Reflections on
Functional Modelling in SOCCA

Maarten Steenhuis
student number 8839670

Leiden University, Department of Computer Science
Niels Bohrweg 1
2333 CA Leiden, The Netherlands

August, 1995

Contents

1 Introduction

In recent years students and staff from the Computer Science Department of Leiden University have been working on a modelling method for the description, analysis and specification of the software process. This method (called *SOCCA*: *Specification of Coordinated and Cooperative Activities* [Engels 94]) is at present still under development.

A SOCCA model describes the software process from three different perspectives: the data perspective, the behaviour perspective and the process perspective. It was decided to use a separate modelling formalism for each of the different perspectives. This kind of modelling is not uncommon in the realm of object-oriented modelling techniques, like Object Modelling Technique (OMT) [Rumbaugh 91] or OOA/OOD [Coad 91].
Until now most work on SOCCA has been devoted to the models for the data and behaviour perspectives. Two existing formalisms have been adopted to achieve this.
Class diagram modelling, based on EER concepts, has been chosen for the **data perspective**. This technique has the advantage of being reasonably well understood and having a clear diagrammatic representation.
The **behaviour perspective** is covered by the use of *state transition diagrams* (STDs). The SOCCA approach however differs from many behaviour description models using STDs. First of all, a distinction is made between *external* and *internal* behaviour(s) of the objects modelled. Secondly, much effort is made to create a detailed picture of the communication between objects and concurrency of different behaviours in one single object. For this purpose PARADIGM is used, a formalism developed by Groenewegen and others [Groenewegen 86] to model parallel and concurrent processes.

As I started out to work on this thesis, little attention had been paid to the **process perspective** in SOCCA. The general idea was, that as in the case of the structural (or data) and behavioral perspectives an existing modelling formalism should be adopted and possibly adapted to cover for this process or *functional* perspective. At that stage ideas about what formalism to adopt and the modelling capabilities of such a technique were still somewhat vague.

So it was decided that I should

- investigate the modelling capabilities of the candidate formalism for the process perspective, which was already baptized the formalism of *Object Flow Diagrams* (OFDs) in the discussions on SOCCA;
- apply it to the field of software process modelling and especially to the problems the SOCCA research was focusing on;
- show how this formalism of OFDs, which I immediately called OFM or *Object Flow Modelling*, could be integrated into the SOCCA method, and explain the relation it would have to the other two modelling techniques used.

This task in fact proved to be not that simple. Looking back, I see two reasons why the work didn't quite work out the way I planned it:

- I held a different view from people working SOCCA about the phenomena a functional or process model should be able to model, and therefore about the nature of the process perspective. It took some time for me to realize this.
- I soon found out, that in fact there was no one existing formalism of OFDs. In [Engels 94] references were made to two different formalisms: the Visual Action Language (ViAL) and the formalism of Data Flow Diagrams (DFDs). It was still unclear how the formalism of OFDs related to the concepts underlying both.

So in fact what did I do? First of all I tried to make it more clear how a functional model should be incorporated into SOCCA and what it should model. In chapter 2 we therefore take a short look at SOCCA, the general field software process modelling and the different modelling perspectives involved. After this chapter we should have a better view on the phenomena that to be captured by a process or functional perspective and accordingly on the desired properties of the modelling technique used.

Chapter 3 and 4 take a look at the two sources of inspiration mentioned in the discussion on process modelling in SOCCA, and discuss their merits:

- the well known formalism of *Data Flow Diagrams* (DFDs).
- the *Visual Action Language* ViAL, developed at the Technical University of Braunschweig, Germany in the context of the CADDY project.

The chapters should explain why ViAL and the formalism of DFDs are of different nature and what concepts underlying both could be incorporated into the SOCCA method and formalisms.

In chapter 5 I will clarify some of the concepts fundamental to the behaviour model in SOCCA by showing how they are related to the *usage structure* from the class model, and the concepts of *view* and *object situation*. For myself I found great help in the understanding of the behaviour model from the introduction of these concepts.

Chapter 6 discusses how integrity rules expressed by the SOCCA class model could be incorporated into the existing behaviour model of SOCCA, or are already fulfilled by the existing models.

Although it was my original ambition to present and apply a formalism for descriptions of the transformations of objects of the SOCCA model, and how objects and values flow from one transformation to another, little or no work has actually been done on the development of such a technique. In chapter 7 I will try to indicate in what direction the development of a formalism should proceed and how in my view it should relate to the existing submodels.

Finally in chapter 8 the results of the work will be briefly reviewed and some concluding remarks will be made.

2 Software Process Modelling and SOCCA

2.1 Software process modelling

Software process modelling deals with the entire process of software development. Although the subject of software development or software engineering is broadly dealt with in computer science, the field of software process modelling is relatively new; its origins in fact seem to date from the late eighties. In software process modelling the emphasis shifts from the *products* or *artifacts* that are the result of the software development process to the properties of the software process itself. With the field of software process modelling (SPM for short) the software engineering *methodology* evolves from a *generator* of methods, life-cycle models, modelling formalisms and languages, to a true *science* of the software process.

One might ask about the *objectives* for research programs in the field of software process modelling.
In [Curtis 92] five basic categories of objectives for software process modelling, ranging from understanding aids to automated execution support, are presented:

- *Facilitate human understanding and communication*
  Software development is teamwork. And, what may be even worse, the activities involved are usually carried out by team members specialized in different disciplines, having different interests and all too often using different 'languages'. An adequate model of the software process should help to represent the process in a form understandable by humans, and enable them to communicate about what should be done.

- *Support process improvements*
  In analysing processes and defining a basis for their understanding SPM can compare alternative software processes, estimate the impacts of potential changes to a software process without first putting them into actual practice, and assist in the selection and incorporation of technology (e.g. tools)

- *Support process management*
  SPM could help to deliver indicators that enable the monitoring, managing and coordination of the software process, and support development of plans for software development projects (forecast)[1].

- *Automate execution support*
  The software process models established can be the basis for tools and procedures that can automate or facilitate some of the activities involved. Indicators of the performance and guidance of the software project could then be automatically collected. Rules to ensure process integrity could be enforced.

[1] This will demand a defined or idealized project against which the actual project behaviour can be compared

- *Automate process guidance*
  Specification of what should be done by whom, how and at what time is a cumbersome task that is usually left to the person designated as project manager. All too often this task is reinvented any time a new software development project is launched. A clear and formal understanding of the software process can help to automate and facilitate this task; by retaining results from previous endeavours in a repository some knowledge and data could possibly be reused.

The issues raised by these objectives range from comprehensibility to enactability. Consequently, many forms of information must be modelled and integrated to adequately describe the software process. One way to deal with this, is to model the software process from different *perspectives*, and to concentrate on different aspects of the process for each of these perspective.
Curtis *et al*. [Curtis 92] justly discern four different views or *perspectives*:

- a *functional* or *process* perspective, representing what process elements are being performed, and what flows of informational entities (e.g. data, artifacts, products) are relevant to these process elements. In a way this functional perspective coincides with the classical view of the field of software engineering, in which the software process is analyzed by its functional behaviour;

- a *behavioral* perspective, representing when process elements are performed (e.g. sequencing), as well as aspects of how they are performed through feedback loops, iteration, complex decision-making conditions, entry and exit criteria and so forth;

- an *informational* perspective, representing the informational entities produced or manipulated by a process; these entities include data, artifacts, products (intermediate and end), and other objects; this perspective includes both the structure of informational entities and the relationships between them;

- an *organizational* perspective, representing where and by whom (which *agents*) in the organization process elements are performed, the (physical) communication mechanisms used for transfer of entities or messages, and the (physical) media and locations used for storing entities.

These perspectives underlie separate yet interrelated representations for analysing and presenting process information. In my view, the first three of these perspectives are most appropriate for analysing the process as an *idealized* model; here one might concentrate on the artifacts that should be produced, the functionality that produces them, and the behaviour of agents when assigned to a certain *role*, i.e. a coherent set of process elements as a unit of functional responsibility. The organizational perspective is important when dealing with implementation or enactment of the software process, and could be compared to the *system architecture view* in a software development process.

Although software process modelling was a new activity focus of software engineering science, its objects of interest are of course compatible with process analysis and modelling underlying the architecture of many software systems. As is stated in [Engels 94] however, the model should not only account for the behaviour oof the technical parts of the software process, but should actually account for the human components adding to its functionality.

As much as it was recognized in the field of software engineering, the different perspectives important to software process modelling can effectively be dealt with by a *multi-paradigm approach*, i.e. using different kind of modelling and different types of modelling formalisms to account for each perspective.
The advantage of such an approach is a certain ease of modelling. One simply chooses the modelling language or formalism best fit to express the phenomena one is concentrating upon. The drawback is an extra model complexity: it is hard to interrelate the different perspectives when modelled in different languages.

Using a multi-paradigm approach certain conditions must be met:

- there must be a kind of *orthogonality* between the different modelling formalisms. Phenomena modelled by one (sub)formalism should not be modelled in another (sub)formalism. In fact, as the alternative formalisms were chosen for each perspective, phenomena modelled by one formalism should in general be harder to model in another formalism. Further on, we should try and minimize our modelling effort.

- it should be possible to *interrelate* the techniques used. In developing a *method* for the description of a software process using different modelling formalisms, interrelating the models can be considered a separate task. And it is certainly not a trivial one.

- it should be clear what *perspective* will be covered by which modelling approach or *paradigm*. This is not always obvious. In the *Design* of a system in OMT, object modelling and behaviour modelling can be added to models that account for the system architecture in the organizational perspective of the system.

There are a few more criteria one might add to the list:

- One must demand from all modelling formalisms that they constitute a firm basis for the understanding of and discussion about the related phenomena involved. This can best be met by using well-understood modelling languages or formalisms, having proved themselves in more than one specific context, and possibly of a diagrammatic or visual form[2].
- Formalisms must be able to deal with a considerable level of complexity.
- Models should be able to cope with very different software processes organized in accordance with the wide range of established software engineering methods or methodologies[3].

[2] As is argued in [Hennemann 91], p. 24-25 this usually helps to a better understanding. In [Petre 95] however, there is an interesting discussion on this viewpoint.

[3] There are 2 different interpretations in literature for the term *software engineering methodology*. Sometimes a methodology refers to a consistent complex of different methods. In other discussions the term is reserved for its proper sense: a science or study of different methods. Here of course we refer to its first meaning

2.2 SOCCA

Having taken a quick look at the objectives of software process modelling and the features and requirements for a multi-paradigm approach we can now take a closer look at SOCCA.
As was stated in the introductory chapter SOCCA is at present still under development. So in many aspects it is not a complete method for the analysis and specification of the software process.

SOCCA is a multi-paradigm software process specification method which handles three different perspectives: the *data, process* and *behaviour* perspective.

Let us first shortly review the formalisms adopted so far.
For the static and structural description of the processes involved, SOCCA uses class or rather *object modelling*, taken from OMT [Rumbaugh 91], and by that it is confessing to the object-oriented paradigm. Consequently the static description not only shows the bare data structure of the system and its components, but also the operations or methods by which it can be manipulated. This formalism can deal with complex structures, and has a diagrammatic representation. Furthermore, it has risen a wide interest and is therefore reasonably well known and much commented upon.

The specification method or modelling formalism PARADIGM was adopted for the behaviour perspective. PARADIGM was originally developed for and restricted to the specification of parallel processes. It is defined and presented in [Groenewegen 86]. Several applications of this formalism can be found in [Steen 88] and [Morssink 93].
The use of PARADIGM certainly is not as widespread as for instance Harel's Statechart technique [Harel 87], which was adopted in OMT for dynamic modelling. And, in my view, its diagrammatic representation has not the elegance of Harel's Statecharts.
Still, there were reasons for adopting this formalism and not use OMTs dynamic modelling:

- PARADIGM was conceived, used and elaborated at Leiden University, so there was much interest in and expertise on the formalism.
- PARADIGM seems better equipped for the modelling of complex dynamical systems. Coordination and communication between different processes can be modelled very sophisticated using its notions of subprocess, trap and manager process.

As was noted in the Introduction little attention has been paid to the *process* or *functional* perspective, and the adoption of a modelling method for this perspective. Both a better understanding of the phenomena to be covered by the model and the modelling capacities of such a formalism are needed.

Now there are two factors hampering this understanding, in my view:

- The literature on SOCCA has not been very clear yet about the research *objectives*, which will influence the phenomena one wants to study or model and the techniques used for modelling. However with the adoption of PARADIGM as a tool for the modelling of model dynamics, the emphasis is likely to be put on the *analysis* of software processes, rather than on the enactment or design of computer-aided software development environments.

- There is a possible source for misunderstanding when discussing the functional or process models in SOCCA (and similar multi-paradigm methods like OMT). Actually, one can consider two types of functional or process models.
  The first one deals with the *process* or *functional* perspective of the software process that was discussed in the previous section. The other 'process perspective' has to do with understanding one model in terms of other models, i.e. *interrelating* the different paradigms. So when modelling behavioral phenomena, like state changes in an object, one is anxious to *understand* these phenomena in terms of other models, like the object model. However, in the analysis of the effect of predefined export operations on objects, who will certainly find their place in any functional model, both perspectives will again coincide.

The rest of this thesis should contribute to the understanding of the concepts involved, by clarifying some problems, and give hints for their solution or understanding, however incomplete.

3 Data Flow Diagrams and Data Flow Modelling

3.1 Data Flow Diagrams

Data flow diagrams (DFDs) are a well-known and widely used notation for the functionality of an information system. The diagrams, also known as 'bubble charts', have become popular with their use in the SA/SD method [Constantine 79][4]. A data flow diagram is a graph showing the flow of data values from their sources through *processes* that transform them to their destination. So data flow diagrams are function-oriented: they do not show control information such as the time at which transformations are made, or decisions among alternate paths. A data flow diagram does not show the organization of data values into structures. The attractive and rather intuitive graphical notation makes DFDs easy to use.
A data flow diagram is particularly useful for showing the high-level functionality of a system and its breakdown into smaller units. One could say that the use of DFDs is more or less 'natural' in top-down methods, because breakdown and refinement are somehow inherent to the formalism[5].

The DFD notation is not standardized. Since the formalism is widely used there are many slightly different definitions and notations. Here I shall follow the notation and terminology of functional modelling in OMT [Rumbaugh 91]. The basic elements of DFDs (shown in Figure 1) are:

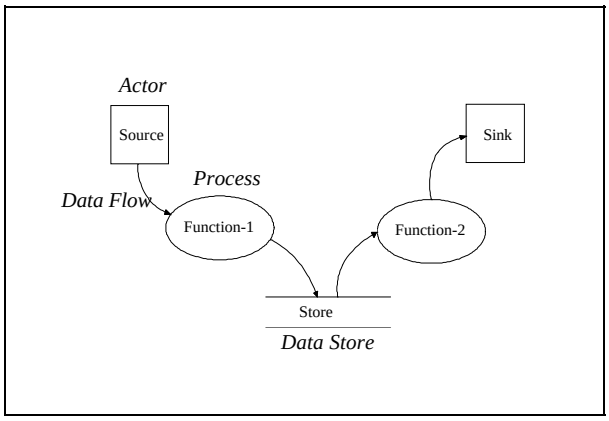

Figure 1. Elements of data flow diagrams

- *processes* (or *bubbles*), used to represent functions or transformation of data values;
- *data flows* (or *arrows*), used to connect the processes, and designating the flow of data or information from one process to another;
- *actors* (or *terminators*), that are the external boundaries to the system represented, that drive the data flow by producing or consuming values. These actors lie on the 'boundary' of the data flow graph, and terminate the 'flow' of data as sources and sinks.
- *data stores* represent values 'tucked away' from the direct influence of the system for sometime, and to be used later on.

---

[4] sometimes slightingly called 'doing bubbles and arrows'

[5] The discussion on ViAL in the next chapter will show a formalism (or rather language) having a bottom-up or constructive nature.

To get a view on the modelling power of DFDs and the use of high-level functional modelling in software process modelling an example of a data flow model is given in the next section.

3.2 The ISPW-6 and ISPW-7 case

The modelling efforts of the research on SOCCA have been concentrating to the problem descriptions for software process modelling known as the ISPW-6 and ISPW-7 cases. The ISPW-6 software process example [Kellner 91] was constructed in conjunction with the 6th International Software Process Workshop. It consists of a rather informal description of a realistic software change process, to be used as an example problem to which various modelling approaches can be applied, and through which they can be evaluated and compared. The example focuses on the designing, coding, unit testing and management of a rather localized change to a software system. The ISPW-7 case [Kellner 91a] provides a set of extensions to the ISPW-6 example.

The core problem for the ISPW-6 case is the change of a single code unit (e.g. a module) in an existing software system. The change is executed by a project team, and initiated by an external authority, called the Configuration Control Board (CCB). The problem is presented as a consistent collection of steps or tasks performed by the different members of the project team.
Eight different steps or tasks are given in the example problem:

- *Modify design*
  This step involves the modification of the design for the code unit affected by the requirements change received from the CCB. The modified design will be reviewed, and ultimately implemented in code. This step may also modify the design based upon feedback from the design review.
- *Review design*
  This step involves the formal review of the modified design.
- *Modify code*
  This step involves the implementation of the design changes into code, and compilation of the modified source code into object code. It may also be based on feedback from testing, indicating that additional source code modifications are required.
- *Test unit*
  This step involves the application of a unit test package on the modified code unit, and the analysis of the results. If all tests are successfully completed, then the unit has successfully passed. In that case the example process has come to an end. Steps beyond unit testing, such as integration testing, are beyond the scope of the example core problem.
- *Modify unit test package*
  This step involves the modification of the actual unit test package for the affected code unit, in accordance with the modifications made to the test plans and objectives. Subsequent iterations of this step may be based upon feedback from testing, indicating that additional modifications to the unit test package are required.
- *Modify test plans*
  This step involves the modification of test plans and objectives to include testing of the specific capabilities related to the requirements change underlying this software modification. The test plan is the blueprint for the actual unit test, as the design is for the actual code unit.
- *Schedule and assign tasks*
  This step is a project management function. It involves developing a schedule for the work to be undertaken, and assigning individual tasks to specific staff members.

- *Monitor progress*
  This step involves the project manager monitoring progress and status of work. This monitoring is based upon notification of completion of each step, together with the (formal) results of the step undertaken.

As the description of the various tasks is process oriented, the process description is captured very naturally in the dataflow diagram of Figure 2[6].

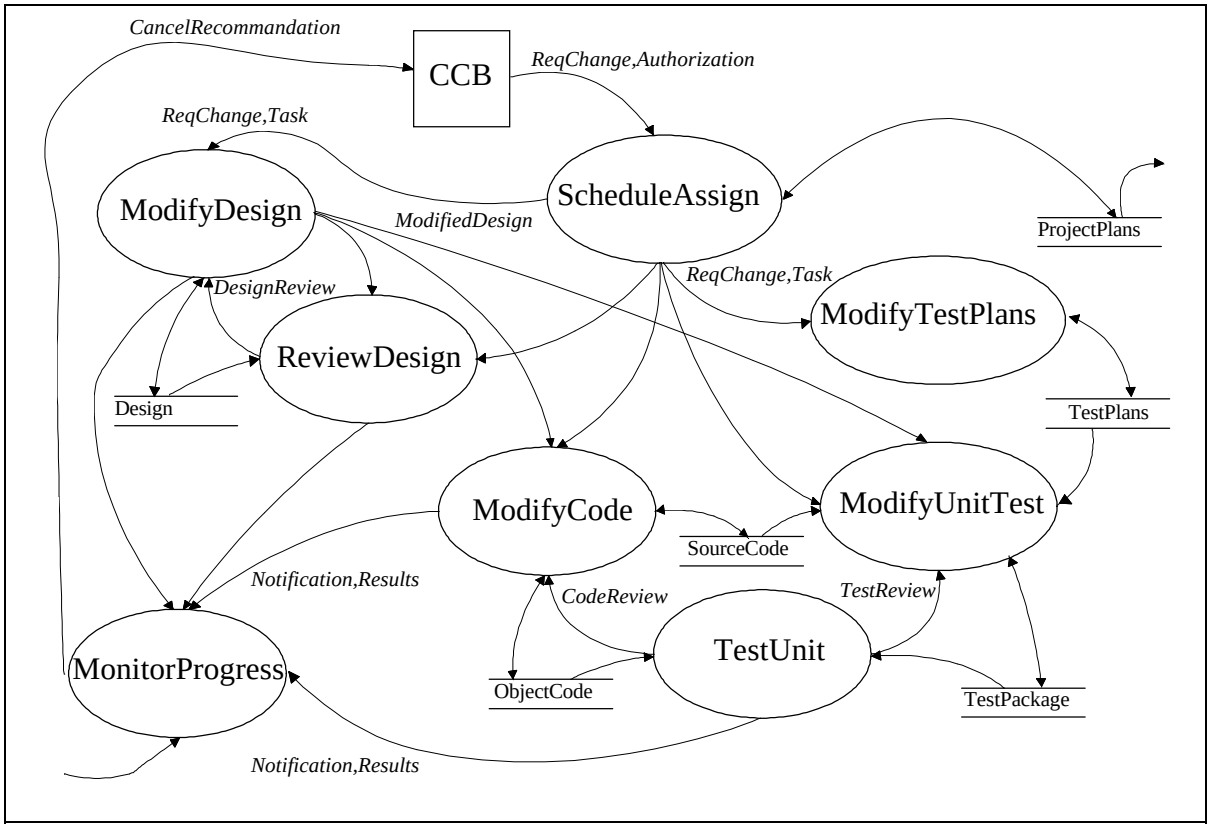

Figure 2. Dataflow model for the ISPW-6 case

One can see quite clearly from the diagram there is a remarkable anomaly in the ISPW-6 process description when compared to the usual *create-review* control cycles of the waterfall-like models. In this process description the modified design is handed down directly both to the *ReviewDesign* step and the *ModifyCode* and *ModifyUnitTest* steps. This effectively means that coding and testing activities could be based on incorrect design.

[6] The reader may have noticed, that the placement of the task or process symbols ('bubbles') in the dataflow diagram is suggestive: the four central tasks *ModifyDesign*, *ReviewDesign*, *ModifyCode* and *TestUnit* should be reminiscent of the well known *Waterfall Model* for the software engineering process [Royce 70]. This is one of the attractive features of diagrammatic techniques like the dataflow model: it gives a kind of high-level intuitive grasp on the complexity of processes, sometimes using rather loose associations. Note however the danger in using these associations, as discussed in [Petre 95]

Of course in any realistic software engineering process there is a distinct possibility for this to occur. Here however a simple measure could be taken to prevent this by handing the modified design to the *ModifyCode* and *ModifyUnitTest* steps only after it is approved of in the *ReviewDesign* step.
In fact this correction was made in the ISPW-7 extensions to the process example. Figure 3 shows the correspondingly adapted dataflow diagram.

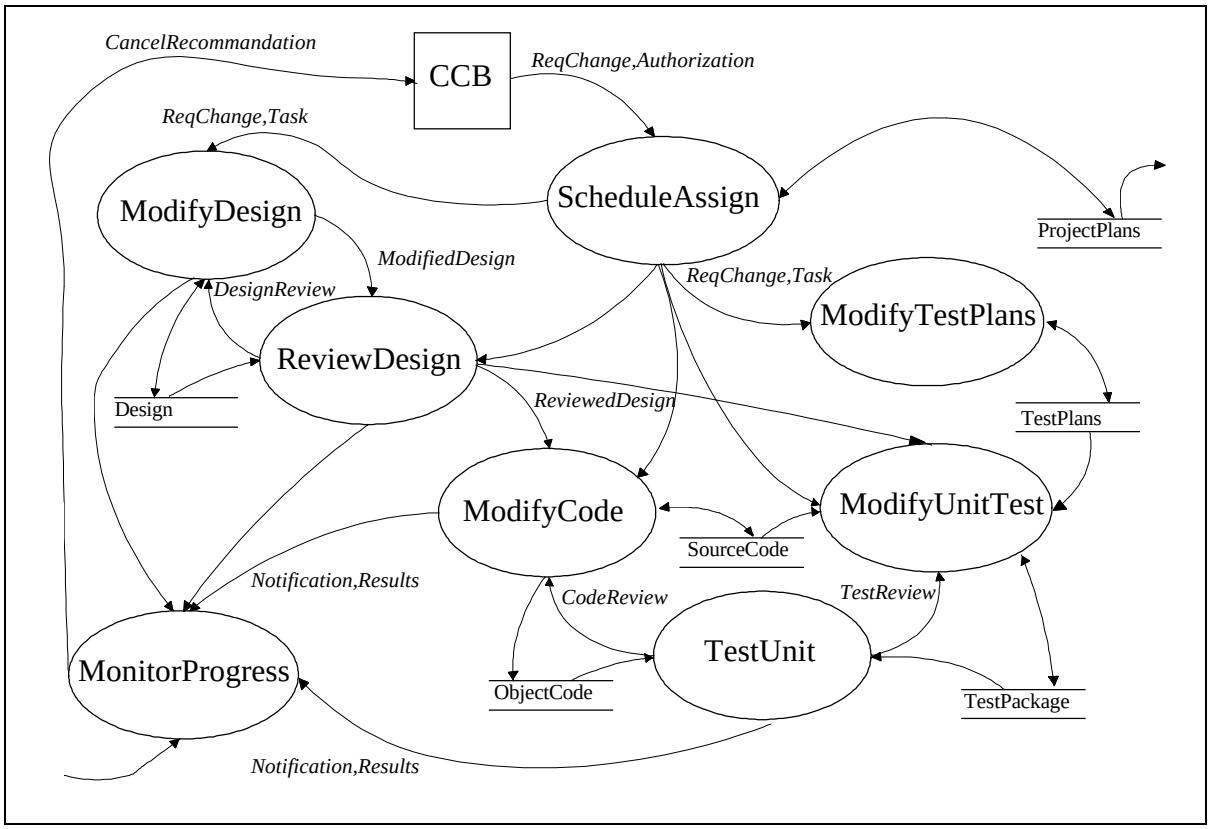

Figure 3. Dataflow model for the ISPW-7 correction.

Now the reviewed design is handed over to the *ModifyCode* and *ModifyUnitTest* steps. There is no danger of creating source code from unapproved design.

To show the abstraction and refinement capabilities of DFDs I have made two abstractions from the model in Figure 3.

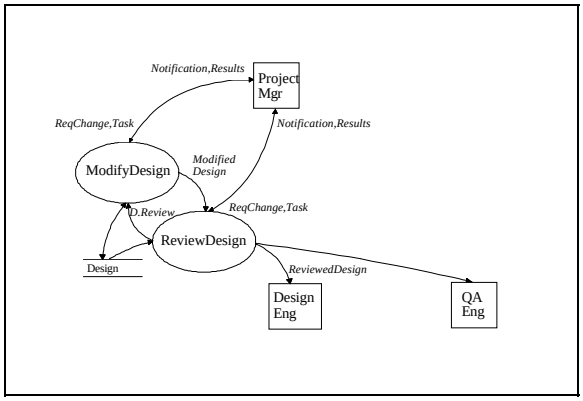

Figure 4. Abstraction for DesignEngineer

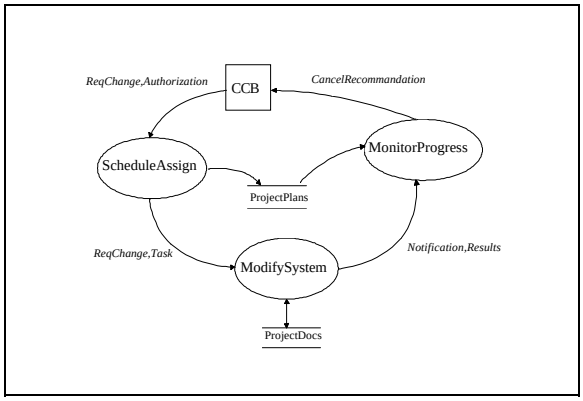

Figure 5. Abstraction for ProjectManager

Figure 4 shows an abstraction of the process model, if one is interested only in the activities of the DesignEngineers working on the Design for the system. The tasks of the QA Engineers (responsible for the testing activities), the DesignEngineers working on the system Code and the ProjectManager's tasks have been left out. In fact, these agents have been abstracted out as 'black boxes' and modelled as actors in the DFD. Now certainly, Figure 3 might be considered a refinement of this model. However, modelling the agents as actors suggests, that we are not 'interested' in their behaviour here.
In contrast, in Figure 5, we see some of the processes abstracted away to concentrate on the tasks of the ProjectManager. Now here elements from the model of Figure 3 can actually be considered a refinement for this model; we have split the task ModifySystem in the various subtasks to be perfomed in modifying the system.
Of course in any real design or analysis process refinement and abstraction need to be defined more precisely than we have done here. For rules of how to refine to various levels of abstraction for dataflow diagrams one can consult e.g. [Yourdon 94].

3.3 Data Flow Modelling

The examples of the previous section show how DFDs are an attractive graphical notation for capturing, in a fairly immediate and intuitive way, the flow of data and the operations involved in any information system, and therefore are an easy to use analysis and modelling tool.

However, as Ghezzi *et al*. state [Ghezzi 91], DFDs lack a precise semantics. Sometimes their syntax, i.e. the rules for composing bubbles, arrows and boxes, and their refinement and abstraction mechanisms, is defined precisely, but their semantics is not. This gives rise to inconsistencies in the applied modelling. Note the difference in the 'communication' between the *ModifyDesign* and *ReviewDesign* steps in Figure 3 on the one hand and the 'communication' between the *ModifyCode* and *TestUnit* steps on the other hand. In the first case the ModifiedDesign is handed over to the *ReviewDesign* step by a dataflow *arrow*; in the second case the modified ObjectCode is handed over via the *datastore* ObjectCode. Both solutions seem plausible as there is no precise semantics clearly discerning one from the other.

In several ways difficulties in the use of DFDs have been tried to overcome. In [Ghezzi 91] these attempts are classified as follows:

- *Usage of a complementary notation to describe those aspects of the system that are not captured adequately by DFDs*
  In fact this is the approach of OMT and similar design and analysis techniques, and consequently the approach of SOCCA. However when introducing DFD-like modelling in SOCCA we have to make clear what problems can be handled by which modelling subformalism, and what are the relations between the different formalisms.
- *Revising the traditional definition of a DFD, to make it fully formal.*
  To define the exact semantics of these DFDs we have to formalize the syntactical rules for the composition of DFDs, adhere semantics to the basic elements and to the syntactical composition rules.
- *Augmenting the DFD model in order to cope with aspects that are not captured by its traditional version*
  This is the approach taken in the development of a language like ViAL. Here traditional concepts from DFDs were enriched by specific symbols for the update of data stores, and control or signal constructs as well as error handling techniques. We will examine ViAL in the next chapter.

Data flow diagrams as used in OMT were augmented with additional constructs, too. In [Rumbaugh 91] two additional flow elements were defined in the diagrams to create the link with the control and structure submodels: the *object creation* arrow, and the *control flow* arrow. The following diagrams show how both elements may be used to model part of the process described by the ISPW-6 and ISPW-7 cases[7].

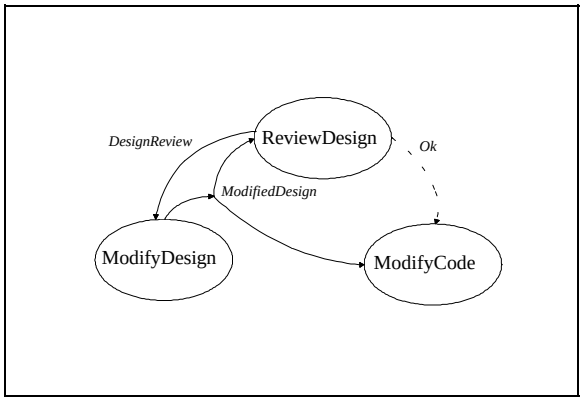

Figure 6. Control flow

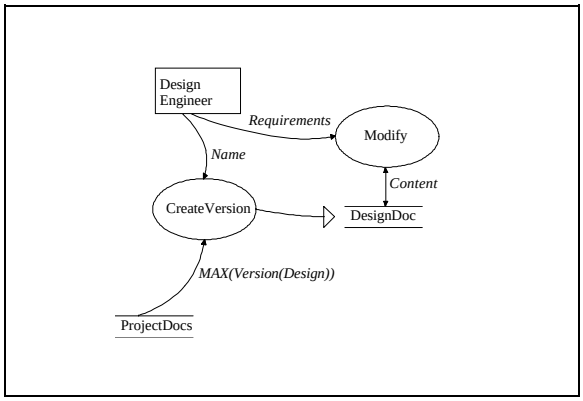

Figure 7. Object flow designating creation

Figure 6 shows how to incorporate *control flow* into the diagram. The modified design created in the ModifyDesign process is handed over to both the Review and the ModifyCode process. However, the ModifyCode process only is 'activated' if the Review process designates the Design as successful; in other cases a DesignReview indicating flaws in the design is handed back to the ModifyDesign process.

[7] Both examples actually do not exactly comply to the behaviour description of the SOCCA model in [Engels 94].

In Figure 7 the CreateVersion process creates a new version of a DesignDocument in response to a request from the DesignEngineer. This new document is then 'handed over' to the Modify process, indicated by the special arrow 'creating' the data store DesignDocument.

The intuitive appeal of DFDs make them particularly suited as an analysis tool. The 'bubbles' of the diagram can be refined to finer and finer levels, until some satisfactory analysis level has been reached. When used in multi-paradigm analysis and design methods, at some level the relation to other models has to be formulated.
In OMT ([Rumbaugh 91], section 6.6) this is done in an informal way by declaring the processes at a certain 'atomic' level to be 'equivalent' to (a subset of) the operations defined in the object submodel (and used in the behaviour submodel).
It remains unclear however, how exactly to interrelate the concepts of the dataflow diagrams used, i.e. processes, data stores, data flow arrows and actors with the concepts object, relationship or link, and operation from the class model, and the concepts introduced in the behaviour model.

One way to define the interrelation, is to relate the three different models, *class model*, *behaviour model*, and *data flow model* to one single underlying formal specification formalism or enactment language. This would be possible, if all the concepts were clearly understood in an informal way. As I belief this is not yet the case, I will start paving the way in chapter 7.

4 Visual Action Language and the EER model

ViAL, or *Visual Action Language*, was developed in connection with a research project called CADDY (Computer-Aided Design of non-traditional Databases) at the University of Braunschweig, Germany [Engels 92]. Here ViAL was implemented as part of a prototype design environment for information systems. A complete description of the language may be found in [Hennemann 91][8] ; for a comprehensive introduction see [Engels 95].

Why was it developed ?
In function-oriented[9] design and analysis paradigms it was customary to start the design and analysis of information systems with a high-level *data model*, and a high level *functional model*. It was soon realized that for a correct analysis sophisticated high-level data models were needed. Thus we saw the development of new methods and languages to account for complex structural descriptions. Many approaches were based on the ER model [Chen 76]. Extensions to this model were made, in many cases adding inheritance structures to the ER model. These models are generally known as *enhanced* or *extended entity-relationship* models (EER models).
For the *dynamical* or *functional* part of the design *data flow diagrams* were customary[10].

The objectives of the CADDY project included the construction of a computer-aided design environment based on an extended entity-relationship model. An integration gap was felt between the usual formalisms for the specification of functionality and the constructed EER models. Functional specifications should obey inherent integrity constraints imposed by the structure specification. For such inherently correct specifications a specification language had to be developed.
So a Visual Action Language was conceived.

Constructed models in ViAL should have [Engels 95]:

- a specification of actions *highly integrated* with the database schemata. The language had to have as its fundamental building blocks so-called *elementary actions*, the 'minimal' functions that mapped one consistent database state into another.
- an *intuitively comprehensible representation*. For this a visual, diagrammatic representation was chosen, adopting concepts of data flow diagrams.
- *inclusion of arbitrary data* queries with respect to the database.

This made ViAL strongly tied to the EER model used in the CADDY project[11].
Before looking at the language itself we therefore will have a quick look at this EER model, and compare it to the class or object model used in SOCCA.

---

[8] where the language is abbreviated VAL

[9] or data-oriented design and analysis. Here I mean more 'classical' methods like SA/SD in contrast to object-oriented methods, or methods with a heavy interest in behaviour.

[10] The past tense is a little out of place here. In fact this is still today the main paradigm in information system development. Again the contrast with oo-methods is to be stressed.

[11] A complete formal syntax and semantics of this EER model can be found in [Gogolla 91].

4.1 The EER model

The 'semantic' data model used is an extension to the well known ER-model [Chen 76], viewing the universe of discourse as consisting of *entities* and *relationships* among them. The entities and relationships are classified into entity, c.q. relationship *types* or classes, having certain properties as defined by the model or *schema*. Information about entities or relationships is expressed by a set of attribute-value pairs. So an *attribute* can be formally defined as a function which maps from an entity set or a relationship set to a value set. In fact, in the EER model *multi-valued* attributes are allowed, i.e. attributes delivering a *set*, *list* (ordered set) or *bag* (sometimes called a multiset) of values.
Entities partaking in a relationship may be restricted by *multiplicity constraints*; they specify how often an entity can be participant in a specific relationship type. The different roles of the entities participating may be indicated by adding *role names* to the relationship.
To this classical concept of the ER model some new constructs are added. The result is a data model with increased expressiveness. This model can be expressed in a diagram according to ER-conventions, with some extensions.
Basically the extensions to the data model encompass two concepts.

Specialization and generalization
To provide modelling primitives for specialization and generalization the concept of **type construction** is introduced. The general form of type construction is given by the diagram in Figure 8, where i_1 , . . . , i_n are already defined or *basic* entity types, called *input* types.

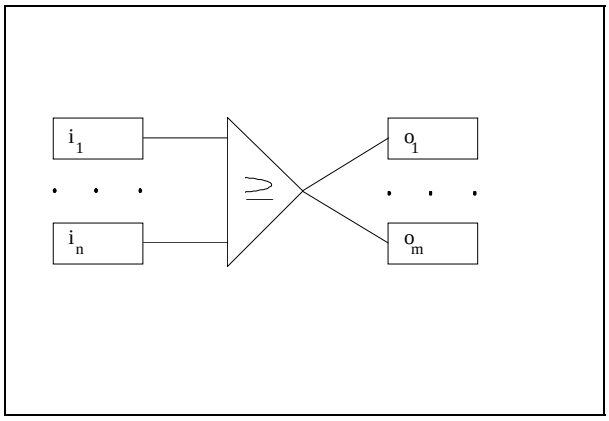

Figure 8. Type construction

Starting with these types, the *output* types o_1 , . . . , o_m are constructed.
All the entities from the input types are put together and distributed over the output types. The entities from the output types are not new entities, they already exist (in the input types) but will be seen in a new context, the output types. *Every* input type entity is assumed to be in *at most one* of the output types. So the following expressions will hold:

$$\cup_{k=1,..,n} i_k \supseteq \cup_{l=1,..,m} o_l \quad \text{and} \quad (\forall\, o_k , o_l , k \neq l)\ o_k \cap o_l = \varnothing$$

Indicated by the inclusion symbol $\supseteq$ in the triangle, the inverted direction need not hold, but can be explicitly required by an additional constraint.
Using this concept of type construction the well known concepts of *generalization* or superclasses (n > 1, m = 1), *specialization* or subclasses (n = 1, m = 1) and *partition* (n = 1, m > 1) can be defined. Complete hierarchies of type constructions are allowed. However, they must not contain cycles.

Components

The other added concept in the EER model is that of a **component**[12]. Certain attributes can hold entities or objects for value. Every basic or constructed entity type can have components. Components arise in diagrams like Figure 9, showing the entity *Personal Computer*, having the attribute *Manufacturer* and the component *CPU*, itself an entity of type *Processor*.

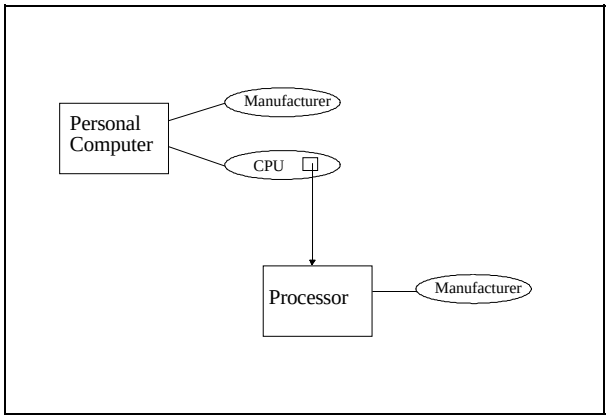

Figure 9. Components

One can compare the modelling power of the EER schemata with that of the OMT class diagrams used in SOCCA (of course *operations* or *methods* will not be presented in the former). Figure 10 gives the class diagram for the SOCCA entities found in [Engels 94]. We will translate this diagram partly into a corresponding EER-diagram.

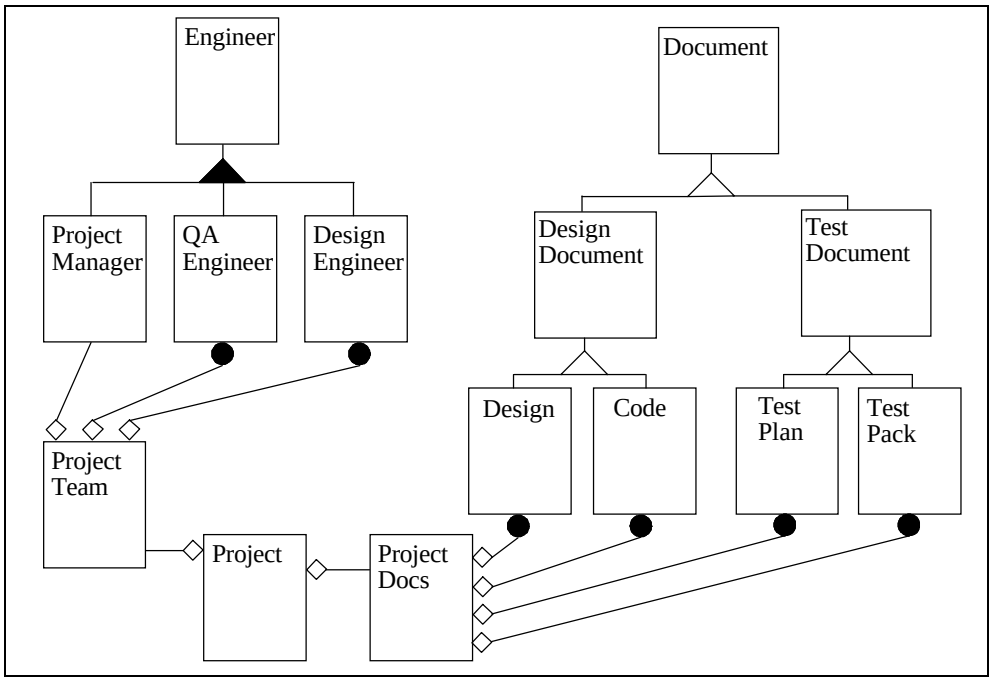

Figure 10. Class diagram showing SOCCA objects and links

[12] Although the concept of a component is an extension to the ER-model, Chen's original model contained *existence dependency* of entities, which is comparable to the concept of components.

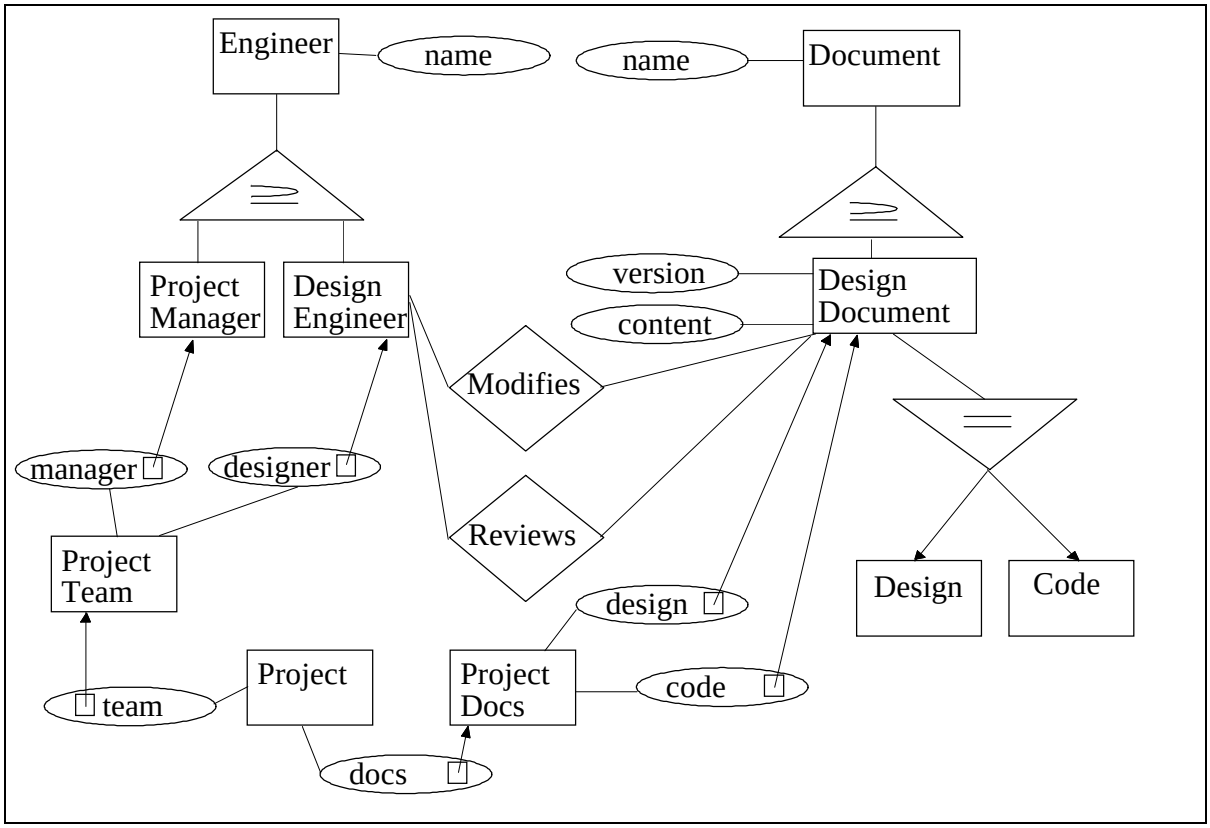

Figure 11. EER diagram showing SOCCA entities and relationships

As should be obvious to the reader both diagrams do not model exactly the same objects and relationships. The *Modifies* and *Reviews* relation are added in the EER diagram of figure 4; this diagram does not contain the *Test Documents* nor the *QA Engineer*. Multiplicity constraints are given in the class diagram of figure 3, but lack in the EER diagram of figure 4. They could be added however: the syntax of the EER model does provide them (although the notation is not defined in any of the referenced works).

Some other more interesting remarks can be made:

- there is a difference in modelling power of the type construction from the EER model, and the corresponding superclass/subclass constructions in the adopted OMT modelling.
  The potential for *overlapping membership* of the ProjectManager and DesignEngineer classes cannot be modelled using type construction from EER.
  To see why, consider the two modelling potentials of the EER type construct: specialization or generalization.
  (i) If ProjectManager and DesignEngineer would be *specializations* of the Engineer entity type, both would belong to different output types of the associated type construction, and consequently have no members in common.
  (ii) If Engineer would be considered a *generalization* of Projectmanager and DesignEngineer, both Projectmanager and DesignEngineer would have to be input or basic types. However, the EER model will not allow basic entities to be of the same type.

More in general, there is a distinct difference between the interpretation of type membership of entities in the EER model, and class membership in OMT's object model, as was noted in [Ebert 94]. In a *type-theoretic* approach, which the EER model is advocating, each object has exactly one class to which it belongs. In the *set-semantic* approach of object-oriented models like OMT, an object is considered a member of all its superclasses.

- although there might be a *formal* equivalence between the *component* construct of EER diagrams and the *aggregation* construct in OMT's class diagrams, there seems to be a *conceptual* difference.
  The reader might have noticed the difference in the construction of the aggregate class ProjectDocs in Figure 10 and Figure 11. In the class diagram, the components of ProjectDocs are from the classes Design and Code, both subclasses of DesignDocument. In the EER diagram, both components are considered to be of type DesignDocument, but *will be partitioned into* the specializations Design and Code.
  In my feeling, the strong suggestion of their role in the component construction of ProjectDocs (by the attribute names) makes this partition look somewhat superfluous, while in the class diagram the subclass partitioning of DesignDocument is somehow the indicator for their role.

The last remark suggests a more economical approach in using inheritance hierarchies in the models. When specialization is used just to clarify the role of a class in some relationship, this might be considered bad design practice[13]. Indeed the component construct in the EER model shows how this can be done: by assigning role names to the associations in the class model.

When compared to the OMT modelling capacity, this is even more true for the SOCCA models. The 'role switching behaviour' of an object or entity can be modelled quite sophisticated using different internal behaviours for an object in different roles. As an example one might look at the behaviour of DesignEngineer in the dynamic model in [Engels 94]. The two different internal behaviours *int-design* and *int-review* somehow represent the different relations the DesignEngineer has to the DesignDocuments: he/she is *Modify*-ing documents or *Review*-ing documents (or both at the same time: a distinct possibility in the SOCCA models). The external behaviour for DesignEngineer is the manager for this 'role switching behaviour'.

[13] although the construction of analysis or design models is in all cases somewhat arbitrary

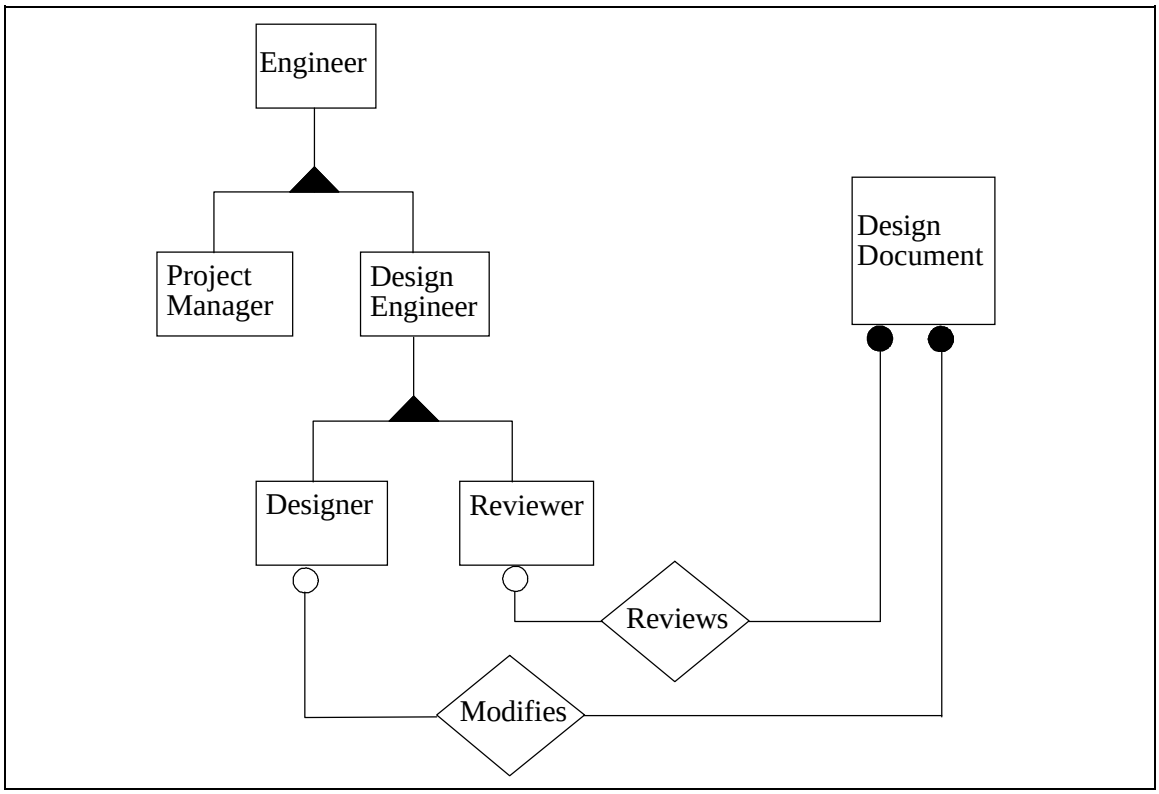

Figure 12. An alternative class diagram

An alternative way to shape this behaviour in a SOCCA model is suggested by the class diagram of Figure 12. Here we have refined the different roles of a DesignEngineer a structural perspective. Adhering to this model, the *external* behaviour of Designer and Reviewer in the corresponding dynamical model would be somewhat like the 'old' *internal* behaviours *int-design* and *int-review* of DesignEngineer.

4.2 Elementary actions

The EER schema sets structural constraints for the information system adhering to this model. It tells us about the structure of the model (in terms of entities, relationships and attributes). By adding dynamics to our model we specify a way to get from one state or instance of the model to another. One way to do this is by the use of so called *basic* actions. The modifications made to the model instance by the use of basic actions are (cf. [Engels 90])

- insertion or deletion of an instance of an entity or relationship type together with its attribute values
- addition or removal of a component of an instance of an entity type
- the update of attribute values of existing database objects: entities or relationships
- insertion or deletion of the membership of a database object in a type construction

These basic actions form a complete set of operations: using these basic actions *any* state change of the EER model instances could in principle be executed[14].
The basic actions describe the modification of exactly one object (entity, relationship). After the execution of such an action the new model instance may not be a correct one. This means that this local modification may violate the global structure as prescribed by the EER schema. In this case additional basic actions, sometimes called *update propagations* may be needed to yield a new and correct model state[15].
Wolff [Wolff 89] has shown, that minimal sequences of basic actions starting and resulting in a correct model instance can be defined, and in fact be automatically constructed from any given EER model. These minimal sequences of basic actions are called *elementary actions*.

The elementary actions take account of rules like:

Insertion of objects
- after the insertion of an entity any required (i.e. not optional) components must be inserted
- the insertion of an entity being the input type of a partition, must result in the insertion of an output entity of one of the partition output types

Deletion of objects
- before deleting an entity, any relation it participates in must be deleted first
- before deleting an entity, any entities belonging to output types corresponding with the entity must be deleted first
- all components belonging to the entity must be deleted first

Update of objects, i.e. update of attribute values
- no elementary actions are constructed[16]; the basic *update* action will suffice

---

[14] i.e. it is my firm belief that this could be proven in a formal way

[15] The notion of elementary actions is not unique for the EER model. In [Chen 76] some rules for 'consistent' insert, update and delete operations were already formulated for the ER-model.

[16] As stated in [Wolff 89]. It shows however, that in spite of the careful construction of elementary actions from the EER schema, a functional model completely adhering to the integrity constraints from the model still was not constructed. The update of *key attributes* must not be allowed, without having checked first the uniqueness of the key value over the database.

4.3 The language ViAL

Having taken a glimpse of the EER model and the concept of elementary actions we are now ready to look at the language ViAL.
ViAL is a visual programming language to construct complex transactions on databases that can be considered instances of EER models.
The language offers the following constructs, as building blocks from which one can compose functionality in the corresponding diagrams:

- *queries*, functions on instances of the EER models that constitute values, value sets, or even objects (entities, relationships) or sets of objects;
- *actions*, i.e. basic actions for the insertion and deletion of objects, the update of attribute values, and the addition and removal of components;
- a declaration and invocation construct, which enables *abstraction* of compositions into procedures;
- *data flow edges*, that can connect the aforementioned constructs.

Some of the transactions to be defined in the language are 'pre-modelled' and can be used by a designer using the language as primitive building blocks for more complex operations. These 'pre-modelled' transactions correspond to the elementary actions defined in section 4.2.

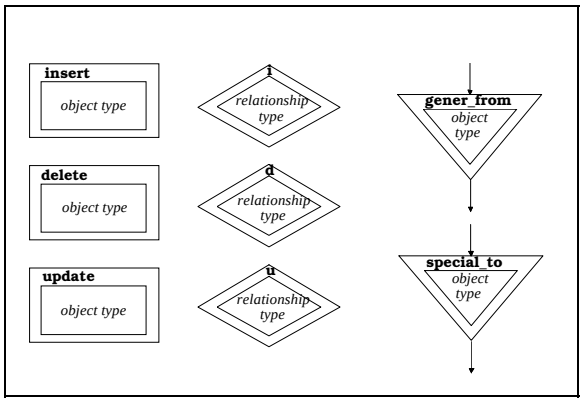

Figure 13. ViAL basic actions

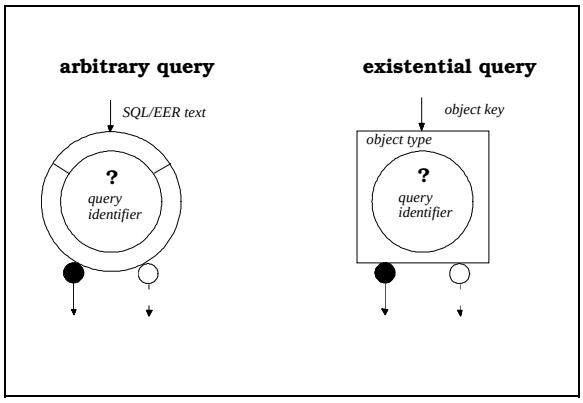

Figure 14. ViAL representation of queries

Figure 13 shows the symbols that are used in ViAL for the basic actions (insertion, deletion, and update of entities and relations). All these basic actions can be simply and automatically derived from a given EER model. These basic actions can be 'enriched' with some constructs, e.g. setting (required) attribute values when inserting an entity.
To select values from the database, or to test for database states, ViAL offers the query constructs of Figure 14[17]
An *existential query* tests the database for the existence of entities of a specific type, using a value for the key attribute from the type. As a result of this query, either the object(s) themselves or a signal value or error are produced.

[17] Actually the different ways queries can be built are a little more complicated. Here two main 'types' are shown.

So the SQL/EER[18] phrase roughly equivalent to an existential query is

select e
from e **in** ENTITY
where e.KeyAttribute = *keyvalue*
if e **exists then** e **else** error

In addition to these existential queries, the user of ViAL is free to formulate arbitrary queries of the database. The result of this query can be an error or signal value, or a data or object flow[19]. The visual symbol of Figure 14 is used to incorporate the query in the definition of the transaction; the query itself can separately be defined using either the SQL/EER language or a hybrid (partly graphical, partly textual) formalism.

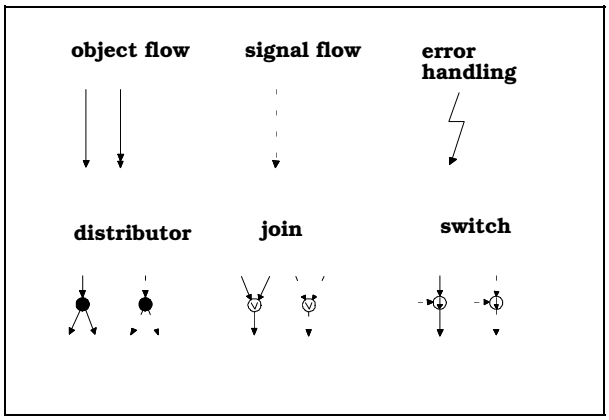

Figure 15. ViAL connector (flow) symbols

Queries and actions are interconnected using the flow symbols of Figure 15. Remember the result of a query can be a (set of) value(s) or a (set of) objects, or a signal or error.

Via the *object flow edges* objects or sets of objects flow through the specified processes. Multi-value oriented processing is supported: doubly arrowed flow edges represent object/value sets. To connect these multi-valued or multi-object flows to the basic actions, the basic actions can be starred (i.e. marked with an asterisk) to denote the iterative application of the actions, like in the symbolic code statement

foreach e **in** s **do update** e.someattribute **with** e.someattribute+1

Control constructs can be made with the error and signal edges. *Signal edges* handle boolean values. *Edges for error handling* could connect an action to a standard error handling procedure in an application.

---

[18] SQL/EER is a SQL-like query language defined over the EER calculus of [Gogolla 91]. Consult [Hohenstein 92] for a formal definition of the language.

[19] more or less equivalent to the formal concept of a *range* in [Gogolla 91]

The *switch* represents an operator to constrain data flow. The ongoing information (data or signal flow) is delivered only if the signal carries a 'positive' value (true) , indicated by the symbolic statement:

if s **then** d **else** ⊥

where ⊥ is a special *null* value[20], and d stands for the ongoing data or object flow.

The *distributor* takes an ingoing signal and puts it through to the outgoing edges. Via the *join* operator the different 'branches' of information flow can be recollected. Only one of the incoming edges is allowed to carry value. The *distributor-join* constructs could therefore best be compared to guarded **cobegin** .. **coend** constructions, where only one of the parallel branches is thought to 'fire' for any database instance.

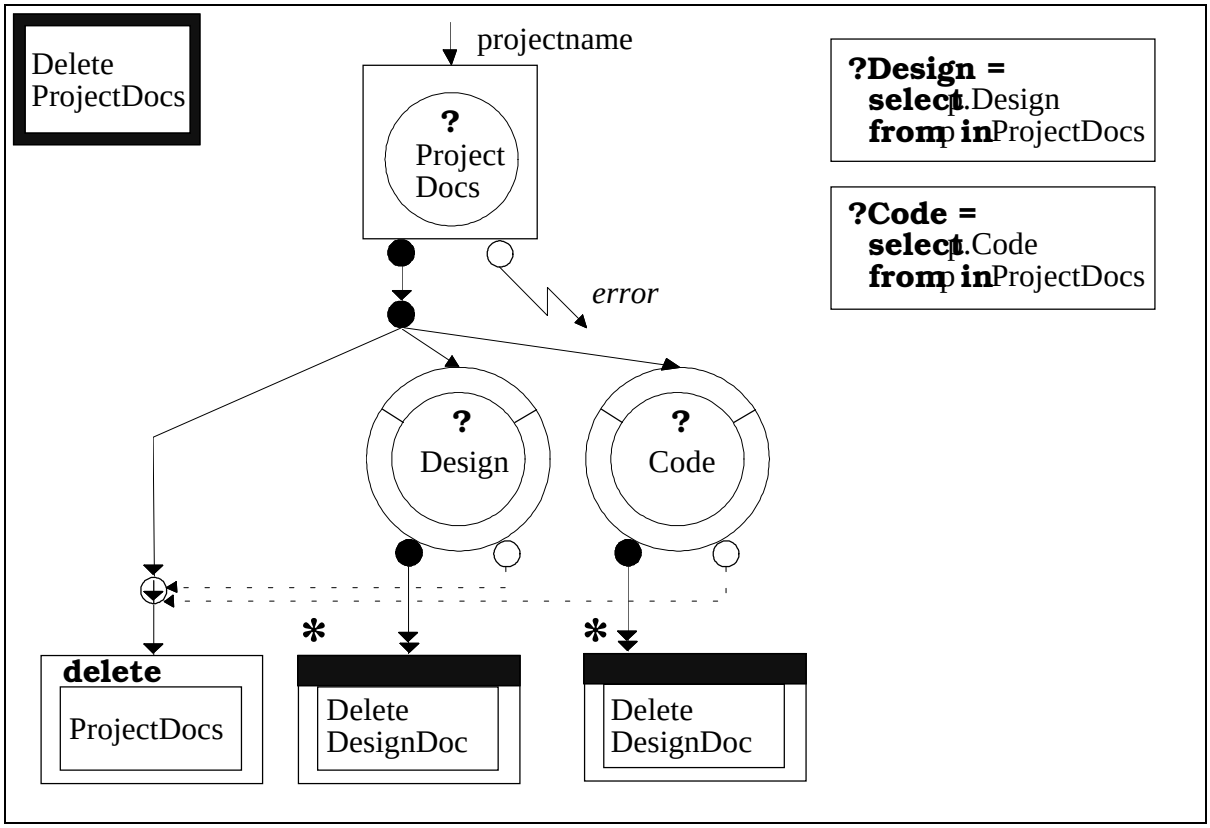

Figure 16. Delete ProjectDocs elementary action

This section is concluded with an example giving an idea of the ViAL language constructs. It is based on the EER model of Figure 11, although somewhat simplified.

[20] cf. [Gogolla 91], p. 376

Example.
The elementary action modelling the deletion of a ProjectDocs document file for a Project.

To delete a ProjectDocs document file, we first need one. So with the existential query *ProjectDocs* we select an entity of this type by its key attribute, which is supposed to be called *ProjectName*[21].
If the entity is not found, it cannot be deleted, and an error signal is forced.
In the other case, when the entity is found, we cannot simply delete it, because it possibly has *Design* and *Code* components.
So first we must delete the accessory DesignDocuments, which in turn can have either Code or Design specializations or a Document generalization. However, we will assume that an elementary delete action for DesignDocuments has already been defined, and simply call it in our ViAL definition. The resulting complex action *Delete ProjectDocs* is shown in Figure 16.
Observe the use of the multi-object flow and the asterisk for the deletion of the components.
The example both shows definition/declaration (Delete ProjectDocs) and invocation (Delete DesignDocs) of a complex transaction.

4.4 An evaluation of ViAL and its constructs

The central question directing my work on SOCCA was:

Can we use ViAL to model the functional perspective, i.e. the functionality of the software process, and add something extra to SOCCA's modelling power in addition to the class and behaviour model ?
My conclusion is: no, we can't. And there are good reasons.

First of all, ViAL constructs model both functionality and behaviour.
With its signal and error flows it is strongly control oriented. In SOCCA however, there was already a satisfactory and to some extent superior modelling technique for the control flow of the software process: the behaviour model with its PARADIGM constructs.
There are some shortcomings in ViAL's handling of control constructs:

- ViAL's constructs (query and action) are conceived from a 'global view' on the database adhering to the data model. So there is no 'local' mechanism to influence the control. structure. In fact, the 'objects' in SOCCA are active components, and their control structure or behaviour depends on local as well as global state.
- There seems to be no clear semantics for the diagrams[22], especially when concurrency of behaviour is concerned. The example from the previous section shows this to some extent. It is clear from the diagram, that the *delete* basic action for the ProjectDocs object cannot precede the ?Design and ?Code queries. However, there is no syntactical reason why this action must not precede the elementary actions Delete DesignDocument.
  Furthermore, the result of the ?Code query might be influenced by the 'firing' of the Delete DesignDocument action 'triggered' by the ?Design query. And lastly, there is no semantical framework for the concurrent behaviour of the two complex transactions Delete DesignDocument in the example.

[21] not modelled in Figure 11, however

[22] I have not consulted [Gerlach 92], where a ViAL interpreter over a database adhering to the EER model was described

Second, ViAL is a 'constructive' language. It is designed to build complex transactions from simpler ones. As we have seen in chapter 3, DFDs were naturally top-down oriented and therefore more suited for analysis and design.
Moreover, the 'constructive' task, i.e. the packaging of behaviour into tasks or operations is already performed by the specification of external and internal behaviours in the behaviour model.

And third, the modelling primitives of ViAL for actions on the EER schema instances are the basic actions for the EER model, which are to some extent 'unnatural' from an object oriented point of view. Indeed, the operations from the class model must be considered the 'basic actions' for the modelling of SOCCA functionality and behaviour.
In any case, the basic actions for the specialization, partitioning and generalization are superfluous in the set-semantic interpretation of the class model of SOCCA (cf. section 4.1).

So we cannot simply adapt ViAL to our needs. But there are some interesting new ideas we can learn from the quick look at ViAL:

- it should be possible to generate some 'blueprint for behaviour' for our objects, just by studying the possible instantiations of the class model, similar to the way elementary actions were conceived and implemented as basic building blocks in ViAL.

- it is possible to construct a dataflow-like diagrammatic language with a better formal relation to the underlying data model than the original formalism of dataflow diagrams.

- inclusion of indicators for 'erroneous' behaviour, like the *error flow* in ViAL, might help to pin-point expected weak spots in behaviour design.

5 Behaviour modelling and some additional concepts

In the preceding chapters we had a brief look at the SOCCA class model for (part of) the ISPW-6 case, and made only a short mention of the behaviour model. Nevertheless we saw some remarks about what concepts should be handled by the behaviour model, and what the relation between a class model and functional model on the one hand, and the behaviour model on the other hand should be. Let us take a closer look at behaviour modelling in SOCCA.

Based on the operations from the class model, in SOCCA one can build State Transition Diagrams (STDs) showing all possible sequences in which these so-called *export operations* might be 'executed upon' an object belonging to the corresponding class. This is roughly equivalent to the use of STDs for behaviour modelling in most object-oriented methods (see e.g. [Graham 94] for an overview). However, behaviour modelling doesn't stop there.
For the implementation of its export operations or methods, an object will undergo some state change, possibly alter some of its attributes and induce state changes in other objects, by calling upon their methods. This 'calling behaviour' is explicitly modelled as a sequential process in the so-called *internal behaviours* corresponding to each of the export operations of the class.
The corresponding behaviour model allows for *asynchronous communication* between the objects, i.e. the effect of the calling behaviour will generally not show itself immediately or 'during' the call, and for *concurrency of behaviour* within one object. The restrictions necessary to ensure the 'correctness' of behaviour are then modelled using PARADIGMs coordination structure of managers, subprocesses and traps (which is adequately described in [Engels 94]).

An interesting question is how one should arrive at the specification of the internal behaviour. From [Engels 94], p. 6 one may read:

from studying where the export operations are imported, the various internal behaviours of the operations of a class are modelled as STDs exhibiting all possible sequences of calling imported operations

One may wonder if there is a systematic way or *method* by which one can arrive at the (possibly partial) specification of the internal behaviour from the external behaviour of the objects involved, and the import/export diagram SOCCA provides. I believe there is. To see why, we first take a look at the import/export diagram, and the concept of a *behavioural view* as introduced by Ebert and Engels [Ebert 94].

5.1 Import/export diagrams

In [Engels 94] the class diagram modelling based on EER-like modelling techniques is extended by defining what is called the *uses* relationship, a new binary relationship type. This *uses* relationship is shown in diagrams like the one in Figure 17.

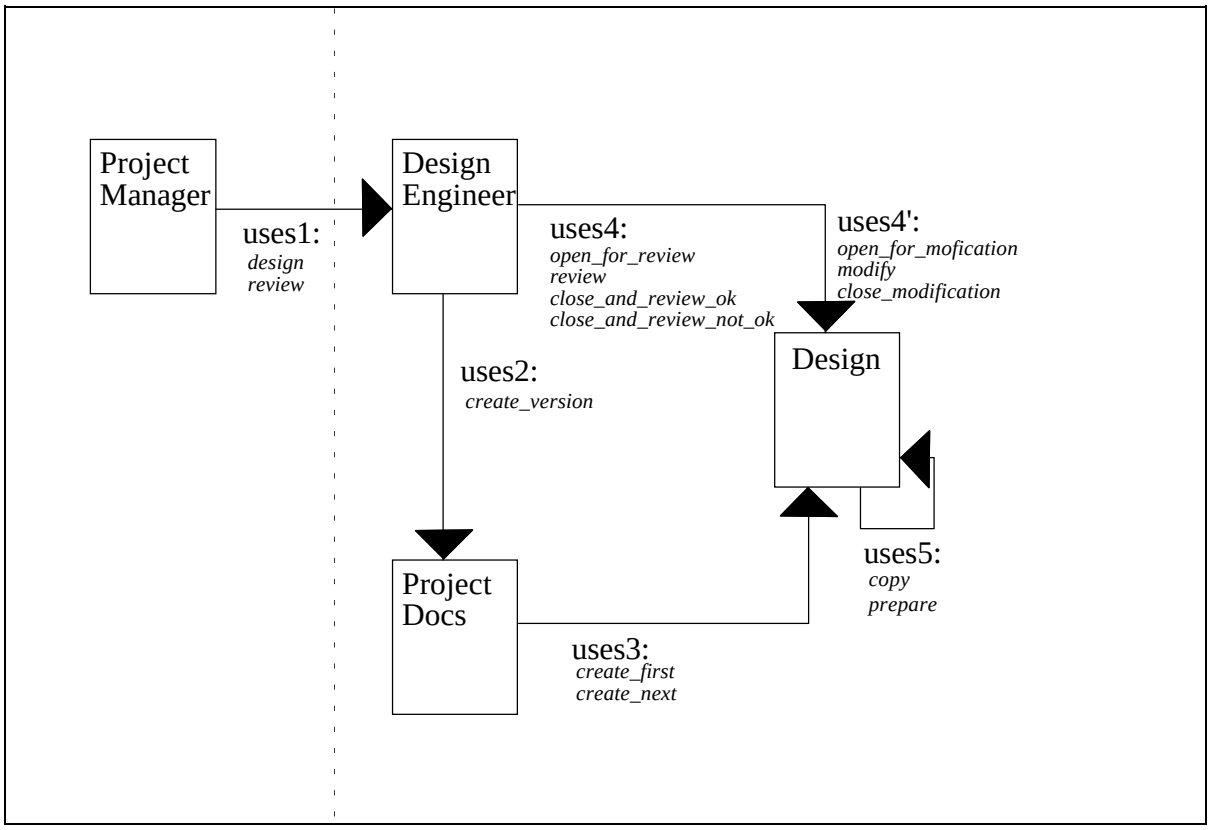

Figure 17. Import/export diagram

Each *uses* relationship has an 'attribute' *uses_list* for keeping the list of actually imported operations. As is remarked on p. 9 of [Engels 94] 'the list of operations may differ between instances of the same type depending on the role of this instance'.
I have some comments on this use of import/export diagrams:

- It is not clear why this *uses* relationship and consequently the import/export diagram is considered part of the class diagram. If the attribute *uses_list* is dependent on the role of the instances of the classes, this role must not be considered part of the class structure. In fact the diagram seems roughly comparable to the *event flow* diagram used in dynamic modelling in OMT, which is part of the behaviour model.
- The *role switching* behaviour of the DesignEngineer does suggest the possibility of defining the partition Reviewer / Designer as possible specializations of DesignEngineer, as was done in section 4.1.
- By making the import/export diagram part of the dynamic model, the use of the calling graph of Figure 18 might be considered. In fact it seems that when the import/export diagram of Figure 17 is constructed, one has all the necessary information to construct the calling graph of Figure 18, which of course contains more information.

The calling graph shows us exactly what operations from other objects are needed for the implementation of export operations. The *review* method of DesignEngineer uses 4 operations from Design for its implementation: *close_and_review_Ok*, *close_and_review_not_Ok*, *review*, and *open_for_review*. Note that the calling graph says nothing about the sequencing of these 'calls'.

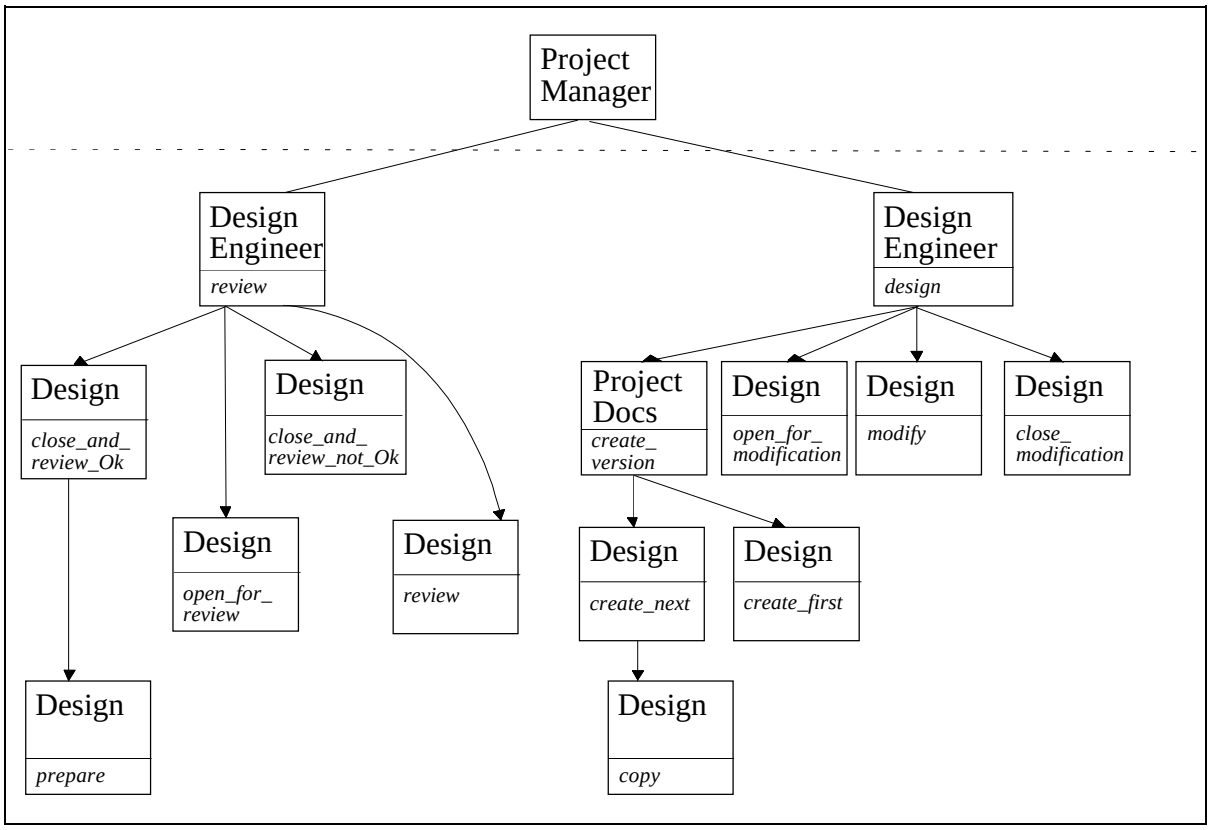

Figure 18. Calling graph as an alternative for the import/export diagram

5.2 Behavioural views and internal behaviour

The notion of a view is a well known concept from database literature. The classical ANSI/SPARC architecture was largely motivated by the intention to allow different users to have a different view on the same database.
Ebert and Engels [Ebert 94] have generalized the concept of a view to the object-oriented paradigm. They define a *view* on a class C as a simplified variant C', which can be derived from C in a methodical way.
The simplest view is one for which:

- the *attributes* of C' are a subset of the attributes of C (and retaining their proper *data sorts*);
- the *(export) operations* of C' are a subset of the operations of C ;
- the *state transition diagram* for C' is the projection of the STD of class C to the restricted set of operations of C', i.e. the STD of C, where all the operations which are not in the subset of operations of C' have been replaced by unlabelled transitions (ε-transitions)[23].

Now for each pair of classes where one class is using operations from another class for the implementation of its methods, a corresponding view can be constructed.

[23] this 'abstracted view' corresponds to the notion of restriction (cf. [Milner 89], ch. 2) or abstraction (cf. [Baeten 87], ch. 5) known from communication theory.

From the calling graph of Figure 18 one sees that the following views can be constructed:

- the *DesignEngineerToDesign* view for the implementation of the *review* method as the view DesignEngineer (as a reviewer) 'holds on' Design;
- the *DesignEngineerToProjectDocs* view for the implementation of the *design* method as the view DesignEngineer 'holds on' ProjectDocs;
- the *DesignEngineerToDesign* view for the implementation of the *design* method as the view DesignEngineer (as a designer) 'holds on' Design;
- the *ProjectDocsToDesign* view for the implementation of the *create_version* method;
- the *DesignToDesign* view for the implementation of the *close_and_review_Ok* method;
- the *DesignToDesign* view for the implementation of the *create_next* method.

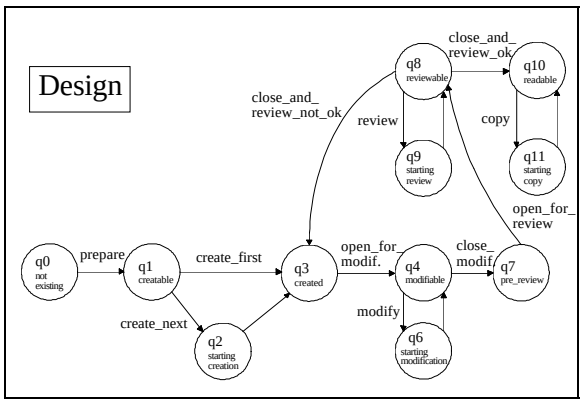

Figure 19. Design's STD

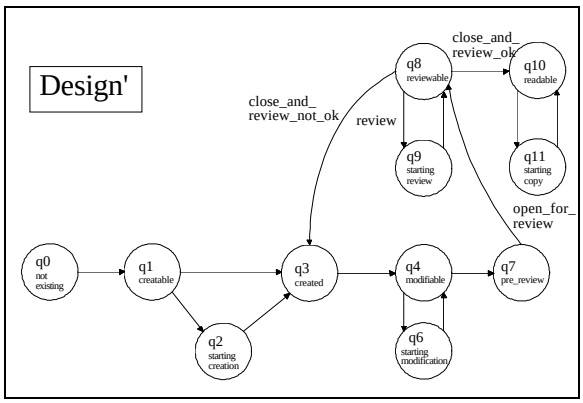

Figure 20. Reviewers view on Design

Figure 19 shows the state transition diagram with all Design's export operations. Figure 20 shows the corresponding view that DesignEngineer holds as a Reviewer. The subset of operations for the corresponding abstraction is the set {*open_for_review*, *review*, *close_and_review_not_Ok*, *close_and_review_Ok*}. All other export operations (and consequently the transitions in the diagram) in the view are substituted by unlabelled transitions.
Now 'from the point of view' of the Reviewer some of these unlabelled (or ε-transitions) and some of the states are redundant. The Reviewer has no influence over the transition (q3 → q4) nor over the transition (q4 → q7) because both are ε-transitions. So for the sake over the Reviewer the 'complete ε-path' (q3 → q4 → q7) or (q3 → q4 → q6 → q4 → q7) might be replaced by one ε-transition[24].

[24] We may not however completely 'identify' state q3 with state q7, because the operation *open_for_review* is not applicable in state q3. One could say that in state q3 the object Design is not 'vulnerable' to *open_for_review*. See [Milner 91], chapter 2 for a discussion on this subject.

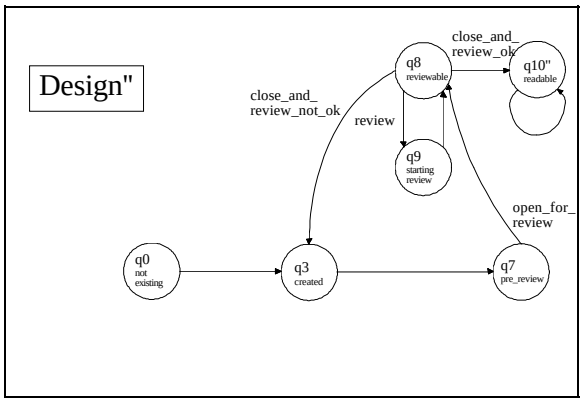

Figure 21. Reduced view on Design

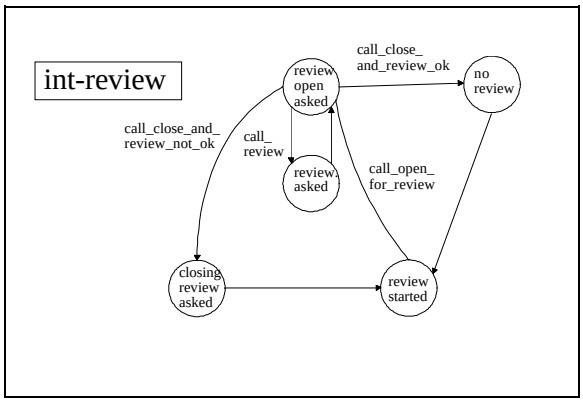

Figure 22. Another look at int-review

If we now remove all the redundant states and transitions and replace them with their 'equivalent' ε-transitions we arrive at the view of Figure 21[25]. What we now have is the 'natural image' of the internal behaviour of the *review* method. Each state and each transition corresponds to a state or transition in the *int-review* STD. To see this we compare the diagram of Figure 21 to that of the int-review STD of Figure 22. There is a natural correspondence between most states and transitions. Of course the *call*-transitions correspond to their counterparts in the Design"-view.
We can see the direct correspondences

state in Design"		state of int-review
reviewable	↔	*review open asked*
starting review	↔	*review. asked*
readable	↔	*no review*
pre_review	↔	*review started*[26]
closing review asked	↔	*created*

However, there is a difference between the *int-review* STD as given in [Engels 94] and the one in Figure 22. In [Engels 94] there is a transition marked *call_open_for_review* as a transition from the state *closing review asked* to the state *review open asked*. Here such a transition is only possible after the ε-transition leading from the state *closing review asked* to the state *review started*. For the sake of its communication behaviour however, the internal behaviour constructed is equivalent to the *int-review* behaviour of [Engels 94][27]. So the only thing that is not captured in this 'natural image' of the internal behaviour of *int-review* is its instantiation/deletion behaviour.

[25] One might argue about the ε-transition (q10 → q10). We could leave out this transition.

[26] called *review asked* in [Engels 94]. This state label appears twice however.

[27] Again, this equivalence can be given a formal basis using the notion of bisimulation from [Milner 89] or [Baeten 86].

Before proceeding, let us recapitulate the steps to 'construct' the internal behaviour:

(i) from the *calling graph*, deduce from the usage structure which operations are actually imported and used for the implementation of the internal behaviour.
(ii) construct the corresponding *abstraction* of the external behaviour of the server object, by replacing all unused internal operations by an ε-transition.
(iii) now remove *redundant* states and ε-transitions (cf. [Milner 89], p. 39)
(iv) add *default instantiation and deletion behaviour* to complete the framework for the internal behaviour

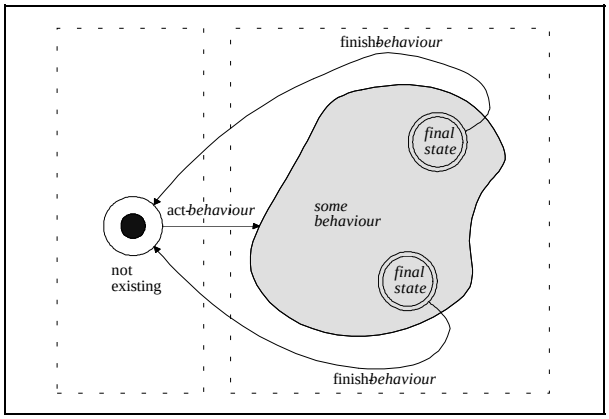

Figure 23. Internal behaviour initial framework

This default instantiation and deletion behaviour is shown in Figure 23. The marked state corresponds to the *not_existing* state of the internal behaviour to be constructed. From this state the act-*behaviour* transition will initiate some internal behaviour. In this internal behaviour then, some states will be marked *final*, from which a return to the *not_existing* state will be allowed, using a finish-*behaviour* transition. This instantiation and deletion behaviour has a 'natural subprocess and trap structure' with respect to the corresponding management process from the external behaviour of the object, symbolically shown by the dashed lines in the diagram. The corresponding subprocesses can be constructed by leaving out the act-*behaviour*, respectively the finish-*behaviour* transitions from the internal behaviour.

Using this 'construction method' we now reconstruct the internal behaviours for the other operations relying on the usage structure in the calling graph of Figure 18, and compare these constructed internal behaviours with the internal behaviours given in [Engels 94].

We start with the simpler ones: the int-*close_and_review_Ok* (int-*Ok* for short) and int-*create_next* internal behaviours.

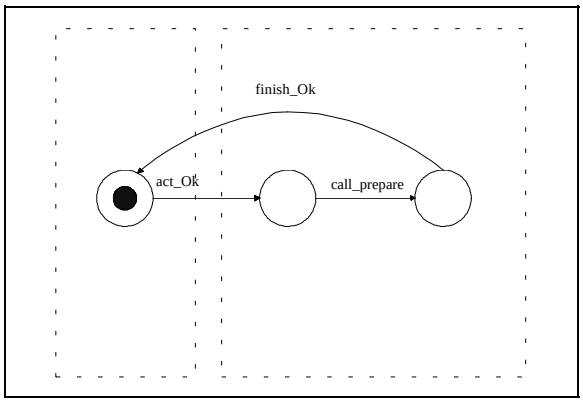

Figure 24. int-Ok internal behaviour

Figure 24 shows the 'constructed' internal behaviour for int-Ok. We can see that it is exactly the same behaviour as given in [Engels 94], p. 15. The dashed lines show the symbolic trap structure with respect to the management of this process by the Design external behaviour (same instance).

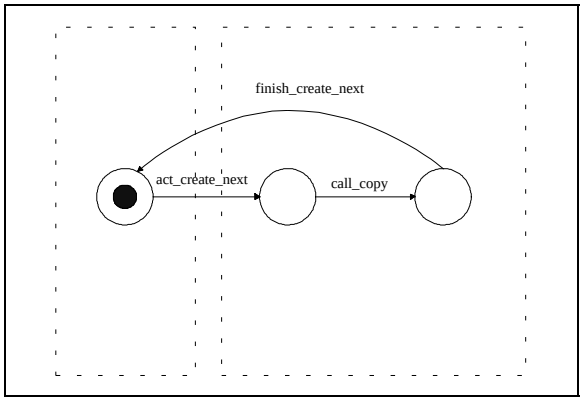

Figure 25. Framework for int-create_next

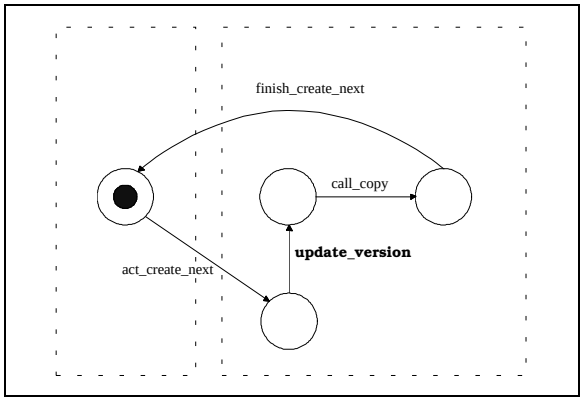

Figure 26. int-create_next internal behaviour

In Figure 25 we see the constructed internal behaviour for int-*create_next*. Figure 26 shows the corresponding internal behaviour from [Engels 94], p. 15. The difference of course is the *update_version_no* transition modelling some 'perfectly local' control, administration or state change in the internal behaviour. This 'extra' behaviour shows the 'double functionality' of the internal behaviours:

- They are the expression of the relation between the object and its 'acquaintances', the objects it is communicating with. one might even say that the internal behaviours int-*review* and int-*design* 'model' the corresponding *Review* and *Modify* relations from the EER diagram in Figure 11. This communication behaviour is captured by the internal behaviour framework, that can be constructed using the method of this section.
- The internal behaviours model some local control, administration or state change in connection with the desired functionality of the corresponding operation.

The next example shows the framework for and the actual behaviour of int-*create_version* (cf. [Engels 94] p. 15). First we construct the corresponding abstraction from the external behaviour of DesignDocument and the calling graph of Figure 18.

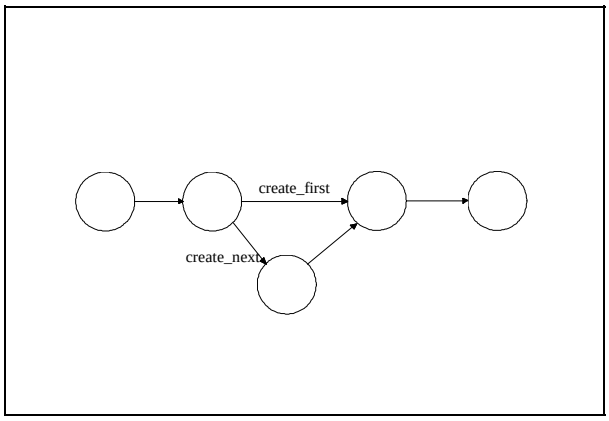

Figure 27. Abstraction from Design behaviour

We have abstracted out all redundant ε-transitions, and indeed one may wonder why the 'initial' and 'final' ε-transitions have not been abstracted out. As far as the 'client-server' communication between ProjectDocs and DesignDocument is concerned, they could be abstracted out. However, they should remind the reader of the fact that the *create_first* and *create_next* operations are only applicable in one definite state of the external behaviour of DesignDocument, and that any 'caller' on this behaviour should 'wait' (although asynchronically) before the actual state admitting the calls has been reached. However, for the framework construction for the int-*create_version* behaviour, we may indeed abstract them out, as is shown in Figure 28.

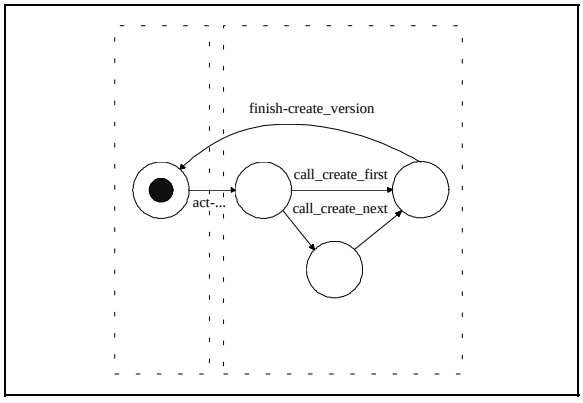

Figure 28. Framework for int-create_version

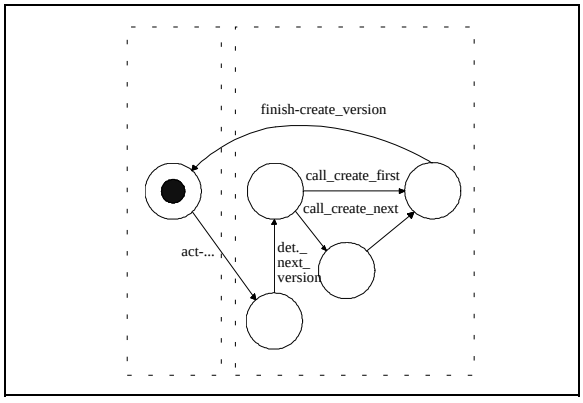

Figure 29. int-create_version actual behaviour

The actual internal behaviour of Figure 29 deflects from this framework in the same way as was the case for the int-*create_next* internal behaviour in the previous example: the *det_next_version* state transition expresses the need for some 'local' administration.

The examples of this section might suggest, that the internal behaviour might actually be constructed in a deterministic way, up to some adjustments needed for the local administration. Unfortunately, this is not the case. Note that for the implementation of its *review* operation, DesignEngineer is only relying on Design as a 'server object', as was the case in all the examples in this section.

For the DesignEngineer acting as a Designer, it is somewhat more complicated. In this case DesignEngineer is relying on two 'server objects' for the implementation of its *design* internal behaviour: Design (calling *open_for_modification*, *modify* and *close_modification*) and ProjectDocs (calling *create_version*). In this case we cannot build the framework for the internal behaviour using a straightforward construction from the corresponding behaviour abstractions.
The only thing we can conclude beforehand, is that the internal behaviour will include some concurrent interleaving of these behaviour abstractions. However, the possibilities for interleaving are further restricted by the PARADIGM communication already set up, the 'deep structure' of the calling graph[28], and the functionality requirements one wants to add for the behaviour.

It might be an interesting idea for future research, possibly using the correspondence between the behavioural view from [Ebert 94] and the abstraction concept from communication theory, to find out how the 'deep structure' of the calling graph restricts the framework for internal behaviour as used in this section.

5.3 Object situation and internal behaviour

The last section of this chapter will shortly review the concept of *state change* for an individual object.
When concentrating on a specific perspective in system models, one is often inclined to look upon every aspect of the system from this perspective. So from the data perspective it is natural to view every state of an object reflected in a specific and recognizable set of values for its attributes, or even to introduce an attribute to 'encode' the state of an object.
When using a multi-paradigm modelling approach for system analysis and design, one can also regard the various submodels involved as *complementary* (or orthogonal, cf. section 2.1).
In such an approach it is natural to regard everything we 'know' about the objects as the collection of features modelled in the various submodels.

This view of complementary models for the description of systems, was advocated for the OMT method by Ebert and Engels [Ebert 94], and formalized using the concept of *object situation*.
In their model at a given point in time, an object *o* is in a precisely defined situation *os*, when it has a concrete state *q* and a concrete value *f*, where the function *f* models the function assigning values to the attributes in a given situation *q*.
In this view, the state of the object does not exclusively determine the value of an object, and neither does the state of an object determine the value it holds for its attributes. The way the value of an object *o* may change then depends on its *situation*, i.e. it depends both on the value *f* that *o* has, and on the state *q* that *o* is in.

[28] which in this case tells us we must call *create_version* before *open_for_modification* as the external behaviour of DesignDocument dictates

Using SOCCA as a modelling method things even get more complex than in the OMT case. In SOCCA the object situation does not only depend on both the value function for its attributes, or the (external) state it is in, but also on the various states its internal behaviours may have and the number of internal behaviours that are activated at some time. The model allows for concurrency of internal behaviour. However, the restrictions modelled by the PARADIGM structure must be obeyed.
As infomation is encoded in the objects adhering to the SOCCA model in various ways, i.e. by attribute value, instantiation of acquaintances (related objects or relations), and external and internal state, there are various ways to model the functionality involved. To get at terms with these encoding mechanisms I constructed the following example objects (indeed an remarkable trio) and named them The Odd Transmitter, The Odd Receiver, and the Odd Transceiver. Their collective behaviour can be modelled in a SOCCA model; however the class model will be a very trivial one.

The Odd Transmitter has an *external* operation called *send*, by which it is asked to pass the value it has for its attribute *this_attribute* on to the Odd Transceiver, who will do nothing but to pass it on as soon as possible to the Odd Receiver, who will encode it into its corresponding attribute *this_attribute*. The value for this attribute can be any positive integer value.

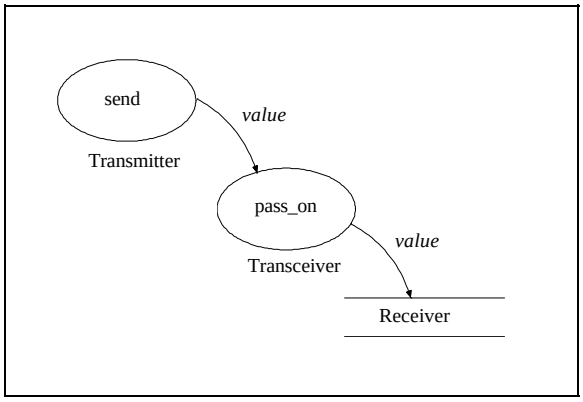

Figure 30. Dataflow diagram for the odd trio.

The functionality of this *send* operation is given in the dataflow diagram of Figure 30. It shows how the send process will pass on the value to the *pass_on* process of the Odd Transceiver, who will pass it on to the Receiver. Note how the 'passive role' of the Odd Receiver is designated by its being modelled as a data store.

The simplest way to model this functionality in the behaviour of the Odd Trio, is to model it as function calls with value passing. In such a model, the *send*(value) operation would initiate some int-*send*(value) internal behaviour calling a *pass_on*(value) operation from the Odd Transceiver. Subsequently the corresponding internal behaviour would call on some *receive*(value) operation from the Odd Receiver, to bring the value to its destination.
However, the Odd Trio will do things differently.

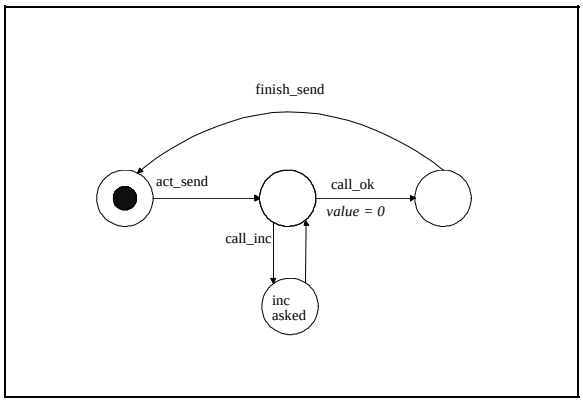

Figure 31. int-send (Odd Transmitter)

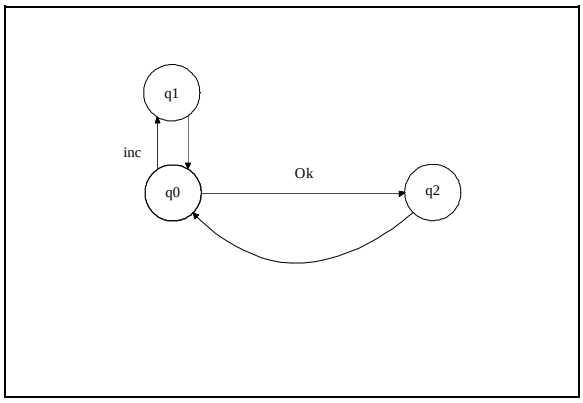

Figure 32. Odd Transceiver external STD

They decide to transfer the value not by value passing, but by 'transmission of behaviour'.
The internal behaviour for the *send* operation on the Transmitter will hold a copy of *this_attribute*, and gradually decrease this value by one until a value of 0 has been reached (cf. Figure 31). Meanwhile, for every substraction it calls upon the *inc* operation from the Transceiver. Upon finishing, a call is made to the *Ok* operation on the transceiver, and the send operation is finished. So not the value itself is transmitted or communicated, but the sequence of +1 operations leading to the value[29]. This 'encoding' could have been done otherwise, e.g. by factoring the value into its prime factors. The point is, we do not actually communicate the value itself, but a *behaviour prescription constructing the value*.

Now the Odd Transceiver does not actually 'hold' the value before passing it on.

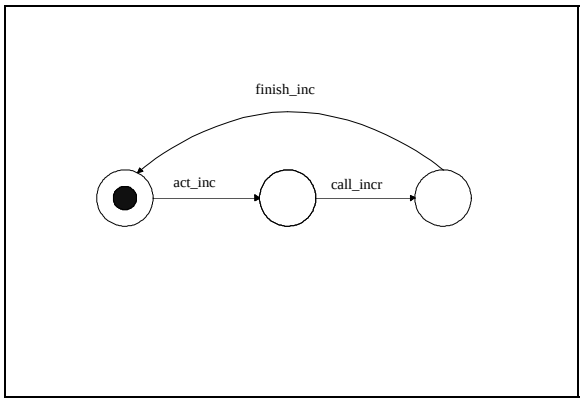

Figure 33. int-inc (Odd Transceiver)

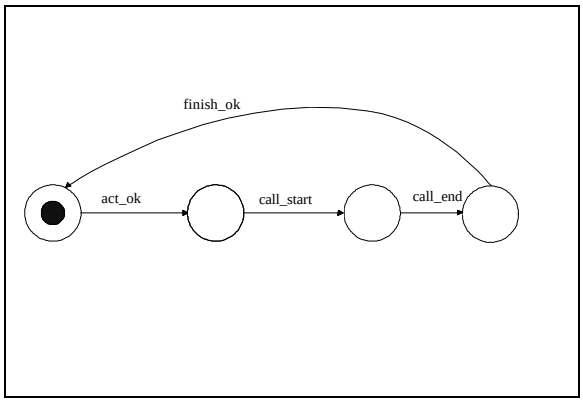

Figure 34. int-ok (Odd Transceiver)

[29] of course we must suppose some preceding behaviour for the Transceiver to 'clear' before accepting inc operations. This behaviour is not given here.

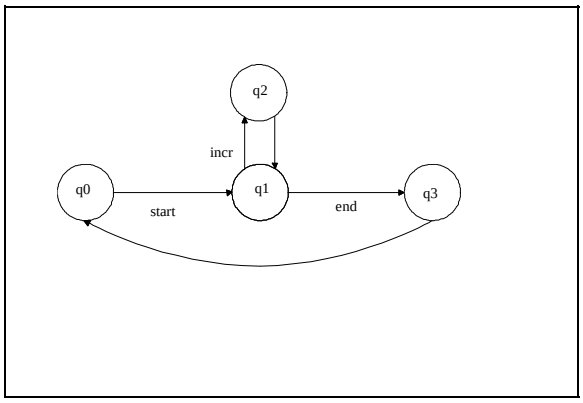

Figure 35. Odd Receiver external STD

For every call to *inc* it will start its int-*inc* internal behaviour. We will assume no PARADIGM restriction for the multiple instantiation of this behaviour. However, this int-*inc* behaviour is not allowed to finish off immediately: it will make a call to the *incr* operation of the Receiver, and this call is restricted in the 'standard' SOCCA way, in which the Receiver external STD acts as a manager. However, initially we assume the Odd Receiver to be in its external state q0, and is not allowed to answer to the call-*incr* calls from the Odd Transceiver, until some state transition to the state q1 has been made. Note that the Odd Transmitter will initiate as much int-inc internal behaviours as the value of the original attribute we wanted to transfer. So here in a way the number of int-*inc* internal behaviours started is the encoding of the original attribute value.
The call to the *Ok* operation on the Transceiver will trigger the int-*ok* internal behaviour of the Odd Transceiver. This behaviour will call the *start* operation on the Receiver, to bring the in the state ready to handle all the *incr* calls waiting from the int-*inc* internal behaviours. When all these behaviours have been dealt with we will alow the transition (q1 $\rightarrow$ q3) on the Receiver external STD. And on the Receiver some int-*incr* internal behaviour will of course 'restore' the value of some attribute.

This (indeed very artificial) example shows us:

- The possibilities for the encoding of infomation into internal states, external states (or rather communication states) and attributes are more complex than in the 'standard' object-oriented models. So in the SOCCA models the concept of object situation is in fact more complex than in other object-oriented models.

- It is possible to 'transfer value' by 'communicating behaviour': in fact the collective behaviour of the odd trio is an expression of the functionality of the DFD of Figure 30. There are two other possibilities: passing value directly by returning value upon calling and allowing for value parameters in the calls, or by the creation of 'value holding' objects, flowing from transformation to transformation.

6 Structural integrity and behaviour

In chapter 4 we saw how one of the points of focus in the CADDY project was the maintenance of data model instance integrity as prescribed by the EER schema. This was done by 'packaging' basic actions into so-called *elementary actions*. To define and build these and other complex database transaction the language ViAL was conceived and implemented.
We then argued that it was one of the restrictions to put on the *behavioral* model that it should provide for a mechanism to safeguard against violations of the database integrity using the communication between PARADIGM manager- and subprocesses.

What would be needed is a kind of 'generic behaviour' or 'blueprint behaviour' for the instantiation and deletion of objects. The following example shows how one may set about to define such a generic behaviour.

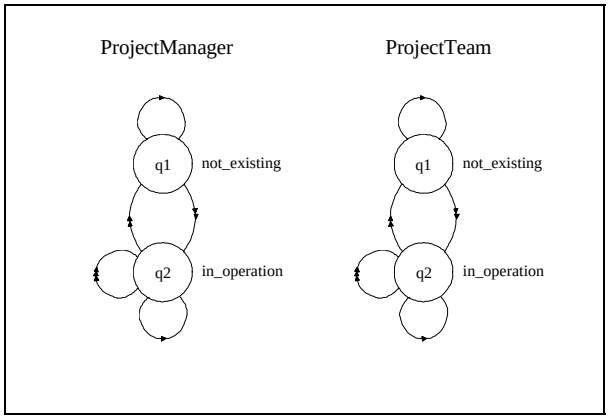

Figure 36. Generic external behaviour

In Figure 36 we see what I have called 'generic behaviour' of two objects from the SOCCA model: ProjectManager and ProjectTeam. Their external behaviour shows only two states: they are either there (*in_operation*) or they are not (*not_existing*).

In the *in_operation* state there are three possible transitions:
- from *in_operation* to *not_existing* (the double arrow), indicating a *delete* method;
- from *in_operation* to *in_operation* (the triple arrow), indicating the call of an arbitrary operation, and thereby possibly starting some internal behaviour;
- from *in_operation* to *in_operation* (the single arrow), indicating a sojourn in this state, without any 'action'..

And likewise in the *not_existing* state, there are two possible transitions:
- from *not_existing* to *in_operation* (the double arrow), indicating a *insert* method;
- from *in_operation* to *in_operation* (the single arrow), indicating a sojourn in this state, naturally without any 'action'.

However, ProjectManager and ProjectTeam may not independently go about and change states. They are bound to some form of collective behaviour by the restriction (specified in the *class model*) that for each ProjectTeam there must be one and only one ProjectManager and vice versa. This may be expressed as a behaviour restriction in PARADIGM by defining a subprocess and trap structure for the collective behaviour of the objects, and by defining a manager process to enforce the restriction.

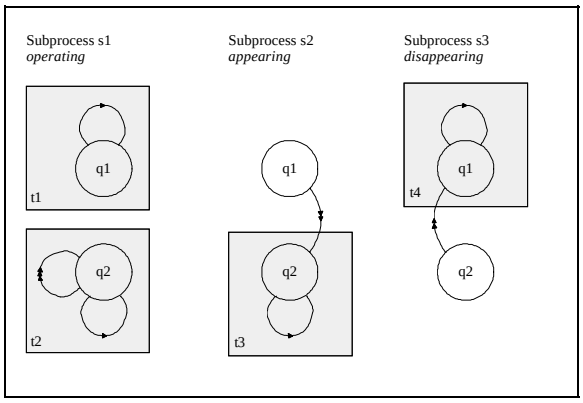

Figure 37. Subprocess and trap structure

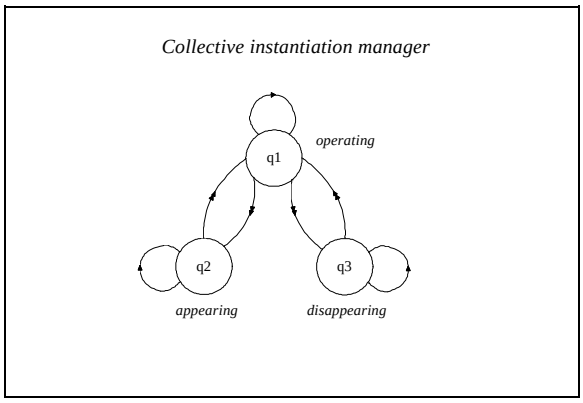

Figure 38. Manager process

This behaviour restriction in shown in Figure 37 and Figure 38. There are three subprocesses for the each of the two objects ProjectManager and ProjectTeam. The three subprocesses bear obvious names: *operating*, *appearing* (or instantiating) and *disappearing*.
Now there are three possible states, as the manager process indicates.

- The manager state *operating* (q1), corresponding to the state where both ProjectManager and ProjectTeam are in the corresponding subprocess (labelled s1) and consequently either 'instantiated and operating' i.e. responding to export operation calls (other than those causing instantiation or deletion) or are both *not_existing* (and remain that way);
- The manager state *appearing* (q2), when both ProjectManager and ProjectTeam are forced to 'come alive', i.e. make the transition *non_existing* → *operating*. Note how they may not respond to other export operations in this subprocess;
- The manager state *disappearing* (q3), when both ProjectManager and ProjectTeam are forced to 'go to rest', i.e. make the transition *operating* → *non_existing*. In this subprocess the normal export operations are also forbidden. So the initiation of new internal behaviours is not possible in this case. We suppose that all internal behaviours have a act-*operation* and finish-*operation* initiation and finishing behaviour, and that the internal behaviour is managed in the 'normal' way by the object behaviour. The object ProjectManager will only be allowed to make the transition *operating* → *non_existing* when all its internal behaviours have come to a 'non existing' state.

The trap structure (for both the objects ProjectManager and ProjectTeam) indicates the possible transitions the manager process can make:

trap t1 is a trap from *operating* to *appearing*;
trap t2 is a trap from *operating* to *disappearing;*
trap t3 is a trap from *appearing* to *operating*;
trap t4 is a trap from *disappearing* to *operating*.

So, like in the case of the WODAN change enactment manager [Wulms 94] we now have a manager process managing two processes, both acting as managers to their internal behaviours themselves. We could say that the 'Collective instantiantion manager' is the behaviour model for the 1-1 relation between ProjectTeam and ProjectManager.

In some cases however the restrictions as defined in the class diagram are much more subtle.

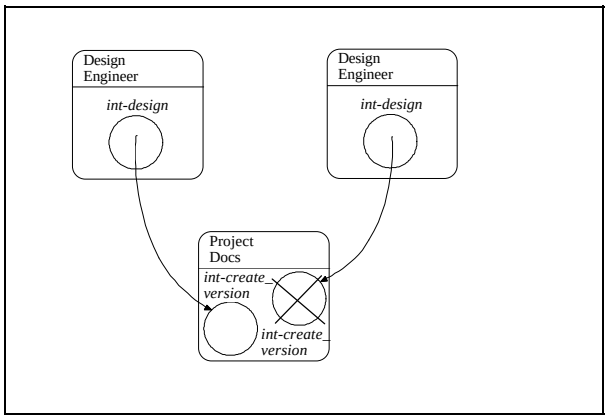

Figure 39. 2nd DesignEngineer must wait

Although this was not in fact modelled by the SOCCA class diagram of Figure 10 we would normally suppose that the attribute combination *name+version* would be a *partial key* or *qualifier*[30] uniquely determining the object DesignDocument as a component of ProjectDocs. So there should not be two DesignDocuments belonging to the same set of ProjectDocs having the same value for both their *name* and *version* attributes.
This restriction is indeed enforced by the behaviour of the objects as defined in [Engels 94]. Consider the following scenario, indicated in Figure 39.
An 'unthoughtful' ProjectManager 'calls upon' two different DesignEngineers DE1 and DE2 to start their *int-design* internal behaviour for the same DesignDocument and bearing the same name at about the same moment. Both DE1 and DE2 can carry out their *act-design* internal operation and start their corresponding internal behaviour. Now one of them, say DE1 is fastest and calls the object ProjectDocs[31] first with a *create_version* call. ProjectDocs starts its (first) internal behaviour, making the internal transition *act-create_version* and promptly returns to its *neutral* state.

Now for a *new* internal behaviour *int-create_version* (for the same document name) it can not immediately make the transition *act-create_version* again upon the reception of the *create_version* call from DE2. It must wait for the internal behaviour *int-create_version* to enter its trap t7 (cf. [Engels 94], p. 20) which creates a DesignDocument with a separate version number for each *create_version* call is receives. So here the uniqueness of the partial key *name+version* is enforced by the ProjectDocs manager process. It is as would the external behaviour of ProjectDocs (acting as a manager for *int-create_version*) act as a *lock manager* on the update of the key values for DesignDocument.

[30] see e.g. [Elmasri 89] section 3.3.4 or [Rumbaugh 91] section 3.3.5

[31] consider the case where there is only one object instantiated for this class

The two examples of this chapter support the following observations:

- it is very well possible to construct behaviour restrictions on the dynamic models for the various objects that ensure that the restrictions of the class model are enforced for any external use one can make of the object instances[32], and it in fact can be done quite elegantly using PARADIGM.
- when analysing the example problem which was modelled in [Engels 94], i.e. a small subproblem of the ISPW-6 case, one can see that indeed implicit and explicit restrictions of the class diagram in Figure 10, are enforced by the behaviour and communication.

The next question then is: but how do we guarantee the enforcement of this 'structural integrity' of the behaviour ?

There are three ways to ensure this:

- in relying on the 'craftsmanship' of the designer/analyzer, responsible for the dynamic modelling, as is done in the construction of the dynamic models in [Engels 94];
- in finding an algorithmic way to generate 'generic' or 'blueprint' behaviour from the class diagram for all objects, in much the same way as elementary actions were constructed from the EER model in [Wolff 89], and use this as a starting point for the construction of the dynamic model;
- in combining both approaches, where the design from craftsmanship is compared to an algorithmically constructed design with a verification method.

The construction of 'generic instantiation and deletion behaviour' from a class diagram is not straightforward however, as it was in the case of the construction of elementary actions from the EER model. There were only two operations to consider in this construction: the insertion and deletion of database elements (entities, relationships, components). For an object-oriented modelling method however, many different operations may lead to instantiation or deletion of objects. The design of these operations is 'put at the discretion' of the analyzer/designer, who is not only interested in the compliance to the class model, but also in the compliance to the functional model.

So in my view the most promising way, is the third alternative:

- as a first step, generate blueprint behaviour from the class diagram;
- as a second step, use craftsmanship to design detailed behaviour, and prove that this detailed behaviour complies to the desired blueprint behaviour.

[32] the first example shows that there are 'states' where both objects are 'not yet' fully 'instantiated'. However, they are not allowed to respond to any calls of their export operations before this is the case

7 Functional Modelling in SOCCA

In chapter 3 we saw how the technique of dataflow modelling is an adequate tools for the *analysis* of functionality of complex systems. In a top-down manner, subsequently refining the level of description we can break down the high-level desired or observed functionality into a fine-grain complex of functional elements, the 'atomic' processes.
In chapter 3 we observed how in OMT *functional modelling* a relation was made between these

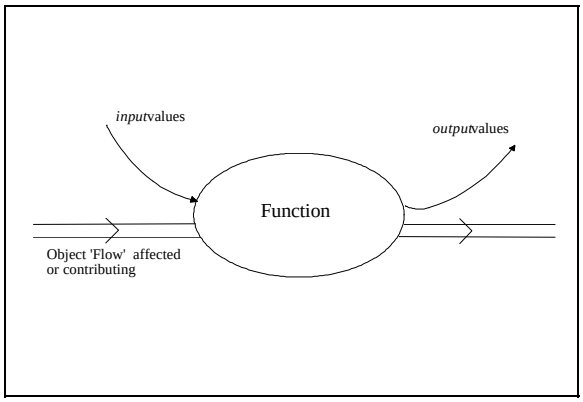

Figure 40. The idea of a function

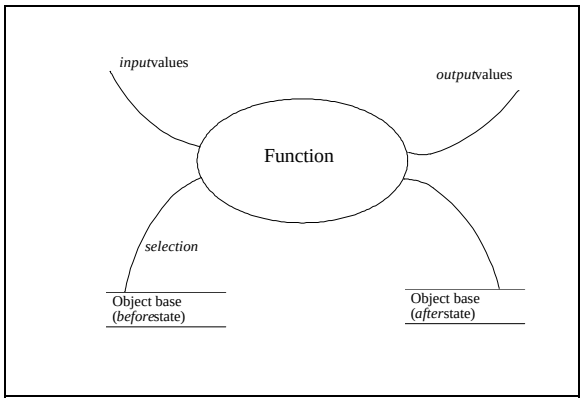

Figure 41. Function with data stores

atomic processes and the 'elements of behaviour' of objects, i.e. its operations and transitions between states. However, this relation was rather informal; there was no attempt for a more formal approach. In that same chapter we noticed how this 'informal' status of data flow diagrams was also noticed by others [Ghezzi 91] as one of the drawbacks of the use of data flow diagrams.
In this chapter I will try to say something about a possible approach to 'formalize' the use of data flow diagrams and to sharpen the relation between 'atomic' processes and the other two submodels of SOCCA. To complete such a description is a sound task, however, for which I neither had the necessary background in the construction of formal models, nor the available time. And so, the ideas will remain very sketchy.

7.1 The idea of a function

First we try to get an idea of a functional 'component' of a functional model in SOCCA, which can be considered an object-oriented method. This idea of a function is in my view very well captured in the diagram of Figure 41. A function or process is in its most general form the *transformation of data values* (as indicated by the top arrows) with a necessary *side-effect* on a distinct *set* of objects (as indicated by the lower arrows). This side effect is a *status* change or *configuration* change in the affected objects, but may also involve the creation or deletion of objects.
Naturally there are simpler functions, i.e. those just transforming data values, and not having any side effects, or the other way around, functions just intended to change the state of a set of objects and 'not computing anything'.

To analyze or specify a given function, we must indicate:

- *what* value transformation is made;
- the 'selection' of objects that either is to be *affected* by the side effect, or that are *contributing* to the value transformation;
- *what* status change the affected objects will make.

To visualize the 'outside viewer's' look on a function we could further exploit the expressive power of the data flow diagrams by adding the symbol for a data store to the diagram of Figure 41, as indicated. Here, the symbol for the data store is indicating a 'substate' of the object base, i.e. the aforementioned 'selection' of objects holding some status 'before' and 'after' the process. This *object base* is the 'object model' we create for or software process, i.e. the totality of model components.
As visualized, the arrow *from* the data store indicates this 'selection' of objects; the arrow *to* the data store indicates the status change of the affected objects, and possibly the creation and deletion of some objects. For the moment, we only worry about the 'functionality' of this 'transaction', as is natural from the analyzers or designers point of view[33].

Now there are two important considerations to be made:

- primitive functionality is a *local* phenomenon.
  Although the visualization is 'from the outside', i.e. holding a *global view* in the terminology of [Engels 94], p. 6, in reality the *active components* of our (object-oriented) model are (some of) the objects themselves. When we specify or analyze the functionality at this lowest level we therefore must take a *local* or *nearby* view on the model, and take a viewpoint 'from the inside', i.e. studying the model the way the active component 'sees' the model . This view is different from the one used for ViAL specifications; in ViAL we construct functionality from the 'outside', or, in SOCCA terms, from a *global* point of view

- functionality is *restricted*
  We are free to analyze and specify the functional model independently from the other models (the *orthogonality* of the submodels and perspectives), but only to a certain extent: the 'functionality' we specify 'knowing' the relation of the 'atomic' components with elements from our structural or behavioural submodels must be in accordance with the restrictions of those submodels.
  Therefore we may not create or delete objects in a way that is not allowed by our class or object model, and indeed may not model a sequence of state changes in the side effect that in violation with the state transition diagrams of our behavioural model.

[33] To restrict ourselves to the functionality means, that for the moment we are not interested in 'implementation' aspects like concurrent transactions

7.2 The elements of specification

So what are the elements with which to specify functionality ? The 'basic building blocks' we use can be classified according to the three 'tasks' we mentioned in the previous section:

- elements to build values from other values, i.e. transforming input values to output values
- elements to 'select' objects, which we will call *queries*, like they are called in [Rumbaugh 91] (cf. chapter 3) or ViAL, and to generate values from objects
- elements that transform object (base) states

In accordance with the EER model for which a formal semantics was developed in [Gogolla 91], the class or object model for SOCCA can be considered a class or object model over a *data signature*. In short, such a data signature consists of value domains for the value types one wants to consider modelling, and operations and predicates defined on those domains and generating other values from these domains. Those primitive data signatures over which our models might be defined are e.g. the **int** and **bool** domains with all their usual operations and predicates.
For the use in our class model we can extend these primitive types with **set**, **list** and **bag** constructors to account for sets, ordered lists or multisets defined over our primitive data domains.

The operations, predicates and constructors used can transform values in a well-defined way. They all can be considered 'primitive' functions of our specification or analysis model, i.e. the first 'type' of elements. The outcome of such a transformation applied to a value furthermore has a well-defined data type (or data sort).

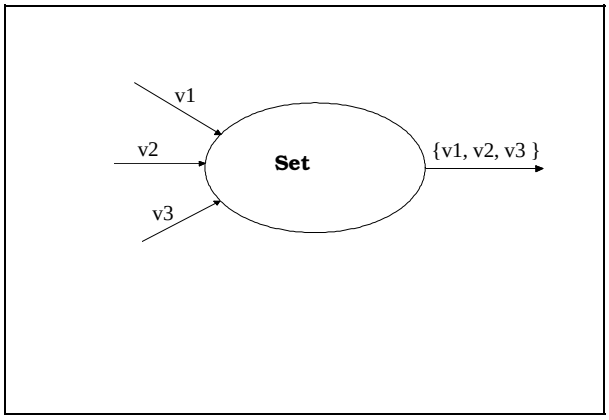

Figure 42. Set construction

These transformations or functions can be represented rather 'naturally' in a data flow symbol like the one in Figure 42, showing how one can arrive at the *set* { v1, v2, v3} from the input values v1, v2, v3.
Note however that the representation is limited, as there is no natural 'order' defined for the input values in our diagrams. The *set* {v2, v1, v3} would mean the same thing, but the *list* <v2,v3,v1> is indeed different from the *list* <v1,v2,v3>.

Again, using data values of a **set of**, **list of** or **bag of** type, one can build other sets, bags or lists using the well-known set operations $\cup$ or $\cap$, or values of type **int** or **bool** using the well-known aggregate functions over sets, bags or lists like CNT (counting elements) or MAX. So from this 'toolbox' of primitive functions we can build more complex structures in combining the functions by the well-known mechanism of function composition, also 'naturally' to be captured in our diagrams, by 'attaching' the 'output arrow(s)' from one 'process' or 'bubble' to an 'input arrow' of another. There are however some restrictions:

- Data flows to and from processes are typed. We therefore can only connect the output flow from one process to the input flow of another process if they are *type compatible*. These type compatibility rules have to be formulated: one can imagine a rule according to which any object subclass can be used as an input type where the superclass is expected.

- Composition rules must adhere to the restrictions from the behaviour model. We cannot allow sequential composition of processes in which the sequence for application is not compatible with the operation sequence prescribed in the SOCCA behaviour model.

So from input values we can build other values or sets of values. But where do these values come from ? They may be 'abstracted' from the analysis by viewing every object in the system as a data source or data sink, which we do not further analyze or specify. In that case we are done. But we want more; we want our model to show how data are delivered and transformed by the objects. Remember we are looking at the objects from 'inside'. So for every process we want to analyze, we assume there is an object performing it as an operation or a sequence of operations. Processes that will require 'teamwork' from different objects must then be regarded as processes on the aggregate object representing the 'team'.

It remains to find out what data can be delivered by an object.
Inspiration for DFD models on how to 'extract' values from the objects in a functional way can be found in the high-level *functional* datamodels known from database theory (see e.g. [Elmasri 89], chapter 15).
First of all, there are the *attributes*. Every attribute (from the viewpoint of the object in question) might be considered a 0-ary function. Actually, it is a little more complicated because not in every object situation a value has to be delivered (cf. chapter 6). For these cases we might adopt the special null value ($\perp$) like in [Gogolla 91], because in principle we want all attributes to be callable for every object situation if we want to. Note that including sets, lists and bags into our data signature allows for the use of multi-valued attributes.
The invocation of attribute *methods* from outside the object will assume we add export operations for the call to these attribute methods to any state of our external STDs where they are applicable. They do not however induce state change.
The value to be returned by these attribute calls will be assumed to return immediately.

The other way to 'generate value from an object' is to call an *operation*. Until now this was not done for the operations in a SOCCA model, but they might be considered data generating functions, and assigned a data type for their 'output value'. Of course there will be 'typeless' operations, whose only effect will be the state change of an object. So as a 'data generator' they may be treated like attributes. But in this case there must be a side effect: by calling an operation, the object will undergo a state change, and induce state changes in its 'acquaintances', i.e. the objects it is communicating with either by calling upon their respective export operations, or by 'forcing' them to different behaviour by the communication mechanisms set up in our PARADIGM specification.

The last thing to consider is, how objects may call upon their acquaintances. To set up any communication with objects contributing to its methods, the object must use its *access paths* , i.e. find the objects to communicate with using its relations or associations with other objects.
In the same way as it is done in the functional data model, one might define a function for any relation defined for an object, that will deliver for value the set of objects partaking in the relation with the objects. As an example how an object might generate value from its acquaintances, the diagram of Figure 7 shows how the implementation of the *create_version* operation for ProjectDocs will query its *design* components to determine the value for the version number to assign to the new DesignDocument to create.
By 'calling upon' its *design* component, a set of objects (components) will be returned, to which the *version* attribute function will be applied. This will deliver a set of values, the *maximum* value of which will be the basis to construct the new version number from.

8 Conclusions and suggestions for further research

The work on this thesis started out with the observation that 'some things were missing' from the SOCCA method as explained in [Engels 94]. More specific, as was explained in chapter 2, there was no clear *functional perspective* expressed in the SOCCA models, and furthermore, it was unclear how the functionality of the example software process from [Engels 94] was related to the class and behaviour models developed.

Most of my work has been devoted to find out *what* exactly 'was missing' in the SOCCA method. And, admitted, I have not found a clear answer. Most of the desired functional features seem to be 'hidden' in the behaviour and communication model, which were already designed, perhaps unwittingly, with a clear concept in mind on the functionality to be expected from the model.

From the discussion in chapter 3 on Data Flow Diagrams it should be clear, that for the analysis of software process modelling a separate concentration on functional aspects is indeed useful, and that the technique of data flow modelling is a valuable tool. Using this tool a more detailed analysis of the ISPW-6 and ISPW-7 example process models can be performed, as there are many features there that are still not captured in the current SOCCA models.
To integrate this technique of data flow diagrams with SOCCA's class and behaviour modelling should be an objective for further research on SOCCA. As indicated in chapter 4 the ViAL approach is in my opinion not the road to follow. Modelling complex transactions in ViAL has too much of an algorithmic or *procedural* flavour, where a more functional or *applicative* approach is desired. Moreover, ViAL's control structures have a number of shortcomings that make it less applicable for the SOCCA situation.

In chapters 5 and 6 I clarified the some aspects of behaviour:

- the behaviour that is necessary to maintain the communication requirements for the model once the export operations and external STDs have been accomplished;
- the restrictions for the behaviour that can be derived from the class model;
- the behaviour that is the expression of algorithms for the realisation of other functionality .

Here too, there are some interesting questions remaining. It is still unclear how the 'deep structure' of the calling graph further restricts the freedom of modelling for the internal behaviour. It might be an interesting subject for further study, possibly using the methods of formal communication theory. And, it might be an interesting endeavour to actually derive blueprint behaviour for all the SOCCA objects from the class diagram and start the construction of export STDs from this behaviour.

In chapter 7 a very modest start has been made for the imbedding of the technique of dataflow diagrams in SOCCA's methods. There is still much to be done in this field.

It is still unclear how one of the central themes of an applicative approach, i.e. parameter binding in functional composition, should be related to SOCCA's call-operation asynchronic conventions. And second, the concept of object situation is much more complex than in the case of other object oriented methods like OMT, because of SOCCA's internal behaviour concept and will have to be formalized.

9 References

[Baeten 86] J.P.M. Baeten, *Procesalgebra: een formalisme voor parallelle, communicerende processen*. Deventer : Kluwer, 1986

[Chen 76] P.P. Chen, *The Entity-Relationship Model - Toward a Unified View of Data*. In: *ACM Transactions on Database Systems*, 1(1), March 1976, p. 9-36.

[Coad 91] Peter Coad, Edward Yourdon, *Object-Oriented Analysis*. 2nd ed. Englewood Cliffs, NJ : Yourdon Press, 1991.

[Constantine 79] L.L. Constantine, E. Yourdon, *Structured Design*. Englewood Cliffs, NJ : Prentice-Hall, 1979.

[Curtis 92] Bill Curtis, Marc I. Kellner, Jim Over, *Process Modelling*. In: *Commun. ACM* 35, 9 (Sept. 92)

[Ebert 94] Jurgen Ebert, Gregor Engels, *Structural and Behavioural Views on OMT-classes*. In: Elisa Bertino, Susan Urban (eds.), *Object-Oriented Methodologies and Systems*. Proc. Int. Symp ISOOMS 1994. LNCS, Berlin : Springer, 1994, p. 142-157

[Elmasri 89] Ramaz Elmasri, Shamkant B. Navathe, *Fundamentals of Database Systems*. Redwood City, California : The Benjamin/Cummings Publishing Co. Inc., 1989

[Engels 90] Gregor Engels, *Elementary Actions on an Extended Entity--Relationship Database*. In: H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science, International Workshop*, LNCS 532, Berlin : Springer, 1990, p. 344-362

[Engels 92] Gregor Engels, Perdita Löhr-Richter, *CADDY: A Highly Integrated Environment to Support Conceptual Database Design*. In: G. Forte, N. Madhavji, and H. Müller (ed.), *Proc. 5th Int. Workshop on Computer-Aided Software engineering, Montreal, Canada*. IEEE Computer Society Press, 1992, p. 19-22

[Engels 93] Gregor Engels, Perdita Löhr, *Visual Specification of Complex Database Actions*. In: J. Eder, C.A. Kalinichenko (eds.), *Proc. 2nd Int. East/West Database Workshop, Klagenfurt, September 1994*. Workshops in Computing. Berlin : Springer, 1995, p. 303-314.

[Engels 94] Gregor Engels, Luuk Groenewegen, *SOCCA: Specification of Coordinated and Cooperative Activities*. Technical Report 94-10. Leiden University, Department of Computer Science, 1994.

[Gerlach 92] K. Gerlach, *Ein Interpreter für visuell spezifizierte komplexe Aktionen auf EER-Datenbanken*. Diplomarbeit. Technische Universität Braunschweig, 1992.

[Ghezzi 91] Carlo Ghezzi, Mehdi Jazayeri and Dino Madrioli, *Fundamentals of Software Engineering*. Englewood Cliffs, New Jersey : Prentice Hall, 1991.

[Gogolla 91] Martin Gogolla, Uwe Hohenstein, *Towards a Semantic View of an Extended Entity-Relationship Model*. In: *ACM Transactions on Database Systems*, 16:369-416, 1991.

[Graham 94] Ian Graham, *Object Oriented Methods*. 2nd edition. Wokingham, England : Addison-Wesley , 1994.

[Groenewegen 86] Luuk Groenewegen. *Parallel Phenomena* 1-14. Various Technical Reports. Leiden University, Department of Computer Science, 1986-1991.

[Harel 87] David Harel, *Statecharts: a visual formalism for complex systems*. In: *Science of Computer Programming* 8 (1987), 231-274.

[Hennemann 91] Christiane Hennemann, *Entwurf und Implementierung einer Sprache zur visuellen Spezifikation von Aktionen auf erweiterten Entity-Relationship-Datenbanken*. Diplomarbeit. Technische Universität Braunschweig, 1991.

[Hohenstein 92] Uwe Hohenstein, Gregor Engels, *SQL/EER - Syntax and Semantics of an Entity-Relationship-based Query Language*. In: *Information Systems* 17(3):209-242, 1992.

[Kellner 91] Marc I. Kellner et al., *ISPW-6 Software Process Example*. In: Takuya Katayama, ed., *Proceeding of the 6th International Software Process Workshop : Support for the Software Process*, IEEE Computer Society Press, 1991.

[Kellner 91a] Marc I. Kellner et al., *ISPW-7 Software Process Example*. 7th International Software Workshop. Yountville, California, 16-18 October 1991.

[Milner 89] Robin Milner, *Communication and Concurrency*, New York : Prentice Hall, 1989.

[Morssink 93] P.J.A. Morssink, *Behaviour Modelling in Information Systems Design : Application of the PARADIGM Formalism*. Ph.D. Thesis, Leiden University, Department of Computer Science, 1993.

[Petre 95] Marian Petre, *Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming*. In: *Commun. ACM* 38, 6 (June 95), p. 33-44

[Royce 70] W.W. Royce, *Managing the Development of Large Software Systems: Concepts and Techniques*. Proceedings IEEE WESTCON, 1970.

[Rumbaugh 91] James Rumbaugh et al., *Object-Oriented Modeling and Design*, Englewood Cliffs, New Jersey : Prentice Hall, 1991.

[Steen 88] M. R. van Steen, *Modelling Dynamic Systems by Parallel Decision Processes*. Ph. D. Thesis, Leiden University, Department of Computer Science, 1988.

[Wolff 89] Matthias Wolf, *Eine Sprache zur beschreibung schema-abhängiger Aktionen in einem erweiterten Entity-Relationship-Modell*. Diplomarbeit. Technische Universität Braunschweig, 1989.

[Wulms 94] Alex Wulms, *Adaptive Software Process Modelling with SOCCA and PARADIGM*. Master Thesis. Leiden University, Department of Computer Science, 1994.

[Yourdon 89] Edward Yourdon, *Modern Structured Analysis*. London : Prentice Hall, 1989.