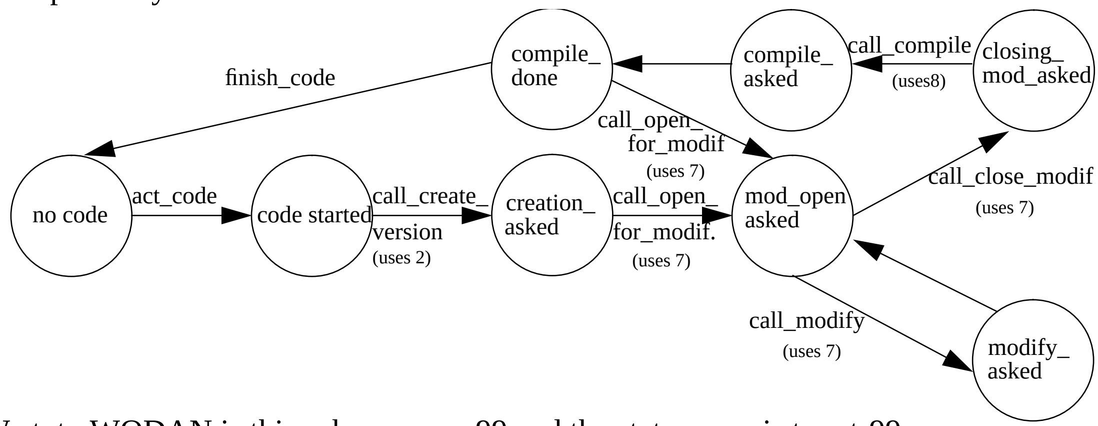


Figure 33. Compiler: STD of the external behaviour

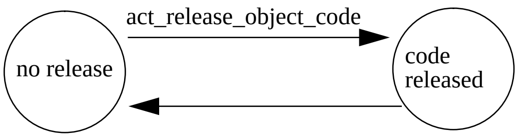
6.3. Designing the internal behaviours of the export operations

After specifying the external behaviours of the classes, the internal behaviours of the operations can be specified. In this example only the various internal behaviours of *code* (from the class *DesignEngineer*), *compile*, *release_object_code* and *test_ok* (from the class *Compiler*) and *schedule_and_assign_task* (from the class *ProjectManager*) will be given. The other internal behaviours are not really interesting within this example as they are highly internal operations which do not communicate with other parts of the model. For the internal behaviour of *schedule_and_assign_task*, only a very rudimentary scheme will be given, as it is not relevant for this example to model the complete behaviour of this complex task. It is only intended to show the process of starting the monitor process (so it is an example of starting one internal behaviour from within another internal behaviour of the same instance of the very same manager). The STD's of the internal behaviours are shown in the figures 34, 35, 36, 37 and 38 respectively.



W.r.t. to WODAN is this subprocess s-99 and the state space is trap t-99

Figure 34. Int-code: STD of its internal behaviour



W.r.t. to WODAN is this subprocess s-82 and the state space is trap t-82

Figure 35. Int-release_object_code: STD of its internal behaviour

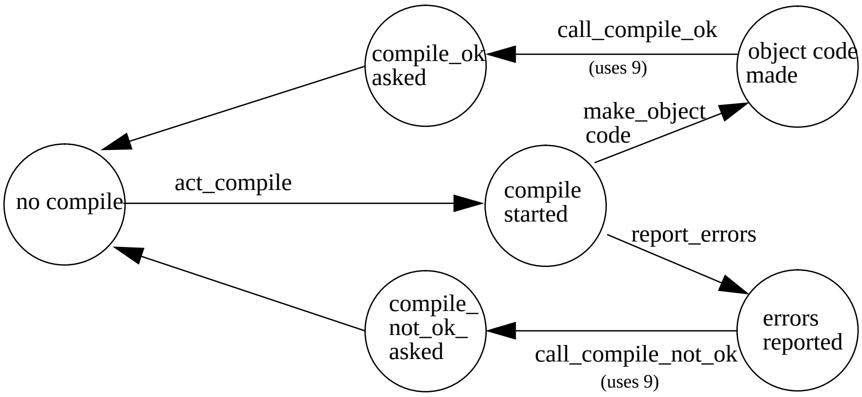


Figure 36. Int-compile: STD of its internal behaviour

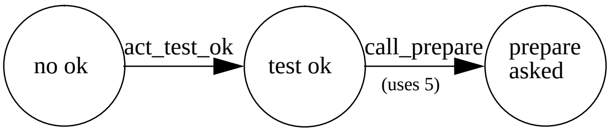


Figure 37. Int-test_ok: STD of its internal behaviour

Note that these STD's not only show what export operations are imported but also in which uses relationship these export operations occur.

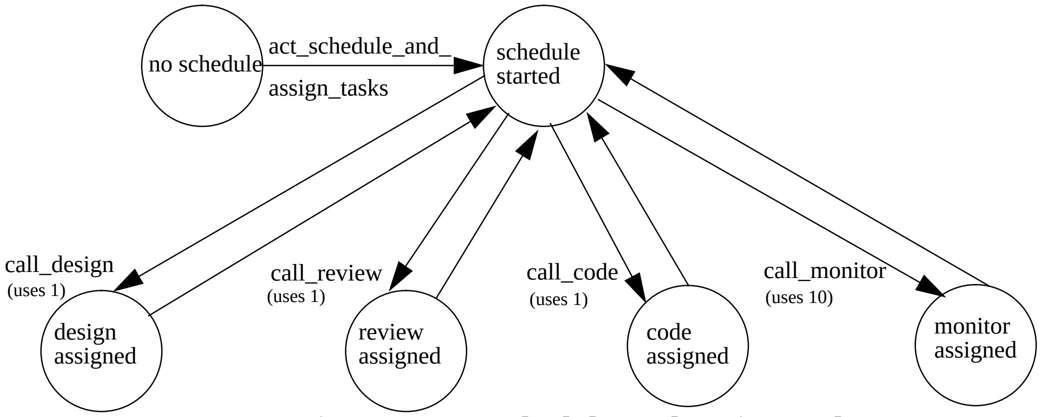


Figure 38. Int-schedule_and_assign_task

6.4. 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 specified. This communication specification is shown in several parts.

The first part of the communication specification shows the communication between the manager process *DesignEngineer* (figure 29 and 41) and its employee processes *int-code* (figure 34 and 40), *int-design*, *int-review* and *int-schedule_and_assign_tasks* (figure 38 and 39).

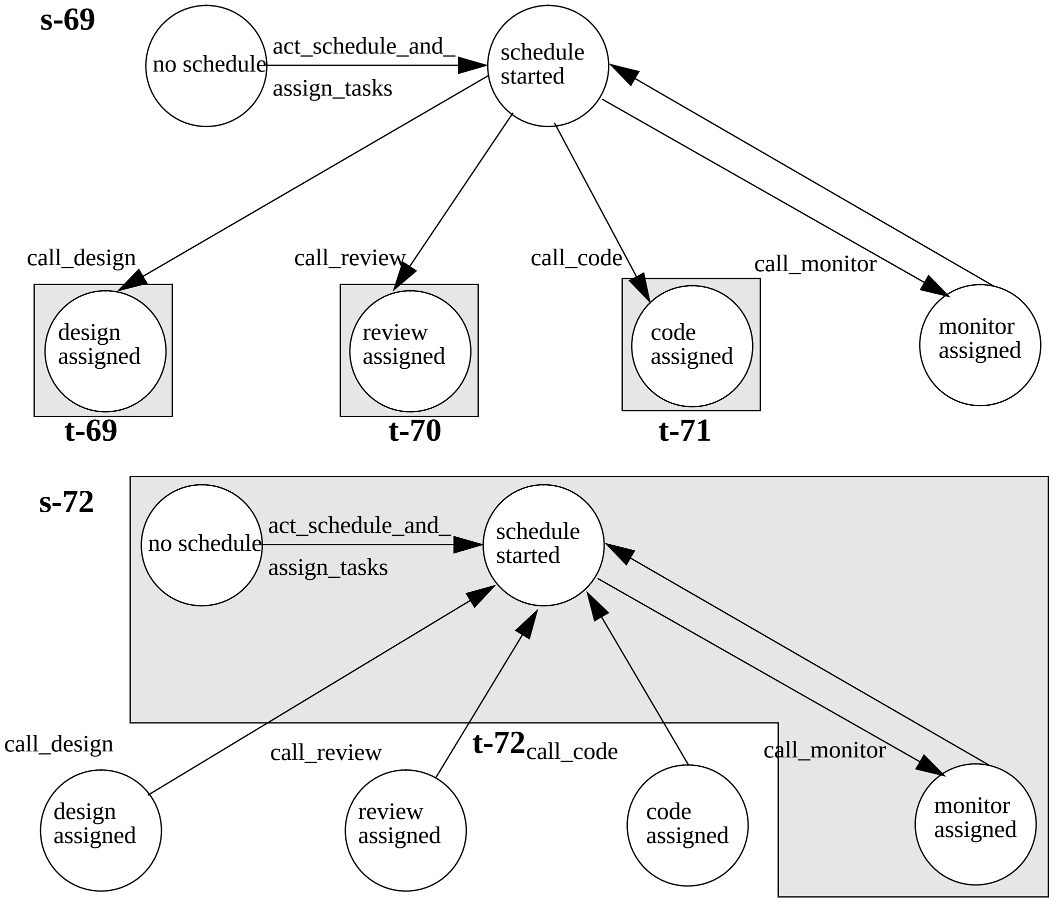


Figure 39. Int-schedule_and_assign_tasks's subp. and traps w.r.t. DesignEngineer

Note that the subprocesses and traps of *int-design* and *int-review* are not given here, because they remain the same as in the original SOCCA example.

The manager *DesignEngineer* waits in its neutral state until *int-schedule_and_assign_tasks*

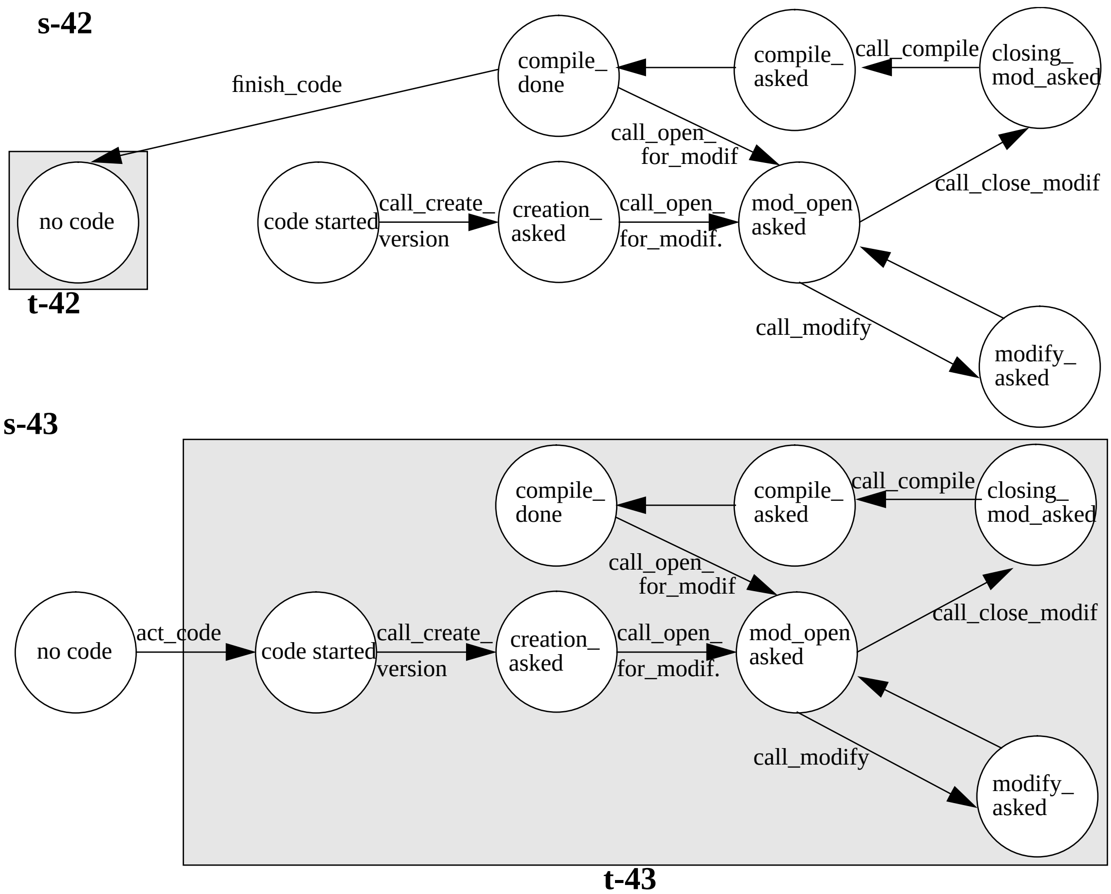


Figure 40. Int-code's subprocesses and traps with respect to DesignEngineer

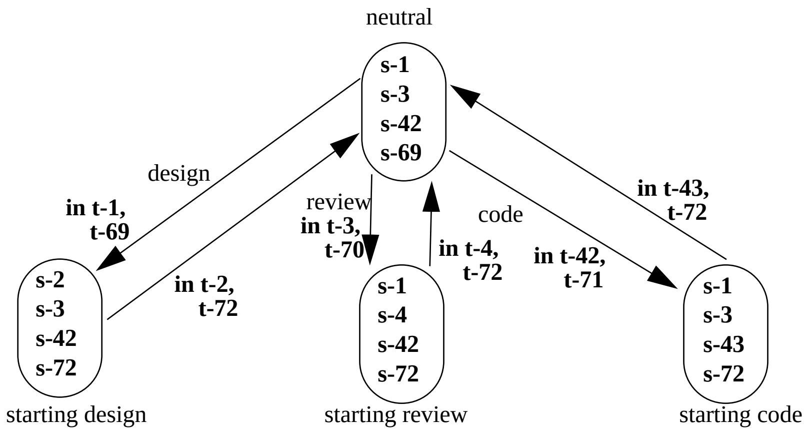


Figure 41. DesignEngineer, manager of int-design, int-review, int-code and int-sched.

calls either *design*, *review* or *code*, parametrized with a name *name1*, by entering the appropriate trap. Assume that a *call_code* transition has been made and that *int-code*, likewise parametrized with *name1*, is in its trap t-42. At that moment *DesignEngineer* can go to its state *starting_code*, prescribing subprocess s-43 of *int-code* and subprocess s-72 of *int-schedule_and_assign_tasks*. As *int-code* was in trap t-42, it will immediately enter subprocess s-43, thereby starting the process of coding a document, and entering trap t-43. Likewise, *int-schedule_and_assign_tasks* will enter subprocess s-72 and therefore it will be allowed to go back to its 'neutral' state *schedule_started* from where it can start other tasks again. As soon as it enters this state it will be in trap t-72, so *DesignEngineer* can go back to its neutral state prescribing s-42 of *int-code* and s-69 of *int-schedule_and_assign_tasks* again, waiting for other tasks to be assigned by the internal behaviour *int-schedule_and_assign_tasks* of the Project-

Manager. *Int-design* and *int-review* can be started analogous.

The second part of the communication specification shows the communication between the manager *ProjectDocs* (figure 30 and 42) and its employees *int-code* (figure 34 and 43), *int-design* and *int-create_version*. Since the last two have not been modified, their traps and sub-

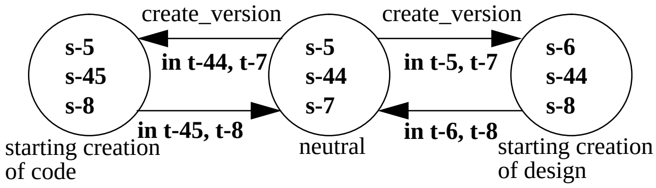


Figure 42. ProjectDocs, manager of int-design, int-code and int-create-version

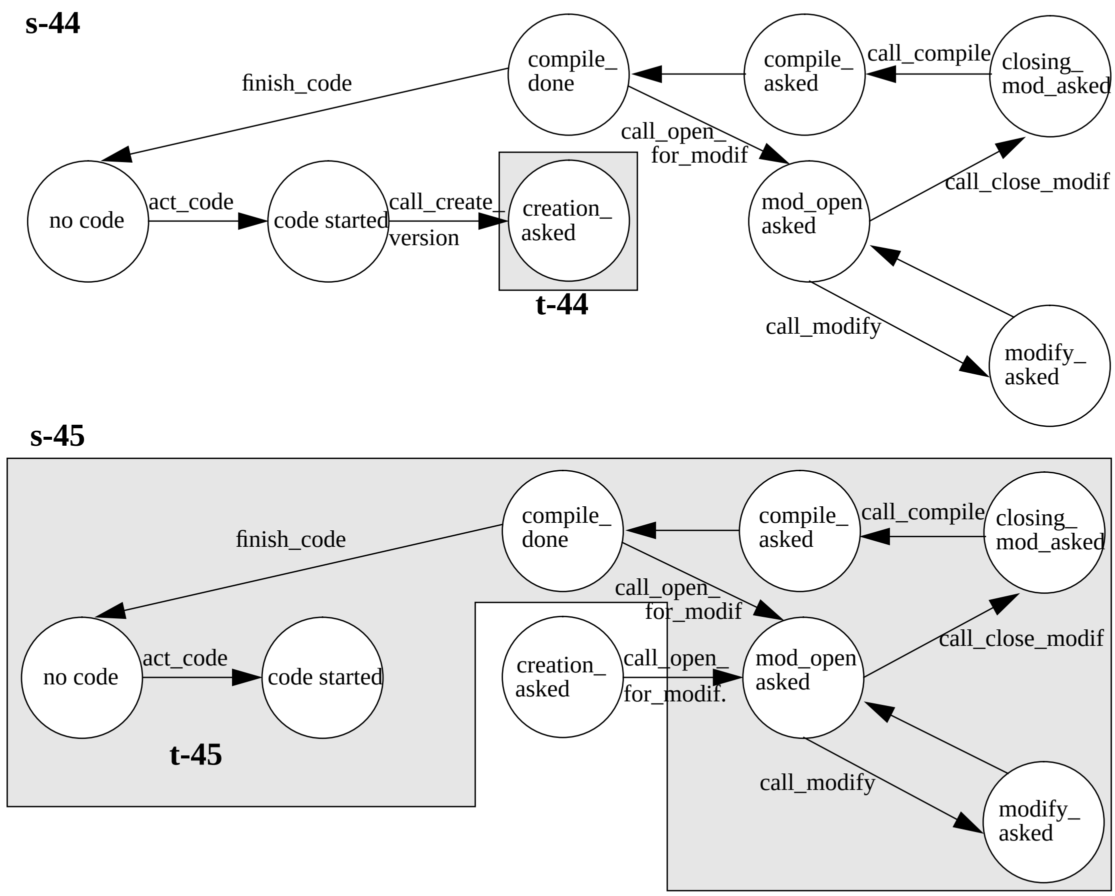


Figure 43. Int-code's subprocesses and traps with respect to ProjectDocs

processes are not given here.

This new version of *ProjectDocs* not only allows *int-design* to make a call to *create_version* by prescribing s-5 to it, but it also allows *int-code* to make such a call by prescribing s-44 to it. As soon as *int-code* or *int-design* performs a *call_create_version* by entering the appropriate trap, *ProjectDocs* will allow its employee to continue by prescribing the new subprocess to it and it will start up the internal behaviour of *create_version*. Since now two employees can perform a *call_create_version*, *ProjectDocs* has two different states which can be reached in one step from the neutral state: one state for the call done by *int-code* and the other state for the call done by *int-design*.

Note that another approach to model *ProjectDocs* is by making use of the concept of roles and views. In stead of having two different transitions modelling the same operation *create_version* and likewise having two states modelling starting a creation, one can also make a model of *ProjectDocs* with one *neutral* state, one *starting_creation* state and one transition labelled *create_version* to model both creating a new version of a code document and of a design document. Whether the transition *create_version* models creating a new version of *code* or a new version of *design* depends on the view one has of the class *ProjectDocs*; at the moment that one views *ProjectDocs* as a manager of *int-code*, following the transition *create_version* yields the creation of a new code document. Likewise, a new design document will be created when *ProjectDocs* is viewed being a manager of *int-design*. This concept of incorporating views and roles into SOCCA can be a topic of future research.

The third part of the communication specification shows the communication between the manager *Code* (figure 31 and 48) and the employees *int-code* (figure 34 and 44), *int-create_version*, *int-create_next (copy call from the next instance of Code)*, *int-create_next (the managers own internal behaviour)*, *int-compile* (figure 36 and 45), *int-release_object_code* (figure 35 and 46), *int-test_ok (other instance,* figure 37 and 47*)* and *int-test_ok (same instance,* figure 37 and 49*)*.

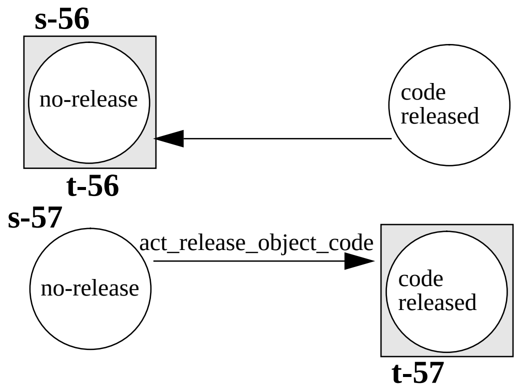


Figure 46. Int-release_object_code's subprocesses and traps with respect to Code

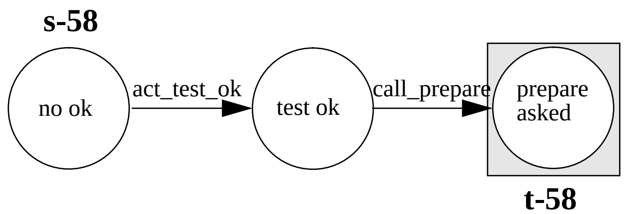


Figure 47. Int-test_ok's subprocesses and traps with respect to Code (other instance)

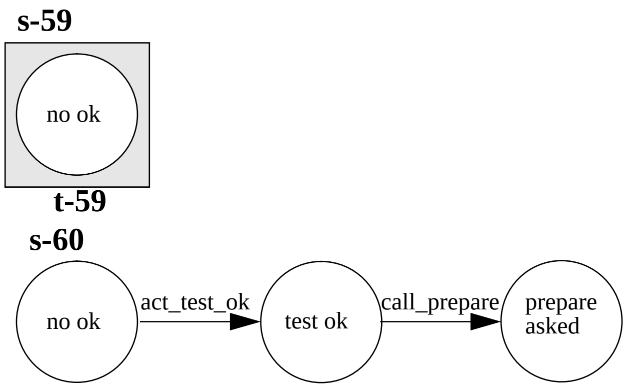


Figure 49. int-test_ok's subprocesses and traps with respect to Code (same instance)

Just like *Design*, one instance of *Code* exists for each version of a code document. The first version starts in the state marked with *** and the other versions start in the state marked with ****. These latter versions are waiting in the state **** until the previous version of *Code* enters trap t-58 of *int-test_ok*. Just like in the case of *Design*, the other traps labelling this tran-

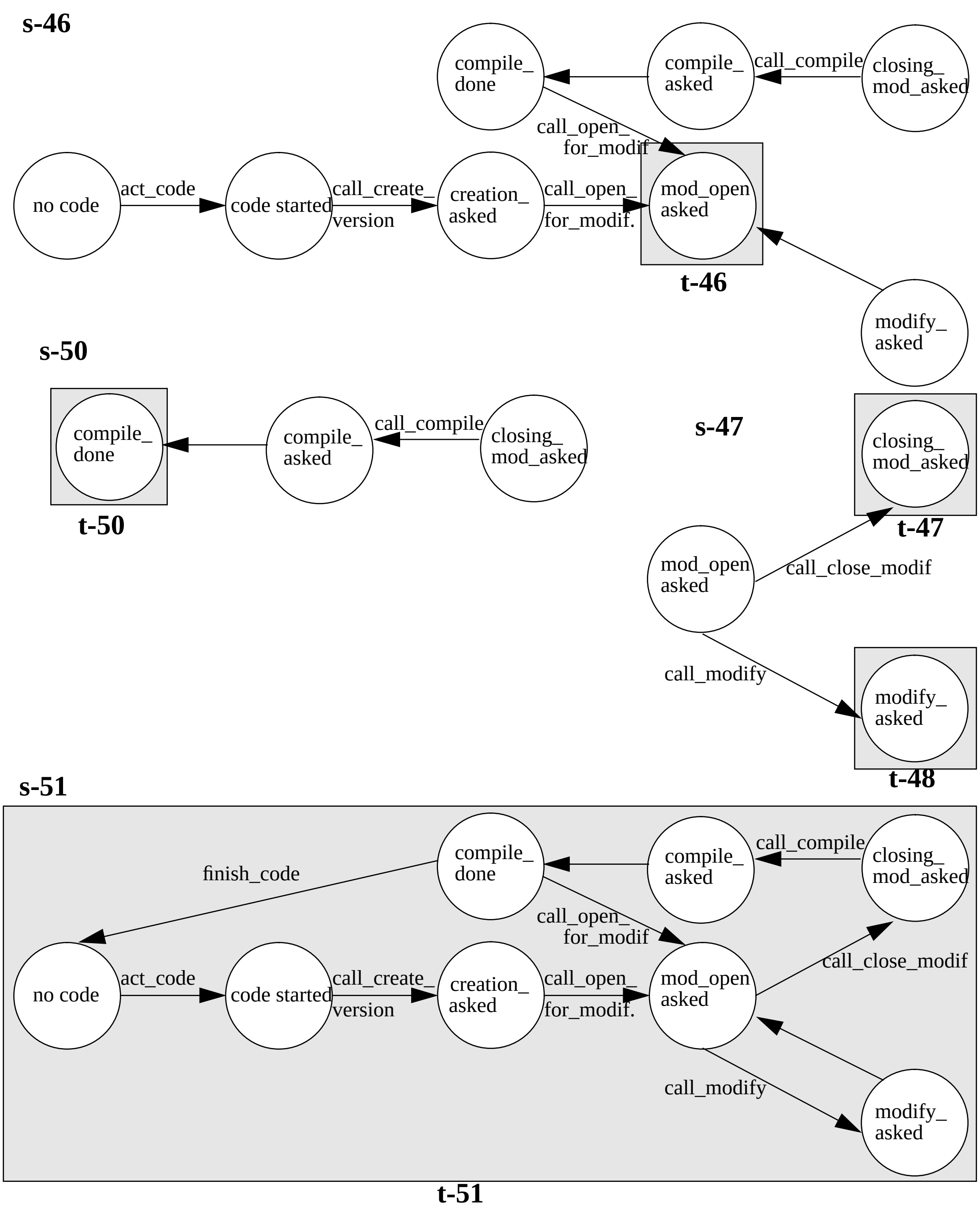


Figure 44. Int-code’s subprocesses and traps with respect to Code

sition from **** to *** do not really matter here, as they consist of the whole state space of the internal processes they belong to.

Since both creating a new version of a code document and modifying it, are modelled exactly like *Design* in the original example, this part of *Code* will not be discussed here. After the process of modifying the code document has been finished, *Code* will be in the state *pre-compile*, waiting for *int-compile* to enter trap t-52 (which means that a possibly existing previous compilation process has been finished). When this trap has been entered, *Code* will prescribe s-53 to *int-compile*, thereby releasing it to start compiling a document, and *Code* will keep *int-code* in s-50 until compiling the document has been finished. When *int-compile* enters its trap t-54 (*compile_not_ok*), *Code* will restart the modify code activity. However, when it enters trap t-53 (*compile_ok*), *Code* will remain prescribing s-50 to *int-code* to permit

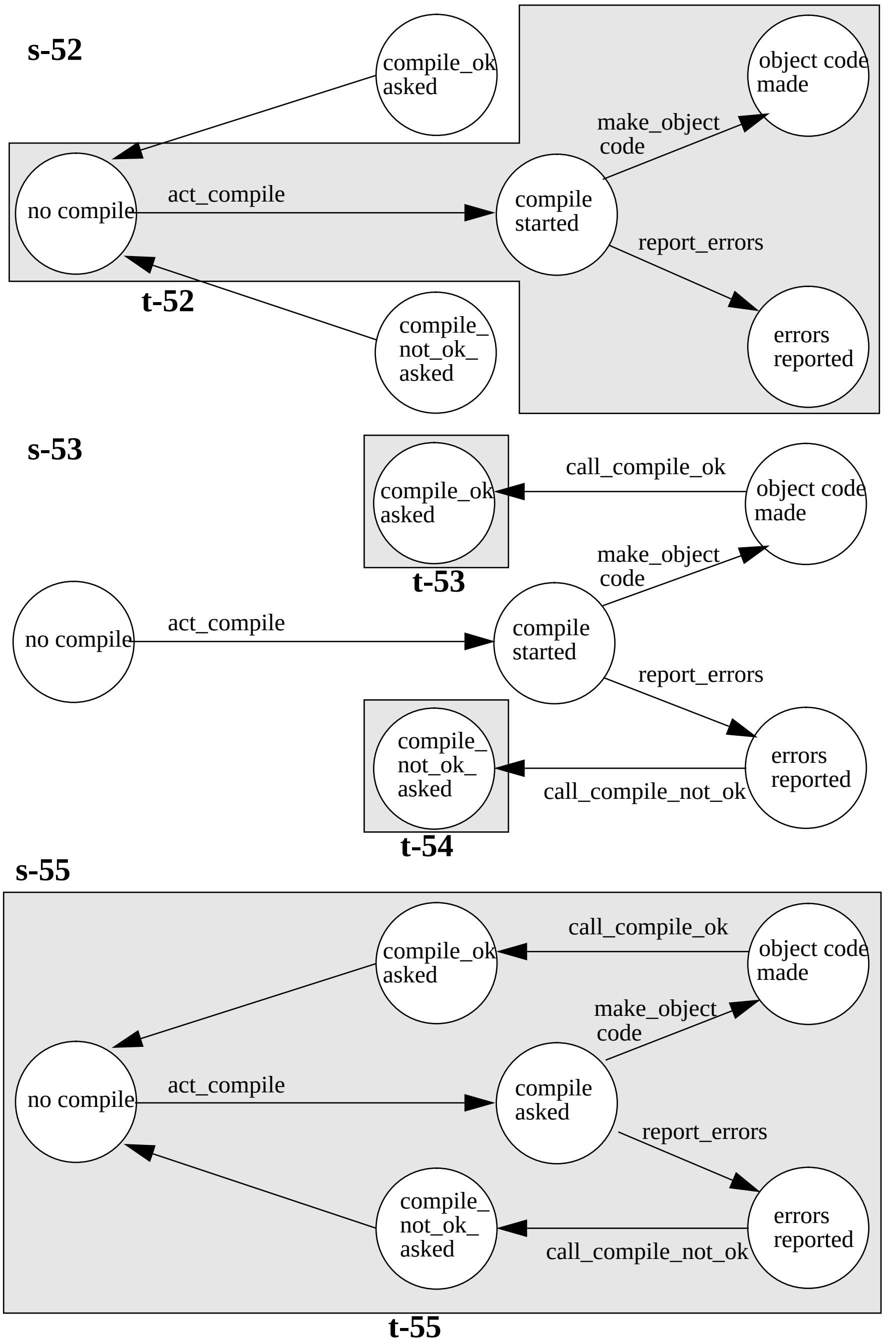


Figure 45. Int-compile's subprocesses and traps with respect to Code

it to leave the state *compile_asked* and enter the state *compile_done*. The manager will now be waiting until *int-compile* has reached its trap t-50, which means that the code has been compiled, and *int-release_object_code* has reached trap t-56, which means that the object code may be released. After prescribing s-57 to *int-release_object_code*, *Code* will wait until *int-release_object_code* has reached trap t-57, which means that the object code has been released. Note that *int-release_object_code* will wait in its first state as long as the design document has not been approved by *Design* (see the discussion of the communication between *Design* and its employees). After releasing the object code, the test round will be started until the internal behaviour of test either performs a *call_test_ok* or a *call_test_not_ok*. Since the behaviour of testing the code document has not been worked out here, no traps for these events are indicated in the figure. However, when such a call has been made, the manager has to continue in the appropriate way; when the test result is not ok, a new modify, compile, test

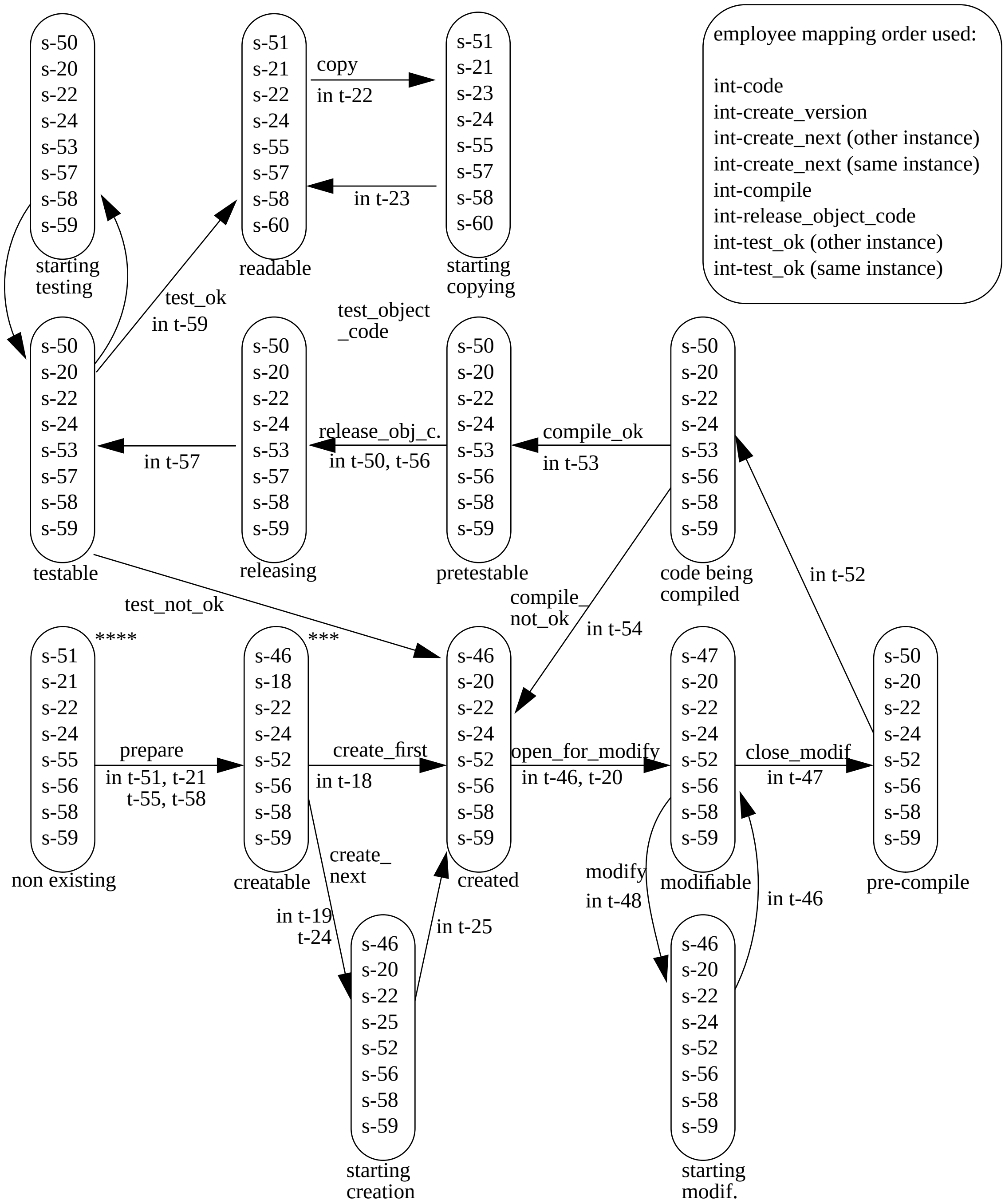


Figure 48. Code: manager of 8 employees.

round has to be started. Otherwise possible copy requests have to be handled. The copy request are modelled in the same way as in the model of *Design* in the original example.

The fourth part of the communication specification shows the communication between the manager *Compiler* (figure 33 and 50) and its employees *int-compile* (figure 36 and 51) and *int-code* (figure 34 and 52).

Compiler is waiting in its neutral state until *int-code* performs a *call_compile*, parametrized with a document name *name1*, by entering its trap t-63. As soon as *int-code* is in trap t-63 and *int-compile* is in trap t-61, *Compiler* will start compiling the document by entering its next state and therefore prescribing the subprocesses s-62 and s-64 to *int-compile* and *int-code* respectively. After *int-compile* has started, *Compiler* will wait until *int-code* has left trap t-63 and entered trap t-64. As can be seen in the previous paragraph about the communication

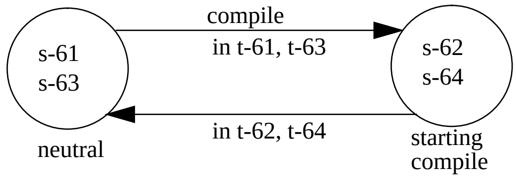


Figure 50. Compiler: manager of int-compile and int-code

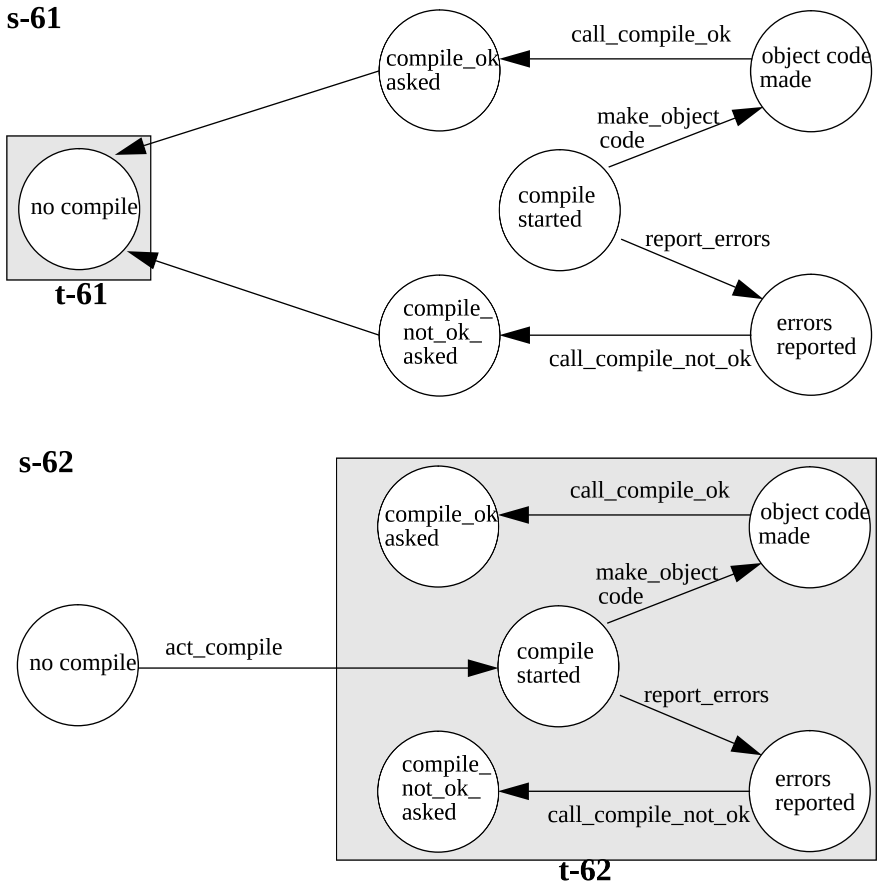


Figure 51. Int-compile's subprocesses and traps with respect to Compiler

between *Code* and *int-code*, this means that *Compiler* has to wait until the compilation has been finished, before it can return to its neutral state.

The fifth part of the communication specification shows the communication between the manager *ProjectManager* (figure 32 and 54) and its employees *int-monitor* (figure 15 and 55) and *int-schedule_and_assign_tasks* (figure 38 and 53).

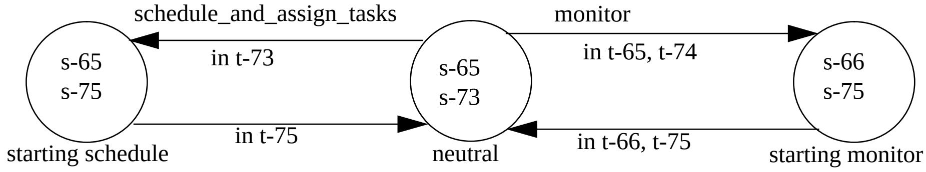


Figure 54. ProjectManager: manager of int-schedule_and_assign_tasks and int-monitor

The manager process *ProjectManager* starts in it neutral state, waiting until the Configuration Control Board (CCB) performs a call to *schedule_and_assign_tasks*. The CCB is the authority which, according to the ISPW-6 example, prompts for the required design and code modification. Its behaviour is outside the scope of this example.

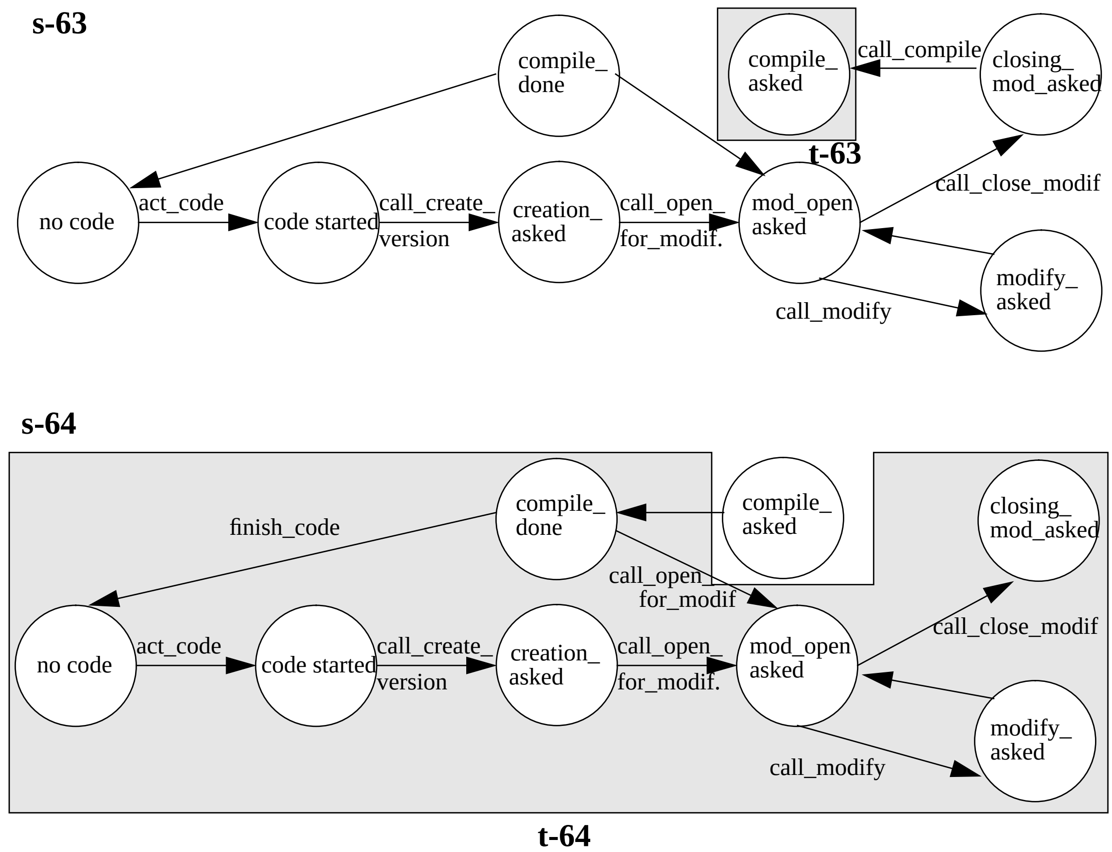


Figure 52. Int-code's subprocesses and traps with respect to Compiler

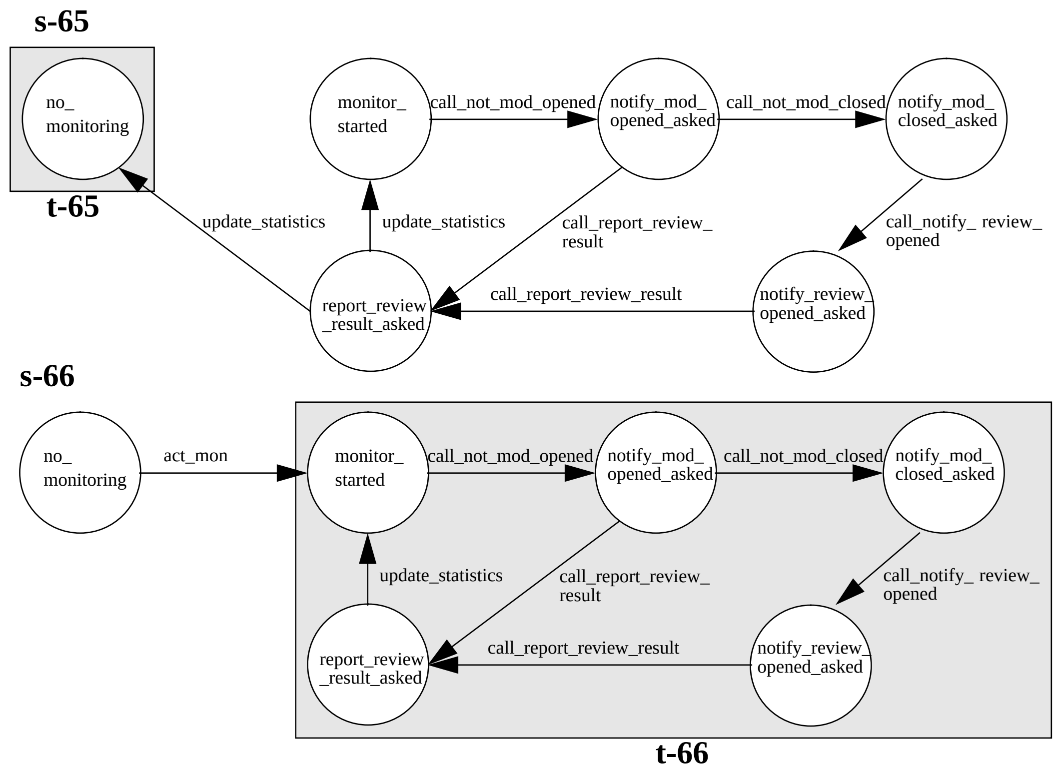


Figure 55. Int-monitor's subprocesses and traps with respect to ProjectManager

At the moment that the CCB performs the call to *schedule_and_assign_tasks*, the manager process *ProjectManager* will go to its state *starting schedule* thereby prescribing subprocess s-75 to *int-schedule_and_assign_tasks*. After a short while, *int-schedule_and_assign_tasks* will enter its trap t-75, thereby allowing *ProjectManager* to return to the neutral state. At some

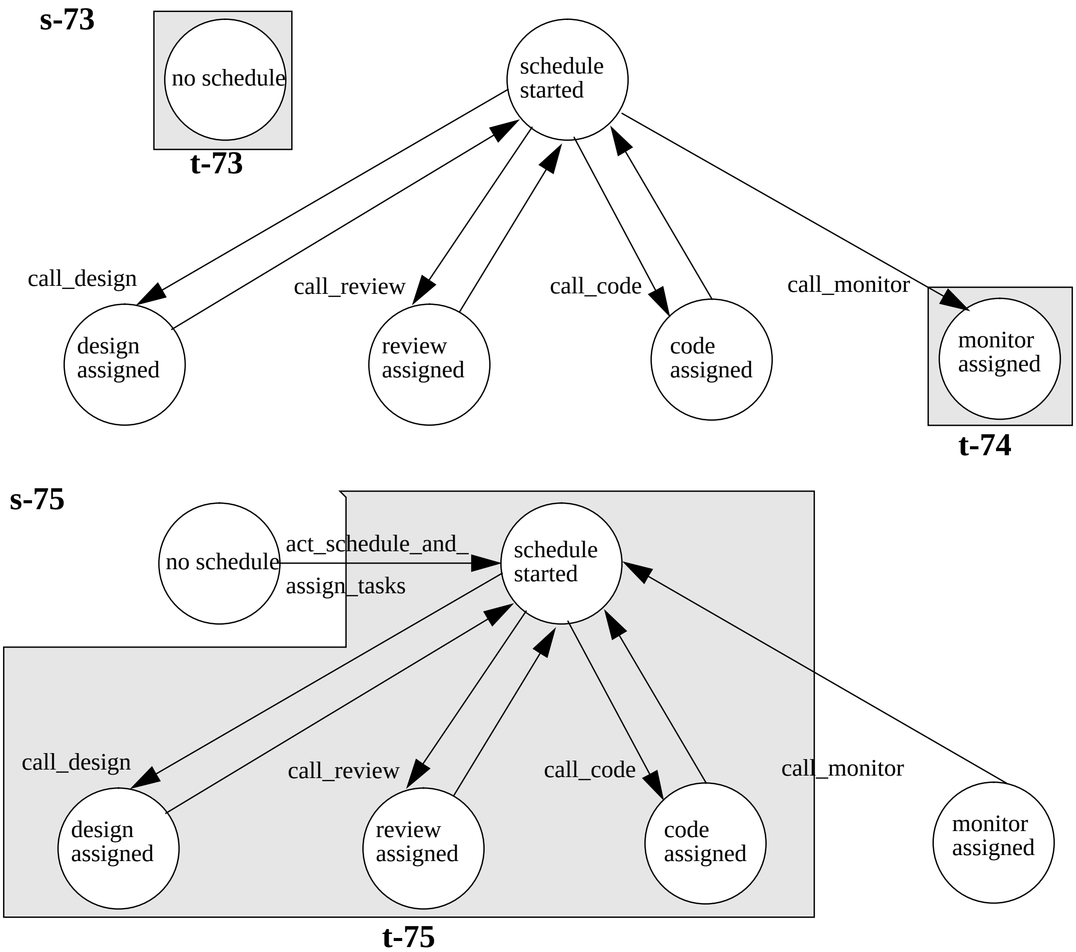


Figure 53. Int-schedule_and_assign_tasks's subp. and traps w.r.t. ProjectManager

given time instant, the internal behaviour of *schedule_and_assign_tasks* will perform a *call_monitor*, parametrized with a document name *name1*. When at that moment the monitor process which is parametrized with *name1* is in its trap t-65, *ProjectManager* can enter the state labelled *starting monitor*, starting the monitor by prescribing subprocess s-66 to *int-monitor* and allowing *int-schedule_and_assign_tasks* to return to its 'neutral' state *schedule_started*. Note that this is an example of one internal behaviour of an operation from an external process controlling the internal behaviour of another operation of that very same external process. This is in accordance with the original SOCCA approach. The advantage of showing the internal behaviour of an export operation which is only called from another internal behaviour of that same external process is that the internal behaviour of that operation becomes visible in the behaviour of the external process.

The sixth and last part of the communication specification shows the communication between the manager *Design* (figure 23 and 57) and its employees *int-design*, *int-review*, *int-create_version*, *int-create_next (other instance)*, *int-create_next (same instance)*, *int-review_ok (other instance)*, *int-review_ok (same instance)*, *int-monitor* (figure 15 and 16) and *int-release_object_code* (figure 35 and 56).

The behaviour of *Design* is being modified for the second time since the first model of it has been designed in [2]. The first modification was required for managing *int-monitor* (chapter 5) and the second time is the current modification, where *Design* has to notify *Code* that the design document has been approved. Because of these two modifications, a new informal comment on the behaviour of *Design* is necessary.

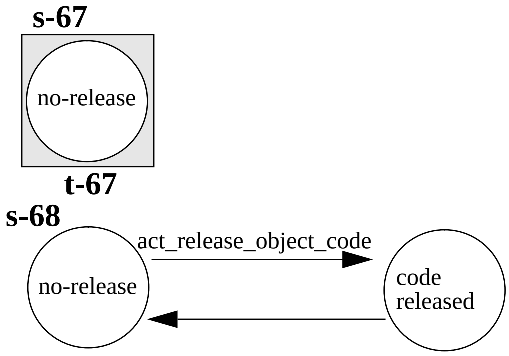


Figure 56. Int-release_object_code's subprocesses and traps with respect to Design

Just as before, every instance of *Design*, except for the first one which starts in the state labelled *, starts in the state labelled **, waiting for the previous instance of *Design* to enter trap t-26 of *int-review_ok*. After this, it will wait until *int-create_version* enters either trap t-18 or trap t-19 and it will continue with the old behaviour until *int-design* enters trap t-9 and *int-create_version* enters trap t-20. Now *Design* enters the state where it has to decide which route has to be taken: the route with a full notification of the monitor or the route with a partial notification of the monitor. As already has been mentioned in the previous chapter, the outcome of this choice depends on the size of the project. Let us assume that the size is large, so the longer route will be taken. In this case *Design* will wait until *int-monitor* enters trap t-31. After this it will prescribe subprocess s-32 to *int-monitor*, thereby forcing it to be notified of all events that will happen. It will now wait for *int-design* to enter trap t-10 to start modifying the document or to enter trap t-11 to close the modification. After trap t-11 has been entered, *Design* will wait again for *int-monitor*, but now for trap t-32 and after this it will wait until *int-review* wants to start a review round by entering trap t-13. When this has happened, *int-monitor* has to be notified again and the review round can really start. If the review of the document turns out to be not ok, a new modify and review round will start. Otherwise the final part of *Design* can be started to manage copying the document and to release a new version to be modified. However, before this end part starts, *Design* will check whether *int-release_object_code* is in trap t-67. Since this is the starting state of *int-release_object_code*, *Design* does not have not to wait at all, but it can prescribe subprocess s-68 to *int-release_object_code* immediately, thereby allowing *Code* to continue after releasing the object code document.

Note that *Design* has no explicit export operation for this notification that the design document has been approved to *Code*. This is because of the fact that the trap in which *int-release_object_code* is waiting, is the first state of it and therefore no transition that can perform a *call_export_operation* does exist. This way to start the behaviour of *int-release_object_code* can be compared with the starting of an internal operation by an external process in the normal way. From this point of view one could say that *int-release_object_code* is merely an internal operation of *Design* and not of *Code*. The fact that *Design* does not have such an explicit export operation is a deviation from the original SOCCA approach. When the original SOCCA approach had been followed, the communication between *Code* and *Design* should have been modelled by means of an explicit export operation of *Design* which would have been called from within the internal behaviour of one of the export operations of *Code*.

6.5. Concluding remarks

In [2], the SOCCA approach has been defined. This SOCCA approach can be split up into two parts:

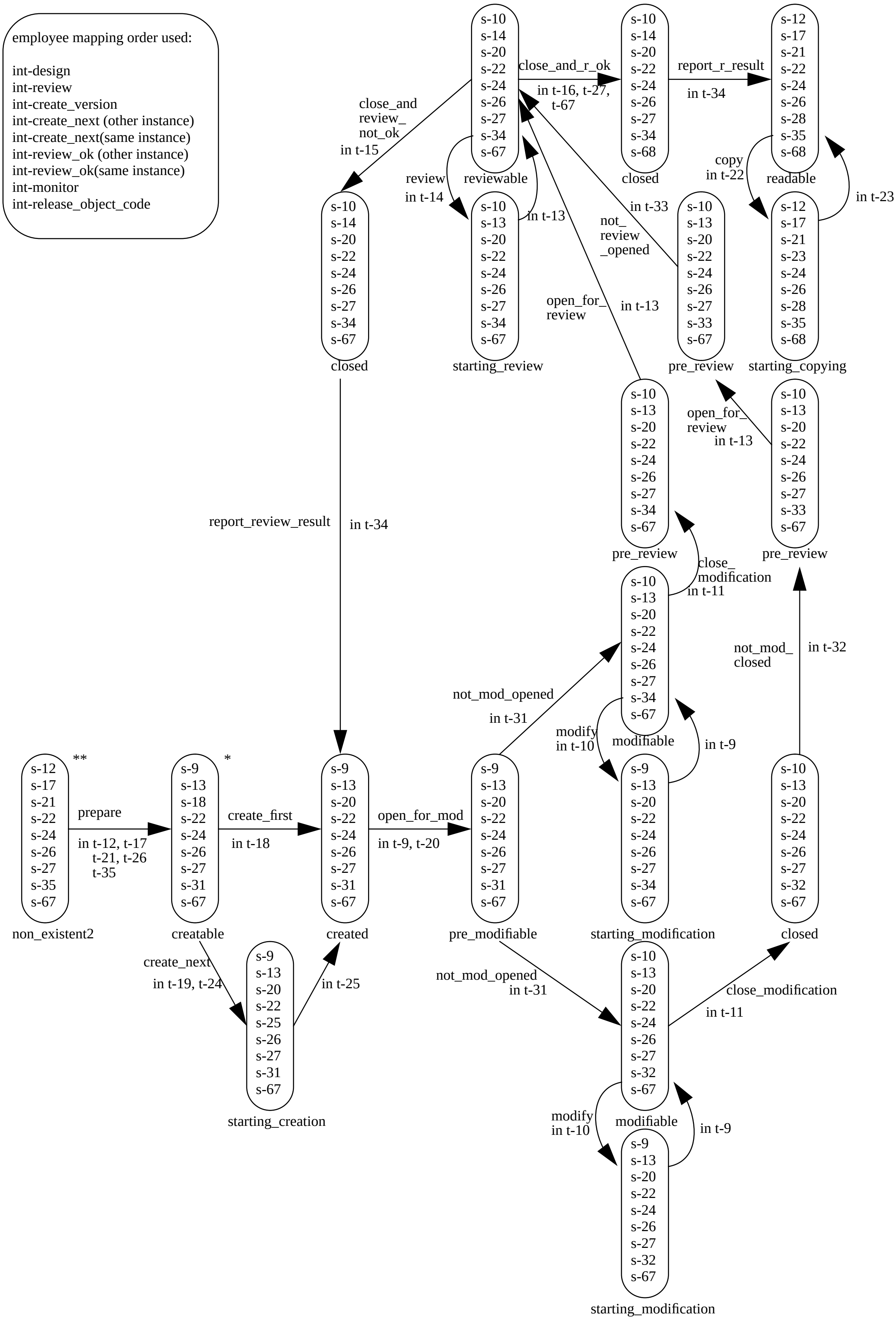


Figure 57. Design: manager of 9 employees

- The SOCCA rules: these rules state that the external behaviours of the classes and the internal behaviours of the export operations of these classes are modelled with STD's and that the communication between the various processes is modelled with PARADIGM on top of them. In the PARADIGM part are the manager processes formed by the external behaviours and the employee processes are formed by the internal behaviours.

- The SOCCA conventions: these conventions are implied by the techniques which have been used in the example in that paper. The following conventions can be identified:

  C1: Let all internal behaviours, denoted with “int-behave”, start with a transition labelled “act-behave” leading from the first state of the STD to the second state of the STD. The external behaviour of the class which exports the operation “behave” keeps the internal behaviour “int-behave” trapped in the first state until the operation has to be started. As soon as the external behaviour arrives in the state after the transition corresponding with starting that internal behaviour, it will prescribe a new subprocess to the internal behaviour, forcing it to leave its first state according to the transition “act-behave”.

  C2: Export operations of a class are called from the internal behaviours of operations from other classes or at least from the internal behaviour of an operation from another instance of the same class. Thus, export operations of one particular object are imported into the internal behaviours of operations from a really different object.

In this chapter, two deviations from these SOCCA conventions have been made. These deviations are not breaking with the formal SOCCA modelling rules as defined in [2], they are only breaking with the conventions as implied by the original SOCCA example in that paper.

The first deviation is a deviation from convention C1: the first state of *int-release_object_code*, which is an export operation of *Code*, is not only a trap in the trap-structure with respect to *Code*, but it is also a trap in the trap-structure with respect to *Design*. Thus, *int-release_object_code* has two managers which keep *int-release_object_code* trapped in its first state in stead of only one manager. This has been done because *Code* has to wait at that moment until the design document has been approved.

This deviation could have been avoided by changing the behaviour of *Design*; give *Design* an export operation called something like *design_doc_approved* and give *Code* a state in which it waits until *design_doc_approved* arrives in a state notifying that the design document has been approved.

The main difference between both methods is that the first method models a direct coordination between *Design* and *Code* via a PARADIGM communication between the behaviour of *int-release_object_code* and the behaviour of *Design* which is not made visible with an explicit export operation in the class diagrams. This in contrast to the second method where the object oriented approach is used to make the PARADIGM communication between *Design* and *Code* visible in the class diagrams.

The second deviation is a deviation from convention C2: in this example, the export operation *monitor* of *ProjectManager* is imported in the internal behaviour of *assign_and_schedule_tasks* of the very same instance of the external process *ProjectManager*. This means that the export operation *monitor* is no real export operation of *ProjectManager*; it is not imported into any other class or instance. From the EER point of view, one could say that *monitor* is an internal operation of *ProjectManager* which should not be visible from outside. However, by making *monitor* an export operation of *ProjectManager*, it is made visible from the external behaviour of *ProjectManager* whenever a new instance of *int-monitor* is started by *ProjectManager*.

Chapter 7
Example 2: changing the model according to the ISPW-7 specification

The ISPW-7 extensions address two issues: teamwork and process change. As within the context of this thesis we are particularly interested in dynamic process modification, we will only concentrate on the process change part of these extensions. The process change extensions have been split up into two parts. One concerns process modification; this is a permanent change to the model. The other part concerns process exception which is a temporary change of the process to handle exceptional circumstances. After such a temporary change, the model has to return to its original behaviour. These two parts will be discussed in the following sections.

7.1. Process modification: problem description

The proposed change in the ISPW-7 example is as follows; in the original ISPW-6 example, coding could start **before** the design document was approved. From now on this restriction is tightened: coding may only start **after** the design document has been approved. This change of the model has to be applied to some process which is being enacted and thereby three cases should be considered (quoted from ISPW-7 example, section 4.2.1):

- ‘The executing process has not yet reached the step affected by the change, so the change will have no immediate impact on the process state’.
- ‘The executing process has reached or passed the steps affected by the change, but for whatever reason, the change is consistent with the existing process state’.
- ‘The executing process has reached or passed the steps affected by the change, and the change is inconsistent with the existing process state’.

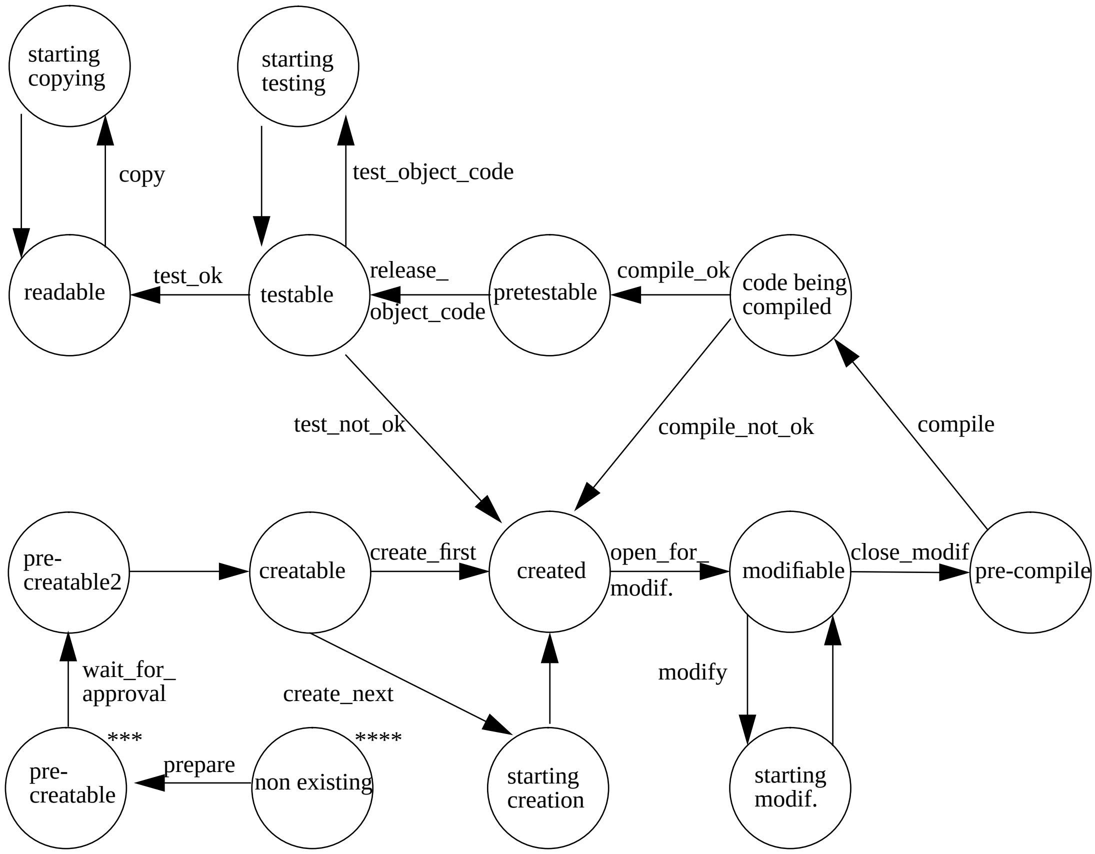
In the following section first a new model for the external behaviour of *Code* will be given and the internal behaviours of the operations will be modified as far as needed. This new model will be referred to as the *ISPW-7 model* or the *ISPW-7 case*. Likewise, the current SOCCA model will be referred to as the *ISPW-6 model* or the *ISPW-6 case*. The ISPW-6 model and parts of it will also be referred to as the *old model* and the ISPW-7 model will sometimes be called the *new model*.

Section 7.3 concentrates on making the transition from the ISPW-6 model to the ISPW-7 model. In that section, the three cases mentioned above will also be taken in consideration and they will be related with the problems as mentioned in section 4.3. The solution to the third problem will be derived from the solutions suggested in section 4.4. When necessary, new (temporary) processes will be designed and the cooperation between *Code* and other parts of the model will possibly also be changed.

7.2. Designing the new model

In the ISPW-6 model, *Code* waits with releasing the object code to the test phase by means of the internal behaviour of *int-release_object_code*: as long as the design document has not been approved by *Design*, *int-release_object_code* can not enter trap t-57 and *Code* will have to wait in the state labelled *releasing*.

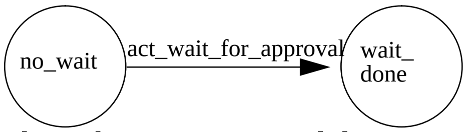
According to the ISPW-7 specifications, *Code* may only start after the design document has been approved. This can be managed by designing a new model. In this new model, *Code* will get an extra export operation called *wait_for_approval* to model waiting for the design document to be approved. In the STD of the external behaviour, the transition corresponding to this operation will be placed before the state labelled *creatable*. The new model of *Code* is displayed in figure 58.



W.r.t. to WODAN is this subprocess s-83 and the state space is trap t-83

Figure 58. Code: new STD of the external behaviour

Note that in this case not only the dynamic description changes but also the static description; the class *Code* gets the new export operation *wait_for_approval*. This new class description will also be defined by WODAN, in exactly the same manner as WODAN defines the new behaviour description. The internal behaviour of *wait_for_approval* is shown in figure 59.



W.r.t. to WODAN is this subprocess s-85 and the state space is trap t-85

Figure 59. Int-wait_for_approval: STD of its internal behaviour

Note also that the state labelled with *creatable* in the old STD of *Code* corresponds to the state labelled with *pre-creatable* in the new STD of *Code*. This means that when the transition of the old STD of *Code* to the new STD of *Code* is made and the process is in the state *creatable* in the STD of *Code* before switching to the new STD, that the process after this transition will be in the state *pre-creatable* in the new STD.

After these modifications of the external and internal behaviours, the new communication has to be specified. In the figures specifying the communication, only the communication between the relevant managers and *int-release_object_code* and *int-wait_for_approval* will be given. The other parts remain the same as in the figures modelling the ISPW-6 case.

First the communication between the manager *Code* (figure 58 and 61) and its employee *int-wait_for_approval* (figure 59 and 60) will be given.

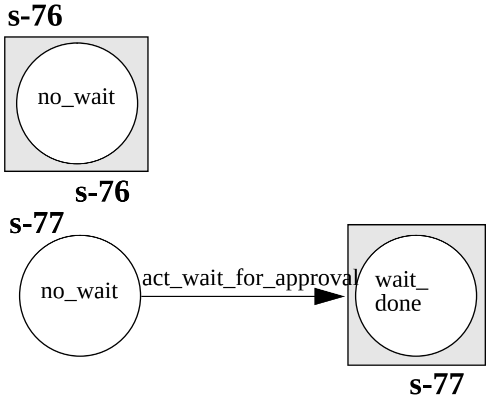


Figure 60. Int-wait_for_approval's subprocesses and traps w.r.t. Code

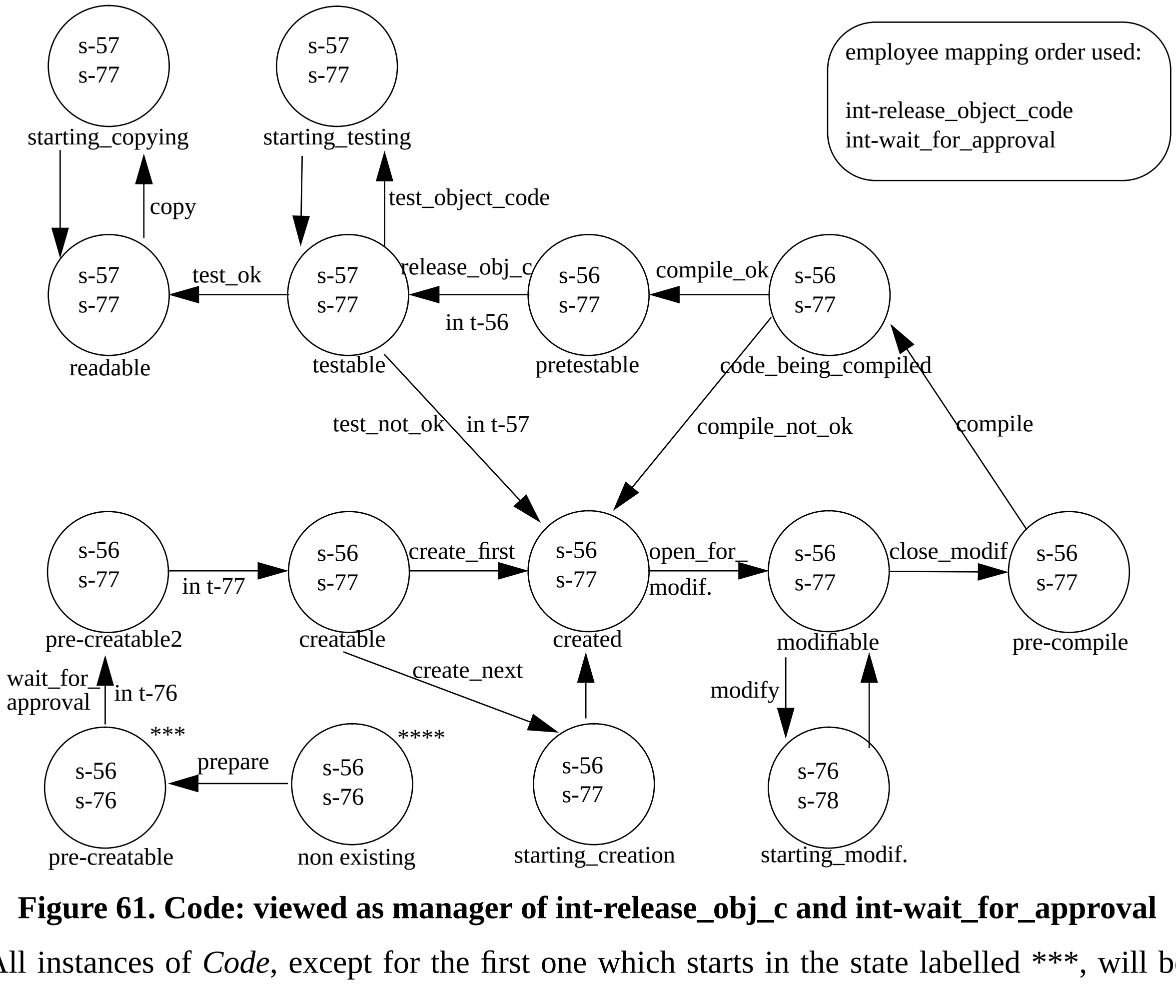


Figure 61. Code: viewed as manager of int-release_obj_c and int-wait_for_approval

All instances of *Code*, except for the first one which starts in the state labelled ***, will be waiting in the stated labelled **** until the previous version of *Code* performs a *call_prepare*. At that moment however, the creation of the new version of a code document will not be started immediately. In stead of this, *Code* will wait in the state *pre-creatable2* until *int-wait_for_approval* has entered trap t-77, which means that the design document has been approved. From that moment on the behaviour will remain almost the same as described in the explanation of the ISPW-6 case; the only other difference is that with this new version of *Code* it is no longer necessary to wait until the design document has been approved before releasing the object code document (since the design document has already been approved).

However, this difference is not visible in the model of *int-release_object_code* or in the communication between *Code* and *int-release_object_code*. It is only visible in the new communication between *Design* and *int-release_object_code; int-release_object_code* is no longer an employee of *Design*, thus *Design* can no longer keep *int-release_object_code* trapped in its first state. Instead, *Design* is now a manager of *int-wait_for_approval*. This new communication structure is shown in figure 62 and in figure 63.

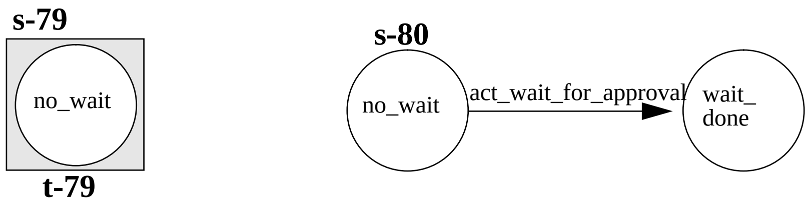


Figure 62. Int-wait_for_approval's subprocesses and traps w.r.t. Design

Except for the fact that *Design* is no longer a manager of *int-release_object_code* but of *int-wait_for_approval* instead, no part of the behaviour of *Design* has been changed. Since from the viewpoint of *Design*, the function of this new employee is the same as the function of the replaced employee, the global behaviour of *Design* does not change at all; it still keeps this employee trapped in the first state until the design document has been approved and after this event, *Design* still allows this employee to move freely through its state space.

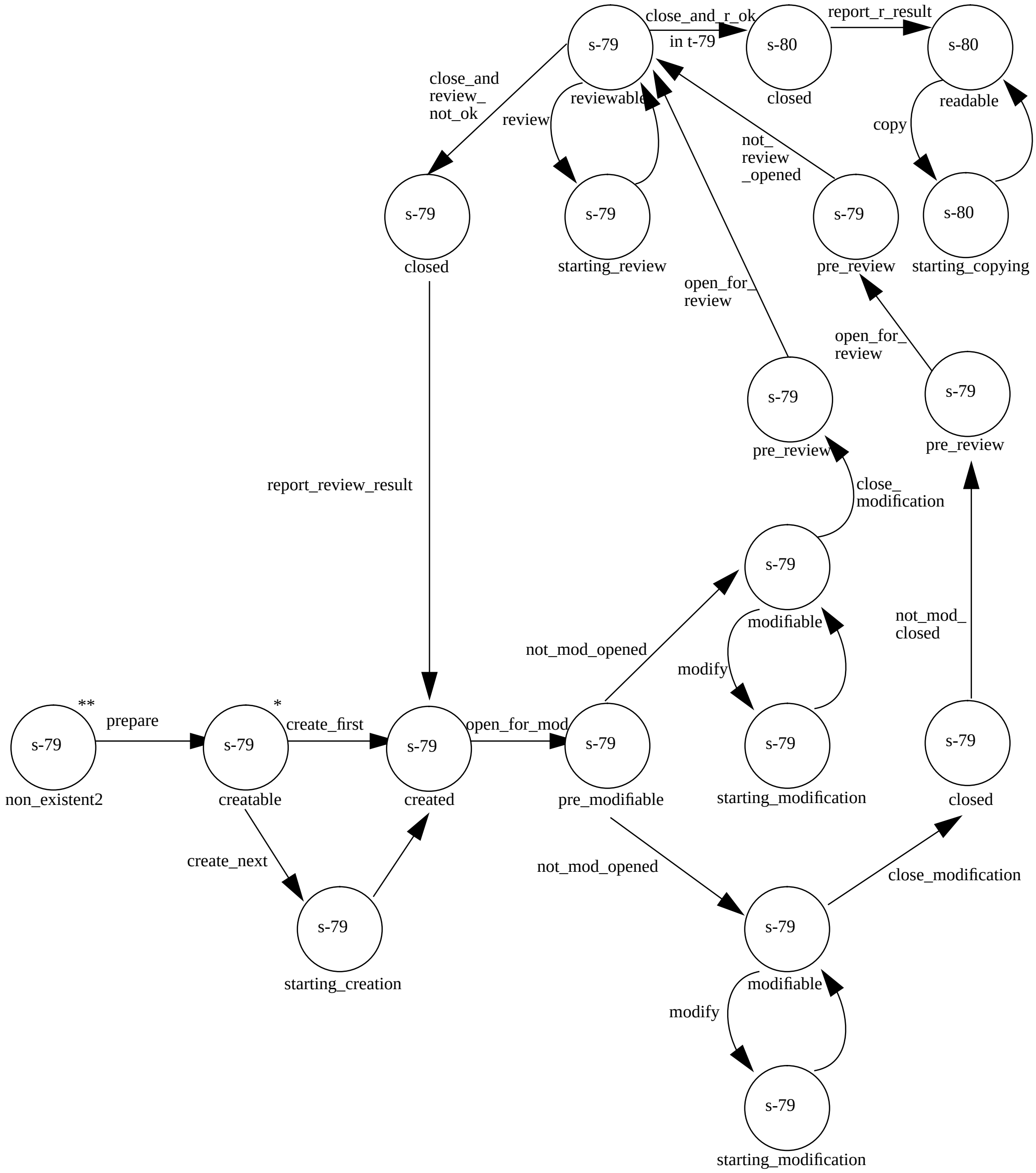
7.3. Introducing the ISPW-7 model.

When this model is introduced during the enactment of a software process model, a problem may arise; since in the ISPW-6 case *Code* could start before the design document was approved, *int-release_object_code* was an employee of *Design* to make it possible to wait for the approval of the design document. In the ISPW-7 case however, *int-release_object_code* no longer waits for the design document to be approved, as *Code* waits with the aid of *int-wait_for_approval* for the design document to be approved. However, when switching from the ISPW-6 case to the ISPW-7 case, *Code* may have past the transition labelled *wait_for_approval* while the design document has not yet been approved. This is problem P3 as defined in section 4.3. This problem corresponds with the inconsistency mentioned as the third case in section 7.1.

In the ISPW-7 specification two options to solve this inconsistency have been suggested: '(1) you may allow the inconsistency to remain, or (2) you may attempt to change the state (by rollback, or whatever) to achieve consistency'. In the following sections, these two solutions will be followed. As it turns out, these solutions are equivalent with some of the solutions mentioned in section 4.4.3

7.3.1. Option 1: do not solve the inconsistency

In this section option 1 will be worked out. For this purpose an intermediate phase of *Code* will be given, which still has the structure of the ISPW-6 model. By choosing the traps from this STD to the STD of the ISPW-7 model in the right manner, it is possible to switch only from the ISPW-6 case to the ISPW-7 case before modify code has started (this situation is



W.r.t. to WODAN is this subprocess s-86 and the state space is trap t-86

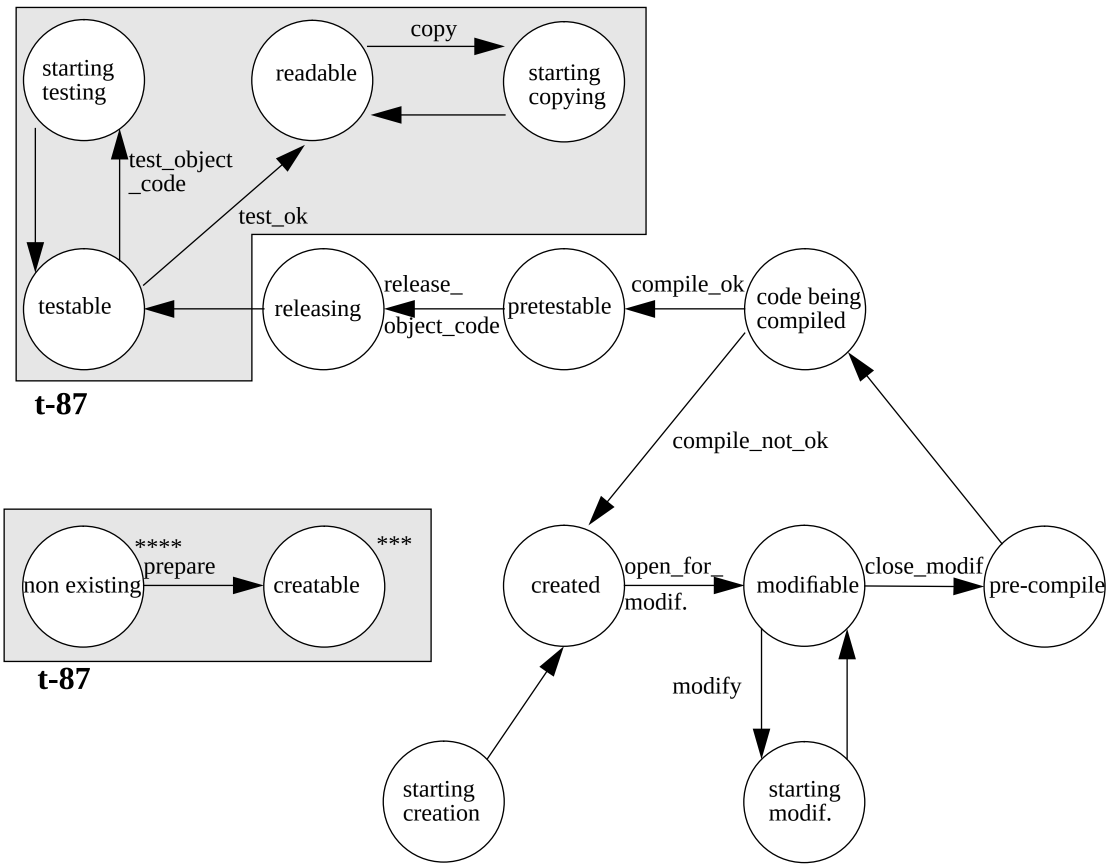
Figure 63. Design: viewed as manager of int-wait_for_approval

equivalent with the first case mentioned in section 7.1) or after modify code has waited until the design document has been approved (this situation is equivalent with the second case mentioned in section 7.1). It is straightforward to incorporate these two cases into the SOCCA model.

As *Code* remains behaving like in the ISPW-6 case in the middle part of this intermediate phase, the inconsistency as mentioned in the third case in section 7.1, will remain in this middle part. As soon as the intermediate phase of *Code* has left the middle part, it will have waited for the design document to be approved and the inconsistency does not arise at all.

This solution to the problem is equivalent with solution S6 from section 4.4.3.

The intermediate phase of *Code* is given in figure 64. Note that this intermediate phase has the same state-action interpreter as the original phase of *Code*. Therefore, the state-action interpreter has not been shown in figure 64.



W.r.t. to WODAN is this subprocess s-87

Figure 64. Intermediate phase of Code

The last step to finish the transition from the ISPW-6 case to the ISPW-7 case is designing WODAN to manage this change. As stated in solution S6 in section 4.4.3, WODAN first has to prescribe the intermediate phase of *Code* and as soon as this intermediate phase has entered its trap, WODAN can prescribe the ISPW-7 model. Note that in the state-action interpreter of WODAN, only the employees of which the STD changes during the transition from the ISPW-6 case to the ISPW-7 case, have been mentioned. WODAN is shown in figure 65.

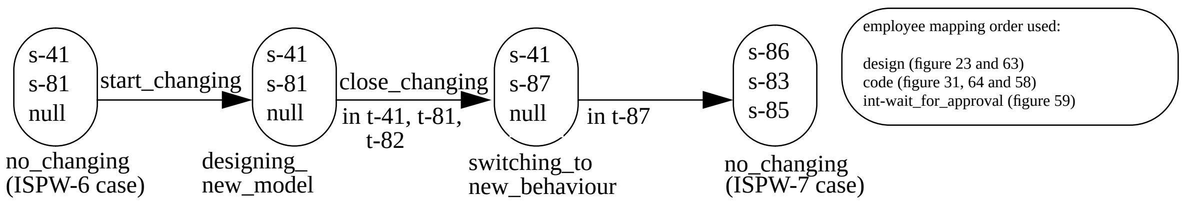


Figure 65. WODAN: viewed as manager of 3 employees

In this example three important notions can be found:

- The only part of *Design* that changes when switching from the ISPW-6 case to the ISPW-7 case, is the state-action interpreter. The states, transitions, strategy and all other parts defining a process remain the same. Since *Design* is a subprocess of an anachronistic process, this means that a subprocess restriction not only defines a restriction on the states, transitions, etc., as defined in [1] but that it also defines a restriction on the state-action interpreter; the anachronistic process of which *Design* is a subprocess, has a not explicitly designed state-action interpreter which minimal consists of the union of the state-action interpreters from the subprocesses s-41 and s-86.
- The subprocess *int-wait_for_approval* is introduced for the first time in the ISPW-7 case.

Before that time it did not exist. Introducing this new process has been done by prescribing the NULL process, defined in section 3.4, in those state where *int-wait_for_approval* did not yet exist and by prescribing *int-wait_for_approval* as soon as the ISPW-7 case becomes active.

- WODAN is not only able to change the dynamic model description part but it can also change the static model description part; in this example the class description of *Code* has been extended with a new export operation called *wait_for_approval*.

7.3.2. Option 2: solve the inconsistency

In this section a solution to the inconsistency mentioned in the third case in section 7.1 will be given. When *Code* already has started before the design document is approved, *Code* will go back to its starting state, thereby discarding all changes made to the code.

Since some employees of *Code* have side effects, these side effects also have to be rolled back. Therefore the behaviour of all employees has to be studied to find out how the side effects of their behaviour can be discarded. Some employees may need extra states and/or transitions to rollback the side effects. If so, subprocesses representing this temporal behaviour will be defined.

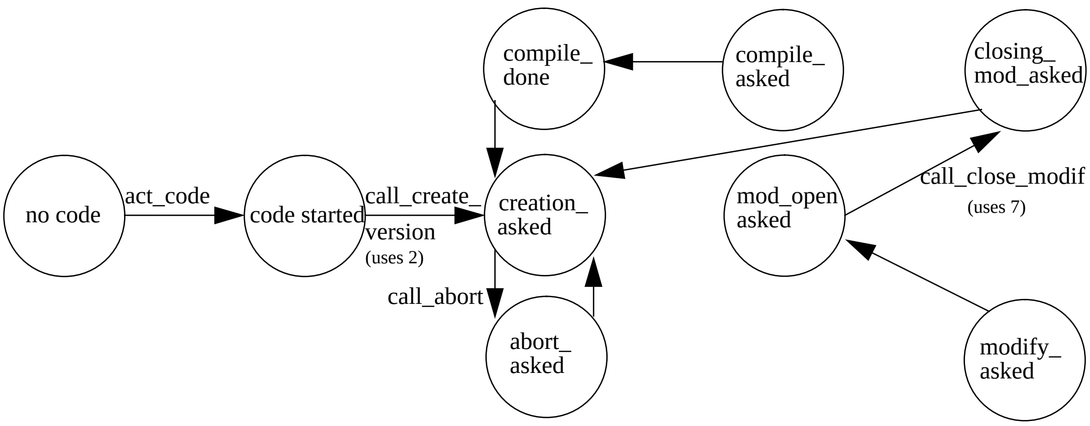
Code needs an intermediate STD for managing the process of rolling back all side effects. This intermediate STD of *Code*, which is displayed in figure 67, will be called *TempCode*, since it is valid only temporary.

This solution will follow the procedure as sketched in solution S8 of section 4.4.3.

Code has the following employees: *int-create_version*, *int-create_next*, *int-code*, *int-compile*, *int-release_object_code* and *int-test_ok*. We will study each employee to determine whether the behaviour of that employee has to be rolled back or not.

As creating the new version of a code document does not really change the code document - it only changes the version number-, the behaviour of the employees *int-create_version* and *int-create_next* does not have to be rolled back.

The employee *int-code* really modifies the code document. Thus, this employee needs an intermediate phase to rollback its side effects when the inconsistency arises. This intermediate phase, which is shown in figure 66, calls *TempCode's* export operation *abort*, which aborts the



W.r.t. to WODAN is this subprocess s-88 and the state space is trap t-88

Figure 66. Intermediate phase of int-code

modify code activity and discards all modifications made to the code document. This means that after execution of the abort operation it seems as if no modification to the code document has been made.

Note that the temporary export operation *abort* also has to be placed in the class description of *TempCode*.

Note also that *int-code* has been designed in such a manner that, when the code document has been opened for modifications, modify code has to be closed before calling abort. This is necessary to assure that only one version of the code document is opened at the same time.

Int-compile reflects the behaviour of an automated tool which can not be modified and thus the behaviour of *Int-compile* can not be modified or rolled back.

WODAN (see figure 71) is designed in such a manner that *TempCode* will only be prescribed when the design document has not yet been approved. If the design document already has been approved before the ISPW-7 case has to be started, the process will be in case 2 of section 7.1 and no inconsistency will arise. Thus, the code document may not be release when *TempCode* is prescribed. Therefore is the transition labelled *release_object_code* removed from *TempCode*. So it is not necessary to rollback the behaviour of *int-release_object_code*.

Note that, as *int-release_object_code* is removed from the behaviour of *TempCode*, WODAN prescribes the NULL process for *int-release_object_code* during the intermediate phase.

The operation *int-test_ok* can only be called when the design document has already been approved, thus no rollback of this operation is necessary.

The communication structure to solve the inconsistency is shown in figure 68 (*int-code* as employee of *TempCode*), figure 67 (two temporary phases of *TempCode* as manager of *int-code* and as employee of WODAN), figure 69 and 70 (two temporary phases of *Design* as employee of WODAN) and figure 71 (WODAN as manager of *Design*, *Code*, *int-code*, *int-*

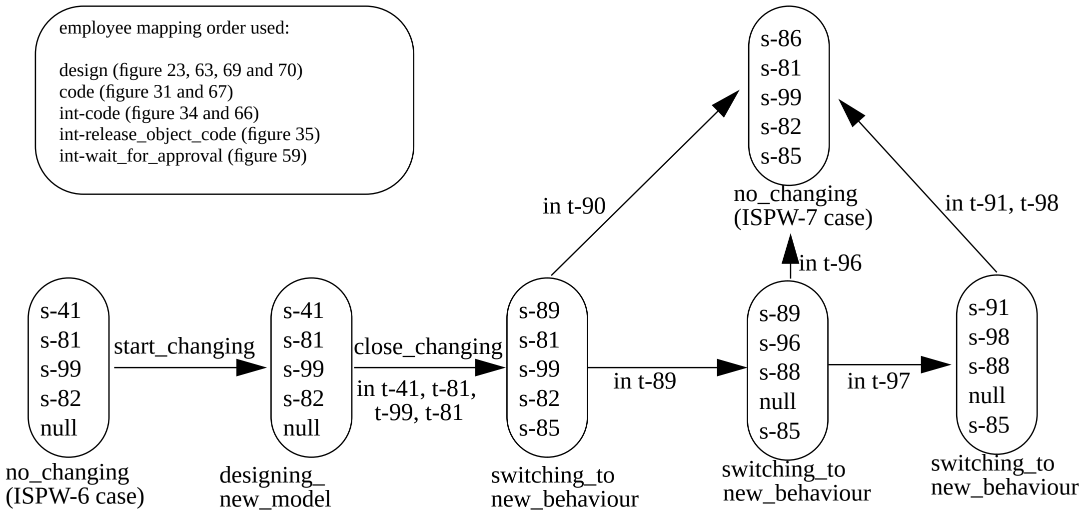
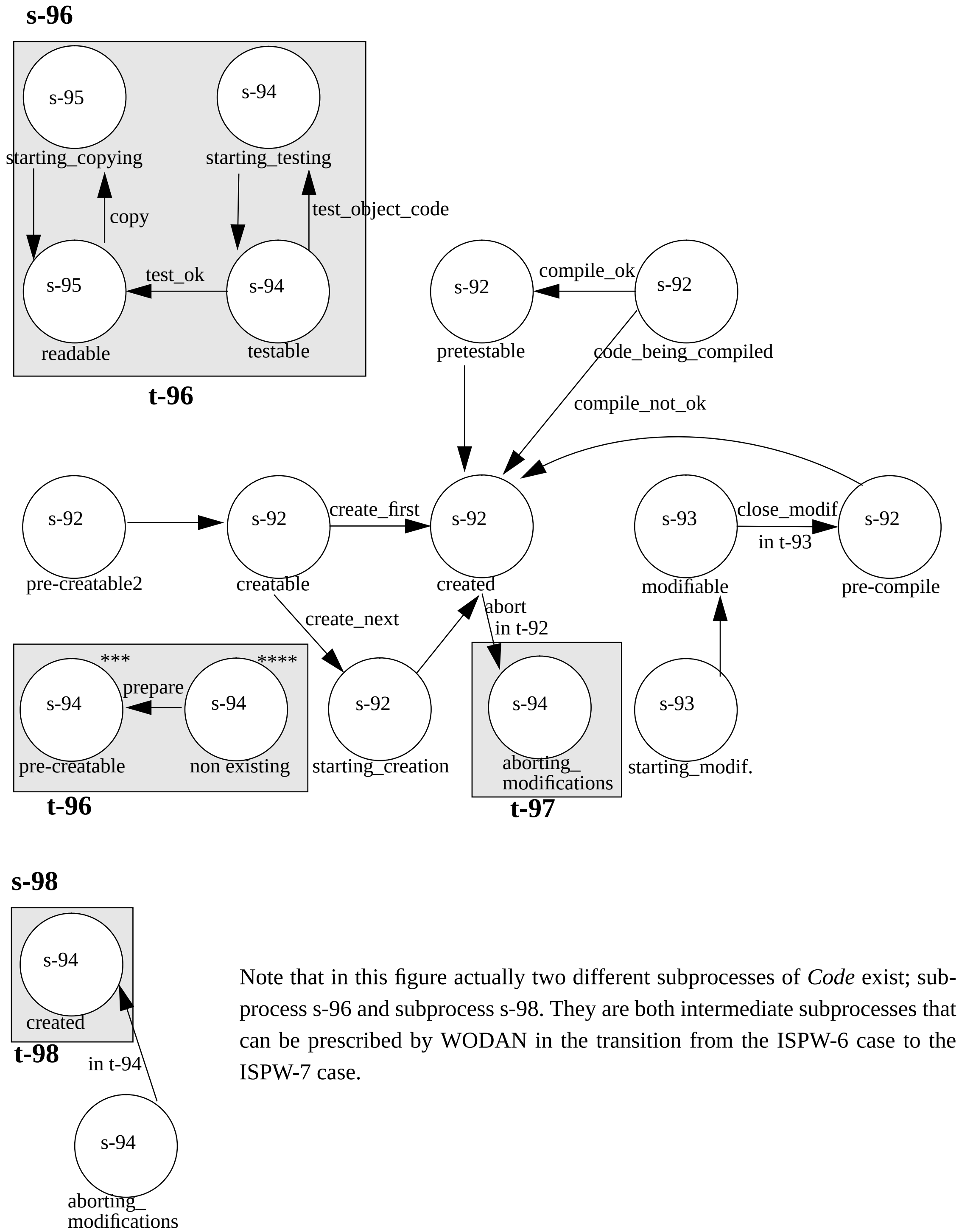


Figure 71. WODAN: viewed as manager of 5 employees

release_object_code and *int-wait_for_approval*).

After designing the new model,. WODAN prescribes the intermediate phase to *Design* to determine what action has to be taken; when *Design* is in trap t-90, the design document will have been approved, which means that no inconsistency arises. Thus the transition to the ISPW-7 case can be finished in that case. Otherwise, the design document has not been approved and therefore it is necessary to abort the modify code process when it has already been started. Thus, WODAN then starts the intermediate phase of the employees *Code* and *int-code* and WODAN prescribes the NULL process to *int-release_object_code*. To determine whether the modify code activity has started or not, WODAN examines the trap of *TempCode*. When *TempCode* is in trap t-96, the modify code activity has not been started, thus the transition to the ISPW-7 model can be finished in that case. Otherwise, WODAN has to wait until



Note that in this figure actually two different subprocesses of *Code* exist; subprocess s-96 and subprocess s-98. They are both intermediate subprocesses that can be prescribed by WODAN in the transition from the ISPW-6 case to the ISPW-7 case.

Figure 67. TempCode: viewed as manager of int-code

the modify code operation is aborted. This operation is aborted only after *TempCode* has entered trap t-97; *TempCode* enters this trap after *int-code* has reached trap t-92, which is entered by *int-code* after calling the *abort* operation.

Note that *int-code* first closes the modify code document when it already has been opened, this is modelled by means of *int-codes* trap t-93 and *TempCode*'s transition from the state modifiable to the state pre-compile.

Further note that *TempCode* has all states and transitions which are necessary for the communication with *int-compile*. These states and transitions are needed as the communication between (*Temp)Code* and *int-compile* does not change during the intermediate phase.

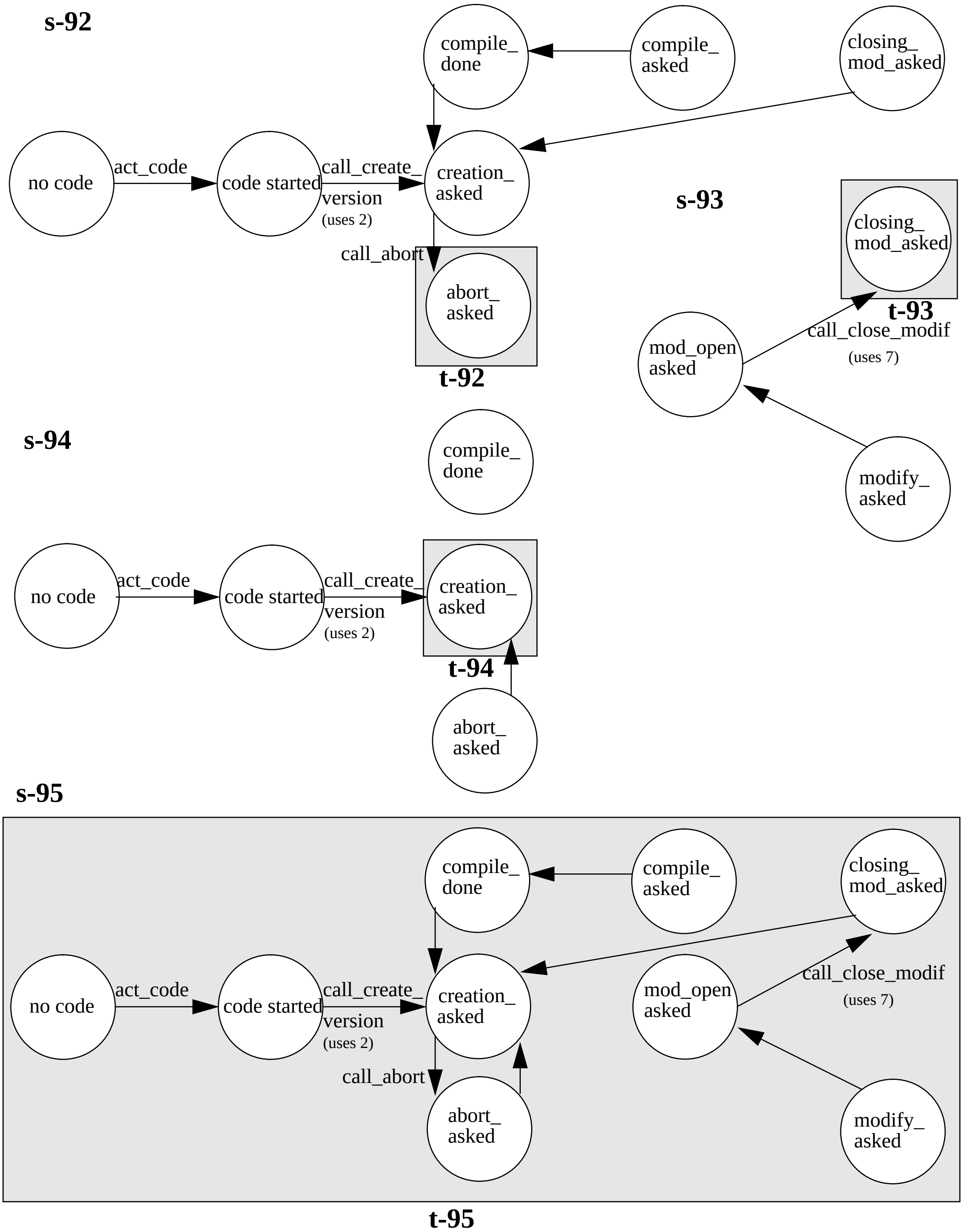
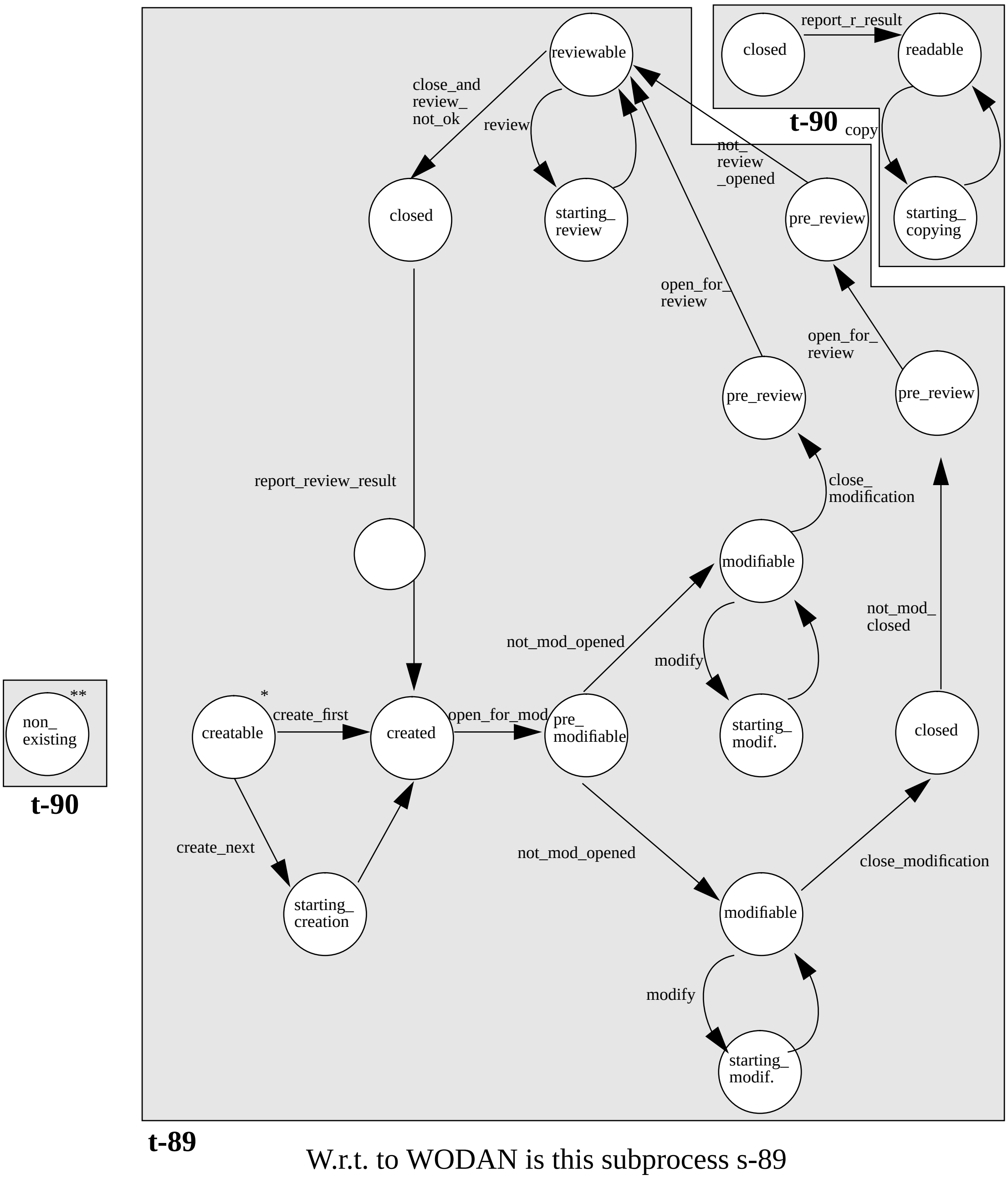


Figure 68. Int-code's subprocesses and traps with respect to TempCode

When *TempCode* has finally entered trap t-97, WODAN will prescribe s-98 to *TempCode* and s-91 to *Design* and then WODAN will wait until the design document has been approved (trap t-91), the abort operation has been finished (trap t-98) and, via the cooperation between *TempCode* and *int-code*, until *int-code* has re-entered the state labelled *creation_asked*; *TempCode* can only enter trap t-98 after *int-code* has entered trap t-94. When this all finally happens, WODAN finishes the transition to the ISPW-7 case.



W.r.t. to WODAN is this subprocess s-89

Figure 69. First intermediate phase of Design

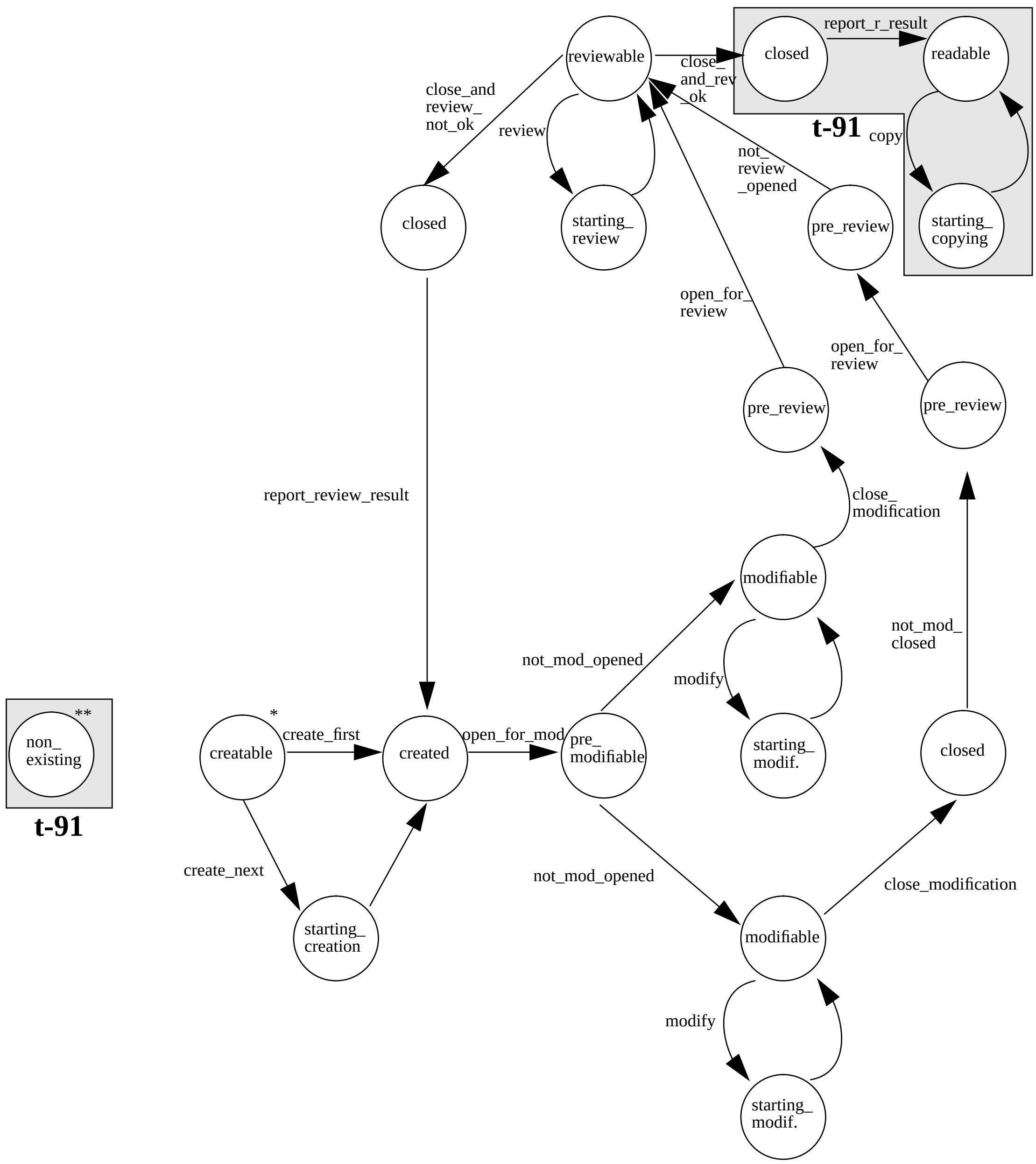
7.4. Process exception: problem description

The second part of the ISPW-7 example concentrates on process exception; this is a temporary change to the model due to an exceptional circumstance.

In SOCCA such a temporary change can be handled as follows:

- Design a model for the process as it should be during the process exception.
- Switch with aid of WODAN to this model at the right time instant.
- As soon as the exceptional circumstance has passed, WODAN can be used to switch back to the standard situation.

As stated above, it is necessary to switch two times to another model; first to the model of the process exception and later back to the model of the standard situation. This switching to the right model can be handled in the same manner as switching to another model during process



W.r.t. to WODAN is this subprocess s-91

Figure 70. Second intermediate phase of Design

evolution. Also in this case, special care should be taken to avoid inconsistencies and if it is not possible to avoid inconsistencies, they should be solved as indicated in chapter 4.

The proposed process exception in the ISPW-7 example is as follows: ‘Suppose that due to unavailability of assigned personnel, it is decided to bypass a follow-up (say second) design review. This decision is made dynamically at the time that this review was scheduled to occur. The rest of the process continues normally in this instance.’ (quoted from section 4.2.2 of [4]).

In the following sections it will be examined how this exceptional circumstance can be incorporated into the current SOCCA model and how WODAN must be designed to switch to the exceptional circumstance and return back to the standard situation as soon as possible.

7.5. Designing the exceptional model

In this section, the model to handle the exceptional situation will be designed. According to the ISPW-7 example, there is not enough personnel to review the design and therefore, reviewing the design should be skipped. This can be modelled in two manners:

- Change the external behaviour of *Design* in such a manner that all parts that handle reviewing the design document will be skipped; make one transition from state 15 (see figure 23 on page 37 for the state numbering) and one from state 25 (both labelled *pre_review*) to the state 20 (*review_closed)* and label these transitions with the export operation *skip_review.* This however has a huge disadvantage; since the various instances of *Design* depend on each other through the behaviour of *int-review_ok* and *int-prepare*, this dependence will be disturbed:

  The behaviour of the next instance of *Design* is initiated by a call to its *prepare* operation performed from the internal behaviour of *int-close_and_review_ok* of the current instance of *Design*. However, as the *close_and_review_ok* operation of the current instance will be skipped when the review process is skipped, the current instance of *Design* will not call the *prepare* operation of the next instance of *Design*, thus the next instance of *Design* will not be initiated. This problem can be solved by changing the next instance of *Design* in such a manner that it does not wait for *int-close_and_review_ok* of the current instance of *Design* to call the prepare operation but that it does wait for *skip_review* of the current instance of *Design* to call the prepare operation. Thus, when the current instance of *Design* is changed in such a way that the review process will be skipped, the next instance of *Design* will also have to be changed, despite of the fact that such a next instance should behave in the normal manner without being aware of any exceptional circumstances.
- Change the behaviour of *int-review* in such a manner that reviewing the document will be skipped but that the communication with other instances of *Design* will not be disturbed. Fortunately this is very easy; as starting the process of reviewing the design document is modelled by means of entering trap t-14 of subprocess s-14 (a subprocess of *int-review*), it is only necessary to remove trap t-14 and the transition to it, labelled *call_review,* from subprocess s-14. This can be done without introducing inconsistencies as *Design* is the only manager that can react to the *call_review* transition; *Design* will be waiting in its state *reviewable* until *int-review* either performs a *call_review*, a *call_review_not_ok* or a *call_review_ok* and on such a call, *Design* will make the appropriate transition. When *int-review* does not perform a *call_review* because that transition has been removed, it will not influence the behaviour of *Design*.

As the second way to handle the exception gives the least problems, this solution will be used. Of course it is not possible to remove a transition and a state from a subprocess. However, it is possible to make a new subprocess that resembles subprocess s-14 without that state and transition and to adapt the state-action interpreter of *Design* to prescribe this temporary subprocess and react to the traps of it, instead of prescribing subprocess s-14 and reacting to the traps of that one. The subprocess of *int-review* to handle the process exception is given in figure 72.

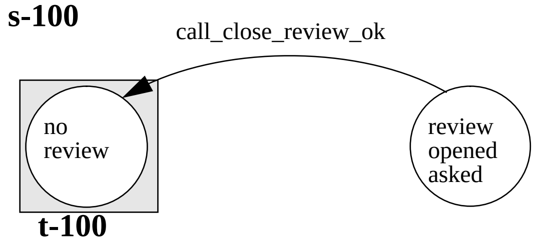
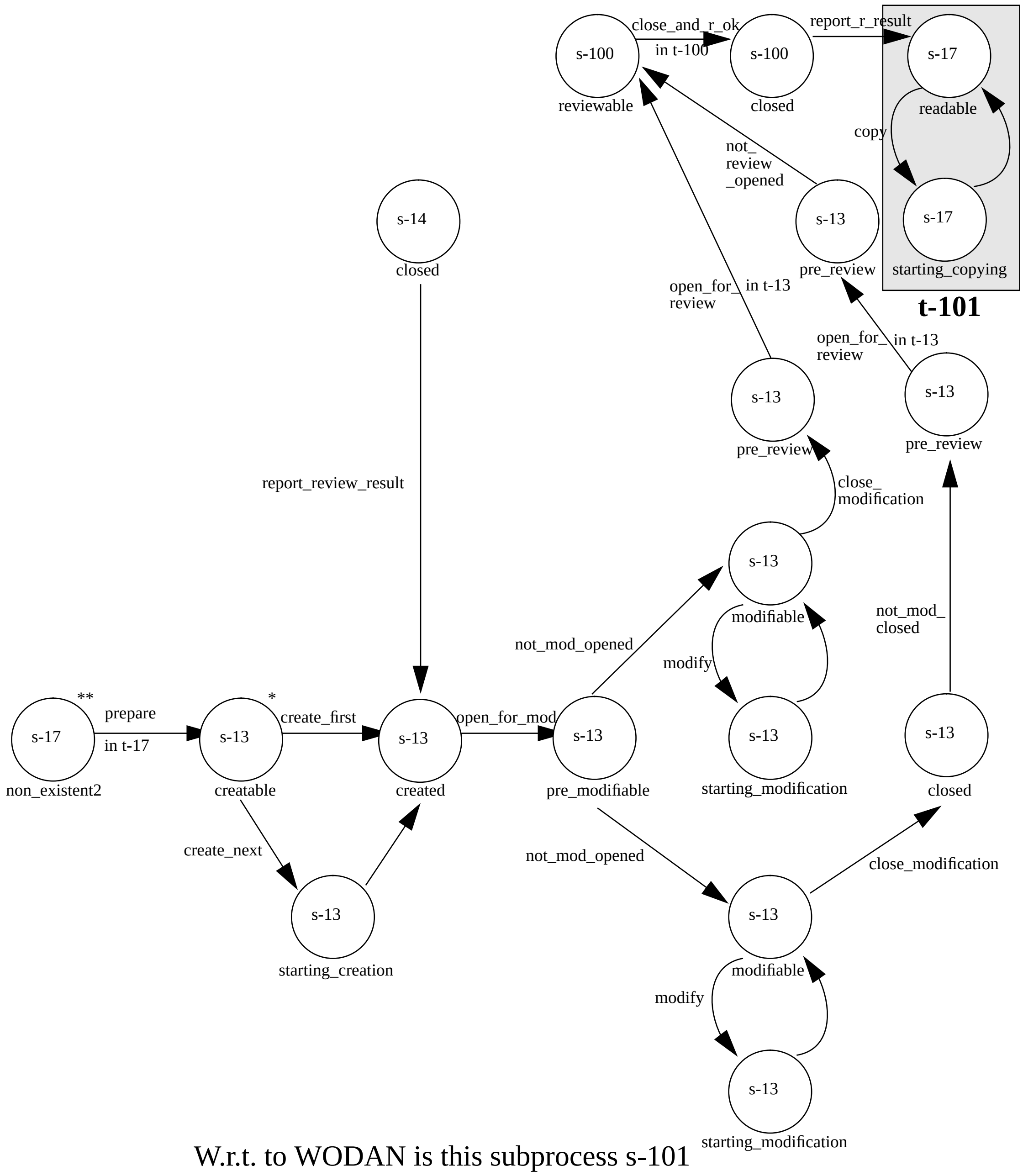


Figure 72. Int-review’s subprocess to handle the ISPW-7 process exception

Note that not only the *call_review* transition has been removed but also the *call_close_review_not_ok* transition. This is because, in accordance with the ISPW-7 example, the behaviour of *int-review* during the process exception has to reflect the behaviour of bypassing a follow-up design review step; it is assumed that the design document has been (re)designed in the right manner and thus that such a follow-up design review step is not necessary.

Design as manager of *int-review* has to be changed too in the exceptional circumstance because *Design* has to prescribe subprocess s-100 in stead of s-14. As s-100 has no trap t-14 nor a trap t-15, the transitions of *Design* that are followed on entering these traps, have to be removed from *Design* to assure that *Design* will only follow the transition corresponding with trap t-100. The state of *Design* that corresponds with starting the review process, can also be removed in the exceptional case, as this state will not be entered.

The STD for *Design* as manager of *int-review* during the exceptional case can be found in figure 73.



W.r.t. to WODAN is this subprocess s-101

Figure 73. Design: viewed as manager of int-review during process exception

7.6. Starting the exceptional case

The last step consists of checking for the problems mentioned in chapter 4 and solving these following the suggested solutions.

As the subprocess for *Design* during the exceptional case has less states then the subprocess for *Design* during the normal case, WODAN can not always switch to the exceptional case immediately. This is problem P2 and the easiest way to solve this problem is applying solution S4; make an intermediate phase of *Design* with a trap consisting of those states that form no problem and design WODAN in such a manner that first the intermediate phase of *Design* will be prescribed and afterwards, the exceptional phase of *Design*.

However, also another problem exists; subprocess s-100 of *int-review* (see figure 72) has less states then subprocess s-14 (see [2]), which is prescribed from the same state of *Design*. This is problem P1 and the easiest way to solve it in this case, is using solution S1; make an intermediate phase of *Design* with a trap consisting of those states in which no problem arises and design WODAN to prescribe first the intermediate phase of *Design* and the standard phase of *int-review* and afterwards the exceptional phase of *Design* and of *int-review.*

The solutions S1 and S4 can be combined to form the intermediate phase of *Design* and to design WODAN.

Note that the STD of *int-review* remains the same during the exceptional phase. The different behaviour of *int-review* is modelled only by means of its new subprocess s-100.

When switching back from the exceptional phase to the normal phase, no problems will arise, so this can be done without further special actions.

WODAN is displayed in figure 75, the ‘normal’ phase of *Design* in figure 63, the exceptional phase of *Design* in figure 73 and its intermediate phase is displayed in figure 74.

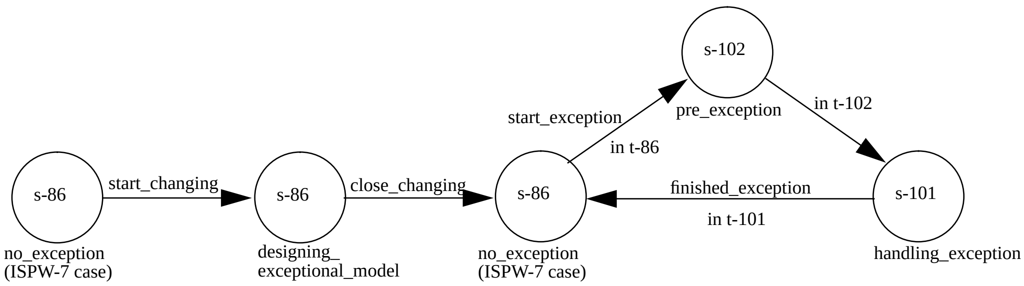
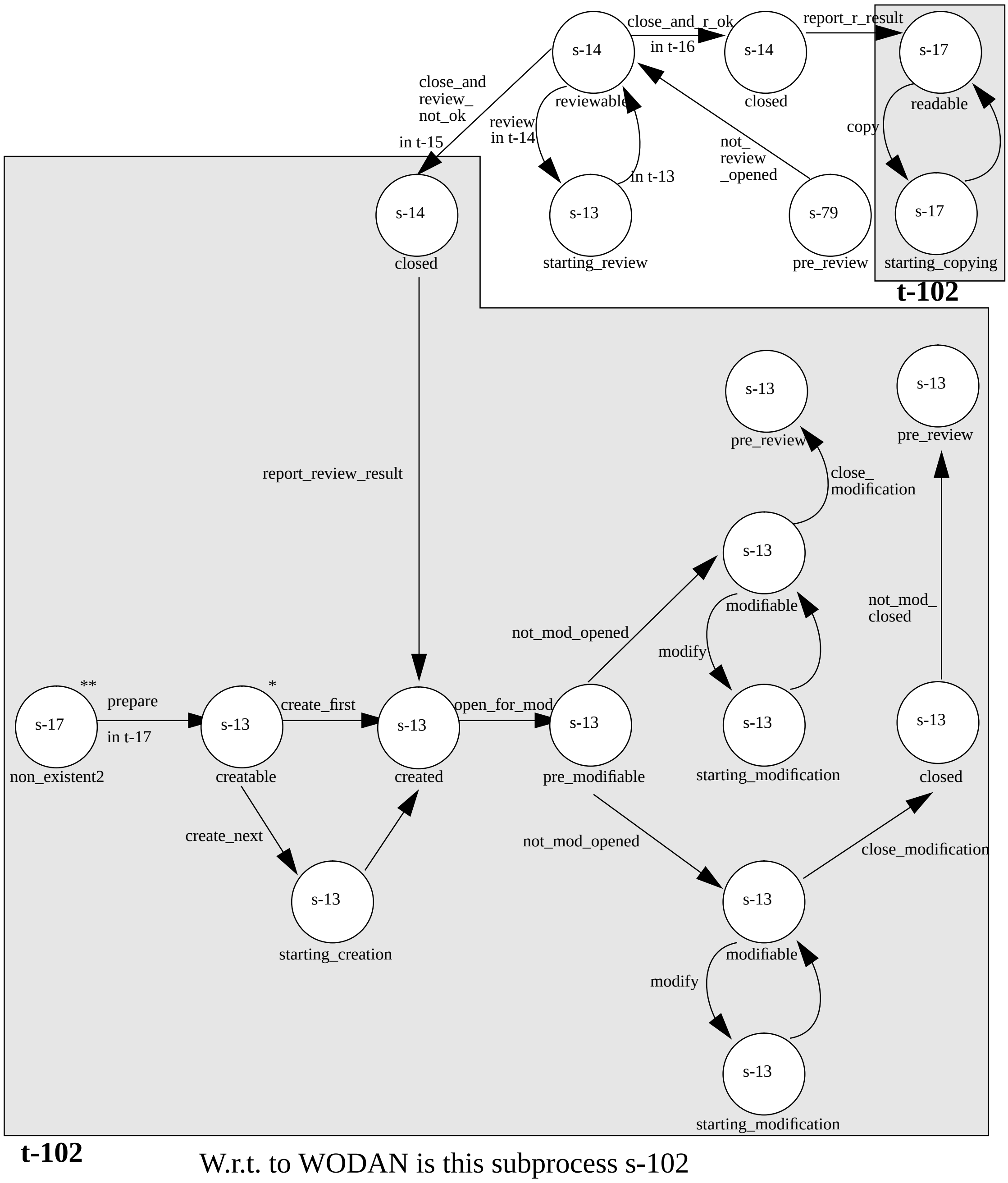


Figure 75. WODAN: viewed as manager of Design

7.7. Concluding remarks

Modelling the change parts of the ISPW-7 example is completely possible within SOCCA. The problems that have been mentioned in the ISPW-7 example correspond with the general problems with process change in SOCCA as mentioned in chapter 4 and the intuitive solutions like allowing the inconsistencies to remain or rolling the process back, correspond with the SOCCA solutions introduced in chapter 4.

The definition of a subprocess has been slightly extended, compared with the original papers defining PARADIGM; in these papers, a subprocess is defined being a restriction to a decision process [1]. However, this definition does not take into account that a process may be the subprocess of a manager process, like an external process which is the subprocess of an anachronistic external process. When this is the case, the process is not only a decision process but it is a decision process with a state-action interpreter. Likewise, the subprocess does not only impose a restriction on the decision process but it also imposes a restriction on the state-action interpreter. Thus, a subprocess is a restriction to a decision process, together with its state-



W.r.t. to WODAN is this subprocess s-102

Figure 74. Design: viewed as manager of int-review during start of exceptional case

action interpreter when the process is a manager process. See also notion 1 on page 63.

WODAN is responsible for all possible changes of the model. It does not only change the dynamic perspective by defining new external and internal processes, new subprocesses of the internal processes and new state-action interpreters to model the changed communication but WODAN can also change the static perspective by adapting the class structure; WODAN can add and remove export operations from a class description whenever necessary and WODAN can even define or remove complete classes. WODAN will probably also be able to change the process perspective of a SOCCA model, which is described by means of object flow diagrams.

Chapter 8
A very brief comparison of SOCCA with other paradigms

In this chapter the properties of SOCCA with respect to the ability of dynamic process modification will be compared with some other paradigms mentioned in [5]. In [5] there is also a paper considering some requirements for enactment mechanisms [6]. In that paper the notion of so called process variables has been introduced as a fundamental mechanism for process change. In the following section this notion of process variables will be applied to SOCCA and in the sections afterwards SOCCA will be compared with some other existing paradigms.

8.1. The notion of process variables applied to SOCCA

In [6] the notion of process variables to control process change, has been introduced. According to that paper, process variables are not intended as a concrete feature of a process definition formalism but they represent a set of features needed by any effective process definition formalism. Process variables should have the following features:

- They are introduced in the “text” of a process definition and can be associated there with type specifications, constraints and default values.
- They need to represent a wide variety of different aspects of a process, like products produced by a performance of the process, tools to be used in performing the process, project specific goals or subgoals or various process definition fragments.
- Process variable binding needs to be incremental and multi-faceted. For example, a process variable might be first bound to a type specification, constraining the actual values to which it can be bound, and have actual values conforming to that specification bound later during enactment.

Process variables are mainly intended to represent yet undefined aspects of the process that is being defined by the process definition formalism.

In chapter 3 the concept of anachronistic external and internal processes has been introduced. The external and internal process descriptions that define the SOCCA process specification are just subprocesses of these anachronistic processes. When applying the concept of process variables to these notions found in SOCCA, the anachronistic processes can be regarded as being process variables with the explicitly designed internal and external process descriptions being the values that can be bound to these process variables. For example the STD of the internal behaviour of *int_monitor1* can be regarded as the value that is initially bound to the process variable representing the monitor process. When some time later, the STD of *int_monitor1* gets replaced by the STD of *int_monitor2*, it can be said that from that moment on the STD of *int_monitor2* is the value bound to the process variable representing the monitor process.

The binding mechanism, to bind the process variable value to the process variable, is modelled explicitly by WODAN and its state-action interpreter. So within SOCCA, this binding mechanism itself is a part of the model instead of a property of the enactment mechanism. This property makes SOCCA a reflective specification mechanism.

The notion of process variables can also be applied to the class perspective of the SOCCA model; the static structure of the model can change when the model evolves, an example of such a change can be found in chapter 7. To achieve such a change of the static structure, one can say that a process variable exists which value is the current static structure. When WODAN has to change the static structure, WODAN only has to bind a new value to this special process variable.

The notion of process variables can also be applied to WODAN itself; WODAN is in fact an infinitely large decision process which can follow any arbitrary path through its state space. Which path will be followed, and therefore how the model evolves, depends on the input -specifying the process which has to be modelled, the wanted process change, etc.- WODAN receives from somewhere outside the model. However, despite the fact that WODAN is infinitely large, only a finite part of WODAN is relevant, and therefore visible, at any time instant. Thus, the currently visible part of WODAN can be considered being a value bound to a process variable representing WODAN.

When considering this from the viewpoint of the SOCCA approach, one could also say that a not explicitly designed version of WODAN exists of which only one subprocess is visible each time; the currently visible part of WODAN is only another subprocess of the not explicitly designed version of WODAN and starting the design of a new model and the transition to the new evolution stage in which the new model is prescribed, is triggered by prescribing a new subprocess of WODAN from somewhere outside of the model.

8.2. Comparing SOCCA with Document Flow Model (DFM)

In [7] three primary objectives have been mentioned which guided to the development of DFM. These objectives are process mobility, framework for change and simplicity. According to [7] these objectives can be reached when the enactment system has two basic properties:

- Independent processes
- Asynchronous communication

By independent processes it is meant that all processes are first class citizens within the model: there is no master process and new processes can join the evolving network. At a first glance SOCCA does not have this property, since the external processes are managers of the internal processes and furthermore, WODAN is a manager of both the external and internal processes. However, when comparing SOCCA and DFM in more detail it can be seen that somehow the external processes in SOCCA correspond with the actons in DFM and the internal processes in SOCCA correspond with the internal behaviours of the actons in DFM. Since the external processes in SOCCA are first class citizens in relation to each other -no external process is the master of another external process and new external processes may join the model-, this property of independent processes is fulfilled. As WODAN is mainly used to guide the process evolution, one can neglect the fact that WODAN is a manager of everything in the normal case when the process is just enacting and no evolution has to happen.

As the communication in SOCCA between the various processes is PARADIGM communication, which is a form of asynchronous communication, all communication in SOCCA is asynchronous so also the second property is fulfilled in SOCCA.

Thus, the two basic properties of DFM are fulfilled in SOCCA, which makes SOCCA powerful enough to satisfy the three primary objectives that guided to the development of DFM.

8.3. Comparing SOCCA with SPADE

Another process centred environment is SPADE which is centred on a language, SLANG, based on high level petri nets [8]. In SPADE the ability to process change is established via modularization of the SLANG model: to change a part of the process during enactment, enactment of the module to be modified can temporarily be stopped and after modification, enactment of the module can continue. This has the disadvantage that the enactment partially has to be stopped for a while, which is not necessary in SOCCA when changing a process.

One of the main features of SLANG is the possibility to model time constraints. For example to automatically abort some function after a timeout period. Such time constraints can also be used in SOCCA; an STD which is used to model the behaviour of a process in fact is a decision process with i.a. a sojourn mechanism that determines the time instant when the next transition is going to be followed. One can make use of this fact by making an abort transition leading from a state which possibly has to be aborted to some other state. This transition can then automatically be followed when the STD is still in the state where the abort transition starts at the moment that the timeout period elapses.

Chapter 9
Conclusions and further research

This thesis shows that incorporating the concept of anachronistic processes together with WODAN and its state-action interpreter into SOCCA, gives SOCCA the reflectivity necessary for process evolution [9]. This reflectivity property is in fact a direct consequence of the existence of WODAN's state-action interpreter, which is a part of the model, describing another part of the model. Thus, within SOCCA, the reflectivity is a natural feature of one of the constituting formalisms, namely of PARADIGM.

Process evolution can lead to inconsistencies in the model after switching from one evolution stage to another evolution stage. These inconsistencies have to be detected by a careful analysis of the effects of the intended change. This possibly can be supported by tools which check for the occurrence of the problems P1, P2 and P3 mentioned in chapter 4 by comparing the old SOCCA processes and the new SOCCA processes with each other. After detecting the inconsistencies, they can be solved or even avoided by following the guidelines mentioned in chapter 4.

Further categorisation of problems as a consequence of process change is a topic for further research.

A SOCCA model can become very large. However, through the modular approach such a large model can still be understood very well. This modular approach also makes it easier to analyse the consequences of an intended change at a local level, without taking any part of the model into account which is not directly related with the changed part.

An interesting approach to make SOCCA models smaller and less complex is by incorporating the concepts of roles and views into SOCCA. This approach has only slightly been mentioned in section 6.4. It can be a topic of further research to incorporate it fully into SOCCA.

In this thesis, the newly introduced concept of anachronistic processes in combination with WODAN to incorporate change in SOCCA has only been used to model process evolution and emergency handling, both mentioned in the ISPW-7 case. An interesting topic of future research is to analyse the application of this change incorporation for model growth and incremental modelling, prototyping in modelling and customizing. Another application of the change incorporation may be reusability by viewing reuse as a certain evolution of a model component and even interoperatibility might be incorporated with the above mentioned change concepts.

Chapter 10
References

1 Groenewegen L.: *Parallel Phenomena 1-14*. University of Leiden, Dep. of Computer Sc., Techn. Rep. 86-20, 87-01, 87-05, 87-06, 87-11, 87-18, 87-21, 87-29, 87-32, 88-15, 88-17, 88-18, 90-18, 91-19. 1986-1991

2 Engels G., Groenewegen L.: *SOCCA: Specifications of Coordinated and Cooperative Activities.* University of Leiden, Dep. of Computer Sc., 1993

3 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

4 Kellner M., Feiler P., Finkelstein A., Katayama T., Osterweil L., Penedo M., Rombach H.: *ISPW-7 Software Process Example*. 7th International Software Workshop. Yountville, California, 16-18 October 1991

5 *Pre-prints of Papers, Third European Workshop on Software Process Technology, EWSPT’94*

6 Dowson M., FernStröm Chr.: *Towards Requirements for Enactment Mechanisms*. In: EWSPT’94

7 Berrington N., De Roure D., Greenwood R., Henderson P.: *Distribution and Change: Investigating two challenges for Process Enactment Systems*. In: EWSPT’94

8 Bandinelli S., Fuggetta A., Ghezzi C., Grigolli S.: *Process Enactment in SPADE*

9 Conradi R., Fernström Chr., Fuggetta A.: *Concepts for Evolving Software Processes*. In [10], 9-32

10 Finkelstein A., Kramer J., Nuseibeh B. (eds.): *Software Process Modelling and Technology.* Research Studies Press Ltd., Taunton 1994

Appendix A
List of figures

1. A general example of an evolution step 13
2. The general situation considered in this part 15
3. Situation b1: the first part of E1 and E2 is the same 17
4. The manager is different during EVS2 17
5. Problem situation P1 19
6. Example of solution S1 20
7. Example of solution S2 21
8. Example of problem P3 22
9. Example of solution S6 23
10. Example of solution S7 24
11. Example of solution S8 25
12. Int-monitor1: STD of the internal behaviour 27
13. Design: only viewed as manager of int_monitor1 28
14. Int-monitor1: subprocesses and traps w.r.t. Design 29
15. Int-monitor2: new STD of the internal behaviour 30
16. Int-monitor2: subprocesses and traps w.r.t. Design 31
17. Design2: only viewed as manager of int_monitor2 32
18. WODAN: switch to int-monitor2 and Design2 via TempDesign. 34
19. TempDesign: subprocess for transition to the new design 34
20. WODAN: switch to monitor2 and Design3.. 35
21. Design3: only viewed as manager of int_monitor2 35
22. WODAN: switch to ExtendedDesign and int-monitor2. 36
23. ExtendedDesign: only viewed as manager of int_monitor2 37
24. Class diagram: classes and is-a and part-of relationships 39
25. Class diagram: attributes and operations 39
26. Class diagram: classes and general relationships 40
27. Import/export diagram 41
28. Import list 41
29. DesignEngineer: STD of the external behaviour 42
30. ProjectDocs: STD of the external behaviour 42
31. Code: STD of the external behaviour 43
32. ProjectManager: STD of the external behaviour 43
33. Compiler: STD of the external behaviour. 43
34. Int-code: STD of its internal behaviour 44
35. Int-release_object_code: STD of its internal behaviour 44
36. Int-compile: STD of its internal behaviour 44
37. Int-test_ok: STD of its internal behaviour 44
38. Int-schedule_and_assign_task. 45
39. Int-schedule_and_assign_tasks's subp. and traps w.r.t. DesignEngineer 45
40. Int-code's subprocesses and traps with respect to DesignEngineer 46
41. DesignEngineer, manager of int-design, int-review, int-code and int-sched. 46
42. ProjectDocs, manager of int-design, int-code and int-create-version 47
43. Int-code's subprocesses and traps with respect to ProjectDocs 47
44. Int-code's subprocesses and traps with respect to Code 49

45. Int-compile’s subprocesses and traps with respect to Code 50
46. Int-release_object_code’s subprocesses and traps with respect to Code 48
47. Int-test_ok’s subprocesses and traps with respect to Code (other instance) 48
48. Code: manager of 8 employees. 51
49. int-test_ok’s subprocesses and traps with respect to Code (same instance) 48
50. Compiler: manager of int-compile and int-code. 52
51. Int-compile’s subprocesses and traps with respect to Compiler 52
52. Int-code’s subprocesses and traps with respect to Compiler 53
53. Int-schedule_and_assign_tasks’s subp. and traps w.r.t. ProjectManager 54
54. ProjectManager: manager of int-schedule_and_assign_tasks and int-monitor 52
55. Int-monitor’s subprocesses and traps with respect to ProjectManager. 53
56. Int-release_object_code’s subprocesses and traps with respect to Design 55
57. Design: manager of 9 employees 56
58. Code: new STD of the external behaviour 59
59. Int-wait_for_approval: STD of its internal behaviour 59
60. Int-wait_for_approval’s subprocesses and traps w.r.t. Code. 60
61. Code: viewed as manager of int-release_obj_c and int-wait_for_approval 60
62. Int-wait_for_approval’s subprocesses and traps w.r.t. Design 61
63. Design: viewed as manager of int-wait_for_approval 62
64. Intermediate phase of Code 63
65. WODAN: viewed as manager of 3 employees. 63
66. Intermediate phase of int-code 64
67. TempCode: viewed as manager of int-code 66
68. Int-code’s subprocesses and traps with respect to TempCode 67
69. First intermediate phase of Design 68
70. Second intermediate phase of Design. 69
71. WODAN: viewed as manager of 5 employees. 65
72. Int-review’s subprocess to handle the ISPW-7 process exception 70
73. Design: viewed as manager of int-review during process exception 71
74. Design: viewed as manager of int-review during start of exceptional case 73
75. WODAN: viewed as manager of Design 72