Internal Report 09-16 Aug 2009

Universiteit Leiden

Opleiding Informatica

UML Specification of and UML Tooling for Paradigm

Jasper Stafleu

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

1 Introduction **3**

2 A description of Paradigm **4**
2.1 Detailed Behaviour 4
2.2 Global Behaviour and Partitions 5
2.3 Subprocesses and Traps 7
2.4 Management and Consistency Rules 9
2.5 Changeclauses 10
2.6 Hierarchy of Paradigm 11

3 UML Modeling of Paradigm without reconfiguration **14**
3.1 Detailed behaviour 14
3.2 Subprocesses 15
3.3 Partitions and Traps 17
3.4 Component Interaction 19

4 Foreseen Model Adaptation **22**
4.1 McPal 22
4.2 A Producer Consumer for Paradigm 24
4.3 Self-adaptation of the Producer-Consumer 26
4.4 Adaptation in UML 28
4.5 Self-Adapting Producer Consumer in UML 29

5 Unforeseen Adaptation in Paradigm and UML **32**
5.1 Scenario 32
5.2 Shrinking the Buffer 32
5.3 JIT modeling in UML 36

6 Conclusion, Related and Future Work **37**
6.1 Solution 37
6.2 Environment 37
6.3 Related Work 38
6.4 Future Research 38

A Program **41**
A.1 Technologies Used 41
A.2 Component Communication 41
A.3 Making a JavaScript execution thread sleep 43
A.4 Creating a new model 44
A.5 Adaptation 46
A.6 Multiple views 47
A.7 Wrap-up: Possible Improvements in Parallelism 47

B Code **49**
B.1 script/Statemachine.js 49
B.2 script/Paradigm.js 52
B.3 script/Poll.js 57
B.4 script/std.js 57
B.5 script/umlPrint.js 58

B.6 script/script.js 63
B.7 ajaxFiles/sendMessage.php 64
B.8 ajaxFiles/poll.php 66
B.9 ajaxFiles/leavePage.php 66
B.10 ajaxFiles/JITForm.php 66
B.11 ajaxFiles/uploadResult.php 66
B.12 ajaxFiles/updateLocation.php 68
B.13 ajaxFiles/getLocation.php 68
B.14 ajaxFiles/noCache.php 68
B.15 script/getComponents.php 69
B.16 index.php 71
B.17 query.php 73
B.18 img/rounded.php 73
B.19 style.css 73
B.20 sampleMigrFiles/MigrDetSchedExtWorker.migr 74

1 Introduction

UML is a widely recognized modeling standard, used to create abstract models of a system —software or otherwise—, describing its structure and/or behaviour, see e.g. [8]. Its diagrams are generally understood by modelers, making UML the language of choice when documenting systems. A problem with UML models is the restriction to their as-is behaviours; it is completely unclear how to reconfigure a model, other than by complete substitution. This restriction forces modelers to consider in advance which changes to the system might occur in order for the model to remain compliant with future demands on the system.

Paradigm is a component-coordination language in which the components can be reconfigured without knowing beforehand what the reconfiguration is going to be. In addition, by the time the reconfiguration is known, the component need not be halted in order to achieve the desired change, nor do all components need to alter their behaviour simultaneously. This makes Paradigm an ideal language to describe systems which need to be capable of altering their behaviour in the future —regardless of what those changes are going to be— and of modeling long-term multi-step reconfigurations of such systems.

In view of these observations, this Master's Thesis will investigate to what extent UML and Paradigm can be brought together. To that aim, Section 2 introduces Paradigm itself, describing each paradigm model constituent both in theory and through an example based on the Worker-Scheduler model from [3, 2].

Section 3 will handle a translation of these Paradigm constituents into UML constituents, showing that Paradigm models without reconfiguration can be translated consistently into UML models, without losing the separation of concerns of the various types of dynamics within the model. It also shows how the interaction between components can be dealt with without having to halt the execution of the model as a whole —and specifically of the relevant components themselves— while also preventing inconsistent situations. This allows the various components within the model to run concurrently.

Section 4 will describe how Paradigm handles reconfiguration in more detail than shown in Section 2 and how this can be translated into UML. This will be shown using a new example based upon the Producer-Consumer problem (see [6]) with a variable sized buffer. Without reconfiguration, the buffer size will be fixed, but when the model 'recognizes' the buffer is too small, a migration can occur which increases the buffer size. The migration itself is foreseen, even though the moment upon which it is executed is not.

Section 5 will describe a scenario for an unforeseen migration of the Producer-Consumer model from Section 4, and how the concepts from that Section can be used to execute such migrations as well, both in Paradigm and in UML.

Together, these sections provide a complete UML description of Paradigm, and have been used to create a multi-agent environment, specified through UML 2.0, in which Paradigm models can be visualized and both foreseen and unforeseen migrations can be executed upon those models. This environment can be perceived as a proof of concept for the methods used to translate the Paradigm models into UML models. A short description of the more complex concepts used in the environment have been described in Appendix A, while the code itself has been added in Appendix B.

2 A description of Paradigm

This section will describe the component based modeling language Paradigm. To facilitate this description, a Worker-Scheduler model will be used as an example. The model shows the behavior of five components:

- Three Workers that need exclusive access to a resource, also named a critical section
- A Scheduler which issues and revokes the access permissions of the three Workers,
- A McPal component, which allows the model to change (migrate) to any other model.

2.1 Detailed Behaviour

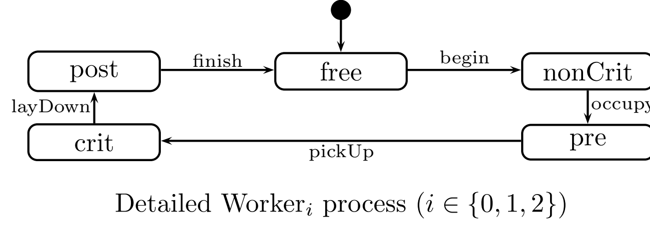
For each component in Paradigm, its behaviour can be split up into two types. The first type is named detailed behaviour and describes the unconstrained, detailed behaviour of the component. This behaviour is described by a state-transitions diagram (STD), which is a quadruple $\langle S, A, T, s_t \rangle$ such that

- $S \neq \emptyset$ is the set of states
- A is a set of actions or transition labels
- T is the set of transitions such that $T \subseteq S \times A \times S$
  $(s_{fr}, a, s_{to}) \in T$ can also be denoted as $s_{fr} \xrightarrow{a} s_{to}$
- $s_t \in S$ is the current state, in a STD's initial configuration $\text{STD}_{t=0}$ this can also be named s_0
- 'Taking' an action $a \in A$ and 'firing' a transition $s_{fr} \xrightarrow{a} s_{to}$ are equivalent atomic operation which require $s_t = s_{fr}$ to hold before the operation and which results in $s_t = s_{to}$

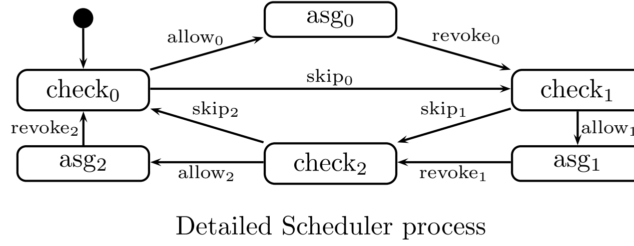
For example, the Worker components at $t = 0$ can be described as

$$Worker_{workerId} = \begin{bmatrix} \{\text{free, nonCrit, pre, crit, post}\}, \\ \{\text{begin, occupy, pickUp, layDown, finish}\}, \\ \left\{ \begin{array}{l} \text{(free, begin, nonCrit),} \\ \text{(nonCrit, occupy, pre),} \\ \text{(pre, pickUp, crit),} \\ \text{(crit, layDown, post),} \\ \text{(post, finish, free)} \end{array} \right\}, \\ \text{free} \end{bmatrix}$$

However, in almost all cases, a graphical representation of the STD will be used instead. A graphical representation of the Worker and Scheduler models is given in Figure 2.1. This figure is drawn in a UML-like style, with the states denoted by the boxes, the transitions as arrows, the actions as labels on the transitions and s_0 pointed at by the black dot.



Detailed Worker$_i$ process ($i \in \{0, 1, 2\}$)



Detailed Scheduler process

Figure 2.1: Detailed behaviour of the Worker$_i$ and Scheduler components

The Worker component starts in state *free*, where it does not need the permission. When the action *begin* is taken, the Worker readies itself in some way for the critical section in state *nonCrit*, after which it enters state *pre* where it waits for the permission to use the critical section to arrive. That eventually occurs and the action *pickUp* is taken; now the state *crit* is reached and the Worker can execute the critical part of its behaviour. It then takes the action *layDown* and reset itself for another iteration in state *post*.

The scheduler uses a round robin strategy to determine whether a Worker wishes to enter the critical section in state $check_i$, allowing it if it does through $allow_i$ and skipping it using $skip_i$. When permission is given, state asg_i is entered, which can only be left by taking the action $revoke_i$ after which the next Worker is checked.

2.2 Global Behaviour and Partitions

The second type of behaviour is named global behaviour, and it presents a view on the dynamicity of the restrictions on the detailed layer. For example, the Worker-Scheduler model requires that at any given time, at most one of the Worker models is allowed to access the critical section. This suggests that each Worker somehow differentiates between (at least) two phases: having the permission (InCS) and not having the permission (OutCS). Also, in order for the Scheduler to consistently determine whether a Worker desires to enter the critical section, a third phase exists (OutCSBlock) which equals the OutCS phase, except that the Worker's "desire" to enter the critical section, as made public, cannot change during this phase.

Each Worker traverses through these phases as if it were a normal statemachine, starting in OutCS, entering OutCSBlock when the Scheduler is checking it, followed by entering InCS if the model was ready to enter the critical section and followed by reentering OutCS otherwise. When InCS is reached, the Worker

is allowed access to the critical section, and when it is done, permission is once again withdrawn and the model reenters OutCS.

This global behaviour appears to be very similar to the detailed behaviour, except that the detailed behaviour does not show how a model is prevented from entering certain detailed states, i.e. how a Worker model is prevented from entering the *crit* state when it does not have permission. The global behaviour does have such distinctions —during the OutCS and OutCSBlock phase, a Worker does not have permission to enter the critical section— and can thus be used to restrict the detailed model to a certain subset of allowed actions during a certain phase, that is, if an action is not in a phase's set of allowed actions, that action can not be taken. In our example, the action *pickUp* is not allowed during the OutCS and OutCSBlock phase, thus preventing the detailed state *crit* from being entered during those phases. Similarly, during the OutCSBlock phase, the "desire" as made public of the Worker to enter the critical section —denoted by being in the state *pre*— is prevented from changing by not allowing the action *occupy*. Finally, not allowing *occupy* during phase InCS ensures the permission to enter the critical section will be returned before a new desire to enter the critical section can be expressed, preventing starvation of other Worker processes.

Paradigm allows the use of sets of global statemachines and multiple restraints with respect to precisely one detailed STD, in order to present a model with multiple views of its phases. For example, if we would want to extend the Worker with an on/off switch in order to skip the $free$ and *nonCrit* states —for which we add the *shortcut* action and the *post* $\stackrel{shortcut}{\longrightarrow}$ *nonCrit* transition— we can create a "view" on the model which has two phases: On and Off, with the first allowing the action and the second disallowing it. Paradigm calls these "views" *Partitions* and the phases *Subprocesses*. The term current subprocess of partition π denotes the subprocess currently determining the allowed actions at the level of that partition; each partition has exactly one current subprocess at any given time. The set of allowed actions of the detailed statemachine is then determined by only allowing an action if each current subprocess allows it.

These observations suggest a formal definition for a partition π: a pair $\langle STD_{global}, AA\rangle$ such that

- STD_{global} is a statemachine $< S_{global}, A_{global}, T_{global}, s_{t,global} >$ as defined in Section 2.1, of which its current state corresponds to the partition's current subprocess
- AA is the set of allowed detailed actions per state of STD_{global} such that

  $$AA = \{(s, A_{allowed}) | s \in S_{global}, A_{allowed} \subseteq A_{detailed}\}$$

  with $A_{detailed}$ the set of actions in the detailed behaviour.
- The set of allowed detailed actions of STD$_{detailed}$ as restricted by partitions $1 \ldots n$ is determined by

  $$\bigcap_{i=1}^{n} \pi_i(AA(s_t))$$

In our example, the partition determining the critical section management from the example is named **CSM**, and can be described at $t = 0$ as

$$\mathbf{CSM}_{workerId} = \left\{ \begin{bmatrix} \{\text{OutCS, OutCSBlock, InCS}\}, \\ \{\text{triv, stay, entering, left}\}, \\ \left\{ \begin{array}{l} (\text{OutCS, triv, OutCSBlock}), \\ (\text{OutCSBlock, stay, OutCS}), \\ (\text{OutCSBlock, entering, InCS}), \\ (\text{InCS, left, OutCS}) \end{array} \right\}, \\ \text{OutCSBlock (iff workerId = 0) or OutCS} \\ \text{(otherwise)} \end{bmatrix}, \begin{bmatrix} \{\text{OutCS, (finish, begin, occupy)}\}, \\ \{\text{OutCSBlock, (finish, begin)}\}, \\ \{\text{InCS, (pickUp, layDown, finish, begin)}\} \end{bmatrix} \right\}$$

Just as with the detailed behaviour, a global behaviour will almost always be visualized, but, unlike detailed processes, a global process is often shown in two figures. The first figure depicts the STD_{global}, as shown in Figure 2.2; in this figure, the starting state is a pseudo-starting state since all Worker_i(CSM) will start in OutCS, except for Worker_0(CSM) which starts in OutCSBlock; upon startup the scheduler will immediately be looking at that worker, see Section 2.3.

The second figure consists of multiple parts; each part representing one subprocess and showing only the detailed actions it allows and the states that can occur as s_t of the detailed STD while being restricted by that subprocess. It also shows the contents of the traps, all of which will be handled more extensively in Section 2.3, which will go into more detail on how subprocesses and traps work. Figure 2.3 shows such a visualization for the subprocesses of CMS.

2.3 Subprocesses and Traps

As mentioned in section 2.2, subprocesses are phases of a model that determine which actions the detailed process is allowed to take. This is a kind of inclusive restriction mechanism: in order for an action to be allowed in the detailed statemachine, it needs to be allowed in each of the current subprocesses, thus if any current subprocess does not allow an action, that action can not be taken. This suggests that a subprocess can be visualised like a detailed process, but showing only the allowed actions. This is done for the three subprocesses of CSM In Figure 2.3. Detailed states which are impossible to be the current state during a subprocess have also been omitted from these figures in order to increase the insight a viewer gets from the figure, even though those states are not explicitly forbidden by the subprocess. For example, in OutCSBlock,

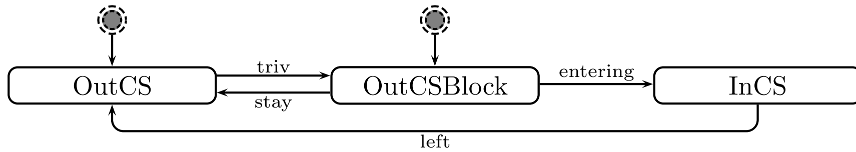


Figure 2.2: Global behaviour of the $Worker_i$ process at the level of Partition CSM, $i \in \{0, 1, 2\}$

the state *crit* is omitted, since it can not be reached. The state *pre* is not omitted, even though it can not be entered during OutCSBlock, since it can be the current state of the detailed process when OutCSBlock first becomes current.

Changing the currently enforced subprocess is done by taking an action in the STD of the partition, but these changes should not be unconditional. For example, as long as the state crit has not been left by the detailed statemachine, we do not want the permission being withdrawn by the scheduler. We therefore do not want the current subprocess of CSM to be changed from InCS to OutCS; this restriction is enforced by *traps*. A trap θ of a subprocess is a set of states such that:

- $\theta \neq \emptyset$
- $\theta \subseteq S_{detailed}$
- $x \in \theta$, $a \in AA(s_t)$ and $x \xrightarrow{a} x'$ imply that $x' \in \theta$

These definitions show that when a trap has been entered, it can not be left again unless the current subprocess —and thus the set of allowed actions AA— is changed. These traps can therefore be used to signify that a subprocess has reached a sort of "goal" and the subprocess can be altered. In our example, we could state that the InCS subprocess has the stated goal of "leaving" the *crit* state, and we can thus define a trap $left$ which contains all the states after *crit*. Reaching this trap signifies the partition's readiness to alter its current subprocess into OutCS. Similarly, the OutCSBlock has two traps, both of which are reached immediately when OutCSBlock is entered. One trap allows the subprocess to be changed to InCS; it contains only the detailed state *pre*, thus

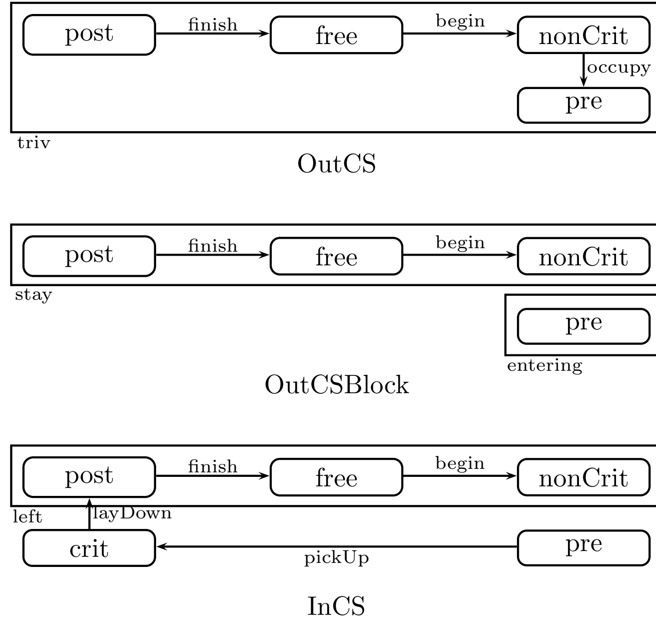


Figure 2.3: Subprocesses of the partition CSM of the Worker process

signifying the detailed model's readiness to enter the *crit* state. The other trap contains all of the other states, allowing the subprocess to be altered back to OutCS, since the Worker is not yet ready to enter the critical section.

These same traps can also be found in Figure 2.2, where they are shown as the actions of the global process. In fact, paradigm allows the alteration of the global state —and thus the current subprocess— only if the corresponding trap has been reached. In this way, the detailed process has a certain amount of control over which constraints are enforced upon itself by the global process. However, traps do not enforce the taking of a global transition; they only allow them. The actual changing of the global state is triggered by the manager, which will be handled in Section 2.4.

2.4 Management and Consistency Rules

The actual change of a subprocess is restricted by the traps it has entered, but orchestrated by a *manager*. A model's managers consist of all the components which can alter the current subprocess of any partition of the model; for example, in the Worker-Scheduler model, we want the Scheduler to be able to alter the Workers' current subprocesses of partition CSM, thus granting or revoking the Workers' permission to enter the critical section. Therefore, the Scheduler is the manager of each of the Workers. In a similar fashion the *employees* of a model can be determined: the Workers are the Scheduler's employees, since they are managed by it.

A manager determines when to alter an employee's current subprocess through its own detailed process. For example, when the Scheduler (see Figure 2.1) is in the detailed state $check_0$, we want the Worker_0 to have OutCSBlock as current subprocess of the partition CSM. If the scheduler then takes the action $allow_0$, the current subprocess of partition CSM of Worker_0 needs to change to InCS, while taking $skip_0$ should change the current subprocess to OutCS, simultaneously altering the current subprocess of Worker_1(CSM) from OutCS to OutCSBlock, since it starts "looking" at that Worker. These changes in subprocesses are not always allowed though, but this can be determined by inspecting the trap information of the Worker; i.e. if $Worker_0$ has entered trap *stay*, the CSM partition's STD does not allow the action *entering* thus the action $allow_0$ should not be allowed either. This type of consistency is enforced by the *consistency rules*:

- A consistency rule consists of exactly one detailed transition $P : s \xrightarrow{a} s'$ with process P, origin state s, resultant state s' and action a, see the definition for STDs in section 2.1
- If this transition is a managing transition, the detailed transition is followed by an asterix ($*$) and a comma-separated list of global transitions $P_i(\pi_{i,j}) : S_{i,j} \xrightarrow{\theta_{i,j}} S^{'}_{i,j}$, with $S_{i,j}$ as origin subprocess of partition $\pi_{i,j}$ of employee process P_i and resultant subprocess $S^{'}_{i,j}$, as restrained by the requirement of trap $\theta_{i,j}$ having been entered
- For a managing detailed transition to fire, the following need to hold:
  - $a \in \bigcap_{i=1}^{n} \pi_i(AA(s_t))$

- $s = P(s_t)$
- $P_i(\pi_{i,j})(s_t) = S_{i,j}$ for each of the global employee transitions
- $P_i(s_t) \in \theta_{i,j}$ for each of the detailed employee processes

- If a managing detailed transition fires, each of the employee's global transitions are triggered simultaneously

- A consistency rule can also contain a *changeclause*, which will be handled in Section 2.5. These changeclauses do not enforce additional restraints on the consistency rule as a whole

For example, the consistency rules for the Workers are:

$$\begin{array}{lrcl} Worker_i : & free & \xrightarrow{begin} & nonCrit \\ Worker_i : & nonCrit & \xrightarrow{occupy} & pre \\ Worker_i : & pre & \xrightarrow{pickUp} & crit \\ Worker_i : & crit & \xrightarrow{layDown} & post \\ Worker_i : & post & \xrightarrow{finish} & free \end{array}$$

Since they are all non-managing transitions, they correspond to the Workers' detailed transitions only. The Scheduler's consistency rules are:

$$\begin{array}{llrcl} Scheduler : & & check_i & \xrightarrow{allow_i} & asg_i \\ * & Worker_i(CSM) : & OutCSBlock & \xrightarrow{entering} & InCS \\ Scheduler : & & asg_i & \xrightarrow{revoke_i} & check_{i+1} \\ * & Worker_i(CSM) : & InCS & \xrightarrow{left} & OutCS, \\ & Worker_{i+1}(CSM) : & OutCS & \xrightarrow{triv} & OutCSBlock \\ Scheduler : & & check_i & \xrightarrow{skip_i} & check_{i+1} \\ * & Worker_i(CSM) : & OutCSBlock & \xrightarrow{stay} & OutCS, \\ & Worker_{i+1}(CSM) : & OutCS & \xrightarrow{triv} & OutCSBlock \end{array}$$

which are managing rules. They show the coupling of detailed scheduler transitions to the global worker transitions, as described earlier. It also shows why Worker_0(CSM) needs to have a different starting current subprocess from the others: if it also started in OutCS, none of the Scheduler transitions could ever fire.

2.5 Changeclauses

Besides changing the current detailed state and subprocesses of employees, consistency rules can also change variables that are internal to the models. Paradigm sets very few restrictions to the form such variables can take, so it can be used in order to store almost any information. For example, if we want to "log" the number of times a Worker has had permission to enter the critical section, we could add a variable to the Worker called *Log*, a number which is

increased by one each time the Scheduler takes action $allow_i$. Paradigm defines these changeclauses through a new extension to an existing rule, in the format

$$P_i[var := newValue]$$

In order to model the "log" into the consistency rules as given in Section 2.4, the rule for the $allow_i$ action would become

$$\begin{array}{rrcl} Scheduler: & check_i & \xrightarrow{allow_i} & asg_i \\ * \quad Worker_i(CSM): & OutCSBlock & \xrightarrow{entering} & InCS, \\ & Worker_i[Log & := & Log+1] \end{array}$$

Particularly, in Paradigm models which are capable of self-adaptation, a specialized component, named McPal (an abbreviation of Managing changing Processes at leisure), contains the variable CRS (or Consistency Rule Set), which holds the current consistency rules. This way, the consistency rules can be altered using a changeclause and the model's behaviour can be altered. Subsequently, when a consistency rule contains a state which does not yet exist, it is implicitly created, therefore changes in the CRS also define new states, transitions, actions, subprocesses and traps. This constitutes one of Paradigm's biggest advantages; not only can separation of concern be utilized in order to accurately describe parallel models in a simple manner, but specialized components can even alter these on-the-fly, keeping the model consistent during these changes if the migration is properly modeled. How this can be done will be handled in more detail in Section 4.1.

While changeclauses can alter many variables in the model, not everything is open for change. The current states and subprocesses can only be changed by taking an action in the relevant STD, since doing this could otherwise create inconsistent models. Also, the number of components and partitions in the model is immutable, since the creation and deletion of information is as yet insufficiently researched. We speak of *Solid Frame Paradigm* as opposed to normal Paradigm in order to highlight this restriction.

2.6 Hierarchy of Paradigm

Paradigm uses separation of concern in order to model concurrent processes. The concerns are separated into five parts:

- The detailed STD shows how the model would work if no restrictions would be enforced upon it. The actions in this STD **enforce** changes in the set of currently entered traps.
- Traps show which goals have been reached in the current subprocess. A change in a trap **allows** global state changes since the current subprocess has achieved a certain goal.
- Global states show which phase the model is currently in and **allows** certain detailed actions. This indirectly **forbids** all the other detailed actions. It also shows which goals (traps) need to be reached in order to continue into another phase.

- Detailed manager transitions **alter** the current global state of an employee model, but are **restricted** in their actions by the traps.
- Change clauses are unconditional changes to the state of the model, **enforced** by detailed (manager) transitions. These changes can be used in order to **migrate** the complete model.

From this list, we can obtain the hierarchy within Paradigm: detailed actions enforce trap changes, traps allow global transitions, which in turn restrict the detailed transitions, which yields a triangular hierarchy within a single component (vertical consistency). Meanwhile, external manager transitions are both restricted by the traps and enforce changes to the global states (horizontal consistency). Finally, change clauses can alter the complete dynamics of the model (third-dimension consistency). This is all shown in Figure 2.4.

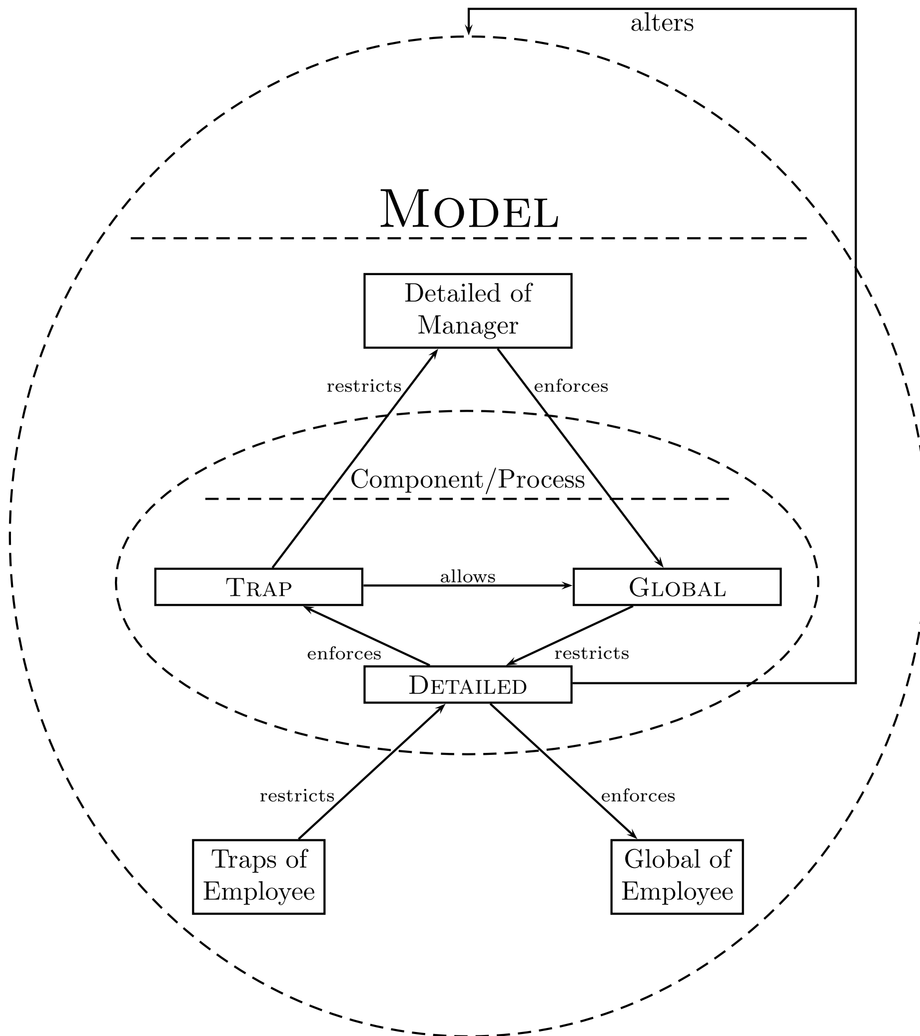


Figure 2.4: Hierarchy of Paradigm

Finally, it needs to be noted that connecting traps (see [3]) are not required in this view. If each state in a trap is also contained in a subprocess, that trap is said to be *connecting* to that subprocess. If a trap is not connecting to a certain subprocess, but it is used as required trap to change the current subprocess to that subprocess by a consistency rule, it might occur that upon entering the new subprocess the current detailed state is not “expected” to occur by the modeler. This could lead to unforeseen —and thus inconsistent— situations. Therefore, the connectivity of traps can be used in order to check the model for inconsistent situations by simply finding out whether the traps in the consistency rules are connecting to the resultant subprocesses; it does not ensure the model is well-formed, but does ensure it is not if non-connecting traps are used by consistency rules.

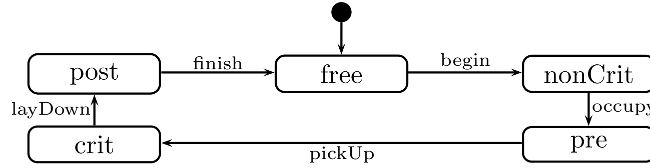
However, we will assume the models we translate into UML are well-formed, and thus traps which are not connecting to resultant subprocesses will not be used in the consistency rules. Since testing for connectivity of traps requires subprocesses to have information about the states they contain, a piece of information not otherwise relevant, this explanation simplified this by removing that information.

3 UML Modeling of Paradigm without reconfiguration

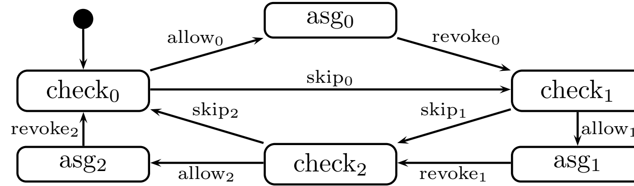
In this section, a UML component will be constructed which can be used to model reconfigurationless Paradigm models. This will be done by first explaining aspects of Paradigm and then determining a method to handle these interactions in UML and the demands on the model made by each of those aspects. The UML component will use a package of classes modeling a standard statemachine, which has states and transitions. States can have a nested behaviour of their own (i.e. another statemachine); the transitions have triggers, guards and activities. The Worker-Scheduler model from [3, 2] will be used both to explain each Paradigm aspect being handled and to show the complete UML solution. The example uses 4 instances of components, three Worker components (named Worker_0 through Worker_2), each of which desire exclusive access to a critical section in order to enter the state crit, and a Scheduler component which determines which Worker actually has permission to enter the critical section.

3.1 Detailed behaviour

A Paradigm model has any number of components, each of which contains exactly one detailed process, describing its behaviour at a detailed level. It uses the notions of state and transition to describe the component's behaviour, ensuring the component is always in exactly one state and allowing the state to change only by firing a transition. To fire, a transition requires the current state to be the source-state and, after firing the transition, the component enters the target-state. In the case of the Worker-Scheduler model, this detailed process is shown in Figure 3.1. This figure resembles the UML model closely, although the triggers, guards and activities of the model are not yet determined. In Paradigm, the method of triggering a detailed transition remains undefined due



Detailed Worker_i process ($i \in \{0, 1, 2\}$)



Detailed Scheduler process

Figure 3.1: Detailed behaviour of the Worker_i and Scheduler components

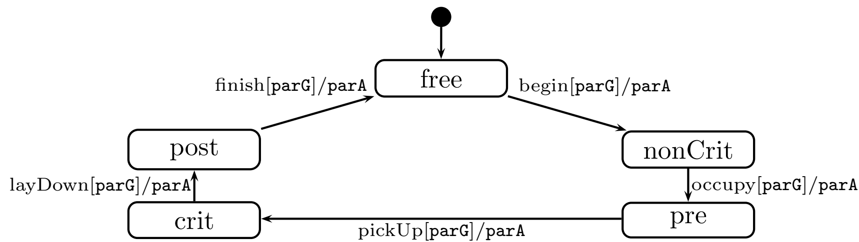


Figure 3.2: UML Model of the Worker component

to a lack of time in a Paradigm model; a component can not "wait" within a state unless all transitions of that component are currently forbidden for some reason. At any time, if any number of transitions are allowed, one of them always fires, atomically and immediately. Therefore, in Paradigm, all transitions are considered triggered at all time, and a trigger definition is not required. On the other hand, the UML STD model is not "timeless", so a trigger definition is required. We will use the action of a detailed transition in the Paradigm model as the trigger of a transition in the UML model. In Figure 3.2 we show a UML model of the worker component, of which the guards and activities have been shown as `parG` and `parA`, respectively. These will be handled in more detail in Sections 3.2 through 3.4.

3.2 Subprocesses

The detailed models are relatively simple to translate, since the Paradigm model and the UML model are nearly the same, but Paradigm does impose a few additional constraints on the transitions. The first of the constraints to be modeled are the subprocesses and partitions. Paradigm components have one or more partitions, each of which has one or more subprocesses and exactly one current subprocess. Each subprocess has one or more traps, which will be handled in Section 3.3. A detailed transition can fire only if each current subprocess contains that particular transition. Using this constraint, we can prevent certain behaviour from occurring. E.g., in the Worker model, we can define a partition (named Critical Section Management, or CSM) in such a way that only one of the subprocesses (InCS) allows the transition from the pre to the crit state. This partition will then be controlled by the Scheduler in some way (see Section 3.4) such that InCS is a Worker's current subprocess of CSM iff the Scheduler is currently allowing that Worker to enter the critical section.

The subprocesses of the partition CSM are shown in Figure 3.3. The InCS subprocess allows the Worker to enter the critical section exactly once, so it needs to return the permission in order to continue. The OutCS subprocess allows all transitions except where it deals with the crit state, and the OutCSBlock equals the OutCS subprocess, but doesn't allow the transition from nonCrit to pre. This allows the Scheduler to determine whether the Worker is ready to enter the critical section, without requiring the worker to suspend most of it's normal behaviour.

These restrictions to our component are handled in UML by the guards on the transitions. They allow the transition to fire if all of the current subprocesses

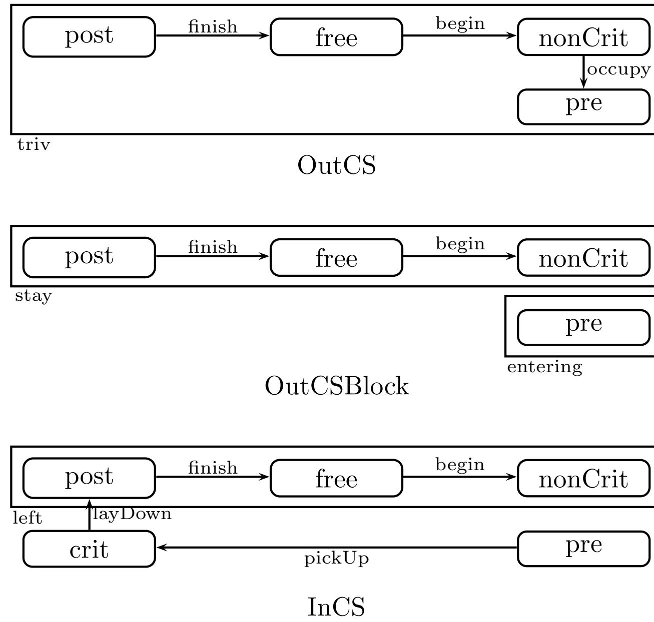


Figure 3.3: Subprocesses of the partition CSM of the Worker process

allow it. This requires the component to contain a model of each partition, keeping track of the subprocesses, the current subprocess and the transitions they allow. These partitions models will themselves be statemachines; the reason for this choice and how these STDs can be obtained will be shown in Section 3.3. For now, only the communication between partitions and the detailed STD is needed, since it shows how the restrictions imposed by the subprocesses are obtained by the detailed statemachine. Partitions have a separate thread in the component, which the detailed STD can communicate with through a port. Figure 3.4 shows the interaction between a detailed process and a partition port; the detailed process asks the partition whether the transition is allowed in the current subprocess and acts depending on the reply. This interaction acts

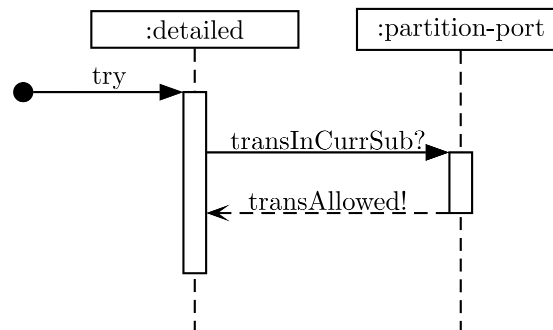


Figure 3.4: The sequence diagram with respect to a detailed transition

as a guard on each of the detailed transitions (`parG` in Figure 3.2), since the transition should not fire if it is not allowed by the current subprocess of the partition. If the model has multiple partitions, multiple asynchronous messages can be sent to all partitions, although the detailed transition needs to wait for each partition to answer before handling the replies. In later sections, we will expand on this diagram in order to show the increasing complexity of the guards, activities and triggers of both detailed and global STD's.

3.3 Partitions and Traps

As mentioned in Section 3.2, Paradigm allows the currently prescribed set of constraints on a component, in the form of the current subprocess, to be altered by another component. The component altering these subprocesses is called the Manager; the component being changed is named the Employee.

In the Worker-Scheduler model, the Scheduler is the manager of all of the Workers. It is capable of altering the current subprocess of the partition CSM, but to do so, it needs to be able to determine whether the current subprocess of a Worker should actually be altered. For example, if the Worker is in the state crit and its current subprocess is InCS, the Scheduler should not change the subprocess to OutCS, since the goal of the InCS subprocess has not yet been achieved in such a way that the models consistency allows the change. Paradigm models these goals in the form of traps; a trap is a subset of states which, once entered, can not be left until the current subprocess changes. This ensures that the trap information received by the manager doesn't change due to detailed transitions of the employee. In Figure 3.3 we can see the InCS subprocess has one relevant trap: left. Once the process has entered this trap, the Scheduler can alter the current subprocess to OutCS, since the stated goal of InCS, entering and leaving the critical section, has been achieved.

In Paradigm, each partition defines a global behaviour, which acts as a statemachine. The transitions of this global process are labeled with the traps the subprocess needs to have entered in order to be able to fire that transition. Figure 3.5 shows this statemachine for the Worker models (the model has pseudo start states; in the case of Worker_0, OutCSBlock is the start state, OutCS is the start state in the case of the other Workers).

In UML, these partitions can be modeled using another instance of the statemachine package. The guards and the triggers of the transitions will be handled in Section 3.4 and the activities can be left empty. These partitions also need to keep track of the transitions allowed by the subprocesses to handle the guards of the detailed transitions.

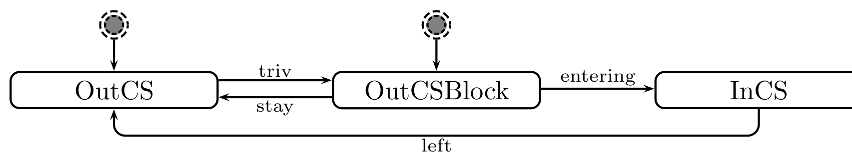


Figure 3.5: Global behaviour of the $Worker_i$ process at the level of Partition CMS, $i \in \{0, 1, 2\}$

The traps interaction can be considered as a separate behaviour of the subprocesses, and can thus be handled by a nested statemachine, in which the traps are the states. Since each subprocess contains at least the trivial trap (a trap containing all states in the subprocess), we can use this trap as the start state for each of the trap-statemachines. The UML model for the Worker component is given in Figure 3.6 (once again with pseudo start states). The transitions in these statemachines need no activities and the guards require the current detailed state to be in the trap, meaning each trap-state needs to keep track of which detailed states are contained within the trap. The triggers of these trap-transitions are a bit more complicated. Since each detailed transition could change the trap information, each detailed transition needs to trigger all of the trap-statemachine transitions. Also, whenever the subprocess is entered, we also enter the triv trap, but this does not need to be the innermost trap. This means we also need to trigger all transitions whenever the subprocess is altered. To handle either of these cases, we define a use case —cascade— which recursively attempts to trigger all of the transitions of the current state of the trap-statemachine. This use case is triggered by the activity part of each detailed transition, as well as by the entry activity of the subprocess state. Figure 3.7 shows the extended version of the sequence diagram of Figure 3.4; it shows the cascade use case is called upon if the transition is allowed.

The only problem with the cascade method arises when considering partially overlapping traps; if we have entered a trap, we can not “go back” in the STD to also enter an overlapping trap. But, since the intersection of these two traps is a trap as well, we can solve this problem by modeling a new trap into the subprocess. An example of this is modeled in Section 4.5.

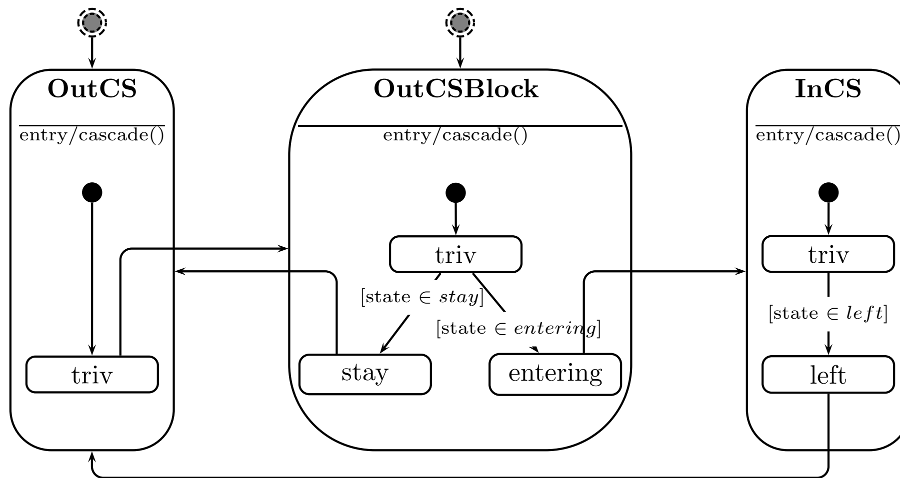


Figure 3.6: The state machine belonging to the global behaviour of $Worker_i(CSM)$

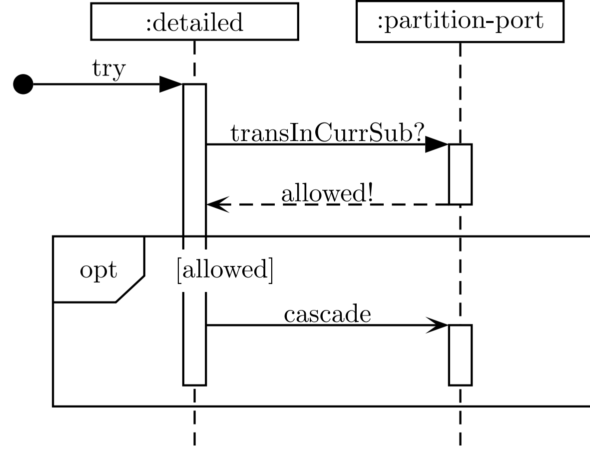


Figure 3.7: The sequence diagram with respect to a detailed transition, with the cascade function included

3.4 Component Interaction

As mentioned earlier, Paradigm processes are capable of interacting. This interaction is modeled in Paradigm by the consistency rules; the rules for the Worker models are:

$$\begin{array}{lrcl}
Worker_i: & free & \xrightarrow{begin} & nonCrit \\
Worker_i: & nonCrit & \xrightarrow{occupy} & pre \\
Worker_i: & pre & \xrightarrow{pickUp} & crit \\
Worker_i: & crit & \xrightarrow{layDown} & post \\
Worker_i: & post & \xrightarrow{finish} & free
\end{array}$$

The rules mention the process, the source state and the target state of each transition. A second form of consistency rule has the same detailed part, but is followed by an asterisk, and then one or more global (employee) steps, separated by a comma. An employee step shows the component, the partition, the subprocess which needs to be the current subprocess, the trap that needs to have been entered and the next current (target) subprocess. The consistency rules for the Scheduler component (using $i+1=0$ if $i=2$) are:

$$\begin{array}{llrcl}
\multicolumn{2}{l}{Scheduler:} & check_i & \xrightarrow{allow_i} & asg_i \\
* & Worker_i(CSM): & OutCSBlock & \xrightarrow{entering} & InCS \\
\multicolumn{2}{l}{Scheduler:} & asg_i & \xrightarrow{revoke_i} & check_{i+1} \\
* & Worker_i(CSM): & InCS & \xrightarrow{left} & OutCS, \\
 & Worker_{i+1}(CSM): & OutCS & \xrightarrow{triv} & OutCSBlock
\end{array}$$

$$\begin{array}{llcl} Scheduler: & check_i & \xrightarrow{skip_i} & check_{i+1} \\ * \quad Worker_i(CSM): & OutCSBlock & \xrightarrow{stay} & OutCS, \\ \quad Worker_{i+1}(CSM): & OutCS & \xrightarrow{triv} & OutCSBlock \end{array}$$

These rules clearly show a connection between a detailed rule (e.g. Scheduler changing the current state from check_i to asg_i) and one or more global rules of any component (e.g. Worker_i changing the current subprocess of partition CSM to subprocess InCS if the current subprocess is OutCSBlock and trap entering has been entered).

In the UML model, we need to translate these consistency rules into triggers, guards and activities. The guards on the detailed transitions of a manager component need to have an additional part requiring each of the employees to have the correct current subprocess in the prescribed partition, as well as having entered the correct trap of that subprocess. After such conditions have been verified for each of the employees, the transition is executed and, by executing, its activity triggers the corresponding global transitions of the employees. Since the global transitions are only triggered when the traps have already been determined to be the correct, the global transitions need no guards for this.

We also need to add ports to the UML models in order to handle the inter-component communication. We need two types; one for each of the partitions and a mirrored one for the detailed manager statemachines. The sequence diagram of these ports is shown in Figure 3.8; whenever a transition is attempted, it first determines if all of its own subprocesses allow the transition. If they do, it requests of all of its employees whether they are in the correct subprocess and have entered the trap of that subprocess needed to execute the transition. The employee's global statemachine port sends back an answer and waits for either an execute or a cancel message to return from the manager. After the manager has received all employees' answers, it determines if all of the employees are in the correct subprocess and trap and if so, it executes the transition, simultaneously sending a trigger to all of the employees. If one of the employees sends a "not correct" message, the manager immediately sends a cancel message to all of the employees and does not fire its own transition. When an employee receives a cancel message, it knows it can stop waiting and starts accepting other manager requests.

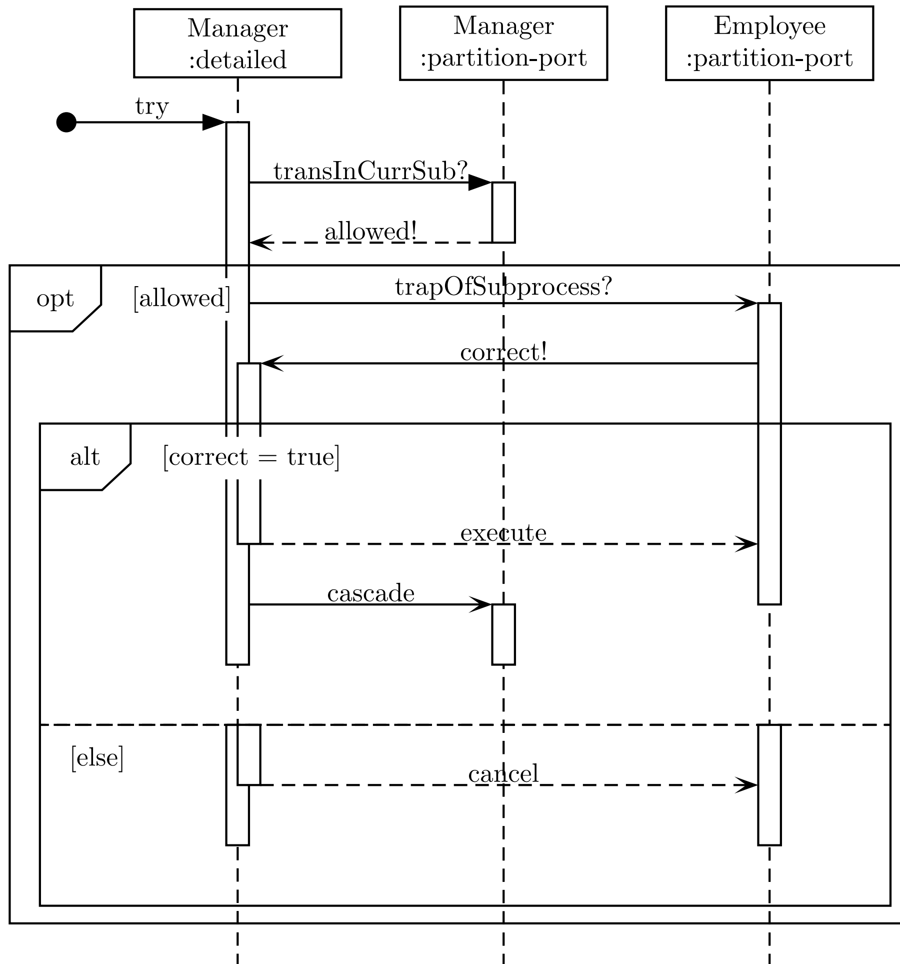


Figure 3.8: The communication sequence diagram of a manager transition

4 Foreseen Model Adaptation

This section will handle the translation of adaptation in Paradigm into the UML model given in Section 3. First, the method used by Paradigm to handle adaptation will be explained and illustrated through an example. The example used will be a Paradigm solution to the producer-consumer problem with a variable buffer size. The size of the buffer will grow when needed through self-adaptation, while the producer and consumer will be able to continue their behaviour during these migrations. After having explained adaptation in Paradigm, the UML model from Section 3 will be altered in order to include this form of behaviour.

In the discussion of adaptation, often we will use the terms PM-constituents and UM-constituents, for Paradigm Model constituents and UML Model constituents. These terms encompass the parts of the respective models that can get added, altered or removed during the adaptation. The PM-constituents are states, transitions, subprocesses, consistency rules, change clauses and the contents of partitions (including traps). UM-constituents are states, transition, triggers, guards, activities and nested statemachines (the trap-statemachines embedded in the states of the partition statemachines). Components and partitions are themselves not constituents because Paradigm does not yet allow the number of components and partitions in its product space to be altered. However, if that was allowed, these would be added to the lists.

4.1 McPal

In order to make Paradigm models capable of adaptation, they need to contain at least one component capable of adding and removing PM-constituents. Paradigm allows any component to do this, but most models contain a specialized component, McPal (an abbreviation of Managing changing Processes at leisure), to handle these actions. McPal has a generic form, as seen in Figure 4.1, and can be split up into four distinct stages; addition, startup, execution and stabilization. These stages use self-management in order to ensure the consistency of the McPal model throughout the migration. This self management occurs on the Migr partitions, as shown in Figure 4.2.

In the addition and stabilization stages, the behaviour of the model is changed. This is done using a change clause; a special Paradigm construct capable of altering an internal variable of the model. The current behaviour specification of the model as described by the consistency rules set (CRS) is one of these internal variables, and can thus be altered using a change clause.

The addition stage occurs between the Observing and NewRuleSet states from McPal. It first acquires the models constituents during and after the

Figure 4.1: A generic McPal model allowing JIT-modeling and self-adaptation

migration, which it then adds to the model using a change clause. This implicitly adds the new PM-constituents needed during the migration. In Paradigm, the acquisition of the new PM-constituents is done either through the addition of new rules by an outside modeler (JIT-modeling) or through a pre-defined step (self adaptation). This section will restrict the discussion to the latter type, of which McPal can have any number (as shown in Figure 4.1 by the asterisk), each corresponding to a different pre-defined migration.

The transition into NewRuleSet extends the rule set of the model to reflect the consistency requirements during the migration, which means it should be able to behave according to the current behaviour, the target behaviour and any behaviour in between those two. This is done using a change clause, making the CRS equal to the union of CRS, the target behaviour Crs_{toBe} and the behaviour needed in order to ensure a consistent migration Crs_{migr}. This makes all of the new rules known, while none of them can be used yet; since changeclauses can not alter the current subprocess(es), the current behaviour as described by those subprocesses is not altered either and thus the 'old' behaviour still holds. Therefore the changes to the model only influence the components behaviour after the migration is properly started. The general form of a consistency rule allowing for this stage is

$$\begin{array}{llcl} McPal: & Observing & \xrightarrow{foreseen_i} & NewRuleSet \\ * & McPal[CRS & := & CRS \cup Crs_{migr} \cup Crs_{toBe}] \end{array}$$

In the startup stage, from NewRuleSet to StartMigr, the migration is started by changing the current subprocess of the Migr partition of McPal to a subprocess determined by the Crs_{migr}. This new subprocess was added to the model during the addition stage and contains all of the new rules required to propagate the migration, including the rules which manage the global transitions of other components into new subprocesses during which the new states and transitions added in the addition stage can be used. The generic form of this rule is

$$\begin{array}{lrcl} McPal: & NewRuleSet & \xrightarrow{start} & StartMigr \\ * \quad McPal(Migr): & StablePhase & \xrightarrow{ready} & ? \end{array}$$

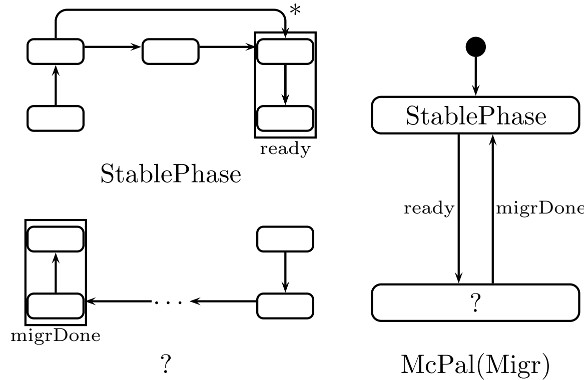


Figure 4.2: The generic form of the McPal Migr partition

where the question mark denotes the new subprocess.

After this, we enter the execution stage, where the actual migration takes place. During this stage, the behaviour of the components is altered in some consistent way, as determined by the rules in Crs_{migr}, towards the new behaviour. The rules for this step can be as simple or complex as the modeler decides; there is no generic form for these other than the usual form of the consistency rules themselves.

Once the execution stage of the migration is finished, McPal enters the state Content, starting the stabilization stage, which acts as the reverse of the addition and startup stages, restricting the CRS to Crs_{toBe} and implicitly removing the states and transitions no longer needed. This stage is generically determined by the rule

$$\begin{array}{lrcl} McPal: & Content & \xrightarrow{stabilize} & Observing \\ * \quad McPal(Migr): & ? & \xrightarrow{migrDone} & StablePhase, \\ & McPal[CRS & := & Crs_{toBe}] \end{array}$$

after which McPal has once again reached its initial state and can execute new migrations as desired.

4.2 A Producer Consumer for Paradigm

The producer-consumer (or bounded-buffer) problem is a well known problem (see e.g. [6]), in which a producer produces an item, delivers it into a buffer and starts producing again. The consumer retrieves an item from the buffer and consumes it, after which it retrieves another item. In a stable situation, the buffer has a fixed size, meaning the producer can add items iff the buffer is not full and the consumer can remove items iff it is not empty.

The example consist of three processes given in Figure 4.3, with a buffer that allows a maximum of *size* items. Buffer should not be able to take any actions unless Producer or Consumer have allowed it to, suggesting a partition (Stable)

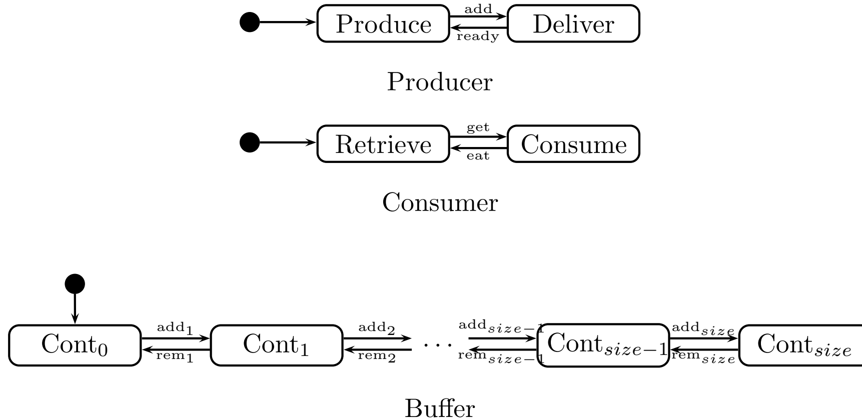


Figure 4.3: Detailed Producer, Consumer and Buffer

of Buffer with three subprocesses, one in which no transitions are present (Idle), one for adding (Fill) and one for removing (Empty) an item. The current subprocess should be altered from Idle to Fill iff the producer delivers an item. This management occurs when the transition from Deliver to Produce is taken and the buffer isn't full, suggesting a trap (notFull, see Figure 4.4) containing all states except Cont_{size}:

$$\begin{array}{lcl} Producer: & Deliver \xrightarrow{ready} & Produce \\ * \quad Buffer(Stable): & Idle_{size} \xrightarrow{notFull} & Fill_{size} \end{array}$$

The model should also ensure only one item is added to the buffer while in subprocess Fill. This is ensured by delegation: if each transition of the buffer changes the current subprocess to Idle, the buffer is prevented from continuing to take any more transitions to another state without first having explicit permission to do so:

$$\begin{array}{lcl} Buffer: & Cont_{i-1} \xrightarrow{add_i} & Cont_i \\ * \quad Buffer(Stable): & Fill_{size} \xrightarrow{triv} & Idle_{size} \end{array}$$

The removal of an item from the buffer by the consumer is modeled similarly:

$$\begin{array}{lcl} Consumer: & Retrieve \xrightarrow{get} & Consume \\ * \quad Buffer(Stable): & Idle_{size} \xrightarrow{notEmpty} & Empty_{size} \\ Buffer: & Cont_i \xrightarrow{rem_i} & Cont_{i-1} \\ * \quad Buffer(Stable): & Empty_{size} \xrightarrow{triv} & Idle_{size} \end{array}$$

Add to these the non-managing transitions of Producer and Consumer

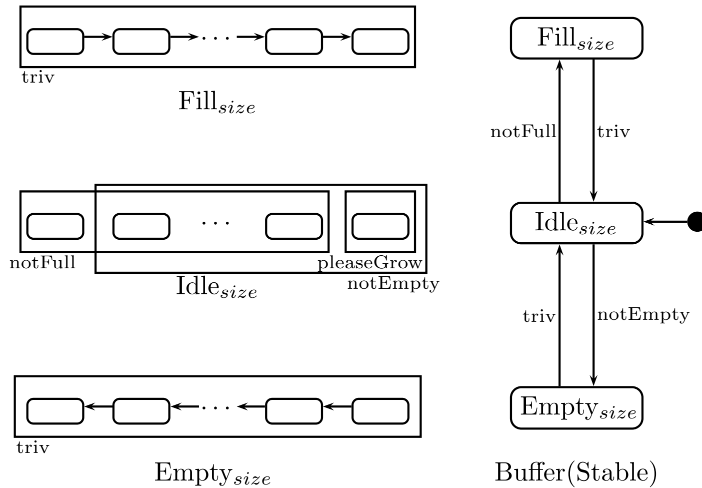


Figure 4.4: The generic form of partition Stable of Buffer

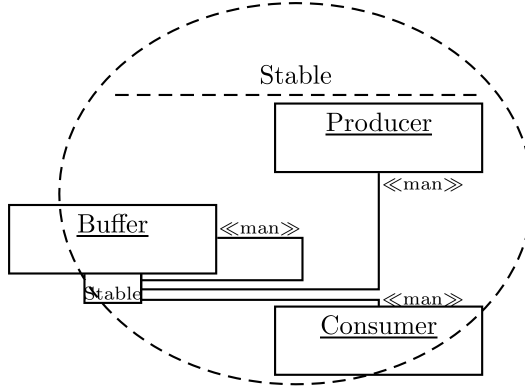


Figure 4.5: Collaboration of the partition Stable

$$\begin{array}{lrcl} Producer: & Produce & \xrightarrow{start} & Deliver \\ Consumer: & Consume & \xrightarrow{eat} & Retrieve \end{array}$$

and the model is complete. Figure 4.4 shows the generic form of the partition Stable of Buffer with all the relevant traps and Figure 4.5 shows the collaboration shown by the components in the model; Buffer is managed by Producer, Consumer and by itself, which alter the current subprocesses of the partition Stable.

4.3 Self-adaptation of the Producer-Consumer

In this section, the producer-consumer model of Section 4.2 will be extended in order to allow it to add a new state to the buffer using self-adaptation. As stated in Section 4.1, McPal handles this adaptation through the use of the transition from Observing to NewRuleSet. That generic transition changes the CRS variable in order to allow the migration to occur, but does so unconditionally. Starting the migration only if the buffer is actually full requires testing the Buffer for fullness, which suggests a trap in the Stable partition; the trap pleaseGrow from Figure 4.4 is used for this purpose. By adding a global transition to the Stable partition going from Idle_{size} back to Idle_{size} we can ensure the migration is only started if the Buffer is currently in trap pleaseGrow without actually changing the current behaviour of Buffer. The rule

$$\begin{array}{lrcl} McPal: & Observing & \xrightarrow{growBuffer} & NewRuleSet \\ * \quad Buffer(Stable): & Idle_{size} & \xrightarrow{pleaseGrow} & Idle_{size}, \\ & McPal[CRS & := & CRS \cup Crs_{grow} \\ & & & \cup Crs_{size+1}] \end{array}$$

models the addition stage in such a way. In this rule, Crs_{size+1} is known; it is another parametrization of the behaviour as stated in Section 4.2. It is also

the desired result of the migration and as such takes the place of the Crs_{toBe} variable in the generic form of the transition. Similarly, the Crs_{grow} variable takes the place of Crs_{migr}. It contains the rules needed in order to ensure a consistent migration, as determined by the startup, execution and stabilization stages. The first of these stages changes McPal's current subprocess of the partition Migr. For this migration, the unknown subprocess from Figure 4.2 has been named Enlarge, see Figure 4.6, which is the only information we need in order to determine the rule for the startUp stage. According to its generic form, this is

$$\begin{array}{llcl} McPal: & NewRuleSet & \xrightarrow{start} & StartMigr \\ * \quad McPal(Migr): & StablePhase & \xrightarrow{ready} & Enlarge \end{array}$$

The execution stage of the migration is relatively simple; we merely need to change the current subprocess of the Stable partition of Buffer to allow for the larger size. This is done by changing the current subprocess from Idle_{size} to Idle_{size+1}, which was not strictly known prior to the migration, but implicitly added by the addition stage:

$$\begin{array}{llcl} McPal: & StartMigr & \xrightarrow{extend} & Content \\ * \quad Buffer(Stable): & Idle_{size} & \xrightarrow{triv} & Idle_{size+1} \end{array}$$

Finally, the stabilization stage is similar to the generic form; with the exception of the increase in the size variable, which is required in order to keep the model internally consistent:

$$\begin{array}{lrcl} McPal: & Content & \xrightarrow{stabilize} & Observing \\ * \quad McPal(Migr): & Enlarge & \xrightarrow{migrDone} & StablePhase, \\ & McPal[CRS & := & Crs_{size+1}], \\ & McPal[size & := & size+1] \end{array}$$

These 3 rules combined make up the Crs_{grow} set. The collaboration as shown with this migration included is shown in Figure 4.7; it shows two different collaborations; the Stable collaboration allows the normal coordination as shown in Figure 4.5, while the Self Adaptation collaboration shows the architecture of the coordination of the migration.

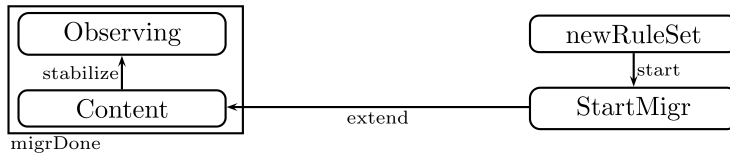


Figure 4.6: Subprocess Enlarge of Migr partition of McPal

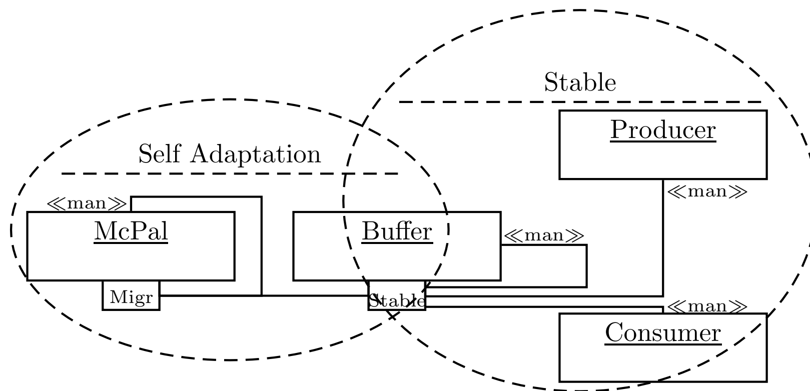


Figure 4.7: Collaboration of the components including the self-adaptation

4.4 Adaptation in UML

To model an adaptation in the McPal component, we need to model each of the 4 stages. Of these, the startup and execution stages need no additional modeling; the component as given in Section 3 is already capable of modeling these stages. However, the addition and stabilization stages use change clauses to alter internal variables and to notify all components of the states, transitions, subprocesses and traps to be altered or removed; these change clauses are not yet present in the UML model.

Since change clauses are unconditional changes of internal variables, it makes sense to model the change clauses as activities of the detailed transitions to which they belong, i.e., when, in Paradigm, McPal uses a change clause to alter the size variable upon taking the Content to Observing transition, the corresponding transition in the McPal UML component should have the same alteration to size as part of its activity.

Whereas this is enough for a simple change clause, the addition stage also implicitly adds the missing PM-constituents. We can model this in UML by creating a use case in which each of the components that needs to alter its behaviour is an actor. The key step in this use case is the creation of new instances of the new UM-constituents. These new UM-constituents will not yet influence the execution threads since the current subprocesses restrict the enabled transitions to those contained within their definition, which does not include them. This also means the order in which the new UM-constituents are created is irrelevant to the consistency of the model and the component can continue its normal behaviour during this stage. The change of the current subprocess towards a subprocess which does allow the newly added behaviour to occur will not be allowed yet for the same reasons, ultimately depending on the McPal component to change its own current subprocess during the startup stage. McPal should therefore wait for the use case to be completely finished, which ensures the overall model remains consistent.

The stabilization stage works similarly, except instead of creating new UM-constituents, it removes the obsolete ones. Since this stage will not start until the model has adopted the new behaviour, the states which are removed do not

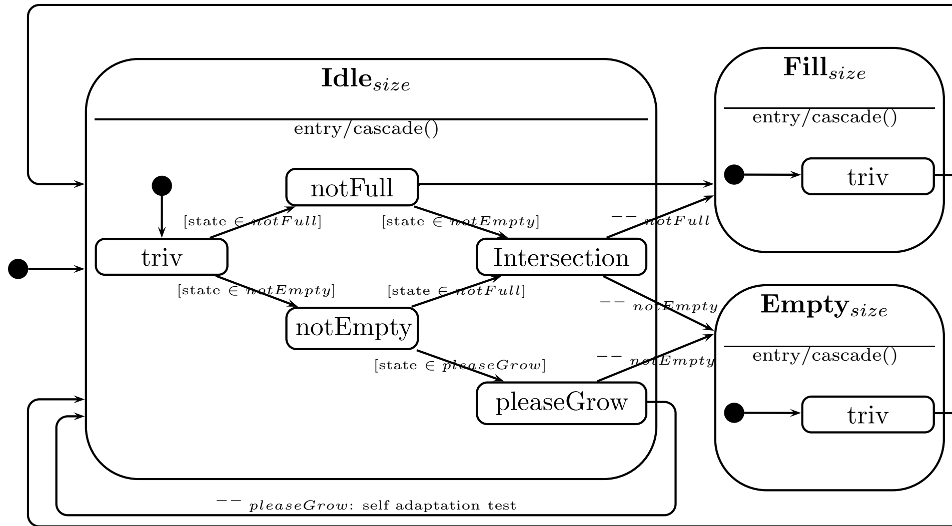


Figure 4.8: UML Component of partition Stable of the Producer-Consumer problem

create an inconsistent model either, assuming the original Paradigm migration was consistent.

4.5 Self-Adapting Producer Consumer in UML

The UML solution to the Stable partition of the model described in Sections 4.2 and 4.3 is shown in Figure 4.8. The trap Intersection was not included in the original Paradigm model, but has been added to the UML component in order to model the partially overlapping notFull and notEmpty traps, using the method mentioned in Section 3.3. While normally the guards on transitions leading into traps require that the state is in the trap, it suffices to set the guards for the transitions into Intersection to determine whether the states are in the overlapping trap not yet entered.

From Idle$_{size}$, four transitions exist that change the current subprocess. Three of these transitions did not have the trap from which they originate as the trap required to have been entered in the Paradigm model's consistency rule. This is because the traps from which they do originate are nested within the notFull or notEmpty traps; these transitions model special occurrences of global transitions constrained by traps, namely those in which the component has also entered another, smaller trap. Those transitions have been labeled with -- *notFull* and -- *notEmpty* respectively.

The transition leading from pleaseGrow into Idle does not actually alter the current subprocess. This transition is still present in the model since the self-adaptation steps in the Paradigm model uses this transition in order to test the model for its desire to grow, thus the UML model will fire this transitions when the self adaptation step is started.

The addition stage of a migration can be modeled in the UML McPal com-

ponent as a use case which distributes the new UM-constituents to the actor components. Figure 4.9 shows a McPal component in which this stage is added to the Observing to newRuleSet transition as an activity; it shows the addition is not finished until McPal is done distributing the new UM-constituents and has entered the newRuleSet state. The distribution itself is handled by an additional signal in the sequence diagram of Figure 3.8; it occurs within the alt-condition, since the transition needs to be allowed, but before McPal sends the Execute message. After each actor is done, they send a reply in order to signify to McPal the addition is done. Then, when all actors have send this message, the McPal thread continues.

In the example, growing the buffer will require a use case that adds the following UM-constituents:

- a new state in the detailed process: Cont_{size+1}
- three new subprocesses of the Buffer component in the Stable transition: Idle_{size+1}, Fill_{size+1} and Empty_{size+1}, as well as their traps and transitions
- a new subprocess of McPal in partition EvolMcPal: Enlarge
- a new detailed transition in McPal going from StartMigr to Content
- new guards on the newRuleSet to StartMigr and Content to Observing transitions of McPal in order to start and stabilize the migrations
- the stabilization use case itself

Because the new transitions can only be taken after this use case is done, these steps can be taken in any order. The stabilization use case is not an UM-constituent, but is nevertheless always defined in the addition use case, since it can only be determined which UM-constituents are obsolete after the migration itself is known. The UM-constituents to be removed in the example consists of all of the McPal parts added by the first use case as well as the Idle_{size}, Fill_{size} and Empty_{size} subprocess-states in the Stable partition statemachine. The stabilization use case is enforced upon the actors using the same message type as the addition stage.

The addition and stabilization use cases allow us to alter the behaviour of the models in a consistent manner. Through the alternating creation and deletion of UM-constituents, we can create even more complex migrations allowing complete reconfigurations of any UML model capable of being translated into

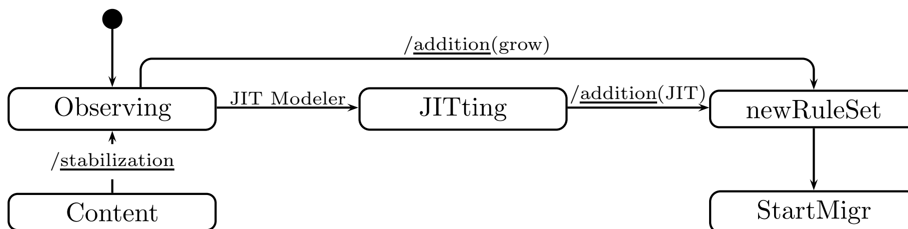


Figure 4.9: McPal UML statemachine before the self adaptation

a Paradigm model. Whereas the method in this Section shows this for foreseen migrations, unforeseen migrations are not handled yet. An example of an unforeseen migration will be handled in Section 5.

5 Unforeseen Adaptation in Paradigm and UML

This section shows that very little additional modeling is required in order to allow the McPal component presented in Section 4 to alter a model in an unforeseen manner. It will do this using the producer-consumer model from that section as an example. The migration will be based on the scenario described in Section 5.1, where the buffer will be shrunk to a smaller size.

5.1 Scenario

An outside modeler decides that the buffer size has grown too large, e.g. size larger than 15, and should be decreased to size 10. This could be done by creating a migration during which the buffer ignores attempts by the producer to deliver products, that is, it somehow does not allow the producer to change the current subprocess of its Stable subprocess. This would last until the consumer had consumed enough items from the buffer in order to reduce the number of items in the buffer to 10, after which the excess states of the buffer would be removed and the migration is done. However, the modeler decides that the producer should not be completely unable to deliver products during the migration; in fact, it is more preferable to have the migration last longer in order to prevent this from occurring. Finally, in order to prevent the Buffer from growing too large in the future, it is decided that maximum of size 15 needs to be maintained. To accomplish all of this, the following requirements to the migration have been set:

1. The buffer is not allowed to grow during the migration
2. The buffer will not shrink by more than one state simultaneously, which will always be Cont_{size}
3. A state can only be removed from the detailed buffer if it contains less than or equal to size−2 items
4. The execution stage will be done when the buffer reaches size 10
5. After the migration, the buffer is allowed to grow as before, except it can not grow larger than size 15

Most notably, the combination of requirements 2 and 3 ensures that the state being removed is never the current state of the buffer and that the producer can always deliver at least one item after a shrinking step has been undertaken. The producer can therefore continue delivering products during the migration, only needing to be delayed during the actual removal of a state or if the buffer is full.

5.2 Shrinking the Buffer

To model this behaviour, we first need to determine Crs_{toBe} and Crs_{migr}. The former is equal to the Crs_{size} from Section 4.2 where size = 10. To determine Crs_{migr}, we start by determining the PM-constituents that need to be added to the Buffer, Producer and Consumer during the addition stage.

The Idle_{size} subprocesses of Buffer will need to be temporarily changed to different subprocesses, which we'll denote as $\mathrm{Idle}_{migr1,size}$. The generic form

of these subprocesses is shown in Figure 5.1. When compared to Idle_{size}, it is evident that a pleaseShrink trap has been added to the subprocesses; it contains all the states in which shrinking can occur —all but Cont_{size-1} and Cont_{size}— and is used in order to test for requirement 3. The pleaseGrow trap however is not present in the subprocess, since growth is not allowed by requirement 1.

The detailed Buffer needs additional transitions tasked with self-management, because the existing transitions alter the current subprocess of the Stable partition to Idle_{size}, while we need them to alter it to $\text{Idle}_{migr1,size}$ during the migration. Since these transitions are not present in the Fill_{size} and Empty_{size} subprocesses, we also need additional subprocesses $\text{Fill}_{migr1,size}$ and $\text{Empty}_{migr1,size}$ which allow these new transitions to occur instead of the old ones. Visually, neither of these new subprocesses nor Buffer seem to differ from Figures 4.3 and 4.4, but the consistency rules do change:

$$\begin{array}{lrcl}
Buffer: & Cont_{i-1} & \xrightarrow{add} & Cont_i \\
* \quad Buffer(Stable): & Fill_{migr1,size} & \xrightarrow{triv} & Idle_{migr1,size} \\
Buffer: & Cont_i & \xrightarrow{remove} & Cont_{i-1} \\
* \quad Buffer(Stable): & Empty_{migr1,size} & \xrightarrow{triv} & Idle_{migr1,size}
\end{array}$$

Producer and Consumer need similar alterations. Their current subprocesses of the Migr partition will be changed from Stable to Migr_1, which differs from subprocess Stable only in that the transitions responsible for the management of Buffer are replaced by transition with these rules:

$$\begin{array}{lrcl}
Producer: & Deliver & \xrightarrow{ready} & Produce \\
* \quad Buffer(Stable): & Idle_{migr1,size} & \xrightarrow{notFull} & Fill_{migr1,size} \\
Consumer: & Retrieve & \xrightarrow{get} & Consume \\
* \quad Buffer(Stable): & Idle_{migr1,size} & \xrightarrow{notEmpty} & Empty_{migr1,size}
\end{array}$$

These four new rules ensure that the Producer and Consumer can continue producing/consuming even though the Buffer is currently in subprocess $\text{Idle}_{migr1,size}$ instead of Idle_{size}. Together with the new subprocesses, they determine the behaviour shown by Buffer, Producer and Consumer during the migration when not being shrunk. The new PM-constituents are implicitly distributed to the various processes by McPal during the addition stage. McPal also adds PM-constituents to itself in order to model the other three stages. The second of these, the execution stage, will be begun by McPal when entering a state Shrinking from the StartMigr state, see Figure 5.2, which also changes all the subprocesses of the Migr partitions to the new current subprocesses and the current subprocess of the Stable partition of Buffer to $\text{Idle}_{migr1,size}$:

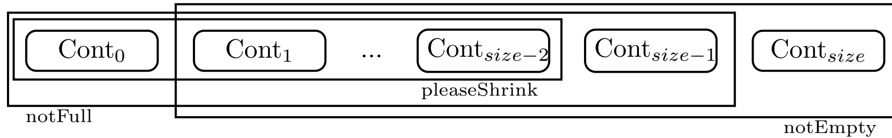


Figure 5.1: Generic form of subprocess $\text{Idle}_{migr1,size}$

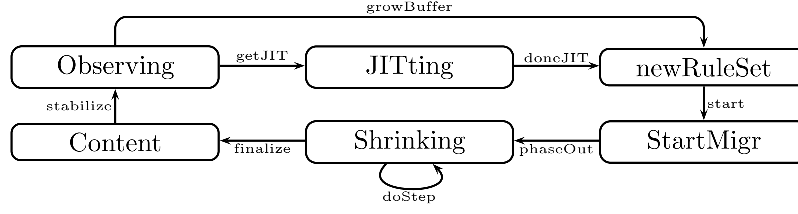


Figure 5.2: McPal during the migration

$$\begin{array}{llrcl} & McPal: & StartMigr & \xrightarrow{phaseOut} & Shrinking \\ * & Producer(Migr): & Stable & \xrightarrow{triv} & Migr_1, \\ & Consumer(Migr): & Stable & \xrightarrow{triv} & Migr_1, \\ & Buffer(Stable): & Idle_{size} & \xrightarrow{triv} & Idle_{migr1,size} \end{array}$$

The transition going from Shrinking to itself will manage a single shrinking step. It acts like the StartMigr to Content transition in the foreseen migration from Section 4.3, but in reverse; if Buffer is in trap pleaseShrink, it changes the current subprocess of the Stable partition of Buffer from $\mathrm{Idle}_{migr1,size}$ to $\mathrm{Idle}_{migr1,size-1}$ and reduces the size variable by one, ensuring requirement 2 and 3. To ensure this will not shrink the Buffer beyond size = 10, we need to restrict this rule to $n \geq 11$, and get:

$$\begin{array}{llrcl} & McPal: & Shrinking & \xrightarrow{doStep} & Shrinking \\ * & Buffer(Stable): & Idle_{migr1,n} & \xrightarrow{pleaseShrink} & Idle_{migr1,n-1}, \\ & & McPal[size & := & size - 1] \end{array}$$

When size 10 has been reached, the migration is done (requirement 4) and McPal needs to enter state Content to start the stabilization stage. We test for this situation by including a global transition which changes the current subprocess of Stable from $\mathrm{Idle}_{migr1,10}$ to Idle_{10}. In addition to the test for the size, it also resets the Buffer to its original behaviour, so the Producer and Consumer need to be reset as well in order to allow the correct consistency rules. Also, due to requirement 5, McPal needs to change the contents of Crs_{15} to prevent the foreseen growth migration from occurring when the buffer is at size 15. The new value of this variable will be $\mathrm{Crs}_{15,new}$:

$$Crs_{15,new} =$$

$$Crs_{15} - \left\{ \begin{array}{llrcl} & McPal: & Observing & \xrightarrow{growBuffer} & NewRuleSet \\ * & Buffer(Stable): & Idle_{15} & \xrightarrow{pleaseGrow} & Idle_{15}, \\ & & McPal[CRS & := & CRS \cup Crs_{grow} \\ & & & & \cup Crs_{16}] \end{array} \right\}$$

We can now define the following consistency rule to take care of all this:

$$\begin{array}{llrcl} McPal: & & Shrinking & \xrightarrow{finalize} & Content \\ * & Buffer(Stable): & Idle_{migr1,10} & \xrightarrow{triv} & Idle_{10}, \\ & Producer(Migr): & Migr_1 & \xrightarrow{triv} & Stable, \\ & Consumer(Migr): & Migr_1 & \xrightarrow{triv} & Stable, \\ & & McPal[Crs_{15} & := & Crs_{15,new}] \end{array}$$

Since we now know the detailed behavior of McPal during the migration, we can determine the contents of the subprocess Shrink, see Figure 5.3. The rules for the startup and stabilization stages are responsible for changing the current subprocess of the Migr partition of McPal to this new subprocess, and can be determined by the generic forms given in Section 4.1:

$$\begin{array}{llrcl} McPal: & & NewRuleSet & \xrightarrow{start} & StartMigr \\ * & McPal(Migr): & StablePhase & \xrightarrow{ready} & Shrink \end{array}$$

$$\begin{array}{llrcl} McPal: & & Content & \xrightarrow{stabilize} & Observing \\ * & McPal(Migr): & Shrink & \xrightarrow{migrDone} & StablePhase, \\ & & McPal[CRS & := & Crs_{toBe}] \end{array}$$

Finally, we can determine the rule for the addition stage. Crs_{toBe} was already determined, and Crs_{migr} contains all of the rules given above. These rules can now be added to the model by the rule

$$\begin{array}{llcl} McPal: & JITting & \xrightarrow{doneJIT} & NewRuleSet \\ * & McPal[CRS & := & CRS \cup Crs_{migr} \cup Crs_{toBe}] \end{array}$$

While this rule adds the new PM-constituents by changing the contents of CRS, it does not state how the contents of these Crs_{migr} and Crs_{toBe} is passed on to the model by the modeler. Paradigm handles the defining of these variables implicitly during the JITting stage. The next Section will describe a UML method for this implicit addition of variables to the model.

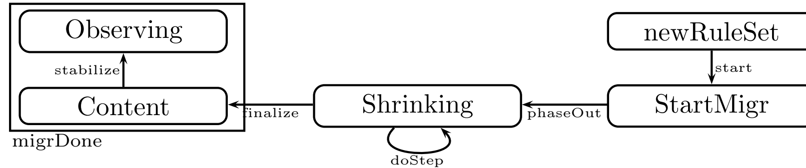


Figure 5.3: Subprocess Shrink of partition Migr of McPal

5.3 JIT modeling in UML

Paradigm handles complex migrations by changing its CRS set through change clauses, which can be modeled into UML as usecases, see Section 4.4. However, the critical difference between unforeseen (or Just In Time: JIT) and foreseen migrations is the addition of the contents of the Crs_{migr} and Crs_{toBe} sets. These are filled by implicitly allowing the modeler to input migrational and target behaviour during the JITting state of McPal. After this, the new PM-constituents are implemented in the model as usual, see Section 4.

To model this in the UML component, we need to add another type of behaviour to the generic component, a behaviour normally only shown by McPal, and then only during the JITting state. This can be done by adding another port to the model, which allows a external modeler to insert an usecase similar to the ones used in Section 4.5. The transition into JITting can only be triggered by the modeler —see Figure 4.9— and will then have an extended activity, that sends a request to the port. The request requires the modeler to enter an usecase in some way (e.g.: through text, programming code, a java-class etc). This usecase is then added to the McPal model in the usual fashion.

6 Conclusion, Related and Future Work

This Master's Thesis illustrates how Paradigm models work and how they can be described through the use of UML constructs. This grants UML modelers access to both Paradigm's separation of concern and its quiescence-less, unforeseen migrations towards other models. This not only makes modeling of complex, parallel processes easier, but it also solves UML's restriction to as-is models, removing the necessity of predicting future model changes.

6.1 Solution

The UML description was achieved using a complex interaction between multiple state machines. Each UML-style Paradigm component contains at least 3 state machines:

- Exactly one state machine for the detailed process.
- One state machine per partition for the global processes, using the subprocesses as states and subprocess changes as transitions.
- One state machine per subprocess, using the traps as states, and each possibility of entering a trap as its transitions.

The interaction between the state machines was achieved through two types of interaction. The first links the activity of a transition to the trigger of a transition of another state machine. For example, entering a new state in the detailed state machine could mean entering a new trap, therefore each detailed transition triggers all transitions of the trap state machines.

The second interaction is woven into the guards of state machines. These can obtain information from other state machines in order to determine whether a transition can be taken. E.g., the trap state machines use their guards in order to ensure that a state change occurs only if the resultant trap state contains the current detailed state.

These two interactions allow state machines to communicate, and were used for all interactions, both within a component —to keep it internally consistent— and between two components, in order to allow for manager-employee dynamics.

Finally, it was shown that a transition's activity could also be used to start a use case, which contains all steps required to execute a change clause. Therefore, foreseen on-the-fly migrations could be executed by the model, while adding a user-input port to the McPal component also allowed us to have the migrations be unforeseen. All these migrations are without quiescence, since they are based entirely on the underlying Paradigm model, and do not require additional constraints on the operation of the model.

6.2 Environment

An environment was created which utilizes the concepts within this Master's Thesis to visualize a Paradigm Model. The code of this environment is given in Appendix B, while Appendix A contains a short description of some of the more complex mechanisms in the environment. Within this environment, some Paradigm models were tested, including the Worker-Scheduler from [3, 2] (both

a stable and migrating version), the Producer-Consumer model from this paper and a hereto forth unpublished pipeline model which contains some highly complex dynamics.

6.3 Related Work

Recently, it has been shown that Paradigm (including migrations without quiescence) can be translated into Process Algebra (see [1, 4]). The trap state machine concept given in this thesis is reminiscent of how process algebra handles traps; both make a split between the detailed states and the traps they are contained in, only to have the two separate methods communicate afterwards.

ParADE (see [5]) is another environment used to visualize Paradigm. Build entirely in Java, it remains closer to the original concepts of Paradigm in its design. Whenever a process desires to take a step, it first finds out which of the transitions from the current state are allowed by each of its protocols, and then takes a random allowed transition. This differs from the environment created for this thesis, in that ParADE only takes allowed transition, while our environment triggers a random transition going from the current state and only then finds out whether the transition is allowed. It depends on the actual model, which method is more efficient; if we can expect an average of less than half of all transitions to be forbidden, the method employed by ParADE is probably less efficient.

ParADE has one advantage in efficiency though: it uses a protocol layer to keep track of the models consistency instead of direct communication between state machines. Whenever a detailed process makes a transition, it communicates all the relevant information (i.e. traps entered and subprocesses changed) to the protocol. This means that when a process determines which transitions are allowed, it only needs to request that information from the protocol. It also means no separate state machine needs to be maintained for the trap information. These two points cause a large reduction in the number of messages required, which increases the efficiency of ParADE significantly. In the current ParADE version however, the protocol layer allows a maximum of one manager to each protocol though, so co-management of an employee (as shown in the aforementioned pipeline model) can not be correctly modeled in ParADE.

6.4 Future Research

The environment shown in Appendix B can be used in order to visualize Paradigm Models, so it presents an easily accessible platform for those unfamiliar with paradigm and allows Paradigm to be used in a larger context, without losing information about the model in the translation to human understandable information when required. This opens up the possibility of cross-discipline research, ranging from using evolutionary algorithms to optimize multi-role business processes —e.g.: every actor explains to a modeler the required input and outputs of his/her work process, and the evolutionary algorithm can find the optimal trap and global processes to maximize the amount of parallel activities— to a new view of the steps within biological or chemical reactions, allowing for greater insight into the various parallel reactions taking place on the various active groups of a molecule during synthesis.

With the addition of an intuitive user interface, the tooling can also be used to allow business modelers to try out complex parallel processes without requiring a deeper understanding of Paradigm, while still allowing automatic testing and deadlock detection (using model-checking of the underlying process algebra). Also, this can allow a modeler to obtain an automated migration from one model to another, since the steps required to do so are easily identified when the from and to models are known; migrational patterns can be identified and applied more easily using new insights gained from the separation of concern in Paradigm. The solid frame requirement of Paradigm might lead to complication though, so some sort of creation and deletion of components and partitions, as well as a renaming operator, need to be more closely researched, both in Paradigm and in UML.

References

[1] S. Andova, L.P.J. Groenewegen, and E.P. de Vink. **Dynamic consistency in process algebra: From Paradigm to ACP** In P. Poizat, C. Canal and M. Sirjani, editors, *Proc FOCLASA '08*. 19pp. To appear in ENTCS, extended version submitted.

[2] L.P.J. Groenewegen and E.P. de Vink. **Dynamic System Adaptation by Constraint Orchestration**. In P.M.E. de Bra and J.J. van Wijk, editors, *CS-Report 08-29*, TU Eindhoven.

[3] L. Groenewegen and E. de Vink. **Evolution On-the-Fly with Paradigm**. In P. Ciancarini and H. Wiklicky, editors, *Proc. Coordination 2006*, pages 97-112. LNCS 4038, 2006.

[4] S. Andova, L.P.J. Groenewegen, J. Stafleu, and E.P. de Vink **Formalizing Adaptation On-the-Fly** Accepted for *Proc FOCLASA '09*, 2009.

[5] A.W. Stam, PhD Thesis **Model-Driven Software Development with Paradigm** To be published

[6] A. Silberschatz, P.B. Galvin, and G. Gagne. **Operating System Concepts, 6th edition: XP Version**. John Wiley & Sons, Inc, 2002. ISBN 0-47-126272-2, pages 108-109.

[7] L.P.J. Groenewegen, A.W. Stam, P.J. Toussaint and E.P. de Vink. **Paradigm as Organization-Oriented Coordination Language**. In L. van de Torre and G. Boella, editors, *Proc. CoOrg 2005*, pages 93-113. ENTCS 150(3), 2005.

[8] M. Fowler. **UML Distilled (3rd edition)**. Addison -Wesley, 2004. ISBN 0-321-19368-7

A Program

The concepts described in this paper have been used to create an environment in which Paradigm models can be simulated and visualised during execution. This appendix will describe the methods used to do so. This environment also acts as a proof of concept for the theories in this paper.

A.1 Technologies Used

In order to obtain an highly distributable environment, we chose to make the environment entirely web-based. This allows us to use the HTTP protocol to easily distribute the component code —written in JavaScript— among the clients , since all a client needs is a web-browser. This also allows the use of HTML elements for UI-widgets and the use of Dynamic HTML to have the page content respond to changes in the model's state, ranging from simple current state changes to subprocess changes. Finally, using the XMLHttpRequest object, we can send and receive short messages to the server in order to create communication channels. This method of client sided asynchronous messaging is commonly known as AJAX.

In order to obtain a high grade of parallelization, server-sided coding was kept to a minimum. The startup definitions of the components are stored in a SQL database, but distributed through a JavaScript file generated using PHP, so that only the definitions for the initialization needs to be centralized.

A.2 Component Communication

Usually, when two components need to communicate, a direct channel can be created between the two. However, using HTTP restricts the communication: clients can only send requests and server can only send replies to those requests. Since all component threads are distributed among the clients, this communication channel is too restrictive, because direct communication between components is not available. We therefore needed to store messages of clients to other clients on the server, which the other client can retrieve at leisure.

To do the latter, we created a **Polling** mechanism in JavaScript. This mechanism allows us to send a Polling Request —using AJAX— to the server once every x seconds, which asks the server whether any messages have been send to this component. Any such messages are then handled by a **Polling Handler**. In this case, the polling handlers are functions which take one of the messages from Figure 3.8 as input and calculate the correct message to send back. This response is stored in the exact same location on the server, allowing the original client to pick up the response using a poll of its own.

The server has a message table which acts as the location to store messages. This table contains 4 fields; the *message*, the message's *source*, its *target* (which the polling mechanism uses to identify the messages send to a component) and an identifier *id*. This last field contains an auto incremented identifier and acts as the primary key. After the first message in a communication has been received, this identifier is send (in the server's reply) to the communication's originator. From that moment on, it acts as a postal box number; if a participant α wants to send a message to participant β, it merely needs to send that message to a

particular box, after which the target field is set to β and the source state to α, so β's polling requests will yield the new message.

The source field is used to obtain anonymity in the channel: only the server knows the participants. When a message is send by one of the participants, the following query is used to store it:

```
UPDATE  messages
SET     target   = IF(source = '$source', target, source),
        source   = '$source',
        message  = '$message'
WHERE   id       = $id
AND     target   = '$source'
```

resulting in the target to be the other participant, without the requesting participant knowing who that is.

A typical message exchange (as based upon Figure 3.8) could be:

1. A manager Man tries to trigger a detailed transition and has checked this against its own subprocesses. It therefore sends a message to the server asking it whether employee Emp's partition Part is in subprocess Sub and in trap Trap. It also sends the target Subprocess SubTarget.

2. The server receives this message and stores it in the messages table in its database. It sends the unique identifier i as a response to the Man's request.

3. Man gets the response and tells the polling handler that when a message with id i comes in, it needs to be handled by the handleDetailedAlt function, which contains the code for the alt box in the sequence diagram.

4. After a while, Emp executes a polling request, asking the server for any messages.

5. The server selects all messages for Emp by checking the *target* field of its messages table and sends them as reply.

6. Emp receives the polling answer and receives the new message, and passes this on to the polling handler. The polling handler executes the handlePartitionOpt function, which checks whether partition Part's current subprocess is Sub and its trap is Trap. Since it is, Emp sends a *correct* message to id i. It also tells the poll handler that the next message with id i needs to be handled by the handlePartitionAlt function and that the new subprocess will be SubTarget if an *execute* message is received. Finally, it also sets the current partition on hold; any other messages that are received by the poll handler for this partition are skipped, automatically reoccurring at the next poll.

7. The server sets the message field of the record with id i to *correct* and switches the target and source fields. Since a response to clients is mandatory, but no useful information can be send, it also sends an empty response to Emp.

8. Man sends a Poll.

9. The server selects and returns all messages for Man.

10. Man receives a message with id i. The poll handler knows to pass this message on to handleDetailedAlt, which recognizes it as an *correct* message, and checks whether all messages of the employees have been received. Since this was the only employee, they are, and it sends an *execute* message to the record with id i. It then executes the transition and starts at the beginning with the next transition.

11. After sending a poll message and receiving the response from the server, Emp obtains the execute message and changes the current subprocess to SubTarget. It then re-enables the partition so that the poll messages that were previously ignored are once again handled in the normal way. Finally, it sends a *delete* message to the server for the message with id i

12. The server receives the *delete* message and deletes the record with id i

A.3 Making a JavaScript execution thread sleep

JavaScript is a single threaded language (per browser window, that is), which has its advantages and disadvantages. An advantage is that we do not have to plan for inconsistency in the model due to one thread having been cleared to take a transition and another thread to take another transition, both of which then occur (which should not be possible). A disadvantage is that having a thread sleep or wait causes the entire JavaScript execution to stall.

Nonetheless, having a STD wait for some signal is sometimes still required (e.g.: an manager waiting for each employee to send an answer to a request). Pseudo-multi-threading is available using the setTimeout and setInterval functions, after which thread control can be returned to the top level (allowing events to gain control over the thread). Then, after a certain amount of time has passed, an instruction predefined in the setTimeout/setInterval call is handled and thread control can be "continue" the earlier thread. However, this is not responsive enough for our purposes, since using this method the component can not act immediately upon receiving a message, but has to wait for the Timeout to occur. We therefore designed a different (but similar) waiting system.

To each STD, an array waitDefs of objects was added; each object containing an identifier, a function to execute when a wakeUp is called, and an object containing any number of arguments to pass to the function. A wait function was added which pushes new elements onto that array, and a wakeUp function which takes an identifier as argument and executes each of the functions within the waitDefs array of which the object's identifier equals the argument.

By cleverly choosing the identifiers and functions, we can simulate incoming messages and immediate responses. For example, a typical execution thread of a manager could look like this (see Figure 3.8 for the sequence diagram corresponding to this thread):

1. A manager tries to execute a transition which manages two employees. It therefore sends 2 trapOfSubprocess messages to the server.

2. The server replies with the identifiers i and j for the communication signals (see Section A.2).

3. The manager adds two waits to its detailed STD, one with id i and one with id j. As functions it passes the handleDetailedAlt function.

4. At the next poll, the manager receives a *correct* reply of one of the employees (corresponding to message i) and it calls wakeUp with the i as identifier. This calls the handleDetailedAlt function, which first checks whether the response was a *correct.* Since it is, the function checks whether all of the employees have responded. Since the STD is waiting (for a message with the identifier j), it apparently has not yet obtained all answers, so the function returns. If the response had been *incorrect*, it would have woken up all other waits and all of those functions would have send a *cancel* message, after which the STD was no longer waiting.

5. At the second poll, a *correct* message is received from the second employee, and the handleDetailedAlt is once again called. Since the STD is no longer waiting for any employees, all employees are send an execute message and the transition fires.

A.4 Creating a new model

This section will describe the databases and tables which contain the initialization definitions of the models. Each component of a model has an entry in the table models of database paradigmmodels, which consists of 6 fields, see Table A.1.

Field	Desciption
id	This field is auto-incremented and acts as primary key. It is not required for anything else
database	Refers to the database in which the definitions for the states and transitions are held. This database also contains the messages table as described in Section A.2.
component	Refers to the base name of the component
numIds	Contains the number of parametrizations, e.g.: Worker_i contains one parametrization
variables	Contains a variable description, which can be referenced to in state and transition definitions
display	a boolean stating whether this component should be displayed in the index page

Table A.1: Definition of the table models of Paradigmmodels

Two fields require additional explanation: the `numIds` field and the `variables` field. The first makes it possible to have an arbitrary number of components of a certain type, e.g.: there were three Worker components in section 3, that all worked the same. The `variables` field allows a single component to have multiple states and transitions, while only one definition exists, For example, the scheduler component needs an equal amount of *asg* states as there are Workers.

Both the `numIds` and `variables` values are inputed by the user when the component is started up, and the states and transitions tables described later in this section can use those values to determine not only the number of states and transitions, but also what the initial state needs to be.

Every model in the environment also has its own database, which contains 3 tables: messages, states and transitions. Messages was handled shortly in Section A.2, but requires no new actions when creating a new model database. We will explain the other two tables and their fields shortly. Table A.2 contains the definitions for the Transitions table.

Field	Desciption
id	This field is auto-incremented and acts as primary key. It is not required for anything else
compName	Contains the name of the component the transition belongs to
stateMachine	Contains the STD within the component the transition belongs to. If empty, the transition is considered to belong to the detailed STD, otherwise this field contains either the subprocess name (for trap STDs) or the partition name (for global STDs)
source	The source state
target	The target state
triggerSig	The signal for the transition
guard	The guard for the transition
activity	The activity of the transition

Table A.2: Definition of the table Transitions of a Paradigm Model

Three fields in this table need further explanation. The first is the *triggerSig* field. This field can have 3 types of values:

- The action label for detailed transitions
- The trap names for global transitions
- `cascade` for trap transitions

The STDs define a trigger function which takes a textual signal as their argument. If called, that function triggers each transition for which the trigger field equals the argument. This shows that the activities of the detailed transitions only need to trigger all transitions with `cascade` as trigger in order to cascade all trap STDs.

This is represented in the two possible values of the activity field:

- `sendTrigger('cascade')` for detailed transitions and trap transitions
- `changeSubprocess(this.from.name, this.to.name);` for global transitions

The traps and detailed transitions both trigger the cascades, just as required of the cascade function. The global transitions change the current subprocess, which requires four actions: changing the global state, disable the trap STD belonging to the previous current subprocess and enable the trap STD belonging to the new current subprocess, followed by a cascade call to that STD. The first action is implicitly taken by the transition, but the other actions need to be executed separately, which the line given takes care of.

Finally, the *guard* field also has three possible values:

- `paradigmGuard(this, '`*list of subprocesses*`', '`*optional list of employees*`');` for detailed transitions
- `getSTD(`*a detailed STD*`).isTrapped(`*a list of detailed state*`);` for trap transitions
- `getSTD(this.from.name).isCurrent(this.trigger);` for global transitions

The function paradigmGuard returns true iff each of the current subprocesses occur in the listOfSubprocesses list, thus evaluating the guard using the `eval` function, we get a simple true or false on whether the transition is allowed. If the optional list of employees is also defined, each of the employees is checked (as described in A.2), the paradigmGuard function returns false and the STD sleeps while waiting for responses of the employees. Iff all of these have send a correct message back, the transition will once again be triggered, but unconditionally, that is, the guard will not be tested.

Table A.3 shows the states table.

Field	Description
id	This field is auto-incremented and acts as primary key. Not required for anything else
stateName	Contains the name of the state
compName	Contains the component the state belongs to
stateMachine	Contains the STD within the component the state belongs to. If empty, the state is considered to belong to the detailed STD, otherwise this field contains either the subprocess name (for trap STDs) or the partition name (for global STDs)
initial	A boolean statement determining whether this state is the initial state, which can use the numIds and variables from the paradigmmodels table upon startup as arguments.

Table A.3: Definition of the table States of a Paradigm Model

A.5 Adaptation

The environment is also capable of on-the-fly migration as described in Sections 4 and 5. It was stated that the migrational information is added to the

model using the activity of the transitions into newRuleSet —the form of which was stated to be a use case—, but how the use case was executed was left undefined. In this environment, we chose to have the activity be evaluable JavaScript, evaluating it using the JavaScript `eval` function when the transition is taken. These activities contain the calls to the object constructors as required. If the use case calls for the creation of a new transition, but the source and/or target states do not yet exist, those states are first created. Similarly, STD's are automatically initiated if states are created which belong to a non-existing STD. Because of this, it is often enough to add only the transitions of the new STD's.

Foreseen migration can be included in the activity field of the database record of the transition, so modeling foreseen migration occurs within the model's definition itself, as required. Unforeseen migrations require a more flexible approach though, since we want the modeler to have complete control over the migration. To handle this, the activity of the transition into the JIT state of McPal (startJit) was set so it opens an upload file field, where the modeler can input a migration. After the modeler had uploaded a file, the guard on the transition into NewRuleSet (doneJit) will request the new definitions of the server, which has generated the required JavaScript from the uploaded file. The start and stabilize transitions are then created with the correct usecases included in their activities, and the rest of the migration can occur.

A.6 Multiple views

The current model configuration can be visualized in any number of ways (i.e.: textual, UML-style STD's, paradigm notation, etc), so some effort went into making sure any of these could be easily implemented. In order to do this, the graphical interface was made dependent upon the underlying objects (statemachines), each of which contains an update function. Whenever this update function is called, it executes all of the functions registered to the STD (which can be done using the addUpdate function), which are responsible for altering their own particular piece visualization.

In particular, an UML-style STD visualization was added to the program. It uses an object defining the drawing of the transitions and states. Each state has a DIV element corresponding to it, whose background denotes whether the corresponding state is the STD's current state. Whenever the objects update function is called, the currently highlighted DIV is returned to the default background color and the state corresponding to the underlying STD's current state, if any, is highlighted. This update function can be registered to the update of one or more STDs using the addUpdate function, and a visualization of the underlying model has been obtained. Using the STD's removeUpdate function, it can also be unregistered.

In a similar way, any type of visualization capable of being represented in Dynamic HTML can be created, and each visualization can be run simultaneously, since the underlying STD's do not change their behaviour.

A.7 Wrap-up: Possible Improvements in Parallelism

Thus, in our environment, each client can be running any number of components, but each component occupies one browser window and therefore has its own JavaScript execution thread. It can also visualize the model in any number

of ways. The server is used only for the initiation of components and as a communication channel, the latter of which makes it the bottle neck of the parallelism objective, since it has only a single thread with which to control all communication.

This problem can be solved in many ways. For starters, another platform could be chosen in order to allow direct communication between components, but this will still need central distribution of the component locations. This also enlarges the security related problems, which the platform chosen solves by using the HTTP protocol and its inherent security.

Another solution could be to introduce multiple servers, at least one for each client. Then, a channel could be handled completely client sided, and the servers could run in parallel. This however would greatly increase the cost of implementation, since a simple browser would no longer be enough.

Finally, another bottleneck for the parallelism is that for each component, all of its detailed STD, its global STDs and its trap STDs run on a single thread. An increase in parallelism could be obtained by also distributing this interaction over multiple clients. This could be set up in a similar manner as described earlier in this Section.

B Code

B.1 script/Statemachine.js

```
function Statemachine (name, allows) {
  var STD = {
    name:         name,
    states:         Array(),
    transitions:     Array(),
    current:        null,
    updateFunctions:   Array(),
    allows:         allows!=''?allows:null,
    waitDefs:       Array(),
    history:        Array()
  };

  STD.State = function(name) {
    if(this.getState(name)) {
      return _e("A state "+name+" already exists in statemachine "+this.
          parent.name);
    }

    var State = {
      name:         name,
      parent:       STD
    };

    State.getName = function() {
      return name;
    };

    this.states.push(State);
    return State;
  };

  STD.getState = function(name) {
    for(var it = 0; it < this.states.length; it++) {
      if (this.states[it].name == name) return this.states[it];
    }
    return false;
  };

  STD.Transition = function(name, from, to, trigger, guard, activity) {
    if(!from || !this.getState(from)) {
      return _e("No state "+from+" (from) found");
    }
    if(!to || !this.getState(to)) {
      return _e("No state "+to+" (to) found");
    }

    var Transition = {
      name:    name,
      from:    this.getState(from),
      trigger:  trigger,
      guard:    guard,
      activity: activity,
      to:    this.getState(to),
      parent:   STD
    };

    Transition.doTrigger = function(trigger, enforce) {
      if(enforce || (this.trigger == trigger && this.parent.current ==
          this.from && this.testGuard())) {
        this.parent.former = this.from;
        this.parent.history.push(this.parent.current.name);
        this.parent.current = this.to;
        eval(this.activity);
        this.parent.update();
        return true;
      }
      return false;
    };
```

```
    Transition.testGuard = function(alsoCheckHistory) {
      if(this.parent.isWaiting()) return false; // prevent evaluation of
          guard if the STD is waiting for some other awnser
      if(alsoCheckHistory && STD.hasBeenIn(Transition.from.name)) return
          true;
      return (this.guard == undefined || eval(this.guard));
    };

    this.transitions.push(Transition);
    return Transition;
};

STD.getTransition = function(name) {
    for(var it = 0; it < transitions.length; it++) {
      if (transitions[it].name == name) return transitions[it];
    }
    return false;
};

STD.getTransitionsFrom = function(from) {
    if (!from) return false;
    if (!from.name) from = this.getState(from);
    var result = Array();
    for(var it = 0; it < this.transitions.length; it++) {
      if(this.transitions[it].from == from) {
        result.push(this.transitions[it]);
      }
    }
    return result.length==0?false:result;
};

STD.isTrapped = function(statesString) {
    var regExp = "(^|,)"+this.current.name+"(,|$)";
    return statesString.match(regExp);
};

STD.isCurrent = function(test) {
    if(!test.name) test = this.getState(test);
    return (test == this.current);
};

STD.trigger = function(trigger) {
    var transitions = this.getTransitionsFrom(this.current);
    for(var it = 0; it < transitions.length; it++) {
      if(transitions[it].doTrigger(trigger)) return true;
    }
    return false;
};

STD.triggerRandom = function() {
    var transitions = this.getTransitionsFrom(this.current);
    if(transitions == false) return false;
    var random = Math.floor(Math.random() * transitions.length);
    return transitions[random].doTrigger(transitions[random].trigger);
};

STD.resetHistory = function() {
   STD.history = Array();
};

STD.hasBeenIn = function(name) {
    if(STD.current == null)     return false;   // not even the
        currently executing this statemachine
    if(STD.current.name == name)  return true;    // is it currently
        there?
    for(var it = 0; it < STD.history.length; it++) {
      if(STD.history[it] == name) return true;    // has it been there
          and currently left it?
    }
    return false;                    // never seen a state by that name
        in the current run
};

STD.removeState = function(name) {
```

```
  var state = STD.getState(name);
  for(var it = 0; it < STD.transitions.length; it++) {
    var temp = STD.transitions[it];
    if(temp.from == state || temp.to == state) {
      STD.transitions.splice(it, 1);
      it--;
    }
  }
  for(var it = 0; it < STD.states.length; it++) {
    var temp = STD.states.shift();
    if(temp != state) {
      STD.states.push(temp);
    } else {
      break; // found the state. Since it is unique, it should be fine
           to simply exit the loop
    }
  }
};

STD.removeTransition = function(from, to, trigger) {
  for(var it = 0; it < STD.transitions.length; it++) {
    temp = STD.transitions[it];
    if(temp.from.name == from && temp.to.name == to && temp.trigger ==
         trigger) {
      STD.transitions.splice(it, 1);
      break;
    }
  }
};

STD.update = function(args) {
  for(var it = 0; it < this.updateFunctions.length; it++) {
    this.updateFunctions[it](args);
  }
};

STD.addUpdate = function(func) {
  if(eval("this."+func)) {
    return _e(func+" already exists");
  }
  if (typeof eval(func) == 'function') {
    this.updateFunctions.push(eval("this."+func+" = "+eval(func)));
  }
};

STD.removeUpdate = function(func) {
  for(var it = 0; it < this.updateFunctions.length; it++) {
    var test = this.updateFunctions.shift();
    if (test.toString() != eval(func).toString()) {
      this.updateFunctions.push(test);
    } else {
      eval("delete this."+func+";");
      it--;
    }
  }
};

STD.wait = function(id, func, vars) {
  this.waitDefs.push({id: id, func: func, vars: vars});
};

STD.wakeUp = function(id, vars, skipWaitFunction) {
  var found = 0;
  var temp = undefined;
  for(var it = 0; it < this.waitDefs.length; it++) {
    temp = this.waitDefs.shift();
    if(temp.id != id) {
      this.waitDefs.push(temp);
    } else {
      it--;
      found++;
      if(temp != undefined && typeof temp.func == 'function' && !(
           skipWaitFunction == true)) {
        var updateVars = temp.func(vars, temp.vars);
```

```
          if(updateVars != false) {
            // function does not return false: set back at start of the
                list and increase it;
            if(typeof updateVars == 'object') {
              for(it2 in updateVars) {
                eval("temp.vars."+it2+" = updateVars."+it2+";");
              }
            }
            this.waitDefs.push(temp);
            it++;
          }
        }
      }
    }
    return found;
  };

  STD.removeWait = function(id) {
    STD.wakeUp(id, {}, true);
  };


  STD.wakeUpAll = function(txt) {
    while(this.waitDefs.length > 0) {
      var temp = this.waitDefs.shift();
      if(temp != undefined && typeof temp.func == 'function') {
        temp.func(txt, temp.vars);
      }
    }
  };

  STD.isWaiting = function(skip) {
    for(var it = 0; it < this.waitDefs.length; it++) {
      if(skip != undefined) {
        eval("var temp = this.waitDefs[it].vars."+skip+";");
        temp = eval(temp);
      }
      if(skip == undefined || !temp) return true;// please don't count
          id = 0, they are placeholders.
    }
    return false;
  };
  statemachines.push(STD);
  return STD;
}

statemachines = Array();
function sendTrigger(trigger) {
  for(var it=0; it < statemachines.length; it++) {
    statemachines[it].trigger(trigger);
  }
}

function getSTD(name) {
  for(var it=0; it < statemachines.length; it++) {
    if(statemachines[it].name == name) return statemachines[it];
  }
  return false;
}

function numSTD() {
  return statemachines.length;
}
```

B.2 script/Paradigm.js

```
function additionUseCase(arr, requestBy) {
  // this adds new states and transitions.
  if(requestBy == undefined) requestBy = "McPal"; // default requester
      name

  for(var it = 0; it < arr.length; it++) {
    var def = arr[it];
```

```
    if(def.compName == undefined || def.compName == '' || def.compName
        == 'self') {   // additions to self
      var STD = getSTD(def.stateMachine);
      if(!STD) {
        STD = Statemachine(def.stateMachine);
      }
      if(def.stateName != undefined) {
        // new states
        STD.State(def.stateName);
      } else if (def.from != undefined) {
        // new transitions
        if(!STD.getState(def.from)) STD.State(def.from);
        if(!STD.getState(def.to)) STD.State(def.to);
        STD.Transition('',def.from,def.to,def.trigger,def.guard,def.
            activity);
      } else {
        _e('unknown addition def type');
      }
    } else {            // additions to send elsewhere
      def.addition = true;
      var mess = Array();
      for(var it2 in def) {
        if(it2 != 'compName') {
          eval('mess.push(it2+":\'"+def.'+it2+'+"\'");');
        }
      }
      mess = "sendMessage.php?db="+database+"&source="+requestBy+"&
          target="+def.compName+"&message="+escape("{"+mess.join(',')+"
          }");;
      send(mess);
    }
  }
  statemachines[0].update('redraw');
  return true;
}


function removalUseCase(arr, requestBy) {
  // Remove states and transitions.
  if(requestBy == undefined) requestBy = "McPal"; // default requester
      name

  for(var it = 0; it < arr.length; it++) {
    var def = arr[it];
    if(def.compName == undefined || def.compName == '' || def.compName
        == 'self') {   // removes of self
      var STD = getSTD(def.stateMachine);
      if(!STD) continue; // if the STD does not exists, nothing needs be
          done
      if(def.stateName == undefined && def.from == undefined) {
        // remove an entire statemachine
        for(var it2 = 0; it2 < statemachines.length; it2++) {
          var temp = statemachines.shift();
          if(temp != STD) {
            statemachines.push(temp);
          } else {
//            break; // found the statemachine. Since it is unique, it
    should be fine to simply exit the loop...
            // ... but actually, this changes the order of the
                statemachines array, so the automatic
            // random triggerer doesn't like it this way and gets
                broken. This way is slightly less
            // optimal, but oh well.
            it2--; // this should of course be done in order to ensure
                all STD's are checked
          }
        }
      } else if (def.stateName != undefined) {
        // remove an state
        STD.removeState(def.stateName);
      } else if (def.from != undefined) {
        // remove transition based upon from, to and trigger
        STD.removeTransition(def.from, def.to, def.trigger);
      } else {
```

```
        _e('unknown remove def type');
      }
    } else {              // removals to send elsewhere
      def.removal = true;
      var mess = Array();
      for(var it2 in def) {
        if(it2 != 'compName') {
          eval('mess.push(it2+":\'"+def.'+it2+'+"\'");');
        }
      }
      mess = "sendMessage.php?db="+database+"&source="+requestBy+"&
          target="+def.compName+"&message="+escape("{"+mess.join(',')+"
          }");;
      send(mess);
    }
  }
  statemachines[0].update('redraw');
  return true;
}



function handleDetailedAlt(txt, vars) {
  if(txt.match(/execute/i) || txt.match(/cancel/i)) {
    send("sendMessage.php?db="+database+"&id="+vars.messId+"&source="+
        vars.src+"&message="+txt);
  } else if(txt.match(/correct/i) && !vars.checked) {
    if(txt.match(/not/i)) {      // subprocess does not allow the
        transition, so cancel all.
      vars.trans.parent.wakeUpAll('cancel');
      send("sendMessage.php?db="+database+"&message=remove&id="+vars.
          messId+"&source="+vars.src);
    } else if(vars.trans.parent.isWaiting('checked==true')) {    // STD
        is still waiting, so still more subprocesses need to awnser the
         correct? message
      return {checked: true};
    } else { // STD is not waiting anymore, so all the messages were
        succesfull, send execute messages and enforce trigger
      vars.trans.parent.wakeUpAll('execute');
      send("sendMessage.php?db="+database+"&id="+vars.messId+"&source="+
          vars.src+"&message=execute");
      vars.trans.doTrigger(vars.trans.trigger, true);
    }
  } else {
    return true;
  }
  return false;
}

function handlePartitionAlt(txt, vars) {
  if(txt.match(/cancel/i)) {
    send("sendMessage.php?db="+database+"&message=remove&id="+vars.
        messId+"&source="+vars.src);
    // canceled, remove channel and continue execution of STD, to do
        this:
    // do nothing, that is, don't execute the transition and don't
        return this to the STD waitlist
  } else if (txt.match(/execute/i)) {
    // transition taken; change subprocess to do this:
    // execute the transition stored in var.trans
    send("sendMessage.php?db="+database+"&message=remove&id="+vars.
        messId+"&source="+vars.src);
    vars.trans.doTrigger(vars.trans.trigger, true); // enforce trigger
  } else {
    return true; // return the listener to the wait stack
  }
  return false;
}

function handlePartitionOpt(messId, mess, src) {
  var STD = getSTD(mess.STD);
  message = "sendMessage.php?db="+database+"&id="+messId+"&source="+mess
      .target+"&message=";
```

```
  if(STD.current != null && getSTD(mess.state).hasBeenIn(mess.trap)) {
    var transitions = STD.getTransitionsFrom(mess.state);
    for(var it = 0; it < transitions.length; it++) {
      if(transitions[it].trigger == mess.trap && (mess.next == undefined
           || mess.next == transitions[it].to.name) && transitions[it].
          testGuard(true)) {
        STD.wait(messId, handlePartitionAlt, {src: src, trans:
            transitions[it], messId: messId});
        send(message+"correct");
        return true;
      }
    }
  }
  send(message+"notCorrect");
  return false;
}

removeStack = {};
function handlePoll(txt, vars) {
  if(txt == '') return false;
  var txtArr = txt.split(";\n");
  for(var it = 0; it < txtArr.length; it++) {
     temp =  txtArr[it].split("=");
    messId = temp[0];
    eval("var mess = "+((typeof temp[1]!='number' && !temp[1].match(/^\s
        *\{.*:.*\}\s*$/))?"'"+temp[1]+"'":temp[1])+";");
    if(typeof(mess) == 'object') {
      if(mess.addition != undefined) {
        // Addition Use Case
        if(additionUseCase(Array(mess))) send("sendMessage.php?db="+
            database+"&message=remove&source="+vars.src+"&id="+messId);
      } else if (mess.removal != undefined) {
        // Removal Use Case
        if(removalUseCase(Array(mess))) send("sendMessage.php?db="+
            database+"&message=remove&source="+vars.src+"&id="+messId);
      } else {
        // Global transition requested
        handlePartitionOpt(messId, mess, vars.src);
      }
    } else if(mess.match(/noListener/i)) {
      send("sendMessage.php?db="+database+"&message=remove&source="+vars
          .src+"&id="+messId);
      for(var it2 = 0; it2 < statemachines.length; it2++) {
        statemachines[it2].removeWait(messId);
      }
    } else {
      var found = 0;
      for(var it2 = 0; it2 < statemachines.length; it2++) {
        found += statemachines[it2].wakeUp(messId, mess);
      }
      if(found == 0) { // nobody listening, send noListener message ...
        eval("var temp = removeStack.var"+messId+"==undefined;"); // ...
             but only if this is not the first time, to allow self-
            messages to be sent. Probably not needed though...
        if(!temp) {
          send("sendMessage.php?db="+database+"&message=noListener&id="+
              messId+"&source="+vars.src);
          eval("delete removeStack.var"+messId+";");
        } else {
          eval("removeStack.var"+messId+" = 1;");
        }
      }
    }
  }
}

function alterWait(text, vars) {
  var trans = vars.trans;
  trans.parent.wait(text, handleDetailedAlt, {src: vars.src, trans:
      trans, messId: text});
  trans.parent.wakeUp(vars.it);
  return false;
}
```

```
function paradigmGuard(trans, subprocesses, globalList) {
  if(trans.parent.isWaiting()) return false; // disallow if the
      statemachine is currently waiting for some awnser
  if(!(subprocesses == undefined && numSTD() == 1)) { // test whether
      current subprocesses exist and allow this transition
    var subArr = subprocesses.split(";");
    var part = {};
    for(var it = 0; it < subArr.length; it++) { // order subprocesses
        according to their partition
      var temp = subArr[it].split(".");
      eval("if(part."+temp[0]+"==undefined) part."+temp[0]+" = Array('"+
          temp[1]+"'); else part."+temp[0]+".push('"+temp[1]+"');");
    }
    for(var it in part) {
      var correct = false;
      eval("var sub = part."+it+";");
      for(var it2 = 0; it2 < sub.length; it2++) {
        if(getSTD(it) != false && getSTD(it).current != null && getSTD(
            it).current.name == sub[it2]) {
          correct = true;
          break;
        }
      }
      if (correct == false) return false;
    }
  }

  if(globalList==undefined)  return true;  // pure employee transition,
       allow at this junction.

  // handle asynchronous trapOfSubprocess signals
  globalList = globalList.split("&");
  var addr = "sendMessage.php?db="+escape(database);
  addr += "&source="+escape(trans.parent.name);

  for(var it = 0; it < globalList.length; it++) {
    var addrTemp = addr;
    var temp = globalList[it].split(".");
    addrTemp += "&target="+escape(temp[0]);
    addrTemp += "&message=";
    addrTemp += escape("{target:'"+temp.shift()+"',STD:'"+temp.shift()+"
        ',state:'"+temp.shift()+"',trap:'"+temp.shift()+(temp.length
        ==0?"'}":"', next:'"+temp.shift()+"'}"));
    trans.parent.wait(-it-1);
    send(addrTemp, alterWait, {src: trans.parent.name, trans: trans, it:
         -it-1});
  }

  return false; // do not allow! Allow will be enforced by follow up
      functions (alter wait etc)
}

function changeSubprocess(from, to) {
  if(getSTD(from)) { // just in case a migration has already removed the
       original subprocess
    getSTD(from).resetHistory();
    getSTD(from).current = null;
    getSTD(from).update();
  }
  getSTD(to).current = getSTD(to).getState('triv');
  getSTD(to).update();
  sendTrigger('cascade');
}

var storeUseCases = "";
function handleMigr(txt) {
  document.getElementById("upload").className = "hideIFrame";
  storeUseCases = txt;

}

function getMigr() {
  storeUseCases = '';
  document.getElementById("upload").className = "showIFrame";
```

```
}
```

B.3 script/Poll.js

```
var polls = Array();

function Poll (name) {
  try {
  var Poll = {
    name: name,
    timer: null,
    speed: 0
  };
  } catch(e) {alert(e);}

  Poll.poll = function() {
    send("poll.php?db="+database+"&source="+this.name, this.handle, {src
        : this.name});
  };

  Poll.repeat = function(milisec) {
    if(milisec != undefined) {
      clearTimeout(this.timer);
      this.speed = milisec;
    }
    this.poll();
    if(this.speed && this.speed > 100) {
      this.timer = setTimeout("getPoll('"+this.name+"').repeat()", this.
          speed);
    } else {
      clearTimeout(this.timer);
    }
  };

  Poll._handle = function(message, vars) {  // standard function
    // handled here in order to show get message and removal from
        database
    if(message == "") {
      return false;
    }
    var messages = message.split(";");
    for(var it = 0; it < messages.length; it++) {
      var temp = messages[it].split("=");
      var id = temp[0];
      if(id == "") continue;
      var mess= temp[1];
      send("sendMessage.php?db="+database
          +"&target="+Poll.name   // only remove messages send to you!
          +"&message=remove"
          +"&id="+id,
          (typeof _e == 'function')?_e:alert
            // show the message (uses _e, a generic error function, if
                it exists, otherwise, alert)
      );
    }
  };

  Poll.handle = this._handle;          // overload this!!!
//  Poll.repeat();
  polls.push(Poll);
  return Poll;
}

function getPoll(name) {
  for(var it = 0; it < polls.length; it++) {
    if(polls[it].name == name) return polls[it];
  }
}
```

B.4 script/std.js

```
function _verbose() {
```

```
  var verbose = {
    div:     null,
    stateMachines:  Array()
  };

  verbose.update = function(STD) {
    if(STD.current) verbose.div.innerHTML = STD.name+".current = "+STD.
        current.getName()+"<BR>\n" + verbose.div.innerHTML;
  };

  verbose.draw = function(elem) {
    for(var it = 0; it < this.stateMachines.length; it++) {
      if(this.stateMachines[it] == elem) return true;
    }
    this.stateMachines.push(elem);
    if(verbose.div == null) {
      verbose.div = document.createElement('div');
      verbose.div.id = 'print';
      verbose.div.className = 'report';
      document.body.appendChild(verbose.div);
    }

    var _print = "<DIV STYLE='border: 2px solid black; background: #
        DDDDDD'><B>Start "+elem.name+"</B><BR>\n";
    for(var it = 0; it < elem.states.length; it++) {
      _print += elem.states[it].getName()+"<BR>\n";
    }
    for(var it = 0; it < elem.transitions.length; it++) {
      var trans = elem.transitions[it];
      _print += "Transition ("+trans.name+"): "+trans.from.getName()+"
          --";
      _print += trans.trigger==undefined?"":trans.trigger;
      _print += trans.guard==undefined?"":"["+trans.guard+"]";
      _print += trans.activity==undefined?"":"/"+trans.activity;
      _print += "--> "+trans.to.getName()+"<BR>\n";
    }
    _print += "<B>End "+elem.name+" (current state: "+(elem.current?elem
        .current.getName():"null")+")</B></DIV>\n";
    verbose.div.innerHTML = _print + verbose.div.innerHTML;
    elem.addUpdate("verboseUpdate");
    elem.update();
  };

  verbose.unDraw = function() {
    if(verbose.div == null) return true;
    verbose.div.parentNode.removeChild(verbose.div);
    verbose.div = null;
    for(var it = 0; it < this.stateMachines.length; it++) {
      this.stateMachines[it].removeUpdate('verboseUpdate');
    }
    this.stateMachines = Array();
  };

  return verbose;
}

var verbose = _verbose();
verboseUpdate = function() {
  verbose.update(STD);
}
```

B.5 script/umlPrint.js

```
function UML() {
  var UML = {
    stateMachines:  Array(),
    states:    Array(),
    transitions:  Array(),
    drag:    null,
    offset:    {
          x:  0,
          y:  0
        },
```

```
    begin:      {
          x:  0,
          y:  0
        }
  };

  UML.update = function(STD) {
    for(var it = 0; it < STD.states.length; it++) {
      var div = document.getElementById("uml."+STD.name+"."+STD.states[
          it].getName());
//        if(div == undefined) {
//          UML.unDraw();
//          for(var it = 0; it < statemachines.length; it++) uml.draw(
    statemachines[it]);
//        }
//        else
        div.className = 'state';
    }
    if(STD.current) {
      var temp = document.getElementById("uml."+STD.name+"."+STD.current
          .getName());
      if(temp) temp.className = 'currentState'; // might fail due to
          removal of state
    }
  };

  UML.draw = function(elem, width, x, y) {
    for(var it = 0; it < this.stateMachines.length; it++) {
      if(this.stateMachines[it] == elem) return false;
    }
    this.stateMachines.push(elem);
    if(!width) width = 4;
    if(!x) x = 0;
    if(!y) y = 0;
    for(var it = 0; it < elem.states.length; it++) { // draw the states
      var name = elem.states[it].getName();
      newDiv = document.createElement('div');
      newDiv.id = 'uml.'+elem.name+'.'+name;


      // DB
      var mess = "getLocation.php?state="+newDiv.id;
      func = function(obj, vars) {
        eval("var temp = "+obj+";");
        vars.div.style.left = temp.left+"px";
        vars.div.style.top = temp.top+"px";
        while(UML.transitions.length != 0) {
          div = UML.transitions.pop();
          div.parentNode.removeChild(div);
        }
        for(var it = 0; it < statemachines.length; it++) {
          for(var it2 = 0; it2 < statemachines[it].transitions.length;
              it2++) {
            UML.transitionDraw(statemachines[it].transitions[it2]);
          }
        }
      }
      newDiv.style.top = ((Math.floor(it/width))*100+y)+"px";
      newDiv.style.left = (((Math.floor((it%(width*2))/width)!=0)?(width
          -1-it%width):it%width)*150+x)+"px";
      send(mess, func, {div: newDiv, UML: UML});
      // end DB

/*
      // cookies
      var temp = getCookie('locations');
      if(temp && temp.match("#"+newDiv.id+"=")) {
        temp = temp.split("#"+newDiv.id+"=");
        temp = temp[1].split("#");
        eval("temp = "+temp[0]+";");
      } else {
        temp = undefined;
      }
```

```
      newDiv.style.top = (temp != undefined)?temp.top:((Math.floor(it/
          width))*100+y)+"px";
      newDiv.style.left = (temp != undefined)?temp.left:(((Math.floor((
          it%(width*2))/width)!=0)?(width-1-it%width):it%width)*150+x)
          +"px";
      // end cookies
*/

      newDiv.className = (elem.current&&elem.current.getName()==name)?"
          currentState":"state";

      newDiv.innerHTML = name+"<BR>("+elem.name+")";
      newDiv.style.height = "50px";
      newDiv.style.width = "100px";
      newDiv.setAttribute('onmousedown', 'uml.startDrag(event, this)');
      newDiv.setAttribute('onmouseup', 'uml.stopDrag(this)');
      this.states.push(document.body.appendChild(newDiv));
    }
    for(var it = 0; it < elem.transitions.length; it++) {
      this.transitionDraw(elem.transitions[it]);
    }
    elem.addUpdate("umlUpdate");
    elem.update();
    // return the number of rows it occupies:
    return Math.ceil(elem.states.length/width);
  };

  UML.transitionDraw = function(trans) {
    from = document.getElementById("uml."+trans.parent.name+"."+trans.
        from.getName());
    to = document.getElementById("uml."+trans.parent.name+"."+trans.to.
        getName());
    if(!from || !to) return false;
    title = trans.trigger==undefined?"":trans.trigger;
    title += trans.guard==undefined?"":"["+trans.guard+"]";
    title += trans.activity==undefined?"":"/"+trans.activity;

    var from = {
      x:      parseInt(from.style.left),
      width:  parseInt(from.style.width),
      y:      parseInt(from.style.top),
      height: parseInt(from.style.height)
    };
    var to = {
      x:      parseInt(to.style.left),
      width:  parseInt(to.style.width),
      y:      parseInt(to.style.top),
      height: parseInt(to.style.height)
    };
    if ((from.x >= to.x && from.x <= to.x + to.width) || (from.x <= to.x
         && to.x <= from.x + from.width)) { // vertical
      this.drawLine(
        {
          x: (from.x > to.x)?from.x + (to.x + to.width - from.x)/2 - 1:
              to.x + (from.x + from.width - to.x)/2 - 1,
          y: from.y + ((from.y<to.y)?from.height:0)
        },
        {
          x: (to.x > from.x)?to.x + (from.x + from.width - to.x)/2 - 1:
              from.x + (to.x + to.width - from.x)/2 - 1,
          y: to.y + ((to.y<from.y)?to.height:-2)
        },
        title
      );
    } else if ((from.y >= to.y && from.y <= to.y + to.height) || (from.y
         <= to.y && to.y <= from.y + from.height)) { // horizontal
      this.drawLine(
        {
          x: from.x + ((from.x<to.x)?from.width:0),
          y: ((from.y > to.y)?from.y + (to.y + to.height - from.y)/2:to.
              y + (from.y + from.height - to.y)/2) - 1
        },
        {
          x: to.x + ((to.x<from.x)?to.width:0),
```

```
        y: ((to.y > from.y)?to.y + (from.y + from.height - to.y)/2:
             from.y + (to.y + to.height - from.y)/2) - 1
      },
      title
    );
  } else {
    this.drawLine(
      {
        x: from.x+from.width/2+1,
        y: from.y + ((from.y < to.y)?from.height:0)
      },
      {
        x: from.x+from.width/2-1,
        y: from.y + ((from.y < to.y)?from.height:0) + ((from.height/2)
             *((from.y < to.y)?1:-1))
      },
      title,
      true
    );
    this.drawLine(
      {
        x: from.x+from.width/2+1,
        y: from.y + ((from.y < to.y)?from.height:0) + ((from.height/2)
             *((from.y < to.y)?1:-1))
      },
      {
        x: from.x+from.width/2-1 + (from.width*(from.x<to.x?.75:-.75))
             + (from.x<to.x?3:0),
        y: from.y + ((from.y < to.y)?from.height:0) + ((from.height/2)
             *((from.y < to.y)?1:-1))
      },
      title,
      true
    );
    this.drawLine(
      {
        x: from.x+from.width/2-1 + (from.width*(from.x<to.x?.75:-.75))
             ,
        y: from.y + ((from.y < to.y)?from.height:0) + ((from.height/2)
             *((from.y < to.y)?1:-1))
      },
      {
        x: from.x+from.width/2-1 + (from.width*(from.x<to.x?.75:-.75))
             ,
        y: to.y + ((from.y < to.y)?0:to.height) + ((to.height/2)*((
             from.y < to.y)?-1:1))
      },
      title,
      true
    );
    this.drawLine(
      {
        x: from.x+from.width/2-1 + (from.width*(from.x<to.x?.75:-.75))
             ,
        y: to.y + ((from.y < to.y)?0:to.height) + ((to.height/2)*((
             from.y < to.y)?-1:1))
      },
      {
        x: to.x+to.width/2 - 1 + (from.x<to.x?3:0),
        y: to.y + ((from.y < to.y)?0:to.height) + ((to.height/2)*((
             from.y < to.y)?-1:1))
      },
      title,
      true
    );
    this.drawLine(
      {
        x: to.x+to.width/2 - 1,
        y: to.y + ((from.y < to.y)?0:to.height) + ((to.height/2)*((
             from.y < to.y)?-1:1))
      },
      {
        x: to.x+to.width/2 - 1,
        y: to.y + ((from.y < to.y)?0:to.height) - (from.y<to.y?3:0)
```

```
        },
        title
      );
    }
  };

  UML.drawLine = function(from, to, title, noArrow) {
    var elem = document.createElement("DIV");
    elem.className = "line";
    elem.style.width = Math.max(3, Math.abs(from.x - to.x))+"px";
    elem.style.height = Math.max(3, (Math.abs(from.y - to.y)+(((noArrow)
        &&(from.y<to.y))?3:0)))+"px";
    elem.style.left = Math.min(from.x, to.x)+"px";
    elem.style.top = Math.min(from.y, to.y)+"px";
    elem.title = title;

    if (!noArrow) {
      var arrow = document.createElement("DIV");
      if (from.x < to.x) {
        arrow.className = "harrow";
        arrow.innerHTML = "&gt;";
        arrow.style.width = to.x - from.x + 5+"px";
        arrow.style.textAlign = "right";
      } else if (from.x > to.x) {
        arrow.className = "harrow";
        arrow.innerHTML = "&lt;";
      } else if (from.y < to.y) {
        arrow.className = "varrow";
        arrow.innerHTML = "&or;";
        arrow.style.top = (to.y - from.y - 23)+"px";
      } else {
        arrow.className = "varrow";
        arrow.innerHTML = "&and;";
      }
      this.transitions.push(elem.appendChild(arrow));
    }
    this.transitions.push(document.body.appendChild(elem));
    return elem;
  };

  UML.unDraw = function() {
    UML.stopDrag();
    var div;
    while(this.stateMachines.length != 0) {
      div = this.stateMachines.pop();
    }
    while(this.states.length != 0) {
      div = this.states.pop();
      div.parentNode.removeChild(div);
    }
    while(this.transitions.length != 0) {
      div = this.transitions.pop();
      div.parentNode.removeChild(div);
    }
    for(var it = 0; it < statemachines.length; it++) {
      statemachines[it].removeUpdate('umlUpdate');
    }
  };

  UML.startDrag = function(event, elem) {
    this.drag = elem;
    this.begin.x = event.clientX - elem.offsetLeft + this.offset.x;
    this.begin.y = event.clientY - elem.offsetTop + this.offset.y;
    document.onmousemove = this.dragFunction;
  };

  UML.stopDrag = function(elem) {
    if(this.drag) {
      var mess = "updateLocation.php?left="+parseInt(this.drag.style.
          left)+"&top="+parseInt(this.drag.style.top)+"&state="+escape(
          this.drag.id);
      send(mess);

      var temp = getCookie('locations');
```

```
      if(temp) {
        eval('var regExp = /#'+this.drag.id+'[^#]*($|#)/gi');
        temp = temp.replace(regExp, "\#");
        temp = temp.replace(/#+/g, "\#");
        if(temp == '#') temp = "";
      }
      temp = (temp?temp:"")+"#"+this.drag.id+"={left:'"+this.drag.style.
          left+"',top:'"+this.drag.style.top+"'}";

      setCookie('locations', temp, 365); // expires after a year
    }

    this.drag = null;
    document.onmousemove = null;
    while(this.transitions.length != 0) {
      div = this.transitions.pop();
      div.parentNode.removeChild(div);
    }
    for(var it = 0; it < statemachines.length; it++) {
      for(var it2 = 0; it2 < statemachines[it].transitions.length; it2
          ++) {
        UML.transitionDraw(statemachines[it].transitions[it2]);
      }
    }
  };

  UML.dragFunction = function(event) {
    if (UML.drag) {
      if (!event) event = window.event;
      UML.drag.style.left = (event.clientX - UML.begin.x)+"px";
      UML.drag.style.top = (event.clientY - UML.begin.y)+"px";
    }
    while(UML.transitions.length != 0) {
      div = UML.transitions.pop();
      div.parentNode.removeChild(div);
    }
    for(var it = 0; it < statemachines.length; it++) {
      for(var it2 = 0; it2 < statemachines[it].transitions.length; it2
          ++) {
        UML.transitionDraw(statemachines[it].transitions[it2]);
      }
    }
  };
  return UML;
}

var uml = UML(); // create an instance of the class
function umlWaitForIt(STD) {
  uml.unDraw();
  for(var it = 0; it < statemachines.length; it++) {
    uml.draw(statemachines[it]);
  }
}

function umlUpdate(args) { // function to append to the draw functions
    array of the statemachine STD
  if(args == 'redraw') {
    if(uml.waiter) clearTimeout(uml.waiter);
    uml.waiter = setTimeout("umlWaitForIt()", 1000);
  } else {
    uml.update(STD);
  }
}
```

B.6 script/script.js

```
function _e(message) { // an error message function
  document.getElementById('error').innerHTML = message + "<BR>\n" +
      document.getElementById('error').innerHTML;
  return false;
}

function init() {
```

```
  for(var it = 0; it < additionalInit.length; it++) {
    additionalInit[it]();
  }
  var height = 10;
  for(var it = 0; it < statemachines.length; it++) {
    height += uml.draw(statemachines[it], 4, 300, height)*100;
  }
  sendTrigger('cascade');
}

function send(url, func, funcArgs, synchronous) {
  if(func == undefined) func = function(a,b){};
  var req = (window.XMLHttpRequest)?new XMLHttpRequest():new
       ActiveXObject("Microsoft.XMLHTTP");
  req.onreadystatechange = function() {
    if (req.readyState == 4) {
      try{
        if (req.status == 200) func(req.responseText, funcArgs);
        else alert("Error: "+req.status);
      } catch (e) {
      }
    }
  };
  req.open("GET", "ajaxFiles/"+url, !synchronous);

  req.send(null);
  return req;
}

function setCookie(name,value,expireIn) {
  if(expireIn) {
    var expire = new Date();
    expire.setDate(expire.getDate()+expireIn);
  }
  document.cookie = name.split(".").join("_")+"="+escape(value)+((
       expireIn==undefined)?"":";expires="+expire.toGMTString());
}

function getCookie(name) {
  if (document.cookie.length>0) {
    var start = document.cookie.indexOf(name + "=");
    if (start != -1) {
        start = start + name.length + 1;
      var end = document.cookie.indexOf(";",start);
        if (end == -1) end = document.cookie.length;
        return unescape(document.cookie.substring(start,end));
    }
  }
  return undefined;
}

var timer = null;
function setRandom(speed) {
  clearTimeout(timer);
  if(speed != 0) {
    statemachines[0].triggerRandom();
    clearTimeout(timer);
    timer = setTimeout("setRandom("+speed+");", speed);
  }
}
```

B.7 ajaxFiles/sendMessage.php

```
<?
// @ATTR, required:   database, the location of the messages table
// @ATTR, required:   source, the sender
// @ATTR, optional:   target, the intended receiver, optional iff
    removal message
// @ATTR, required:   message, the message
// @ATTR, optional:   id, the update location, insert if not set
// @RETURN, required: id, or the channel number, also the wait id

// first send no caching headers
```

```
require_once "noCache.php";

require_once "/xampp/htdocs/paradigm/query.php";

if(!isset($_REQUEST["id"])) {
  $sql = "
    INSERT
    INTO   messages (source, target, message)
    VALUES (
      '{$_REQUEST["source"]}',
      '{$_REQUEST["target"]}',
      '{$_REQUEST["message"]}'
    )
  ";
  $mysql_insert_id = 0;
  $result = query($sql, $_REQUEST['db']);
  $reply = $mysql_insert_id;
} else if($_REQUEST["message"]=="remove") {
  $sql = "
    DELETE
    FROM   messages
    WHERE id = '{$_REQUEST['id']}'
    AND    target = '{$_REQUEST['source']}'
  ";
  $mysql_affected_rows = 0;
  $result = query($sql, $_REQUEST['db']);
  $reply = "deleted $mysql_affected_rows rows";
} else if ($_REQUEST['message'] == 'cancel') {
  $sql = "
    UPDATE   messages
    SET    target = IF(source = '{$_REQUEST['source']}', target, source),
         source = '{$_REQUEST['source']}',
         message = 'cancel'
    WHERE id = '{$_REQUEST['id']}'
    AND    (
           source = '{$_REQUEST['source']}'
           OR
           target = '{$_REQUEST['source']}'
         )
  ";
  $mysql_affected_rows = 0;
  $result = query($sql, $_REQUEST['db']);
  $reply = "updated $mysql_affected_rows rows";
} else {
  $sql = "
    UPDATE   messages
    SET    target = source,
         source = '{$_REQUEST['source']}',
         message = '{$_REQUEST['message']}'
    WHERE id = '{$_REQUEST['id']}'
    AND    target = '{$_REQUEST['source']}'
  ";
  $mysql_affected_rows = 0;
  $result = query($sql, $_REQUEST['db']);
  $reply = "updated $mysql_affected_rows rows";
}
echo $reply;

/* use below in case of debugging neccesities */
/* ksort($_REQUEST);
unset($_REQUEST["locations"]);
$temp = Array();
foreach($_REQUEST as $key => $val) {
  $temp[] = "$key=$val";
}
$temp = implode("&", $temp);

$sql = "
  INSERT
  INTO   mess_log (tijd, message, reply)
  VALUES (NOW(), '$temp', '$reply')
";
$result = query($sql, $_REQUEST['db']);
*/
```

```
?>
```

B.8 ajaxFiles/poll.php

```
<?
// first send no caching headers
require_once "noCache.php";

// actions first

$sql = "
  SELECT    id,
            message
  FROM      messages
  WHERE     target = '{$_REQUEST["source"]}'
  ORDER BY  message != 'noListener', message LIKE '{%}'
";
require "/xampp/htdocs/paradigm/query.php";
$result = query($sql, $_REQUEST["db"]);

$temp = array();
while(($row = mysql_fetch_assoc($result)) !== FALSE) {
  $temp[] = "{$row["id"]}={$row["message"]}";
}

echo implode(";\n", $temp);

?>
```

B.9 ajaxFiles/leavePage.php

```
<?
require_once "noCache.php";
require_once "/xampp/htdocs/paradigm/query.php";

$reply = "No db or comp found";
if(isset($_REQUEST["db"]) && isset($_REQUEST["comp"])) {
  $sql = "
    DELETE
    FROM    messages
    WHERE source = '{$_REQUEST["comp"]}'
    OR      target = '{$_REQUEST["comp"]}'
  ";
  $mysql_affected_rows = 0;
  $result = query($sql, $_REQUEST['db']);
  $reply = "deleted $mysql_affected_rows rows";
}
echo $reply;
?>
```

B.10 ajaxFiles/JITForm.php

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
    html4/strict.dtd">
<HTML>
<HEAD>
<TITLE>File Upload Form</TITLE>
<LINK REL="stylesheet" TYPE="text/css" HREF="style.css">
</HEAD>
<BODY>
<FORM ID="file_upload_form" METHOD="post" ENCTYPE="multipart/form-data"
    ACTION="uploadResult.php" CLASS='uploadForm'>
<INPUT NAME="file" TYPE="file" ONCHANGE='this.form.submit()' SIZE=0>
</FORM>
</BODY>
</HTML>
```

B.11 ajaxFiles/uploadResult.php

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
    html4/strict.dtd">
<HTML>
<HEAD>
<TITLE>Get and send Usecase</TITLE>
<?
// some usefull variables to set:
$map = "/xampp/htdocs/paradigm/upload/";
$max = 1048576; // max size = 1Mb
$extAllow = Array(
  "migr",
); // mogelijke extenties (zonder punt), hoofdletter ongevoelig

// Controleren
if (isset($_FILES['file'])) {
  $bestand = explode("\\", $_FILES['file']['name']);
  $bestand = $bestand[count($bestand) - 1];

  $ext = explode(".", $bestand);
  $ext = strtolower($ext[count($ext) - 1]);

  // Toegestaande extensies opvragen
  if($extAllow !== "") {
    $extFout = true;
    foreach($extAllow as $val) {
      if ($ext=== strtolower($val)) {
        $extFout = false;
        break;
      }
    }
  } else $extFout = false;

  if ($extFout) {
     echo "Foutieve extentie van het migratie bestand (moet .migr zijn)"
        ;
  } else {
    if ($_FILES['file']['size'] > $max)  {
      echo "max size error; hou het AUB onder 1Mb";
    } else {
      if(move_uploaded_file($_FILES["file"]["tmp_name"], $map.$bestand)
          !== FALSE) {
        $content = file($map.$bestand);
        $toRem = $toAdd = Array();

        echo "<SCRIPT TYPE='text/javascript'>\n";
        foreach($content as $key => $val) {
          $val = trim($val);
          $key++;
          if(preg_match("/^\w*$/", $val)) {
//              echo "// skip empty line $key ($val)\n";
            continue;
          } else if(stripos($val, "add:") === 0) {
            $mode = "A";
//              echo "// set mode $mode on line $key ($val)\n";
            continue;
          } else if (stripos($val, "remove:") === 0) {
            $mode = "R";
//              echo "// set mode $mode on line $key ($val)\n";
            continue;
          } else if(!isset($mode)) {
//              echo "alert('No mode set, assuming add'); // line $key\n";
            $mode = 'A';
          }
//            echo "// using mode $mode on line $key\n";
          if($mode == 'A') {
            $toAdd[] = "{".$val."}";
          } else {
            $toRem[] = "{".str_replace("'", "\\\\'", $val)."}";
          }
        }

        $temp = "additionUseCase(Array(";
        $temp .= "{stateMachine:'McPal',from:'NewRuleSet',to:'StartMigr
             ',guard:'paradigmGuard(this,\"Migr.StablePhase\",\"McPal.
```

```
            Migr.StablePhase.ready\");',activity:'sendTrigger(\"cascade
            \")'},";
        $temp .= implode($toAdd, ",");
        $temp .= ",{stateMachine:'McPal',from:'Content',to:'Observing',
            guard:'paradigmGuard(this,\"Migr.doMigr\",\"McPal.Migr.
            doMigr.migrDone\");',activity:'sendTrigger(\"cascade\");";
        $temp .= "removalUseCase(Array(";
        $temp .= implode($toRem, ",");

        $temp .= "))'})); storeUseCases = '';";
        $temp = "window.top.window.handleMigr('".str_replace("'", "\\'",
             $temp)."');";
        echo "$temp\n</SCRIPT></HEAD><BODY><P></BODY></HTML>";
      } else {
        echo "failed to move to $map$bestand";
      }
    }
  }
}
require('JITForm.php'); // dies if correct, so if it reaches this, the
    upload failed. Retry please
?>
```

B.12 ajaxFiles/updateLocation.php

```
<?
require_once "noCache.php";
require_once "/xampp/htdocs/paradigm/query.php";

$mysql_affected_rows = 0;
$sql = "
  UPDATE `states`
  SET    `left` = {$_REQUEST['left']},
       `top` = {$_REQUEST['top']}
  WHERE `name` = '{$_REQUEST['state']}'
  ";
$result = query($sql, 'locations');
if($mysql_affected_rows == 0) {
  $sql = "
    INSERT
    INTO `states` (`name`, `left`, `top`)
    VALUES(
      '{$_REQUEST['state']}',
      {$_REQUEST['left']},
      {$_REQUEST['top']}
    )
  ";
  $result = query($sql, 'locations');
}
?>
```

B.13 ajaxFiles/getLocation.php

```
<?
require_once "noCache.php";
require_once "/xampp/htdocs/paradigm/query.php";

$sql = "
  SELECT `top`, `left`
  FROM `states`
  WHERE `name` = '{$_REQUEST['state']}'
  ";
$result = query($sql, 'locations');
if(($row = mysql_fetch_assoc($result)) !== FALSE) {
  echo "{top: {$row["top"]}, left: {$row["left"]}}";
}
?>
```

B.14 ajaxFiles/noCache.php

```
<?
header("Expires: Mon, 26 Jul 1990 05:00:00 GMT");
header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT");
header("Cache-Control: no-store, no-cache, must-revalidate");
header("Cache-Control: post-check=0, pre-check=0", false);
header("Pragma: no-cache");
?>
```

B.15 script/getComponents.php

```
additionalInit = Array();
<?
require_once "/xampp/htdocs/paradigm/query.php";

function varParse($varArr, $arr) { // this got to be too complicated...
  foreach($varArr as $key => $val) {
    foreach($arr as $arrKey => $row) {
      if(($temp = str_replace($key, 'HELEMAALNIETS', $row)) !== $row) {
        $row["id"] .= "_$key";
        $row = str_replace("max($key)", $val, $row);
        for($it = 0; $it < $val; $it++) {
          $temp = str_replace($key, $it, $row);
          foreach($temp as $tempKey => $tempVal) {
            if(strpos($tempVal, "CALC[") !== FALSE) {
              $temp2 = preg_replace("/.*CALC\[([^\]]*)\].*/", "$1",
                  $tempVal);
              eval('$temp2 = '.$temp2.';');
              $temp[$tempKey] = preg_replace("/CALC\[([^\]]*)\]/",
                  $temp2, $tempVal);
            }
          }
          $arr[] = $temp;
        }
        unset($arr[$arrKey]);
      }
    }
  }
  return $arr;
}

function idParse($idAdd, $arr) {
  foreach($idAdd as $key => $val) {
    foreach($arr as $arrKey => $row) {
      if(($temp = str_replace("id$key", $val, $row)) !== $row) {
        $arr[$arrKey] = $temp;
      }
    }
  }
  return $arr;
}

$componentNames = Array();
foreach(explode(";", $_REQUEST["components"]) as $component) {
  echo "var database = '{$_REQUEST["db"]}';\n";
  echo "var temp = function() {\n";

  /* start states */
  $sql = "
    SELECT  S.stateName,
        S.compName,
        S.statemachine,
        S.initial
    FROM  states S
    WHERE S.compName = '$component'
  ";

  $result = query($sql, $_REQUEST["db"]);

  while(($row = mysql_fetch_assoc($result)) !== FALSE) {
    $comps[] = $row;
  }

  if(isset($_REQUEST["var"])) {
```

```
    $comps = varParse($_REQUEST["var"], $comps);
  }

  if(isset($_REQUEST["id"])) {
    $comps = idParse($_REQUEST["id"], $comps);
  }

  foreach($comps as $row) {
    $row = str_replace(Array("'", "\n"), Array("\\'", " "), $row);
    $comp = ($row["statemachine"]=='')?$component:$row["statemachine"];
    $compLong = $comp;
    if(isset($_REQUEST["id"]) && $row["statemachine"] == "") {
      $compLong = $comp."_".implode(",",$_REQUEST["id"]);
      $detailedComp = $comp;
      $detailedCompLong = $compLong;
    }

    if(!isset($componentNames[$comp])) {
      echo "\tvar $comp = Statemachine('$compLong');\n";
      if($comp == $component) {
        echo "\tPoll('$compLong');\n";
        echo "\tgetPoll('$compLong').handle = handlePoll;\n";
      }
      $componentNames[$comp] = true;
    }

    eval ('$temp = '.$row["initial"].';');
    echo ($temp?"\t$comp.current = ":"\t")."$comp.State('{$row["
        stateName"]}');\n";
  }
  /* end states */

  /* start transition */
  $sql = "
    SELECT  T.id,
        T.statemachine,
        T.source,
        T.target,
        T.triggerSig,
        REPLACE(REPLACE(T.guard, '\\r', ' '), '\\t', '') as guard,
        REPLACE(REPLACE(T.activity, '\\r', ' '), '\\t', '') as activity
    FROM  transitions T
    WHERE T.compName = '$component'

  ";
  $result = query($sql, $_REQUEST["db"]);
  while(($row = mysql_fetch_assoc($result)) !== FALSE) {
    $trans[] = $row;
  }

  if(!isset($trans)) {
    echo "_e('No transitions found!!!');";
  } else {
    if(isset($_REQUEST["var"])) {
      $trans = varParse($_REQUEST["var"], $trans);
    }

    if(isset($_REQUEST["id"])) {
      $trans = idParse($_REQUEST["id"], $trans);
    }

    foreach($trans as $row) {
      $row = str_replace(Array("'", "\n",$detailedComp), Array("\\'", "
          ",$detailedCompLong), $row);
      $comp = $row["statemachine"]==''?$component:$row["statemachine"];
      unset($row["statemachine"]);
      foreach($row as $key => $val) $row[$key] = ($val == '')?'undefined
          ':"'$val'";
      echo "\t$comp.Transition(".implode(",", $row).");\n";
    }
  }
  /* end transition */

//  foreach($componentNames as $comp => $true) {
```

```
//      echo "\tstatemachines.push($comp);\n";
//   }
  echo "}\n";
  echo "additionalInit.push(temp);\n";
  echo "sendTrigger('cascade');\n";
}

?>
```

B.16 index.php

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
    html4/strict.dtd">
<HTML>
<HEAD>
<?
if(isset($_REQUEST["components"])) {
  echo "<TITLE>";
  $pageTitle = $_REQUEST["components"];
  if(isset($_REQUEST["id"])) {
    foreach($_REQUEST["id"] as $val) {
      $pageTitle .= "_$val";
    }
  }
  echo $pageTitle." - ";
?>Paradigm UML Tool</TITLE>
<LINK REL="stylesheet" TYPE="text/css" HREF="style.css">
<SCRIPT TYPE='text/javascript' SRC='script/Poll.js'></SCRIPT>
<SCRIPT TYPE='text/javascript' SRC='script/Statemachine.js'></SCRIPT>
<SCRIPT TYPE='text/javascript' SRC='script/Paradigm.js'></SCRIPT>
<SCRIPT TYPE='text/javascript' SRC='script/std.js'></SCRIPT>
<SCRIPT TYPE='text/javascript' SRC='script/umlPrint.js'></SCRIPT>
<!-- SCRIPT TYPE='text/javascript' SRC='script/paradigmPrint.js'></
    SCRIPT -->
<SCRIPT TYPE='text/javascript' SRC='script/getComponents.php?<?
  $temp = Array();
  foreach($_REQUEST as $key => $val) {
    if($key == "locations") continue;
    if(is_array($val)) {
      foreach($val as $key2 => $val2) {
        $temp[] = "$key"."[$key2]=$val2";
      }
    } else {
      $temp[] = "$key=$val";
    }
  }
  echo implode("&amp;", $temp);
?>'></SCRIPT>
<SCRIPT TYPE='text/javascript' SRC='script/script.js'></SCRIPT>
</HEAD>
<BODY ONLOAD='init()' ONUNLOAD='send("leavePage.php?comp="+statemachines
    [0].name+"&db="+database, alert, undefined, true);'>
<FIELDSET STYLE="width: 0px">
<LEGEND>Interface <? echo $pageTitle; ?></LEGEND>
<TABLE>
<TBODY>
<TR><TD><INPUT ID='trigger' VALUE='manager'></TD><TD><BUTTON ONCLICK="
    sendTrigger(document.getElementById('trigger').value);">Trigger</
    BUTTON></TD></TR>
<TR><TD><INPUT ID='randomSpeed' VALUE='0.5'></TD><TD><BUTTON ONCLICK="
    setRandom(document.getElementById('randomSpeed').value*1000);"
    STYLE="white-space: nowrap">Rand Sec</BUTTON></TD></TR>
<TR><TD><INPUT ID='pollSpeed' VALUE='1'></TD><TD><BUTTON ONCLICK="for(
    var it=0;it<polls.length;it++)polls[it].repeat(document.
    getElementById('pollSpeed').value*1000);" STYLE="white-space:
    nowrap">Pol Sec</BUTTON></TD></TR>
<TR><TD COLSPAN=2><BUTTON ONCLICK="for(var it=0;it<polls.length;it++)
    polls[it].repeat(0);">Stop Poll</BUTTON></TD></TR>
<TR><TD COLSPAN=2><BUTTON ONCLICK="statemachines[0].triggerRandom();">
    Trigger Random</BUTTON></TD></TR>
<TR><TD><INPUT ID='toEval' VALUE=''></TD><TD><BUTTON ONCLICK="eval(
    document.getElementById('toEval').value);" STYLE="white-space:
    nowrap">Eval</BUTTON></TD></TR>
```

```
<TR><TD COLSPAN=2><BUTTON ONCLICK="uml.unDraw();">Undraw UML</BUTTON></
    TD></TR>
<TR><TD COLSPAN=2><BUTTON ONCLICK="for(var it = 0; it < statemachines.
    length; it++) uml.draw(statemachines[it], 3, 300, (200*it)+10);">
    Draw UML</BUTTON></TD></TR>
<TR><TD COLSPAN=2><BUTTON ONCLICK="for(var it = 0; it < statemachines.
    length; it++) verbose.draw(statemachines[it]);">Turn verbose ON</
    BUTTON></TD></TR>
<TR><TD COLSPAN=2><BUTTON ONCLICK="verbose.unDraw();">Turn verbose OFF</
    BUTTON></TD></TR>
</TBODY>
</TABLE>
</FIELDSET>
<DIV ID='error' CLASS='report' STYLE='position: absolute; top: 0px; left
    : 1000px; color:red; width: 75%'></DIV>
<IFRAME ID="upload" SRC="ajaxFiles/JITForm.php" CLASS='hideIFrame'></
    IFRAME>
</BODY>
</HTML><?
} else { ?>
<TITLE>Paradigm UML Tool; Select model</TITLE>
<LINK REL="stylesheet" TYPE="text/css" HREF="style.css">
</HEAD>
<BODY>
<?
  $sql = "SELECT *
      FROM models
      WHERE display = 1
  ";
  require_once "/xampp/htdocs/paradigm/query.php";

  $result = query($sql, "paradigmmodels");

  while(($row = mysql_fetch_assoc($result)) !== FALSE) {
    $models[$row["database"]][$row["component"]] = Array("ids" => $row["
        numIds"], "vars" => explode(";", $row["variables"]));
  }

  ksort($models);
  foreach($models as $db => $compArr) {
    ksort($compArr);
    echo "<FIELDSET>\n";
    echo "<LEGEND>$db</LEGEND>\n";
    echo "<TABLE>\n";
    echo "<TBODY>\n";
    foreach($compArr as $comp => $varArr) {
      echo "<FORM METHOD='post'><TR>\n<TD>$comp<INPUT TYPE='hidden'
          VALUE='$db' NAME='db'><INPUT TYPE='hidden' VALUE='$comp' NAME
          ='components'></TD>\n<TD><TABLE><TBODY>\n";
      for($it = 0; $it < $varArr["ids"]; $it++) {
        echo "<TR><TD STYLE='width: 50px'>Id[$it]: </TD><TD STYLE='width
            : 50px'><INPUT TYPE='text' NAME='id[$it]' STYLE='width: 50
            px'></INPUT></TD></TR>";
      }
      echo "</TBODY></TABLE></TD>\n<TD><TABLE><TBODY>\n";
      foreach($varArr["vars"] as $var) {
        if ($var !== "") {
          echo "<TR><TD STYLE='width: 50px'>$var: </TD><TD STYLE='width:
              50px'><INPUT TYPE='text' NAME='var[$var]' STYLE='width:
              50px'></INPUT></TD></TR>";
        }
      }
      echo "</TBODY></TABLE></TD>\n<TD><INPUT TYPE=submit VALUE='Open
          '></INPUT></TD></TR></FORM>\n";
    }
    echo "</TBODY>\n";
    echo "</TABLE>\n";
    echo "</FIELDSET>\n";
  }
?>
</BODY>
</HTML><? } ?>
```

B.17 query.php

```
<?
function query($sql, $db, $server = "localhost", $user = "***", $pass =
    "***") {
  global $mysql_insert_id;
  global $mysql_affected_rows;
  $link = mysql_connect($server, $user, $pass)  or die('Could not
      connect: ' . mysql_error());
  mysql_select_db($db)                 or die('Could not select database: '
       . $db);
  $result = mysql_query($sql)            or die('Query ' . $sql . 'failed
      : ' . mysql_error());
  $mysql_insert_id = mysql_insert_id();
  $mysql_affected_rows = mysql_affected_rows();
  mysql_close($link);
  return $result;
}
?>
```

B.18 img/rounded.php

```
<?php

foreach($_REQUEST as $key => $val) $$key = $val;
if (!isset($w)) $w = 100;
if (!isset($h)) $h = 50;
if (!isset($cw)) $cw = $w/5;
if (!isset($ch)) $ch = $cw;

// create image
$img = imagecreatetruecolor($w, $h);
imageantialias($img, true);

// allocate some solors
$trans  = imagecolorallocate($img, 0xC0, 0xC0, 0xC0);
imagefill($img, 0,0, $trans);
imagecolortransparent($img, $trans);
$white  = imagecolorallocate($img, 0xFF, 0xFF, 0xFF);
$border = imagecolorallocate($img, 0x00, 0x00, 0x00);

// draw the corners and lines between them
imagearc($img, $cw/2-1, $ch/2-1, $cw, $ch, 180, 270, $border);
imageline($img, $cw/2, 0, $w-$cw/2, 0, $border);
imagearc($img, $w-$cw/2, $ch/2 - 1, $cw, $ch, 270, 0, $border);
imageline($img, $w-1, $ch/2, $w-1, $h-$ch/2, $border);
imagearc($img, $w-$cw/2, $h-$ch/2, $cw, $ch, 0, 90, $border);
imageline($img, $cw/2, $h-1, $w-$cw/2, $h-1, $border);
imagearc($img, $cw/2-1, $h-$ch/2, $cw, $ch, 90, 180, $border);
imageline($img, 0, $ch/2, 0, $h-$ch/2, $border);

// fill the corners (non-transparent)
imagefill($img, 0,0, $white);
imagefill($img, $w-1,0, $white);
imagefill($img, 0,$h-1, $white);
imagefill($img, $w-1,$h-1, $white);

//imageellipse($image, $w/2, $h/2, $cw, $ch, $white);

// flush image
header('Content-type: image/png');
imagepng($img);
imagedestroy($img);
?>
```

B.19 style.css

```
.report {
  height:        500px;
  width:         250px;
  overflow:      auto;
  text-align:      left;
```

```
  z-index:        -1;
}
.state, .currentState {
  background:      #F0E68C url('img/rounded.php');
  position:       absolute;
  text-align:      center;
  vertical-align:    middle;
  cursor:         pointer;
  z-index:        2;
}
.currentState {
  background:      #6B8E23 url('img/rounded.php');
  color:          white;
  font-weight:      bold;
  z-index:        2;
}
.line {
  position:       absolute;
  overflow:       visible;
  background:      black;
  border:         0px;
  text-align:      center;
  padding:        0px;
  z-index:        1;
}
.harrow,
.varrow {
  position:       absolute;
  overflow:       visible;
  font-size:       25px;
  left:         -5px;
  top:          -12px;
  font-weight:      900;
  z-index:        1;
}
.harrow {
  top:          -16px;
  left:         -3px;
  z-index:        1;
}
button {
  min-width:       75px;
  width:          100%;
}

.hideIFrame {
  display:        none;
}
.showIFrame {
  border:         0px solid black;
  position:       absolute;
  left:         380px;
  top:          70px;
  z-index:        100;
}
.uploadForm {
  border:         1px solid black;
  width:          225px;
  background:      #AAAAAA;
}
```

B.20 sampleMigrFiles/MigrDetSchedExtWorker.migr

```
Add: McPal add:/remove: is enough, the rest can be viewed as comments
stateMachine:'Migr',from:'StablePhase',to:'doMigr',trigger:'ready',guard
    :'getSTD(this.from.name).isCurrent(this.trigger);',activity:'
    changeSubprocess(this.from.name,this.to.name);'
stateMachine:'Migr',from:'doMigr',to:'StablePhase',trigger:'migrDone',
    guard:'getSTD(this.from.name).isCurrent(this.trigger);',activity:'
    changeSubprocess(this.from.name,this.to.name);'
stateMachine:'doMigr',from:'triv',to:'migrDone',trigger:'cascade',guard:
    'getSTD("McPal").isTrapped("Content,Observing")',activity:'
    sendTrigger("cascade")'
```

```
stateMachine:'McPal',from:'StartMigr',to:'Both1to2',guard:'paradigmGuard
      (this,"Migr.doMigr","Scheduler.Migr.Stable.triv&Worker_0.Migr.
      StablePhase.triv&Worker_1.Migr.StablePhase.triv&Worker_2.Migr.
      StablePhase.triv")',activity:'sendTrigger("cascade")'
stateMachine:'McPal',from:'Both1to2',to:'Content',guard:'paradigmGuard(
      this,"Migr.doMigr","Scheduler.Migr.SchedMigr.finished&Worker_0.CSM.
      OutCS.triv.OutCS")',activity:'sendTrigger("cascade")'

Remove: McPal
stateMachine:'doMigr'
stateMachine:'Migr',stateName:'doMigr'
stateMachine:'McPal',stateName:'Both1to2'
stateMachine:'McPal',from:'NewRuleSet',to:'StartMigr'
stateMachine:'McPal',from:'Content',to:'Observing'



Add: Scheduler
compName:'Scheduler',stateMachine:'Scheduler',from:'check_0',to:'asg_0',
      trigger:'allow_0',guard:'paradigmGuard(this,"Migr.SchedMigr;Migr.
      SchedPhase2","Worker_0.CSM.OutCSBlock.entering");',activity:'
      sendTrigger("cascade")'
compName:'Scheduler',stateMachine:'Scheduler',from:'check_1',to:'asg_1',
      trigger:'allow_1',guard:'paradigmGuard(this,"Migr.SchedMigr;Migr.
      SchedPhase2","Worker_1.CSM.OutCSBlock.entering");',activity:'
      sendTrigger("cascade")'
compName:'Scheduler',stateMachine:'Scheduler',from:'check_2',to:'asg_2',
      trigger:'allow_2',guard:'paradigmGuard(this,"Migr.SchedMigr;Migr.
      SchedPhase2","Worker_2.CSM.OutCSBlock.entering");',activity:'
      sendTrigger("cascade")'
compName:'Scheduler',stateMachine:'Scheduler',from:'check_0',to:'check_1
      ',trigger:'skip_0',guard:'paradigmGuard(this,"Migr.SchedMigr;Migr.
      SchedPhase2","Worker_0.CSM.OutCSBlock.stay&Worker_1.CSM.OutCS.triv.
      OutCSBlock");',activity:'sendTrigger("cascade");'
compName:'Scheduler',stateMachine:'Scheduler',from:'check_1',to:'check_2
      ',trigger:'skip_1',guard:'paradigmGuard(this,"Migr.SchedMigr;Migr.
      SchedPhase2","Worker_1.CSM.OutCSBlock.stay&Worker_2.CSM.OutCS.triv.
      OutCSBlock");',activity:'sendTrigger("cascade");'
compName:'Scheduler',stateMachine:'Scheduler',from:'check_2',to:'check_0
      ',trigger:'skip_2',guard:'paradigmGuard(this,"Migr.SchedMigr;Migr.
      SchedPhase2","Worker_2.CSM.OutCSBlock.stay&Worker_0.CSM.OutCS.triv.
      OutCSBlock");',activity:'sendTrigger("cascade");'

compName:'Scheduler',stateMachine:'Scheduler',from:'asg_0',to:'check_1',
      trigger:'revoke_0',guard:'paradigmGuard(this,"Migr.SchedMigr;Migr.
      SchedPhase2","Worker_0.CSM.InCS.left&Worker_1.CSM.OutCS.triv.
      OutCSBlock");',activity:'sendTrigger("cascade");'
compName:'Scheduler',stateMachine:'Scheduler',from:'asg_1',to:'check_2',
      trigger:'revoke_1',guard:'paradigmGuard(this,"Migr.SchedMigr;Migr.
      SchedPhase2","Worker_1.CSM.InCS.left&Worker_2.CSM.OutCS.triv.
      OutCSBlock");',activity:'sendTrigger("cascade");'
compName:'Scheduler',stateMachine:'Scheduler',from:'asg_2',to:'check_0',
      trigger:'revoke_2',guard:'paradigmGuard(this,"Migr.SchedMigr;Migr.
      SchedPhase2","Worker_2.CSM.InCS.left&Worker_0.CSM.OutCS.triv.
      OutCSBlock");',activity:'sendTrigger("cascade");'

compName:'Scheduler',stateMachine:'Scheduler',from:'asg_0',to:'check_1',
      trigger:'revoke_0',guard:'paradigmGuard(this,"Migr.SchedMigr","
      Worker_0.CSM.Busy.done.OutCS&Worker_1.CSM.Free.triv.OutCSBlock&
      Worker_2.CSM.Free.triv.OutCS");',activity:'sendTrigger("cascade");'
compName:'Scheduler',stateMachine:'Scheduler',from:'asg_1',to:'check_2',
      trigger:'revoke_1',guard:'paradigmGuard(this,"Migr.SchedMigr","
      Worker_1.CSM.Busy.done.OutCS&Worker_2.CSM.Free.triv.OutCSBlock&
      Worker_0.CSM.Free.triv.OutCS");',activity:'sendTrigger("cascade");'
compName:'Scheduler',stateMachine:'Scheduler',from:'asg_2',to:'check_0',
      trigger:'revoke_2',guard:'paradigmGuard(this,"Migr.SchedMigr","
      Worker_2.CSM.Busy.done.OutCS&Worker_0.CSM.Free.triv.OutCSBlock&
      Worker_1.CSM.Free.triv.OutCS");',activity:'sendTrigger("cascade");'

compName:'Scheduler',stateMachine:'Scheduler',from:'idle',to:'check_0',
      trigger:'leaveIdle',guard:'paradigmGuard(this,"Migr.SchedMigr","
      Worker_0.CSM.Free.triv.OutCSBlock&Worker_1.CSM.Free.triv.OutCS&
      Worker_2.CSM.Free.triv.OutCS");',activity:'sendTrigger("cascade");'
```

```
compName:'Scheduler',stateMachine:'SchedMigr',from:'triv',to:'finished',
     trigger:'cascade',guard:'getSTD("Scheduler").isTrapped("asg_0,
     check_0,asg_1,check_1,asg_2,check_2")',activity:'sendTrigger("
     cascade");'
compName:'Scheduler',stateMachine:'SchedPhase2',stateName:'triv'
compName:'Scheduler',stateMachine:'Migr',from:'Stable',to:'SchedMigr',
     trigger:'triv',guard:'getSTD(this.from.name).isCurrent(this.trigger
     );',activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Scheduler',stateMachine:'Migr',from:'SchedMigr',to:'
     SchedPhase2',trigger:'finished',guard:'getSTD(this.from.name).
     isCurrent(this.trigger);',activity:'changeSubprocess(this.from.name
     ,this.to.name);'

Remove: Scheduler
compName:'Scheduler',stateMachine:'Scheduler',stateName:'idle'
compName:'Scheduler',stateMachine:'Stable'
compName:'Scheduler',stateMachine:'SchedMigr'
compName:'Scheduler',stateMachine:'Migr',stateName:'Stable'
compName:'Scheduler',stateMachine:'Migr',stateName:'SchedMigr'



Add: Worker_0
compName:'Worker_0',stateMachine:'CSM',from:'Free',to:'OutCS',trigger:'
     triv',guard:'getSTD(this.from.name).isCurrent(this.trigger);',
     activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_0',stateMachine:'CSM',from:'Free',to:'OutCSBlock',
     trigger:'triv',guard:'getSTD(this.from.name).isCurrent(this.trigger
     );',activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_0',stateMachine:'CSM',from:'Busy',to:'OutCS',trigger:'
     done',guard:'getSTD(this.from.name).isCurrent(this.trigger);',
     activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_0',stateMachine:'CSM',from:'OutCS',to:'OutCSBlock',
     trigger:'triv',guard:'getSTD(this.from.name).isCurrent(this.trigger
     );',activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_0',stateMachine:'CSM',from:'OutCSBlock',to:'OutCS',
     trigger:'stay',guard:'getSTD(this.from.name).isCurrent(this.trigger
     );',activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_0',stateMachine:'CSM',from:'OutCSBlock',to:'InCS',
     trigger:'entering',guard:'getSTD(this.from.name).isCurrent(this.
     trigger);',activity:'changeSubprocess(this.from.name,this.to.name);
     '
compName:'Worker_0',stateMachine:'CSM',from:'InCS',to:'OutCS',trigger:'
     left',guard:'getSTD(this.from.name).isCurrent(this.trigger);',
     activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_0',stateMachine:'OutCS',stateName:'triv'
compName:'Worker_0',stateMachine:'OutCSBlock',from:'triv',to:'stay',
     trigger:'cascade',guard:'getSTD("Worker_0").isTrapped("post,free,
     nonCrit");',activity:'sendTrigger("cascade");'
compName:'Worker_0',stateMachine:'OutCSBlock',from:'triv',to:'entering',
     trigger:'cascade',guard:'getSTD("Worker_0").isTrapped("pre");',
     activity:'sendTrigger("cascade");'
compName:'Worker_0',stateMachine:'InCS',from:'triv',to:'left',trigger:'
     cascade',guard:'getSTD("Worker_0").isTrapped("post,nonCrit,free");'
     ,activity:'sendTrigger("cascade");'
compName:'Worker_0',stateMachine:'Migr',from:'StablePhase',to:'Phase2',
     trigger:'triv',guard:'getSTD(this.from.name).isCurrent(this.trigger
     );',activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_0',stateMachine:'Phase2',stateName:'triv'
compName:'Worker_0',stateMachine:'CSM',from:'OutCS',to:'OutCS',trigger:'
     triv',guard:'getSTD(this.from.name).isCurrent(this.trigger);',
     activity:'changeSubprocess(this.from.name,this.to.name);'

compName:'Worker_0',stateMachine:'Worker_0',from:'free',to:'nonCrit',
     trigger:'begin',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Free;CSM
     .OutCS;CSM.OutCSBlock;CSM.InCS");',activity:'sendTrigger("cascade")
     ;'
compName:'Worker_0',stateMachine:'Worker_0',from:'nonCrit',to:'pre',
     trigger:'occupy',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Busy;
     CSM.OutCS");',activity:'sendTrigger("cascade");'
compName:'Worker_0',stateMachine:'Worker_0',from:'pre',to:'crit',trigger
     :'pickUp',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Busy;CSM.InCS
     ");',activity:'sendTrigger("cascade");'
```

```
compName:'Worker_0',stateMachine:'Worker_0',from:'crit',to:'post',
    trigger:'layDown',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Busy;
    CSM.InCS");',activity:'sendTrigger("cascade");'
compName:'Worker_0',stateMachine:'Worker_0',from:'post',to:'free',
    trigger:'finish',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Busy;
    CSM.OutCS;CSM.OutCSBlock;CSM.InCS");',activity:'sendTrigger("
    cascade");'

Remove: Worker_0
compName:'Worker_0',stateMachine:'CSM',stateName:'Free'
compName:'Worker_0',stateMachine:'CSM',stateName:'Busy'
compName:'Worker_0',stateMachine:'Free'
compName:'Worker_0',stateMachine:'Busy'
compName:'Worker_0',stateMachine:'Migr',stateName:'StablePhase'
compName:'Worker_0',stateMachine:'StablePhase'



Add: Worker_1
compName:'Worker_1',stateMachine:'CSM',from:'Free',to:'OutCS',trigger:'
    triv',guard:'getSTD(this.from.name).isCurrent(this.trigger);',
    activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_1',stateMachine:'CSM',from:'Free',to:'OutCSBlock',
    trigger:'triv',guard:'getSTD(this.from.name).isCurrent(this.trigger
    );',activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_1',stateMachine:'CSM',from:'Busy',to:'OutCS',trigger:'
    done',guard:'getSTD(this.from.name).isCurrent(this.trigger);',
    activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_1',stateMachine:'CSM',from:'OutCS',to:'OutCSBlock',
    trigger:'triv',guard:'getSTD(this.from.name).isCurrent(this.trigger
    );',activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_1',stateMachine:'CSM',from:'OutCSBlock',to:'OutCS',
    trigger:'stay',guard:'getSTD(this.from.name).isCurrent(this.trigger
    );',activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_1',stateMachine:'CSM',from:'OutCSBlock',to:'InCS',
    trigger:'entering',guard:'getSTD(this.from.name).isCurrent(this.
    trigger);',activity:'changeSubprocess(this.from.name,this.to.name);
    '
compName:'Worker_1',stateMachine:'CSM',from:'InCS',to:'OutCS',trigger:'
    left',guard:'getSTD(this.from.name).isCurrent(this.trigger);',
    activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_1',stateMachine:'OutCS',stateName:'triv'
compName:'Worker_1',stateMachine:'OutCSBlock',from:'triv',to:'stay',
    trigger:'cascade',guard:'getSTD("Worker_1").isTrapped("post,free,
    nonCrit");',activity:'sendTrigger("cascade");'
compName:'Worker_1',stateMachine:'OutCSBlock',from:'triv',to:'entering',
    trigger:'cascade',guard:'getSTD("Worker_1").isTrapped("pre");',
    activity:'sendTrigger("cascade");'
compName:'Worker_1',stateMachine:'InCS',from:'triv',to:'left',trigger:'
    cascade',guard:'getSTD("Worker_1").isTrapped("post,nonCrit,free");'
    ,activity:'sendTrigger("cascade");'
compName:'Worker_1',stateMachine:'Migr',from:'StablePhase',to:'Phase2',
    trigger:'triv',guard:'getSTD(this.from.name).isCurrent(this.trigger
    );',activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_1',stateMachine:'Phase2',stateName:'triv'

compName:'Worker_1',stateMachine:'Worker_1',from:'free',to:'nonCrit',
    trigger:'begin',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Free;CSM
    .OutCS;CSM.OutCSBlock;CSM.InCS");',activity:'sendTrigger("cascade")
    ;'
compName:'Worker_1',stateMachine:'Worker_1',from:'nonCrit',to:'pre',
    trigger:'occupy',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Busy;
    CSM.OutCS");',activity:'sendTrigger("cascade");'
compName:'Worker_1',stateMachine:'Worker_1',from:'pre',to:'crit',trigger
    :'pickUp',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Busy;CSM.InCS
    ");',activity:'sendTrigger("cascade");'
compName:'Worker_1',stateMachine:'Worker_1',from:'crit',to:'post',
    trigger:'layDown',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Busy;
    CSM.InCS");',activity:'sendTrigger("cascade");'
compName:'Worker_1',stateMachine:'Worker_1',from:'post',to:'free',
    trigger:'finish',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Busy;
    CSM.OutCS;CSM.OutCSBlock;CSM.InCS");',activity:'sendTrigger("
    cascade");'
```

```
Remove: Worker_1
compName:'Worker_1',stateMachine:'CSM',stateName:'Free'
compName:'Worker_1',stateMachine:'CSM',stateName:'Busy'
compName:'Worker_1',stateMachine:'Free'
compName:'Worker_1',stateMachine:'Busy'
compName:'Worker_1',stateMachine:'Migr',stateName:'StablePhase'
compName:'Worker_1',stateMachine:'StablePhase'



Add: Worker_2
compName:'Worker_2',stateMachine:'CSM',from:'Free',to:'OutCS',trigger:'
     triv',guard:'getSTD(this.from.name).isCurrent(this.trigger);',
     activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_2',stateMachine:'CSM',from:'Free',to:'OutCSBlock',
     trigger:'triv',guard:'getSTD(this.from.name).isCurrent(this.trigger
     );',activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_2',stateMachine:'CSM',from:'Busy',to:'OutCS',trigger:'
     done',guard:'getSTD(this.from.name).isCurrent(this.trigger);',
     activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_2',stateMachine:'CSM',from:'OutCS',to:'OutCSBlock',
     trigger:'triv',guard:'getSTD(this.from.name).isCurrent(this.trigger
     );',activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_2',stateMachine:'CSM',from:'OutCSBlock',to:'OutCS',
     trigger:'stay',guard:'getSTD(this.from.name).isCurrent(this.trigger
     );',activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_2',stateMachine:'CSM',from:'OutCSBlock',to:'InCS',
     trigger:'entering',guard:'getSTD(this.from.name).isCurrent(this.
     trigger);',activity:'changeSubprocess(this.from.name,this.to.name);
     '
compName:'Worker_2',stateMachine:'CSM',from:'InCS',to:'OutCS',trigger:'
     left',guard:'getSTD(this.from.name).isCurrent(this.trigger);',
     activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_2',stateMachine:'OutCS',stateName:'triv'
compName:'Worker_2',stateMachine:'OutCSBlock',from:'triv',to:'stay',
     trigger:'cascade',guard:'getSTD("Worker_2").isTrapped("post,free,
     nonCrit");',activity:'sendTrigger("cascade");'
compName:'Worker_2',stateMachine:'OutCSBlock',from:'triv',to:'entering',
     trigger:'cascade',guard:'getSTD("Worker_2").isTrapped("pre");',
     activity:'sendTrigger("cascade");'
compName:'Worker_2',stateMachine:'InCS',from:'triv',to:'left',trigger:'
     cascade',guard:'getSTD("Worker_2").isTrapped("post,nonCrit,free");'
     ,activity:'sendTrigger("cascade");'
compName:'Worker_2',stateMachine:'Migr',from:'StablePhase',to:'Phase2',
     trigger:'triv',guard:'getSTD(this.from.name).isCurrent(this.trigger
     );',activity:'changeSubprocess(this.from.name,this.to.name);'
compName:'Worker_2',stateMachine:'Phase2',stateName:'triv'

compName:'Worker_2',stateMachine:'Worker_2',from:'free',to:'nonCrit',
     trigger:'begin',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Free;CSM
     .OutCS;CSM.OutCSBlock;CSM.InCS");',activity:'sendTrigger("cascade")
     ;'
compName:'Worker_2',stateMachine:'Worker_2',from:'nonCrit',to:'pre',
     trigger:'occupy',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Busy;
     CSM.OutCS");',activity:'sendTrigger("cascade");'
compName:'Worker_2',stateMachine:'Worker_2',from:'pre',to:'crit',trigger
     :'pickUp',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Busy;CSM.InCS
     ");',activity:'sendTrigger("cascade");'
compName:'Worker_2',stateMachine:'Worker_2',from:'crit',to:'post',
     trigger:'layDown',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Busy;
     CSM.InCS");',activity:'sendTrigger("cascade");'
compName:'Worker_2',stateMachine:'Worker_2',from:'post',to:'free',
     trigger:'finish',guard:'paradigmGuard(this,"Migr.Phase2;CSM.Busy;
     CSM.OutCS;CSM.OutCSBlock;CSM.InCS");',activity:'sendTrigger("
     cascade");'

Remove: Worker_2
compName:'Worker_2',stateMachine:'CSM',stateName:'Free'
compName:'Worker_2',stateMachine:'CSM',stateName:'Busy'
compName:'Worker_2',stateMachine:'Free'
compName:'Worker_2',stateMachine:'Busy'
compName:'Worker_2',stateMachine:'Migr',stateName:'StablePhase'
compName:'Worker_2',stateMachine:'StablePhase'
```