Abstract

Goal of the thesis was to make an inventory of the required functionality for a future environment for creating and using SOCCA models.

Usually such an inventory is made with use cases, which describe the functionality informally. However, because a description of SOCCA models was already at hand, we first tried to describe accurately which models and model fragments are used in SOCCA and more important: which relationships exist between those models and model fragments. This resulted in the meta model for SOCCA.

Certain fragments of this meta model provide an idea for future system components. This, because those fragments represent certain concepts of SOCCA, which will have to be supported by the future environment. Besides these components, other expected system components can be introduced as well.

As the expected system components are known to us, we can model their expected functionality in use cases. This functionality is partially already defined by the concepts from the meta model which they are responsible for. The use cases do now not just provide starting ideas, but present functionality which is more or less a consequence of the way a process is modelled in SOCCA.

All this results in a document useful as a starting point for modelling the future system more precisely: for example in a SOCCA model.

Acknowledgments

I would like to thank dr. Luuk Groenewegen and ir. Ed van der Winden for their guidance and support during my thesis. Also thanks to the SEIS projectgroup for their insights and support.

Many thanks to my mother and father for making it possible for me to study (not only financially) and motivating me. I would also like to thank the rest of my family and friends for their love and support.

Last but certainly not least, I would like to thank my girlfriend Maureen for putting up with me for over two years now, which is not always an easy task. Maureen, I love you.

I could not have done it without you all...

Contents

Chapter 1

Introduction

This document contains the master's thesis from Ezra Schuitema, done under supervision of dr. Luuk Groenewegen, ir. Ed van der Winden and with support of all members of the SEIS project group.

1.1 Goal

The goal of this thesis was to make an inventory of the required functionality for a future environment for creating and using SOCCA models (refer to 1.3 for a small introduction on SOCCA), with the emphasis on the part of the environment responsible for the creation of SOCCA models. This has resulted in a document in which the expected functionality is listed and which is an approach towards the future SOCCA environment (especially a requirements definition document (RDD)). Besides this, some interesting ideas regarding editors and similar environments as well as ideas on the way to make an inventory of the required functionality itself are presented. More on this can be found in chapters 5 and 6.

1.2 Contents

We want to describe the functionality for the future environment for SOCCA with use cases. This is usually the first step for describing new environments, when little is known about the expected functionality. In our case, however, we already have a little knowledge, because an informal description of SOCCA models is at hand. A good description of SOCCA models could provide us with a better understanding of the required functionality of the environment. Therefore, we need a more formalized description of SOCCA models.

A more formalized description (or meta model) of SOCCA is created in chapter 2. This meta model describes the dependencies between the different diagrams and concepts in SOCCA.

The information in the meta model provides some ideas on some system components which can be expected in the future system. Chapter 3 discusses this. Besides the system components which can be derived from the meta model, other expected system components are introduced as well.

In order to provide a detailed description of the required functionality for the system components introduced in chapter 3, use cases are introduced in chapter 4. An enhancement to the way to describe use cases is introduced and the link between the use cases and the system components is made.

The use cases for our future system are actually given in chapter 5 and chapter 6. The first of these two chapters focuses on the functionality of system components related to a user which builds diagrams. Chapter 6 focuses on a user which takes diagrams together and builds (sub) models with them. Both chapters not only contain the use cases, but also a discussion on these use cases an notes on possible implementation is provided. The discussion provides some alternatives and present some choices which have to be made, while the system notes focuses on possible ways to implement the functionality into the future system.

The conclusion to the work done is given in chapter 7.

1.3 A (very) small introduction to SOCCA

SOCCA (Specification Of Coordinated and Cooperative Activities) is a specification formalism for modelling software processes created by prof. dr. Gregor Engels and dr. Luuk Groenewegen at the University of Leiden (the Netherlands).

SOCCA uses three perspectives to model a process: the data perspective, the behaviour perspective and the process perspective.[1] The best from different formalisms is used in SOCCA to model the different perspectives. This is called eclectic modelling ([GE95]). Different formalisms are used, because each formalism is can be used best to model only one of the perspectives. The formalisms are integrated in SOCCA to supply a model which covers all perspectives. It is this separation in components according to the perspectives, which provides the decrease in complexity in modelling software processes and the increase in the changeability and evolvability of such a software process model.

In SOCCA, the data perspective of a process is modeled by class diagrams with uses relationships added to them. The behaviour perspective is modeled by STDs modelling the external and internal behaviour of classes and the communication perspective is modeled by PARADIGM on top of these STDs.

For a far more detailed explanation of SOCCA, examine [EG94]. For information on PARADIGM, look at [Gro91]. Examples of SOCCA specifications are given in [Wul95], [Rij95] and [Höp94].

1.4 Terms and abbreviations used in this thesis

In this thesis, some terms used could lead to some confusion. This, because sometimes the same term could refer to two different concepts. We have tried to avoid this by introducing specialized terms for every concept.

Besides terms which could be interpreted differently, also other terms need some introduction before use. In the list below, all often used terms and abbreviations are explained:

[1]As the process perspective is in development, this thesis will not take it into account. The behaviour perspective can be split up in a part only involved with behaviour and a part involved with communication: the (new) behaviour perspective and the communication perspective. This thesis will use these two perspectives instead of the one 'behaviour perspective'.

building block	The smallest element of a visual language which can be used to draw a diagram [HW96]. The element represents a concept used in the visual language.
building block class	A meta model class which represents a building block. See section 3.3.1 for more details.
chunk	A group of SOCCA diagrams, which are grouped together for a particular reason. Refer to section 3.4.2 for more details.
class diagram	A modelling formalism to model object oriented systems. The class diagrams used in SOCCA are the same as used in the UML (Unified Modelling Language). Refer to [BRJ97] for a description of UML notation. In this thesis, the notation for the class diagrams used for the meta model is somewhat different. Refer to section 2.1.1 for explanation.
constraint	A dependency between two or more classes which cannot be expressed by a relationship in a class diagram.
dependency	Used to indicate some kind of relationship between parts of SOCCA models. Dependencies often involve several relationships and constraints.
diagram	A diagram as it is used in SOCCA.
diagram class	A meta model class which represents a diagram type in SOCCA. See 3.3.1 for more details.
grouping class	A meta model class which represents groups of diagrams. See 3.3.2 for more details.
grouping relationship	A meta model relationship concerned with grouping diagrams. See section 3.3.2 for more details.
inter-diagram relationship	A meta model relationship which relates classes of two or more different diagram types. See section 3.3.1 for more details.
intra-diagram relationship	A meta model relationship which relates classes within one diagram type. See section 3.3.3 for more details.
link	An instance of an inter-diagram relationship. See section 3.4.3 for more details.
meta model	The meta model describes what a well-formed SOCCA model looks like in a class diagram.
meta model class	A class in the meta model.
meta model constraint	A constraint in the meta model.
meta model relationship	A relationship in the meta model.
plug	Plugs are points on building blocks where other building blocks can be connected. For a detailed description refer to section 3.4.1 on page 41.
relationship	A direct dependency between two or more classes in a class diagram. Denoted by a connection between the involved classes.
SOCCA model	A model 'written' in the SOCCA formalism.
STD	(State Transition Diagram) A modelling formalism for modeling behaviour. Refer to [EG94] for more detail.
submodel	A part of a SOCCA model containing all three perspectives. A submodel is a SOCCA model, but does only model a part of the process which needs to be modeled.

Chapter 2

The meta model for SOCCA models

2.1 Introduction

We want to make a system in which SOCCA models can be created. For this, we need to define what a correct SOCCA model is. Informally, the definition of a SOCCA model is already known (look at [EG94] for example). However, this definition has never been formalized. If the envisioned system should support SOCCA to the fullest extend, the formalisation of the relationships between the diagrams is needed.

SOCCA uses existing formalisms, namely object-oriented modelling based on class diagrams, state transition diagrams (STD) together with PARADIGM and object flow diagrams. These formalisms have all been formalized (for example: class diagrams in [BRJ97], PARADIGM in [Gro91] and object flow diagrams also in [BRJ97]) and so this will not have to be done again. However, the dependencies between the models from these formalisms in SOCCA have never been formalized. In this chapter, we will try to create a class diagram in which all existing relationships between those diagrams are given. Whenever the class diagram cannot cover all the properties of some dependency, we will add these properties in separate constraints. The constraints are written in a style resembling **Z**, in order to avoid difficulties in interpretation. We have chosen to add as few constraints as possible and thus we try to model as much as we can in the class diagram. Apart from formalizing the dependencies between the diagrams, the class diagram brings with it another benefit: it can be used for implementing the system, it provides the basis of an internal representation of SOCCA models.

The class diagram is created top-down. We first identify large and complex parts of a SOCCA model and then try to break these parts down in smaller pieces. These pieces can often be split up in parts as well, and so on. This way, we get a class diagram with most classes either grouping different other classes (part-of relationships) or describing a similarity between other classes (inheritance). An instance of the class at the top of this aggregation is identified with the real world notion of a SOCCA model, while the leafs (or bottom) of it represent the building blocks within the different formalisms. Note, that information about the layout of the diagrams and building blocks will not be included in the class diagram. Such information is not needed to understand a SOCCA model and thus not needed to model the dependencies between the diagrams.

In order to get a class diagram which is intuitively correct, we try to identify each class with some real world concept or object. This is not always easy. We also strive for the relationships in the class diagram to have meaningful names.

We have chosen not to model labels and names as separate classes. This was done in order to keep the class diagram smaller and therefore more clear. The correspondences between labels which are sometimes used in SOCCA, are modeled as relationships between the concepts which the labels label.

We have also not added attributes and operations to the classes. As the expected operations were not clear at the time when we created the meta model. They were not necessary as we were not creating the internal model for the system, but only a meta model for SOCCA. In the next step after this thesis, the attributes and operations will be necessary for a useful internal model.

2.1.1 The notation used for the meta model

In this subsection, we will describe the notations used for our meta model, shortly.

The class diagram

The notation for class diagrams used in this chapter, is a frequently used one. But, because no general accepted notation for class diagrams exist, we will explain this notation shortly.

Classes are drawn in rectangle. The name of the class is put inside the rectangle. See figure 2.1.

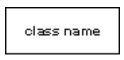


Figure 2.1: The way classes are drawn

We distinguish three types of relationships between classes: part-of relationships, inheritance (or is-a) relationships and common relationships (common relationships actually are all other possible relationships). These three types have their own notation, which is displayed in figure 2.2. The relationship types in that figure are as follows:

- From class A to class B: *common relationship.*
- From class D to class E: *part-of relationship.* (Class E is a part of the class D)
- From class G to class H: *inheritance relationship.* (Class H inherits from class G)

Note, that a name for the relationship only is supplied with the common relationships. Also note, that the direction is important with part-of and inheritance relationships.

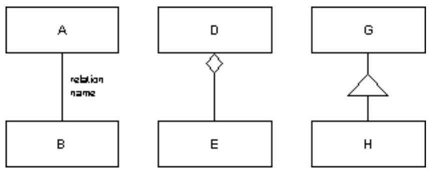


Figure 2.2: Relationships between classes

Relationships can also involve more than two classes. Figure 2.3 shows three of such relationships:

- *Common relationship* between class A, B and C;
- *Part-of relationship* from class D to class E and F;
- *Inheritance relationship* from class G to class H and I.

Note, that a common relationship between three or more classes uses a new graphical notation (the diamond). Like with the relationships between two classes, the direction of the relationship is again important with part-of and inheritance relationships.

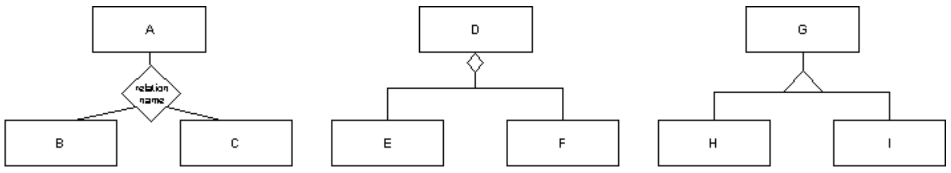


Figure 2.3: Relationships between more than two classes

For the common and the part-of relationships cardinalities can be provided to denote how many object of the class can or must participate in the relationship. The way cardinalities are drawn is shown in figure 2.4.

The meaning of these cardinalities is as follows:

- *cardinality of the relationship to class A:* Exactly one object of class A is related to the object on the other side.
- *cardinality of the relationship to class B:* Either zero or one objects of class B are related to the object on the other side.
- *cardinality of the relationship to class C:* Zero or more objects of class C are related to the object on the other side.
- *cardinality of the relationship to class D:* One or more objects of class D are related to the object on the other side.

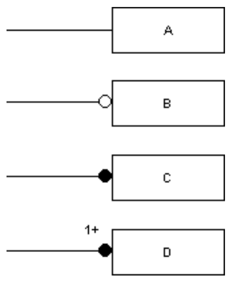


Figure 2.4: The cardinalities of the relationships

There are two concepts important with inheritance relationships. These concepts are:

- **disjoint versus overlapping:** An overlapping inheritance relationship allows objects of the 'general' class to be objects of several 'inheriting' classes at the same time. A disjoint inheritance relationship only allows objects of the 'general' class to be an object of <u>one</u> 'inheriting' class.
- **totality versus in-totality:** A total inheritance relationship requires every object of the 'general' class to be an object of one (or more) 'inheriting' class(es). With an in-total inheritance relationship, an object of the 'general' class does not have to be an object of an 'inheriting' class.

In figure 2.5 four kinds of inheritance relationships are displayed:

- *a:* disjoint and not total;
- *b:* disjoint and total;
- *c:* overlapping and not total;
- *d:* overlapping and total.

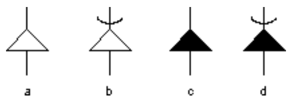


Figure 2.5: The four kinds of inheritance relationships

The constraints

As stated, the constraints are written in style resembling **Z**. Generally, a constraint looks like this:

definition restricted quantifiers | *pre-condition* • *condition*

An example of such a constraint is the following:

$$\forall u : \text{USES};\ c : \text{CLASS};\ o : \text{OPERATION} \quad | \quad (u \ \mathbf{calls}\ o) \bullet \quad (u\ \mathbf{to}\ c) \Leftrightarrow (o\ \mathbf{part\text{-}of}\ c)$$

Here, you can see that the definition of the restricted quantifiers is done with classes. '$\forall u$:USES' means something like: "for all objects u from class USES...".

The pre-conditions are written like the conditions. Used in those conditions are existences of instances of relationships: '(u **calls** o' means something like: "an instance of relationship **calls** relates object u and object o". In the (pre-)conditions these 'instances' can be connected with logical connectives.

Interpretation of the constraint above results in something like:

"For all objects from the classes USES, CLASS *and* OPERATIONS *for which an instance of the relationship* ***calls*** *exist between the object from class* USES *and the object from class* OPERATION*: it must be true, that when an instance of the relationship* ***to*** *exists between the object from the class* USES *and the object from the class* CLASS *then an instance of the* ***part-of*** *relationship from the object from class* OPERATION *and the class* CLASS *must exist and vice versa"*

...or in clear English:

"For all uses-relationships which call an operation: if the uses-operation points to some class, then the operation called by that uses-relationship must be part of that class.

For an introduction on **Z** refer to [Dil94].

2.2 Perspectives within a SOCCA model

Let us first identify the class at the top of the hierarchy. This class represents SOCCA models, so it is called SOCCA MODEL. An instance of this class groups all objects and relationships from one SOCCA model together and thus represents that SOCCA model. So, we have class diagram fragment 2.6 on the next page.

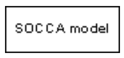


Figure 2.6: Class diagram fragment of SOCCA model

A process can be viewed and described from three perspectives. These are the data perspective, the behaviour perspective and the communication perspective. Another perspective, called the process perspective, does not play a very important role in SOCCA and so is not introduced as an explicit perspective.

Three classes are added to the class diagram fragment of figure 2.6: DATA PERSPECTIVE, BEHAVIOUR PERSPECTIVE and COMMUNICATION PERSPECTIVE. We would like to identify real world objects with all classes, but it is not very intuitive to speak of some perspective as an object. Instances of these classes are sets of diagrams from the SOCCA model. An instance of DATA PERSPECTIVE represents all the class diagrams used in the data perspective of the SOCCA model. An instance of BEHAVIOUR PERSPECTIVE represents all the internal and external STDs used in the SOCCA model. And an instance of COMMUNICATION PERSPECTIVE represents all the managers, employees and subprocesses used in the SOCCA model.

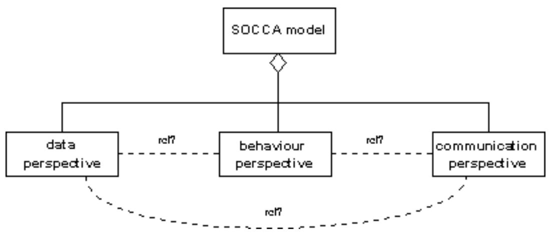


Figure 2.7: Class diagram fragment of SOCCA model

In figure 2.7 the three perspectives have been added to the class diagram. They are all part of the class SOCCA MODEL, indicating that it is possible to distinguish the three perspectives for each SOCCA model. Because it is not possible for one process to have more than one of each of the perspectives, the cardinality of the part of relationships is ‘exactly one’. So, each SOCCA model has one data perspective, one behaviour perspective and one communication perspective.

The dashed lines labeled by ‘rel?’ between the classes concerning the perspectives indicate the existence of some intuitive dependency between those classes. In this case, the perspectives are related, because they are three different viewpoints on the same model; a change in one perspective will change the other perspectives. When the class diagram is complete, these relationships will have been made explicit.

2.3 The data perspective

We now proceed with defining how the data perspective of a process is modeled in SOCCA. For now, we leave out the classes concerning other perspectives and the class SOCCA MODEL and we zoom in on the class DATA PERSPECTIVE.

In SOCCA, the data perspective of some process is modelled by a class diagram. This class diagram models the classes which are important to the modelled process and the relationships between them. An important relationship between classes in SOCCA is the uses relationship. It models which operations can be imported from one class to another. This uses relationship can be modeled in class diagrams, but because it is so important, a special type of diagram is also used.

This type is known as an import/export diagram. An import/export diagram focuses only on the classes and the uses relationships between them and is thus a special view on the class diagram.

To add this to the meta model, we need two new classes: CLASS DIAGRAM and IMPORT/EXPORT DIAGRAM. An instance of the class CLASS DIAGRAM represents a class diagram and an instance of IMPORT/EXPORT DIAGRAM represents a import/export diagram.

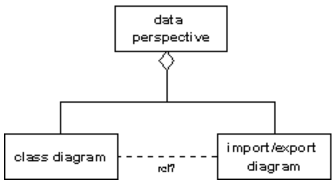


Figure 2.8: Class diagram fragment of data perspective

See the class diagram fragment in figure 2.8. We have added the class CLASS DIAGRAM and the class IMPORT/EXPORT DIAGRAM. As mentioned previously, each data perspective has one class diagram and one import/export diagram (which is a view on the class diagram). Again, a dependency exists between the class diagram and the import/export diagram for some data perspective: for a SOCCA model the import/export diagram contains the classes which are modeled in the class diagram. In section 2.4 this dependency is made explicit.

We could imagine the data perspective to be modeled by several class diagram fragments and several import/export diagram fragments. Then these fragments taken together would span up the whole class diagram. It might be useful to include this in the meta model, if such a thing is really wished for. In our opinion, the inclusion would not involve so many changes to the model.

2.3.1 A class diagram

Now, let us zoom in on the class CLASS DIAGRAM. A class diagram models classes and relationships between those classes. The first idea is to model class diagrams as in figure 2.9. The classes CLASS and RELATIONSHIP are new. The class CLASS represents classes used in class diagrams and import/export diagrams. The class RELATIONSHIP represents any kind of relationship used in a class diagram.

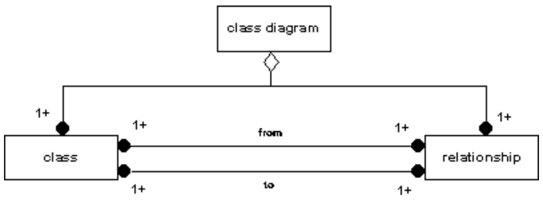


Figure 2.9: Class diagram fragment of class diagram

In the class diagram fragment in figure 2.9 the new classes have been added. Apart from these new classes, two new relationships called *from* and *to* have been added too. The class diagram fragment says the following:

A class diagram can have one or more classes and one or more relationships. An empty class diagram (without any classes) cannot be stored in this representation and is thus not seen as a valid class diagram. A relationship connects at least two classes, at least one at the 'from'-side (the side where the relationship originates) and at least one at the 'to'-side (the side where it is headed). This distinction between the 'from'- and 'to'-side is needed for inheritance and the part of relationship; if A is part of B, then B is not part of A.

As modeled here, we need to specify a direction for common relationships as well. Although this is not usual, it is more of an advantage than a limitation. Most of the time, the name of a relationship implies a direction anyway. And if a relationship involves the same class at both sides, such a direction is needed to indicate the roles of the classes.

From the cardinalities, we can also conclude that a class must have a relationship connected to it. This is because a class with no uses relationship connected to it is not useful in SOCCA.

Attributes and operations of classes are important in SOCCA. They are defined in class diagrams as well. Therefore, the classes OPERATION and ATTRIBUTE must be included.

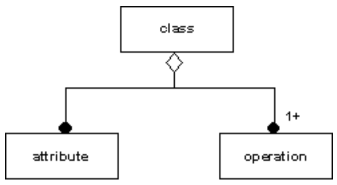


Figure 2.10: Class diagram fragment of class

In the class diagram fragment in figure 2.10, the new classes are added to the class diagram. Because operations and attributes are defined for a specific class, we have made these classes part of the class CLASS. The cardinality of the part of relationship involving the class OPERATION is 'one or more', because a class is not useful in SOCCA, if it does not specify any new operations. The cardinality of the part of relationship involving the class ATTRIBUTE is 'zero or more'. A class without any attributes can be useful in SOCCA.

Back to the class diagram fragment in figure 2.9 on the facing page. We focus now on the class RELATIONSHIP. The different types of relationships are important in SOCCA as well. We can identify four types of relationships used in the class diagrams of SOCCA. These types are the inheritance relationship, the part-of relationship, the common relationship and the uses relationship. We introduce the classes INHERITANCE RELATIONSHIP, PART-OF RELATIONSHIP, COMMON RELATIONSHIP and USES RELATIONSHIP to model them. We add them to the diagram as classes that inherit from the class relationship, because they merely are different types of relationships..

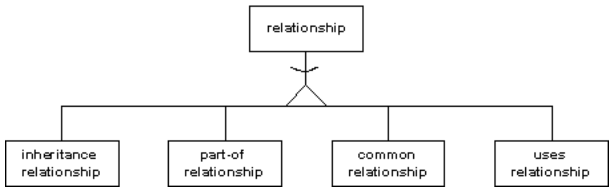


Figure 2.11: Class diagram fragment of relationship

In the class diagram fragment in figure 2.11 the four classes are added. By making the inheritance from RELATIONSHIP to the four new classes total, it is modeled that every relationship in a class diagram is either an inheritance, a part-of, a common or a uses relationship. And because the inheritance is disjoint, no relationship can be of two types simultaneously.

Now, let's put the diagram fragments concerning class diagrams in SOCCA into one complete class diagram for class diagrams. We do this by taking the class diagram fragments in figures 2.9 on the facing page, 2.10 and 2.11 together. The class diagram fragment in figure 2.12 on the next page displays the result. It shows a class diagram modeling the relevant concepts of class diagrams within SOCCA.

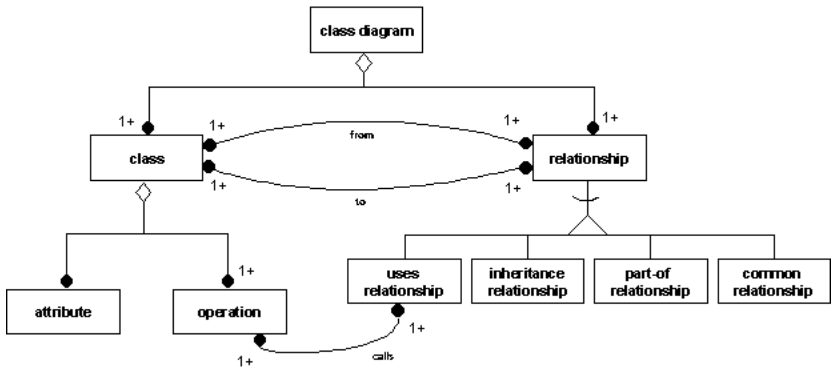


Figure 2.12: Class diagram fragment of class diagram

The relationship *calls* has been added to the class diagram fragment. This relationship is explained in the next section, as we discuss the import/export diagram and the uses relationship.

2.3.2 An import/export diagram

The next step is to focus on the class IMPORT/EXPORT DIAGRAM from the class diagram fragment in figure 2.8 on page 18 and then relate it to figure 2.12 by means of the complete class diagram for the data perspective.

An import/export diagram from a SOCCA model displays classes with their operations and attributes and the uses relationships between those classes. The classes which are displayed are exactly the classes from the class diagram from that SOCCA model. Therefore, an import/export diagram is nothing more than a limited view on the class diagram of a SOCCA model. The reason why this view is mentioned explicitly is that a lot of people like to define their uses relationships in an import/export diagram and not in the class diagram.

To model the import/export diagrams, we do not need any new classes. The classes involved are all defined in section 2.3.1, where all classes involving the class diagrams were introduced.

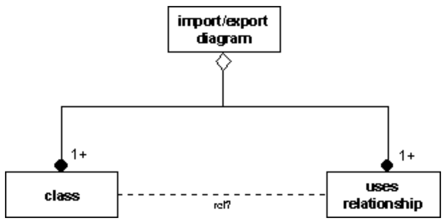


Figure 2.13: Class diagram fragment of import/export diagram

In the class diagram fragment of figure 2.13 the classes CLASS and USES RELATIONSHIP have been added. Because classes and uses relationships are part of import/export diagrams, both classes are part of the class IMPORT/EXPORT DIAGRAM. An import/export diagram contains at least one class, because a class diagram contains at least one class. An import/export diagram should also contain at least one uses relationship, because a class with no uses relationships is not useful in a SOCCA model.

A dependency exists between classes and uses relationships. However, it does not only involve the classes in the diagram fragment of figure 2.13. We also need the class OPERATION. It is included in the class diagram fragment in figure 2.14 on the facing page.

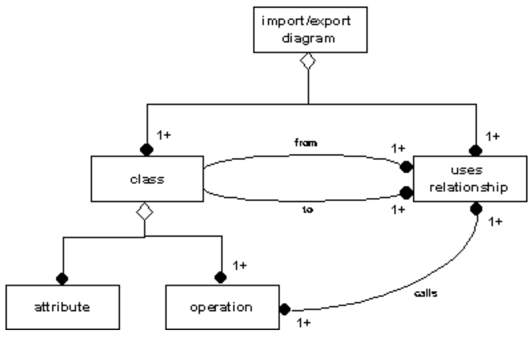


Figure 2.14: Class diagram fragment of import/export diagram

To model the dependency between classes and uses relationships we need three relationships: *from*, *to* and *calls*. The relationships *from* and *to* express the direction of the uses relationship. As the relationship *calls*, indicates which operations are called from the class to which the uses relationship is pointing. Because every useful operation of some class should be called somewhere, the cardinality of *calls* is 'one or more' at the side of USES RELATIONSHIP. The cardinality of *calls* at the side of OPERATION is 'one or more' also, because a uses relationship should always import at least one operation. The cardinalities from the relationships *from* and *to* are not surprising. Each uses relationship should originate from one class and point to another, so the cardinalities at the side of CLASS are 'exactly one'. Each class can be connected to several uses relationships and should be connected by at least one. So the cardinalities from *from* and *to* at the side of USES RELATIONSHIP are 'one or more'.

You might expect the operations to be exported from the class related to the uses relationship by *from* and imported into the class related by *to*. It is indeed the other way around. This is not as illogical as it seems. Look at figure 2.15. Class A uses the operations from class B, while the uses relationship is drawn from A to B. The class related to the uses relationship by *from* is thus not the class from which operations are called, but the class from which the relationship is drawn.

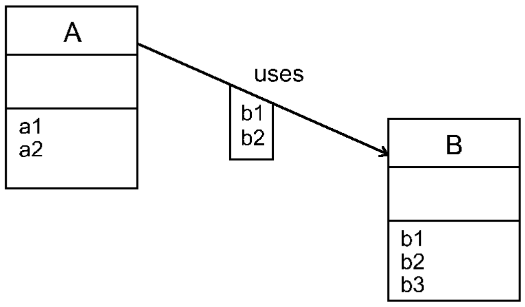


Figure 2.15: The uses relationship

In the meta model, figure 2.15 would be translated as figure 2.16 on the following page. This means: A uses B $\leftrightarrow$ A uses some of the operations of B.

There is something which we cannot conclude from figure 2.14, but which is required for the uses relationship; We need to make sure that if an operation is called by a uses relationship, it is from the class to which the relationship is pointing. This cannot be done in the class diagram and so this has to be forced by means of a constraint. Constraint 2.1 on the following page does this.

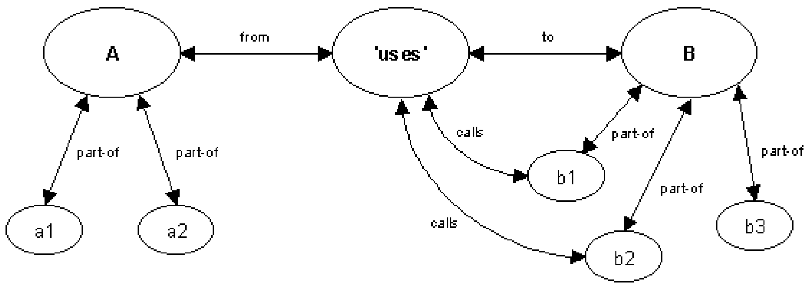


Figure 2.16: Translation of the uses relationship to the meta model

$$
\begin{aligned}
\forall u : \text{USES};\ c : \text{CLASS};\ o : \text{OPERATION} \quad & | \\
(u\ \textbf{calls}\ o)\ & \bullet \\
(u\ \textbf{to}\ c) \Leftrightarrow (o\ & \textbf{part-of}\ c)
\end{aligned}
\tag{2.1}
$$

2.4 Back to the data perspective

Now, we can compose the class diagram fragment for the whole data perspective. This is done by taking the class diagram fragments of figures 2.8 on page 18, 2.12 on page 20 and 2.14 on the page before together.

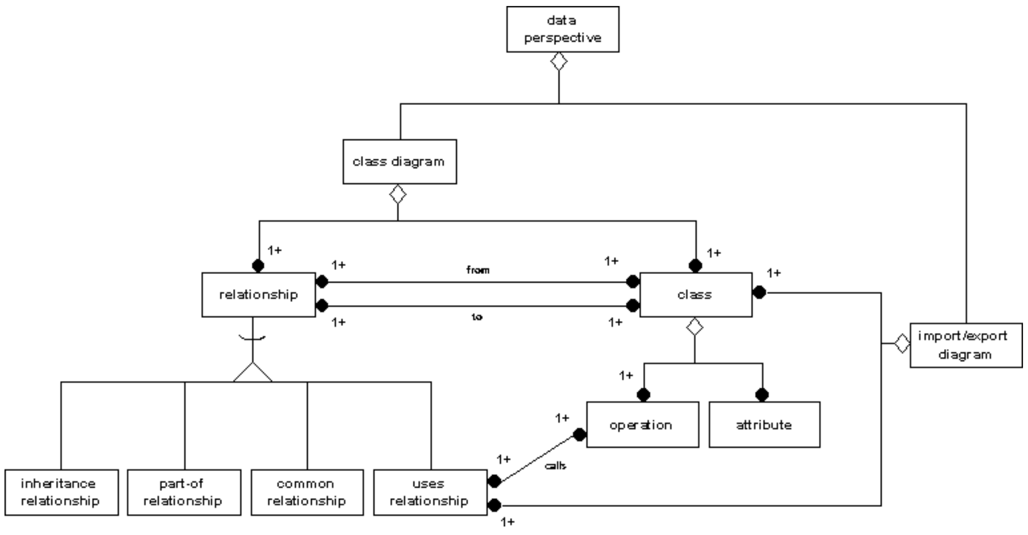


Figure 2.17: Class diagram fragment of data perspective

In the class diagram fragment in figure 2.17, the undefined relationship from figure 2.8 on page 18 has been defined. It is a relationship via the class CLASS. Intuitively: a class diagram from some SOCCA model should contain exactly the same classes as the import/export diagram from that SOCCA model. This, because the import/export diagram is only a view on the class diagram. Unfortunately, this cannot be expressed in the class diagram of the meta model. It tells us that each class is part of some class diagram and part of some import/export diagram which both are part of some data-perspective. But it cannot be concluded, that the class diagram and import/export diagram are part of the same data-perspective (of some SOCCA model). Therefore, another constraint is needed.

$$\forall cl : \text{CLASS};\ cd : \text{CLASS DIAGRAM};\ ie : \text{IMPORT/EXPORT DIAGRAM};\ \bullet$$
$$(\exists dp : \text{DATA PERSPECTIVE}\ \bullet\ (cd\ \textbf{part-of}\ dp \wedge ie\ \textbf{part-of}\ dp)) \Leftrightarrow$$
$$((cl\ \textbf{part-of}\ cd) \Leftrightarrow (cl\textbf{part-of}\ ie)) \qquad (2.2)$$

Constraint 2.2 expresses what we want. From right to left: if a class is part of both a class diagram and a import/export diagram, then a data perspective exists, from which that class diagram and import/export diagram are both part. The constraint also implies another thing. From left to right: if a data perspective exists from which both some class diagram and some import/export diagram are part, then if a class is part of that class diagram, it is part of the import/export diagram as well (and vice versa). This is the correct way to enforce that every class in some class diagram is present in the corresponding import/export diagram. This could not be extracted from the meta model as well.

Note, that on figure 2.17 on the preceding page, apart from constraint 2.2, also constraint 2.1 on the preceding page applies.

2.5 The behaviour perspective

Because the data perspective modeling has been finished, we now proceed with modeling the behaviour perspective (recall the class diagram fragment in figure 2.7 on page 17). In SOCCA this perspective is modeled with STDs. So we need to model STDs in the meta model.

A class STD is needed, which represents STDs in SOCCA. States and transitions are the important parts of STDs and thus two other classes are modeled: STATE and TRANSITION. States and transitions are both part of STDs. In order to get a valid STD, each transition needs to be connected to a state on either side.

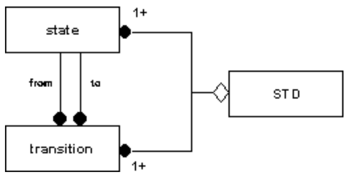


Figure 2.18: Class diagram fragment of STD

See the class diagram fragment of figure 2.18. It shows a way to model STDs in a class diagram. A STD in SOCCA should have at least one state and at least one transition too. Each type of STD in SOCCA requires at least one transition (and thus one state) to exist. Two relationships, called *from* and *to* have been added to model that a transition originates from one state and ends at one state. Each state can have several transitions going in and out.

In SOCCA, there are two types of STDs which model the behaviour: external and internal STDs. External STDs model the behaviour of classes and the internal STDs model the behaviour of operations. Every class should be modeled by at least one external STD and every operation by at least one operation. The behaviour perspective consists of these two types of STDs. The classes EXTERNAL STD and INTERNAL STD will represent external STDs and internal STDs, respectively.

See the class diagram fragment of figure 2.19 on the following page. The two new classes have been added as classes which are part of the class BEHAVIOUR PERSPECTIVE and inherit from STD. Because there exists at least one class in the SOCCA model (at least one class is required in the class diagram), at least one external STD should exist as well. In that one class, at least one operation should exist. So at least one internal STD should exist as well. This is seen in the cardinalities of the part-of relationship.

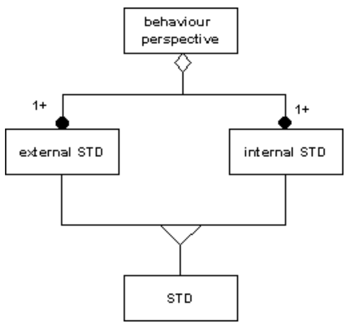


Figure 2.19: Class diagram fragment of behaviour STD

Because internal and external STDs are not the only STDs used in SOCCA and the other STDs are not part of the behaviour perspective, we cannot make the class STD part of BEHAVIOUR PERSPECTIVE.

2.6 Linking the behaviour perspective to the data perspective

We have both modeled the data and the behaviour perspective in SOCCA. As implied in the previous section, these two perspectives are linked, because external STDs model the classes from the data perspective and the internal STDs model the operations. To link the two perspectives we need to know exactly what the relationships between the two perspectives are.

External STDs model the behaviour of classes by letting the transitions refer to operations of that class. That way only sequences of operations specified by the external STD are allowed. So, apart from relationships between external STDs and classes, relationships between a transitions of external STDs and operations exist too. These relationships are modeled in the class diagram fragment in figure 2.20.

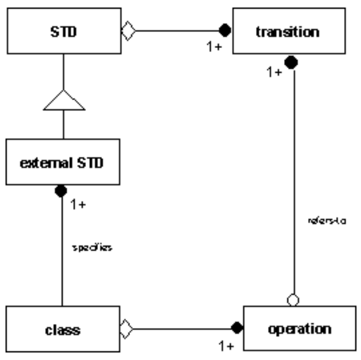


Figure 2.20: Class diagram fragment of linking the two perspectives

In the class diagram fragment in figure 2.20 the new relationships have been added.[1] The first called *specifies* denotes which external STDs specify the class. Multiple external STDs can together

[1]Only the relevant classes for external STDs are modeled.

specify one class and every external STD specifies a class. This is modeled by the cardinalities of the relationship.

The other relationship, called *refers-to*, indicates which transition of an external STD refers to which operation. A transition of an external STD does not have to refer to an operation, but each operation should be referred to by a transition of the external STD which models its class. For a small part this is modelled by the cardinalities. But not all can be modeled in the class diagram fragment. There is no way of modelling that an operation has to be referred to by some transition of the external STD which models its class. Or the other way around: If an operation is referred to by some transition of an external STD modelling a class, there is no way to model that the operation is part of the modeled class. These two requirements are captured by constraint 2.3.

$$\begin{aligned}&\forall o : \text{OPERATION};\ \ c : \text{CLASS} \bullet \\ &\qquad (o\ \textbf{part-of}\ c) \Leftrightarrow \exists es : \text{EXTERNAL STD};\ \ t : \text{TRANSITION} \bullet \\ &\qquad\qquad t\ \textbf{part-of}\ es \wedge es\ \textbf{specifies}\ c \wedge t\ \textbf{refers-to}\ o\end{aligned} \tag{2.3}$$

Besides the dependencies between the two perspectives involving the external STDs, there are also dependencies involving the internal STDs. Internal STDs model the behaviour of operations. This by prescribing specific orders of doing internal operations[2] and calling other operations. Transitions in internal STDs can thus refer to some internal operation (not modeled) or to some other operation. They also could refer to something called an act-operation. This is an operation that initiates the operation itself. It reflects some communication with the external STD which is not modeled here.

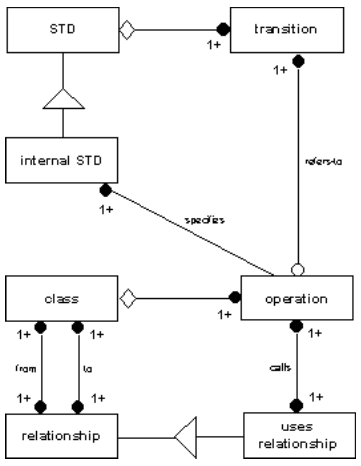


Figure 2.21: Class diagram fragment of linking the two perspectives

In the class diagram fragment in figure 2.21, the two dependencies are added as two relationships: *specifies* and *refers-to*. The cardinalities are like with the external STD: an operation can be modeled by multiple internal STDs and every internal STD models (part of) exactly one operation. A transition of an internal STD can refer to an operation, but does not have to. In order to be useful an operation should at least be called once (like with the uses relationship).

The classes CLASS, RELATIONSHIP and USES RELATIONSHIP are included for the following reason: Only operations available for calling can be called. Which operations are available is defined by uses relationships in the data perspective. In the class diagram there is no way to enforce this, so another constraint is needed.

[2]These internal operations are internal to the operations themselves.

$$\begin{aligned}
&\forall o_1, o_2 : \text{OPERATION};\ c_1, c_2 : \text{CLASS} \mid (o_1 \ \mathbf{part\text{-}of}\ c_1 \wedge o_2 \ \mathbf{part\text{-}of}\ c_2) \bullet \\
&\quad (\exists u : \text{USES RELATIONSHIP} \bullet (u \ \mathbf{from}\ c_1 \wedge u \ \mathbf{to}\ c_2 \wedge u \ \mathbf{imports}\ o_2) \Leftrightarrow \\
&\quad\quad (\exists is : \text{INTERNAL STD};\ t : \text{TRANSITION} \bullet \\
&\quad\quad\quad (is \ \mathbf{specifies}\ o_1 \wedge ct \ \mathbf{part\text{-}of}\ is \wedge t \ \mathbf{refers\text{-}to}\ o_2)))
\end{aligned} \tag{2.4}$$

Constraint 2.4 states that if there exists a uses relationship, that calls some operation of a class, then there exists a transition in an internal STD, that models an operation from that class, which refers to that operation (and vice versa).

You might have noticed, that there exist two relationships called *refers-to* from TRANSITION to OPERATION. This is not a problem, as both are just references to some operation. For clarity, it could be useful to rename both relationships, but as both have compatible cardinalities this is not necessary and both can be combined in one relationship, which only denotes a reference from a transition in a STD to an operation.

The classdiagram fragment in figure 2.22 shows the whole data and behaviour perspective linked to each other.

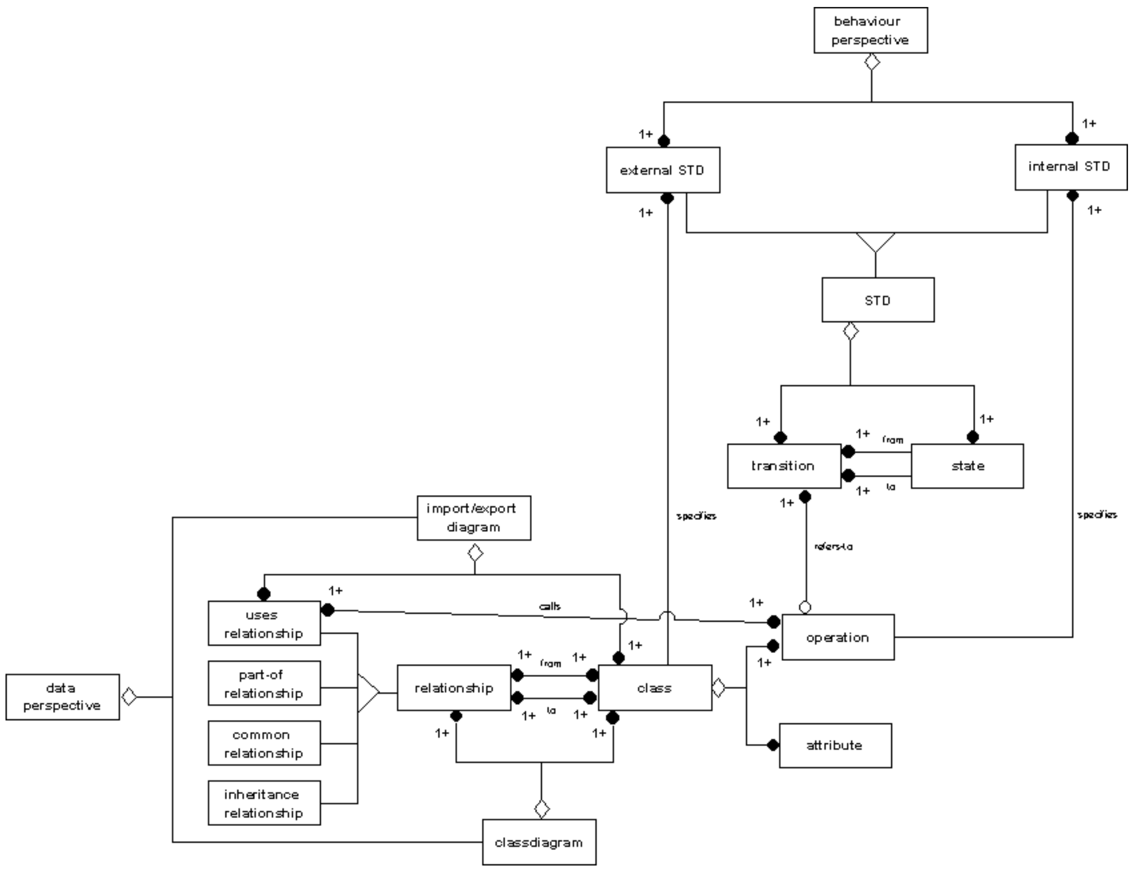


Figure 2.22: Class diagram fragment of linking the two perspectives

2.7 The communication perspective

The last perspective left to model is the communication perspective. Communication in SOCCA is modeled with PARADIGM on top of the STDs of the behaviour perspective. In PARADIGM, there are three types of STDs which are important: managers, employees and subprocesses. Although subprocesses are not really STDs, but rather STDs with some added functionality (they contain 'traps'), we will model them as STDs anyway.

We introduce the classes MANAGER, EMPLOYEE and SUBPROCESS as classes that inherit from STD and are part of COMMUNICATION PERSPECTIVE. This is done in the class diagram fragment in figure 2.23.[3]

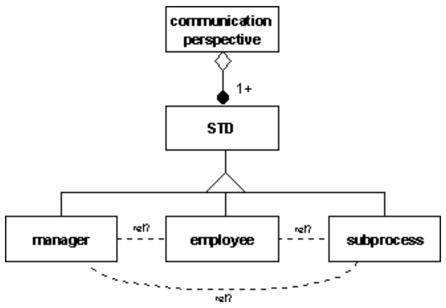


Figure 2.23: Class diagram fragment of the communication perspective

The dashed relationships in the class diagram fragment in figure 2.23 express the existence of a dependency between the three classes. This dependency is the whole idea PARADIGM is based on. It is best described by the following four steps taken from [EG94]:

1. The sequential behaviour of each process is described by an STD.
2. Within each STD, significant subdiagrams (called subprocesses) with respect to coordination with other processes (also described by STDs) are identified.
3. Within each subprocess, (sets of) states, or so-called traps, are identified. These describe situations where an object is ready to switch from one subprocess to another.
4. The possible transitions between subprocesses of all objects, the behaviours of which have to be coordinated, are once more described by a STD, or a so-called manager process.

PARADIGM also supplies us with the concept of a partition. A partition is a set of subprocesses for one employee which are managed by one manager. It is useful to use this concept in SOCCA as well and we will model it in the meta model.

The class diagram fragment in figure 2.24 on the following page defines the relationships between the PARADIGM concepts as accurate as possible in a class diagram. This fragment is used as the communication perspective in the meta model. A class PARTITION must be introduced, as this has not yet been done. This class represents the SOCCA concept of a partition.

In the class diagram fragment of figure 2.24 on the next page, the relationship *manages* denotes the described dependency between a manager, an employee and the partitions. The cardinality of this relationship is 'one or more' for each side. A manager can manage multiple employees and thus multiple partitions. An employee can be managed by multiple managers. For every partition, there is only one employee. This cannot be extracted from the class diagram, so we need a constraint. This constraint is given in constraint 2.5.

$$\forall p : \text{PARTITION};\ \exists_1 e : \text{EMPLOYEE} \bullet e\ \mathbf{manages}\ p \qquad (2.5)$$

[3]The cardinalities and the disjoint inheritance in that class diagram fragment will be explained later.

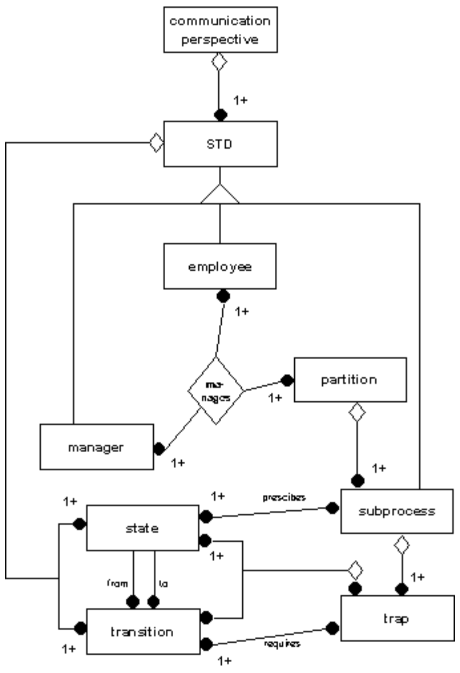


Figure 2.24: Class diagram fragment of communication perspective

Two other relationships in the class diagram fragment of figure 2.24 are *requires* and *prescribes*. The relationship *prescribes* specifies which subprocesses are prescribed by a state of the manager. Each subprocess should be prescribed at least once, thus the cardinality of *prescribes* at the side of STATE is 'one or more'. If a subprocess is not prescribed at least once, then it is useless and it should not be in the SOCCA model. Each state of a manager should prescribe as many subprocesses as there are employees managed by the manager containing that state. To be exact, for every partition/employee couple managed by the manager, one subprocess is prescribed from that partition for each state of that manager. To describe this, constraints 2.7 and 2.8 are included. Although at least one subprocess is prescribed by each state of a manager, not every STD is a manager. And thus every state is not a manager state. Therefore, the cardinality of *prescribes* at the side of SUBPROCESS is 'zero or more'. Constraint 2.6 is needed to enforce that every state of a manager prescribes at least one subprocess.

$$\begin{aligned}&\forall m : \text{MANAGER};\ s : \text{STATE} \mid \\ &\qquad s\ \textbf{part-of}\ m \bullet (\exists sp : \text{SUBPROCESS} \bullet s\ \textbf{prescribes}\ sp)\end{aligned} \tag{2.6}$$

$$\begin{aligned}&\forall s : \text{STATE};\ m : \text{MANAGER};\ p : \text{PARTITION} \mid \\ &\qquad (s\ \textbf{part-of}\ m \wedge m\ \textbf{manages}\ p) \bullet \\ &\qquad\quad \exists_1 sp : \text{SUBPROCESS}(sp\ \textbf{part-of}\ p \wedge s\ \textbf{prescribes}\ sp)\end{aligned} \tag{2.7}$$

$$\begin{aligned}&\forall s : \text{STATE};\ m : \text{MANAGER};\ sp : \text{SUBPROCESS} \mid \\ &\qquad (s\ \textbf{part-of}\ m \wedge s\ \textbf{prescribes}\ sp) \bullet \\ &\qquad\quad \exists p : \text{PARTITION}(m\ \textbf{manages}\ p \wedge sp\ \textbf{part-of}\ p)\end{aligned} \tag{2.8}$$

The relationship *requires* specifies which traps are required by which transitions. Every trap defined has to be required at some transition of a manager (else it is useless), so the cardinality on the side of TRANSITION is 'one or more'. Every transition of a manager has to require at least one trap, but because not all transitions are transitions of a manager the cardinality of *requires* at the side of TRAP is 'zero or more'. Constraint 2.9 states the every transition of a manager requires at least one trap. Constraint 2.10 has been included to make sure that every trap required by some transition, is a trap from a subprocess which was prescribed in the state from where that transition originates.

$$\forall m : \text{MANAGER};\ t : \text{STATE};\ | \quad t\ \textbf{part-of}\ m \bullet (\exists tp : \text{TRAP} \bullet t\ \textbf{requires}\ tp) \tag{2.9}$$

$$\forall s : \text{STATE};\ t : \text{TRANSITION};\ tp : \text{TRAP}\ | \quad (t\ \textbf{from}\ s \wedge t\ \textbf{requires}\ tp) \bullet \exists_1 sp : \text{SUBPROCESS}(tp\ \textbf{part-of}\ sp \wedge s\ \textbf{prescribes}\ sp) \quad) \tag{2.10}$$

Now, we want to make clear that subprocesses are specific views on employees with traps added to them. The new constraints 2.11 and 2.12 say that every state or transition from some subprocess is a state or transition of the associated employee.

$$\forall sp : \text{SUBPROCESS};\ p : \text{PARTITION};\ e : \text{EMPLOYEE};\ s : \text{STATE}\ | \quad (p\ \textbf{manages}\ e \wedge sp\ \textbf{part-of}\ p \wedge s\ \textbf{part-of}\ sp) \bullet (s\ \textbf{part-of}\ e) \tag{2.11}$$

$$\forall sp : \text{SUBPROCESS};\ p : \text{PARTITION};\ e : \text{EMPLOYEE};\ t : \text{TRANSITION}\ | \quad (p\ \textbf{manages}\ e \wedge sp\ \textbf{part-of}\ p \wedge t\ \textbf{part-of}\ sp) \bullet (t\ \textbf{part-of}\ e) \tag{2.12}$$

2.8 Linking the communication perspective to the other perspectives

The communication perspective now needs to be linked to the other perspectives. As mentioned in the beginning of 2.7 the communication in SOCCA is modeled by PARADIGM on top of the STDs of the behaviour perspective. So we need to link the concepts of PARADIGM to the STDs of the behaviour perspective. As is stated in [EG94], the external STDs serve as useful managers and internal STDs serve as useful employees. Therefore we need to identify the two concepts.

In the meta model this corresponds to identifying the classes EXTERNAL STD and INTERNAL STD with the classes MANAGER and EMPLOYEE respectively. This however leads to a problem. In SOCCA, employees which are not really internal STDs must be allowed to exist. Take a look at the following example:

A class can have several external STDs specifying it. Each of these external STDs corresponds to a manager. However, there is a need for a manager to manage these managers for the class. This manager is then also an external STD specifying the class. The managers that are managed by this manager are really employees of this manager. These employees do not correspond to an internal STD.

Likewise, an example in which an operation would have multiple internal STDs can be constructed. In this case, the employees can be managers to (such a manager manages the different employees for the operation). Such a manager does not correspond to an external STD.

So the classes from both perspectives cannot be identified, although most of the time they represent the same concepts. The solution is to let the classes EXTERNAL STD and INTERNAL STD inherit from the classes MANAGER and EMPLOYEE respectively. Both inheritance relationships must not be total. That way, an external STD is always a manager, but a manager not always an external STD. And an internal STD is always an employee, but an employee not always an internal STD.

The classdiagram fragment in figure 2.25 shows the communication perspective and the behaviour perspective linked together.

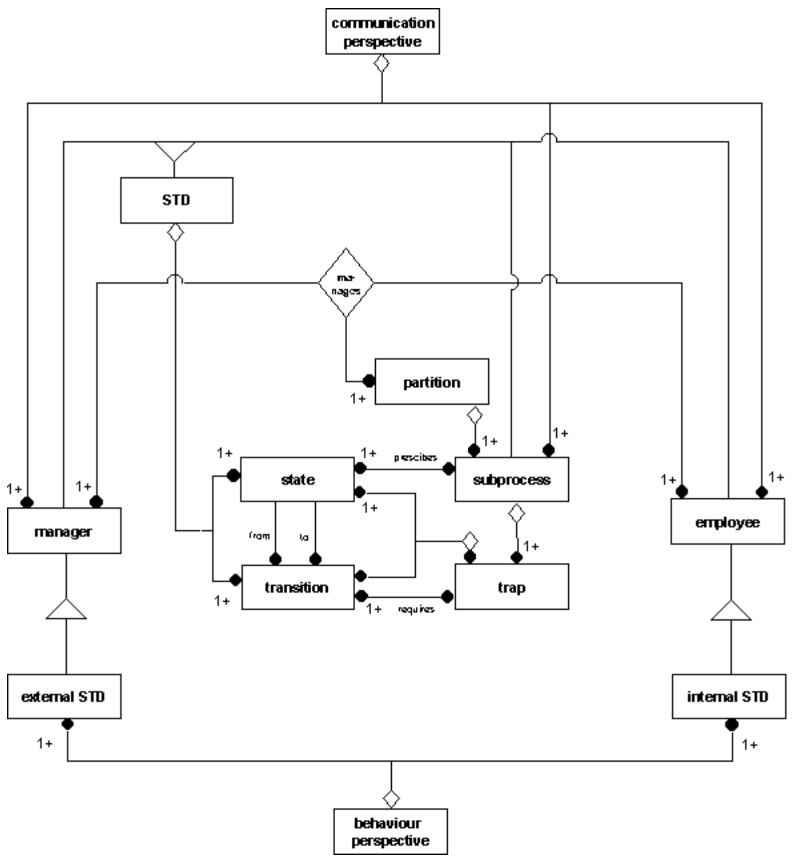


Figure 2.25: Classdiagram fragment of linking the two perspectives.

Because the behaviour perspective is linked to the data perspective, the communication is indirectly linked to the data perspective as well. This does not have to be done separately.

2.9 The complete meta model

As we have linked the perspectives, the complete meta model has been finished. This meta model also includes all constraints introduced in this chapter. Figure 2.26 on the facing page displays the whole meta model. All constraints mentioned in this chapter apply to this meta model.

For a two page version of the meta model see appendix A. The constraints are repeated there too.

2.10 Conclusion.

This concludes the creation of the class diagram for SOCCA models. In the next chapter, this class diagram (and its constraints) will be used to extract demands on the way a SOCCA model is built. The class diagram itself and its constraints already tell what a SOCCA model should

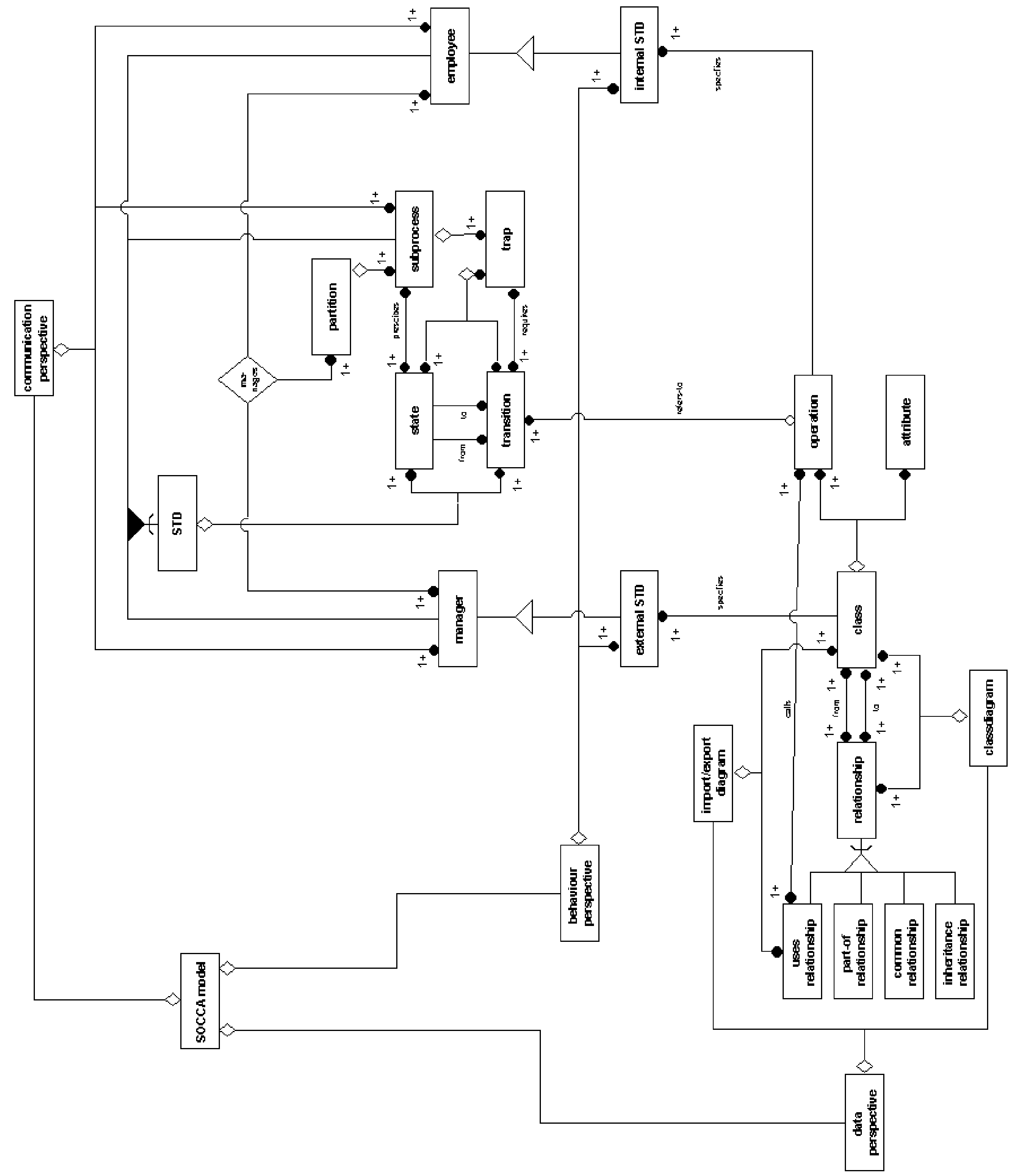


Figure 2.26: The complete class diagram.

consist of, if it should be correct. However, a SOCCA model in construction will seldom meet the demands forced on it by this class diagram. Therefore, in the next chapter, the demands from this chapter will be weakened somewhat.

Chapter 3

Analysing the meta model

3.1 Introduction

The meta model created in the previous chapter was created to formalize the dependencies between the diagrams used in SOCCA. The result is a meta model which can be used to check whether some collection of diagrams is a syntactically correct SOCCA model or not: if a SOCCA model is syntactically correct, then it 'fits' in the meta model.

Besides being a formalization of the dependencies in SOCCA models, the meta model can also be used as a basis for the internal representation and the database. Such an internal representation needs to be build and maintained a system component, usually called an editor.

This chapter analyses the meta model and splits it up in a natural way. This split results in smaller pieces of meta model with dependencies between these pieces. Therefore, it also leads to smaller editors responsible for smaller fragments of the internal representation and editors responsible for the dependencies between these fragments. These editors are identified and described in this chapter too. Besides these editors, other components of the future system are introduced as well. These components cannot be identified with a part of the meta model (like the editors). They are introduced as components which are necessary to examine the internal representations/database and use their contents.

Although the meta model is a useful basis for the internal representation and the database, there are some notions which were not important in the meta model, but are in the internal representation/database. These are discussed in section 3.6.

The functionality of the editors and other components is described only a little in this chapter. The exact functionality of the system (and thus its components) is described by the use cases in the following chapters. Therefore, the use cases are identified with the system components in the last section of this chapter.

3.2 Internal representation and the database

If we examine the needs for the future system, then the first thing we come across is the need for a way to represent (parts of) SOCCA models in a data structure. Such a data structure is called an internal representation. The meta model can serve as the basis for this internal representation, because it contains all needed and useful concepts within SOCCA.

Also needed in the system is a place to store the (parts of) SOCCA models for indefinite time. A place to store data is usually called a database, so we will call our place to store models a database as well. The internal representation serves as a useful 'blueprint' for this database, because it must be possible for the system to get information from the database and convert it to the internal representation. If the structures of the database and the internal representation are largely the same, there is no need for difficult conversion steps.

For now, we will assume that the meta model is also a model for the internal representation and the database. In reality, at least some additions to the meta model are needed in order for it to be a useful internal representation and/or database (see section 3.6). But, for the identification of the useful editors we do not need this.

3.3 Splitting up the meta model

The meta model from the previous chapter is thus also the model for the internal representation of SOCCA models for the system. For creating and editing SOCCA models we need editors. These editors translate the diagrams drawn by the users to a SOCCA model in the internal representation. These SOCCA models will later be stored in the database.

The meta model is quite large. Therefore, we are going to split it up in smaller fragments, with these fragments representing concepts which are more or less independent of the concepts represented in other fragments. For these fragments we can use separate editors. Each of these editors is only responsible for a fragment of the meta model. These editors are easier to define and build than an editor, which is responsible for the complete meta model. Sections 3.3.1 and 3.3.2 present useful split ups of the meta model which will be used to define the system.

By splitting up the meta model in fragments, some relationships and constraints, that are connecting these fragments in the original meta model, come to disappear. Although these relationships and constraints are not visible to the fragments, they still need to be edited and maintained by some part of the system. Section 3.3.3 discusses this.

3.3.1 Split up in diagram types

A SOCCA model models a process by means of various types of diagrams. It is possible to split up any SOCCA model in diagrams according to these diagram types. The basic diagram types which SOCCA uses, are class diagrams and STDs. Furthermore, because SOCCA uses PARADIGM on top of its STDs for specifying the communication, a special kind of STD called subprocess is used as well. No other diagram types are used in SOCCA. Because import/export diagrams are only views on class diagrams and add no special graphical constructs, they are not a diagram type. The STDs used in SOCCA regarding PARADIGM, thus the internal, external, employee and manager STDs are no separate diagram types as well. Although used for different purposes, these kinds of STDs only represent a special view on 'normal' STDs. The graphical constructs used in those views are the same as those used in 'normal' STDs. A partition is no diagram type as well, but only a special way of looking at a special group of subprocesses, thus a view.

If a SOCCA model is split up in diagrams of different diagram types, then the editor which is responsible for the internal representation can be split up in smaller editors. Each of the smaller editors is then responsible for one diagram type and thus for only a small part of the internal representation. So, the internal representation must be split up in parts corresponding to the different diagram types. The meta model serves as a basis for the internal representation, therefore we will split up the meta model according to these parts.

Let us first introduce some classification of the classes used in the meta model. This to make the discussion in this chapter a little easier. The meta model contains three types of classes: classes which represent a diagram, classes which represent building blocks of a diagram and classes which represent groups of diagrams. We will call these classes: diagram classes, building block classes and grouping classes, respectively.

The fragments of the meta model corresponding to the diagram types are constructed in the following way. First, the diagram class representing the diagram type is included. For each diagram type, there exist building blocks. These building blocks are represented by building block classes in the meta model. These building block classes are included in the fragment as well. All the relationships which existed in the meta model between the included classes are then made part of the fragment as well. These relationships are relationships which are involved with the specific diagram type only. Therefore, we will call them intra-diagram relationships.

Class diagrams

The first basic diagram type we want to discuss is the class diagram. The meta model class representing the class diagrams is the diagram class CLASS DIAGRAM. The construction process, described earlier, results in the fragment of the meta model given in figure 3.1.

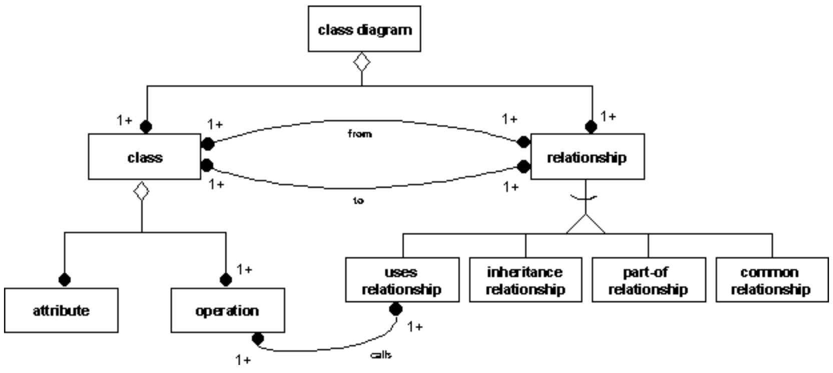


Figure 3.1: The meta model for class diagrams

The building block classes and intra-diagram relationships for class diagrams are the meta model classes (apart from the diagram class CLASS DIAGRAM) and relationships in the fragment of figure 3.1, they are listed in the table 3.1 too.

building block class
CLASS
ATTRIBUTE
OPERATION
RELATIONSHIP
USES RELATIONSHIP
INHERITANCE RELATIONSHIP
PART-OF RELATIONSHIP
COMMON RELATIONSHIP

relationship	class	class
part-of	CLASS DIAGRAM	CLASS
part-of	CLASS DIAGRAM	RELATIONSHIP
part-of	CLASS DIAGRAM	ATTRIBUTE
part-of	CLASS DIAGRAM	OPERATION
from	CLASS	RELATIONSHIP
to	CLASS	RELATIONSHIP
inheritance	RELATIONSHIP	USES RELATIONSHIP
inheritance	RELATIONSHIP	INHERITANCE RELATIONSHIP
inheritance	RELATIONSHIP	PART-OF RELATIONSHIP
inheritance	RELATIONSHIP	COMMON RELATIONSHIP
calls	USES RELATIONSHIP	OPERATION

Table 3.1: The building block classes and intra-diagram relationships for class diagrams

Because import/export diagrams are only a special view on class diagrams, we have modeled them as such in the meta model. Although import/export diagrams are not really a special type of diagram, we can identify the fragment of the meta model which models the import/export diagrams just as we did with the class diagrams. Figure 3.2 on the following page shows this fragment. The meta model class representing the import/export diagrams is the diagram class IMPORT/EXPORT DIAGRAM.

Note, that in the meta model the class RELATIONSHIP appears between the meta model class USES RELATIONSHIP and the meta model class CLASS. Because the uses relationship is the only type of relationship existing in the import/export diagrams, this generalization has been left out. In the meta model the class USES RELATIONSHIP inherits from RELATIONSHIP. Therefore, in the

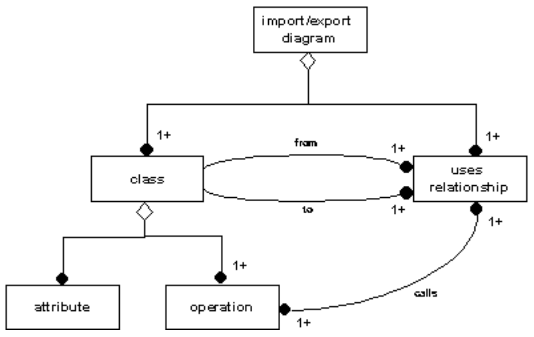


Figure 3.2: The meta model for import/export diagrams

fragment of figure 3.2, the class USES RELATIONSHIP has inherited the meta model relationships *from* and *to* as well.

The building block classes and intra-diagram relationships for import/export diagrams are given in the table 3.2.

building block class
CLASS
USES RELATIONSHIP
OPERATION
ATTRIBUTE

relationship	**class**	**class**
part-of	IMPORT/EXPORT DIAGRAM	CLASS
part-of	IMPORT/EXPORT DIAGRAM	USES RELATIONSHIP
part-of	CLASS	OPERATION
part-of	CLASS	ATTRIBUTE
from	USES RELATIONSHIP	CLASS
to	USES RELATIONSHIP	CLASS
calls	USES RELATIONSHIP	OPERATION

Table 3.2: The building block classes and intra-diagram relationships for import/export diagrams.

STDs

STDs are the other basic type of diagrams used in SOCCA. Figure 3.3 shows the fragment of the meta model which is responsible for modelling STDs. It exists of three classes and four relationships of which only two classes and two relationships represent the building blocks and the relationships between them. The small number of classes and relationships involved is a result of the simple structure of a STD.

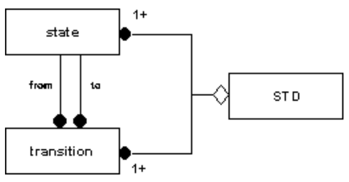


Figure 3.3: The meta model for STDs

The diagram class representing STDs in the meta model is the class STD. The building block classes and intra-diagram relationships for STDs are listed in the table 3.3.

In SOCCA, STDs are used for modelling the behaviour and the communication perspective. For this purpose, several kinds of STDs with different functionality have been introduced: external

building block class
STATE
TRANSITION

relationship	class	class
part-of	STD	STATE
part-of	STD	TRANSITION
from	TRANSITION	STATE
to	TRANSITION	STATE

Table 3.3: The building block classes for STDs

STDs, internal STDs, managers and employees. Although the transitions and states of these STDs have a different meaning for each kind, the syntax of diagrams are the same: they are normal STDs. Therefore, the diagram classes corresponding with these kinds of STDs inherit from the meta model class STD. These diagram classes are: EXTERNAL STD, INTERNAL STD, MANAGER and EMPLOYEE. The fragment of the meta model in figure 3.4 shows the inheritance. This fragment is responsible for modelling all the kinds of STDs.

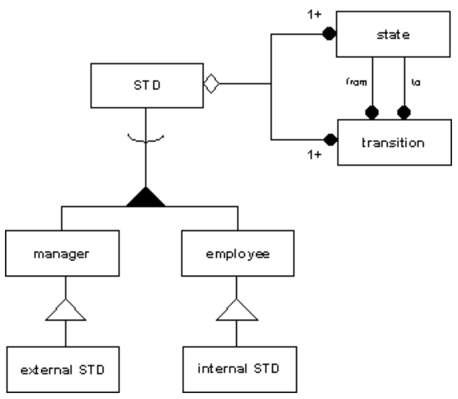


Figure 3.4: The meta model for all kinds of STDs

The meta model fragment in figure 3.4 shows that every special kind of STD is a STD. Therefore, the part-of relationships from the meta model class STD to its building block classes are also inherited by the diagram classes representing the special kinds of STDs. These STDs thus also have states and transitions part of them. Therefore, the building block classes for STDs are building block classes for every kind of STD as well. The same is true for the intra-diagram relationships.

Subprocesses

In the meta model, the class SUBPROCESS inherits from STD too. Therefore, subprocesses contain states and transitions as well. The reason why subprocesses were not discussed in the previous section is that a special construct is used in the syntax of subprocesses. This construct is called a trap and is not part of the general syntax of STDs.

Figure 3.5 displays the meta model fragment which is responsible for modelling the subprocesses. Like the diagram classes modelling the different kinds of STDs, the class SUBPROCESS also inherits from STD. Therefore, the building block classes and intra-diagram relationships for subprocesses include those of the STDs. However, the meta model class SUBPROCESS also has the class TRAP part of it. This class is a building block class too. There are also more intra-diagram relationships. The part-of relationship from SUBPROCESS to TRAP for example.

In the table 3.4 all building block classes and intra-diagram relationships for subprocesses are listed.

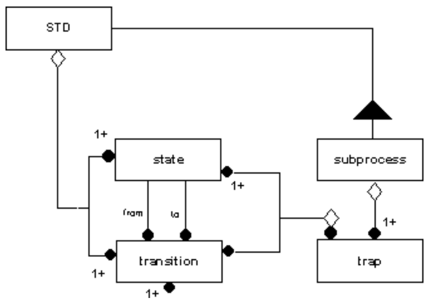


Figure 3.5: The meta model for subprocesses

building block class
STATE
TRANSITION
TRAP

relationship	class	class
part-of	STD	STATE
part-of	STD	TRANSITION
part-of	SUBPROCESS	TRAP
from	TRANSITION	STATE
to	TRANSITION	STATE
part-of	TRAP	STATE
part-of	TRAP	TRANSITION

Table 3.4: The building block classes and intra-diagram relationships for subprocesses.

3.3.2 Split up in groups of diagrams

Instead of splitting a SOCCA model in diagrams of the different diagram types, it is also possible to split it in groups of diagrams. Such a split up is only useful, if it is useful to group some diagrams together. We can think of two different purposes of grouping diagrams together:

The first purpose for making a special group of diagrams is to focus on a limited parts of the model. Such a focus could be on some special class in the model or on some special behaviour, for example. For this purpose, the group of diagrams must be more or less 'complete'. Complete in the sense, that the group of diagrams is a small SOCCA model on its own: it is possible to simulate and analyse it. Therefore, a group of diagrams for this purpose should contain diagrams from all three perspectives.

Because in a SOCCA model all diagrams are related, taking a special group of diagrams from it will result in some dependencies not being satisfied in that group. Simply, because some related diagrams are not included in it. While analysing or simulating such a group, these dependencies should be disregarded or at least not result in errors.

In the meta model as it is now, there is no room for these kind of groups yet.

The other purpose for grouping diagrams is to catalogue diagrams in groups. A few examples are: grouping diagrams according to SOCCA model, grouping diagrams according to author, grouping diagrams according to class, etc. Because the user will define these groups, any criteria will do. For this purpose, the groups do not have to contain diagrams of all of the three perspectives and thus will not always be useful to analyse or simulate.

In the meta model, some of these 'cataloguing' groups can already be distinguished. These groups are:

- The SOCCA model group type, grouping diagrams together which belong to one SOCCA model.

- The data perspective group type, grouping diagrams together which belong to the data perspective of one SOCCA model.
- The behaviour perspective group type, grouping diagrams together which belong to the behaviour perspective of one SOCCA model.
- The communication perspective group type, grouping diagrams together which belong to the communication perspective of one SOCCA model.
- The partition group type, grouping subprocesses together to partitions.

We will call the classes and relationships associated with these groups grouping classes and grouping relationships, respectively. These types of groups are special in the way, that they can only contain a few types of diagrams and sometimes also only a limited number of them. This is shown in figure 3.6.

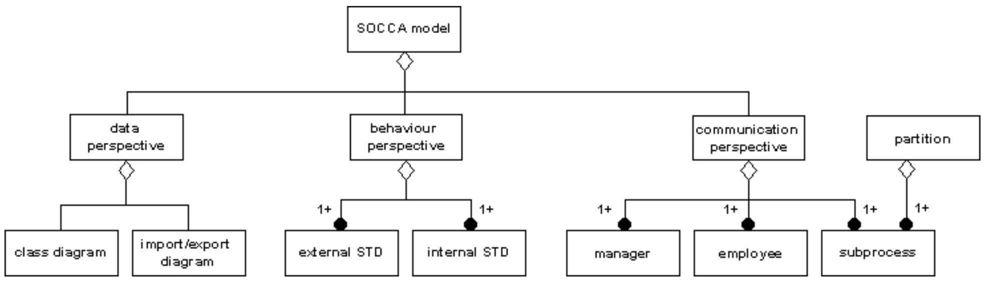


Figure 3.6: The grouping classes, diagram classes and the grouping relationships

Although presented as two different purposes for grouping, there seems to be some overlap: what if the user wishes to catalogue the diagrams according to special focuses on the SOCCA model? Or how about defining other catalogues then already presented in the meta model?

It seems useful to allow the user to group diagrams in every way desired. In that light, the groups belonging to the two purposes are only two possible ways to group diagrams. By allowing every kind of group to exist, the number of ways to group diagrams is infinite. It must be possible to analyse these general groups. Simulation must also be possible, but only with group which can be simulated.

In figure 3.7, a small example of possible groups is given.

In figure 3.7, the small filled squares represent the diagrams used in some SOCCA model. The ellipses, rounded squares and circle containing the diagrams represent examples of groups of diagrams. These groups can overlap and contain each other, which means that some diagrams are part of more than one group. Note, that in this figure only one SOCCA model (the large circle) can be seen. In the system, several SOCCA models coexist in the database and diagrams can be part of several SOCCA models as well.

In this figure, it might be possible to simulate group I and II, as they contain diagrams of all of the three perspectives. This of course, is also true for the SOCCA model group represented by the circle. Group III might be a class diagram with an external STD for one class of that class diagram and group IV might be a group of diagrams which have been made by one specific author.

As mentioned above, apart from grouping diagrams according to perspective and grouping subprocesses in partitions, there is no way to represent these groups of diagrams in the meta model as it is now. However, there should be a component of the system which is responsible for creating and maintaining groups of diagrams. For now, this component would only be responsible for the grouping classes and grouping relationships. More discussion on the groups and the limitation of the current meta model is given in sections 3.4.2 and 3.6.2.

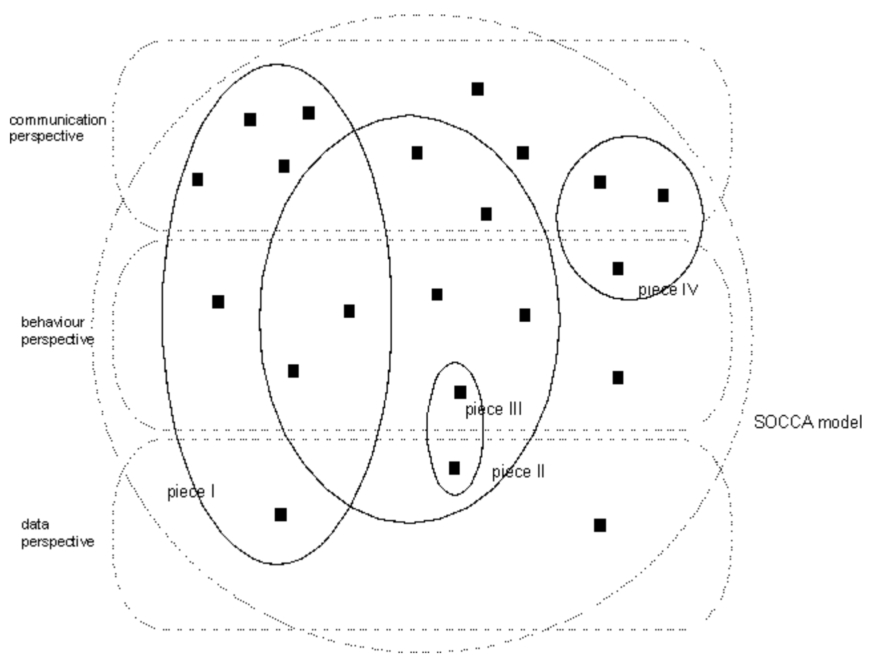


Figure 3.7: Splitting up a SOCCA model in useful pieces

3.3.3 Dependencies between the model pieces

If we split up a SOCCA model in smaller pieces (like diagrams or groups of diagrams), we disregard some dependencies between these pieces. The dependencies which are disregarded, are those which cross the border of the pieces: the two parts of model dependent on each other are part of different pieces of the model.

As each SOCCA model is modeled by the meta model, each possible dependency is modeled in this meta model as well. If we split up the SOCCA model in its diagram types (as in subsection 3.3.1), the relationships from the meta model presented in table 3.5 are disregarded as they do not appear in any of the meta model fragments presented in section 3.3.1. These relationships represent dependencies between different diagrams and therefore do not fit in a meta model fragment which is responsible for only one diagram type. Because we will reference to these relationships further in this thesis, we will give these relationships a specific name: inter-diagram relationships. Other dependencies between two diagrams are represented by the constraints belonging to the meta model. These constraints are listed in appendix A.

If a split up is made in groups of diagrams (as in subsection 3.3.2), again certain similar relationships and constraints are disregarded. In this case, the inter-diagram relationships, which connect diagrams belonging to one group, will also be part of the piece of model represented by that group of diagrams. Only those inter-diagram relationships and constraints which connect diagrams that are not in the same group are disregarded in the split up.

A separate component of the system is responsible for these inter-diagram relationships and constraints connecting diagrams or different groups of diagrams. This component is thus also responsible for the inter-diagram relationships and for the constraints of the meta model as a whole. This component is discussed in subsection 3.4.3.

relation	class	submodel	class	submodel
specifies	EXTERNAL STD	external std	CLASS	classdiagram + import/export diagram
specifies	INTERNAL STD	internal std	OPERATION	classdiagram + import/export diagram
refers-to	TRANSITION	std (external or internal)	OPERATION	classdiagram + import/export diagram
prescribes	STATE	std (manager)	SUBPROCESS	subprocess
requires	TRANSITION	std (manager)	TRAP	subprocess
manages	MANAGER EMPLOYEE PARTITION	manager employee subprocesses	← ternary ← relationship	

Table 3.5: The inter-diagram relationships

3.4 The editors

As a result of the proposed split ups of SOCCA models, the editor for SOCCA models can be split up in smaller editors. Each of these smaller editors is responsible for a part of the meta model. These parts have been presented in 3.3. Although the different editors will have to cooperate in order to get a complete and correct SOCCA model, most of the time they can work independently of each other. Therefore, the process of creating a SOCCA model is facilitated. Besides for being smaller, the editors are also less restrictive as they are not responsible for all of the restrictions laid upon a SOCCA model by the meta model, but only for some of them.

These editors and their functionality are discussed in this section. Some important concepts regarding these editors are also introduced.

Note, that the descriptions of the functionality of the different editors is rather limited. In the following chapters concerning the use cases, this functionality and ideas on possible implementation are discussed in much more detail.

3.4.1 Diagram editors

The fragments of the meta model presented in section 3.3.1 are meta models for the separate diagram types. As the meta model more or less defines the database which will be used, each of these fragments can be identified with a small part of this database. In such a part of the database it is possible to store diagrams of one specific type. As there are parts for every diagram type used in SOCCA, it is possible to store all diagrams necessary for a SOCCA model. The other parts of the whole meta model only indicate how the different diagrams relate to each other and group them together in perspectives and models.

To make use of these meta models for diagram types an interface with the user is needed. Such an interface must supply the user with possibilities to create diagrams of each type and translate them to the meta model. Such an interface is usually called an editor. And because this editor is used for diagrams, it is common to call it a diagram editor.

An important concept in diagram editing is: syntax directed editing. A syntax directed editor is an editor which uses the syntax of the diagram created to guide and force the user. Syntax direction can be very restrictive, but also more mild forms are used. Syntax directed editing is used in editors for visual languages.

In the master's thesis of A. Hartog and M. Wijnakker ([HW96]), a syntax directed editor for a whole range of diagrams is described and created. Their editor can be used for every type of diagram which can be described by a special grammar language. The thesis also describes a possible user interface for the editor. Because their editor provides a lot of functionality which is useful for our future diagram editors, their ideas are used as a basis for our future diagram editors in this thesis.

The editor uses building blocks and plugs. The term building block has been introduced before, but the term plug has not. This is done in the following subsection. Both the building block and the plug concept will be used further on in this thesis.

The plug concept

In a diagram editor we need to manage the connections between the different building blocks of the specific diagram type we are editing. We need to make sure that only building blocks which are allowed to be connected, can be connected by the user. Therefore, we introduce the concept of plugs.

Plugs are points on building blocks where other building blocks can be connected. Plugs can be positively or negatively charged. A positively charged plug can only connect to a negatively charged plug and vice versa. For every type of connection a separate type of plug exists. Only plugs of the same type can connect. With these two limitations (only plugs which are oppositely charged and of the same type can connect), it can be achieved that only building blocks which are allowed to be connected, can be connected.[1]

Some building blocks require some plug to be connected. If all plugs, which need to be connected actually are connected, the diagram is a correct diagram (according to the syntax). Checking for problems in the layout of a diagram is thus as easy as checking for unconnected plugs.

In the meta model, we have encountered the building blocks as objects from the building block classes. Plugs are optional connections to other building blocks and can be identified with the intra-diagram relationships. These relationships relate two building blocks, whereas connected plugs do the same. An unconnected plug is a possible intra-diagram relationship which has not been created yet.

Because we want to identify the plugs with the intra-diagram relationships presented earlier, we need to change the original definition from [HW96] slightly. Besides being points on building blocks where other building blocks can be connected, they must also be used for connections between building blocks and the diagrams.

Functionality of the diagram editors

Diagram editors are needed for creation of diagrams in the SOCCA system. They can be made responsible for the parts of the meta model which involve diagrams of one type. These parts consist of diagram classes, building block classes and intra-diagram relationships.

The functionality of the diagram editors should resemble the functionality of most graphical editors. Like all editors, the diagram editors must provide options to create, open and close diagrams. It must of course be possible to add, edit and remove building blocks of the diagram type to the diagram on the canvas and it must also be possible to connect and disconnect these building blocks. Also, it must be possible to reuse parts of diagrams by cutting, copying and pasting.

As mentioned before, syntax directed editing is an important concept in diagram editing. The editor from [HW96], which serves as an example for our future editor, makes use of this concept.

There are other things which also make it easier to work with an editor. A few examples are: spell checking, scrolling, right click menus with every type of building block, etc. Some of this functionality is presented in chapter 5, which discusses the use cases. In [HW96] some other ideas on functionality are presented as well. We have chosen only to include the most basic and the most special functionality in our descriptions of the editor. Other useful functionality can always be added later.

[1]Instead of the opposite charges, an analogy with the opposite sexes was made in the original definition of in [HW96]. We think the analogy with positive and negative charges is more appropriate.

3.4.2 Chunk editor

The split up of a SOCCA model in interesting groups of diagrams described in section 3.3.2 calls for an editor which allows the user to create and maintain groups of diagrams. This editor we will call the chunk editor and is discussed here.

As the chunk editor will be responsible for all groups of diagrams, it will also be responsible for the groups of diagrams represented by the grouping classes and the grouping relationships. If we adapt the meta model to let it also represent every group of diagrams (as proposed in subsection 3.3.2 and discussed in section 3.6.2), the chunk editor is responsible for the new grouping classes and diagrams as well.

To understand the functionality of the chunk manager we must first explain the concept of chunks.

The chunk concept

The concept of 'chunks' arose with the idea of allowing the user to create every group of diagrams useful to him. These groups of diagrams are called: 'chunks'. General chunks can contain every kind of diagram in any number. This way the user is completely free to group whatever diagrams wished for. However, there are some special types of chunk constituting certain predefined grouping classes. These chunks are:

- SOCCA model;
- data perspective;
- behaviour perspective;
- communication perspective;
- partition.

These chunks can only contain the diagrams which can be grouped by the according to their predefined type. Chunks of these specific types are special in the way that they represent some specific part of the SOCCA model (or the SOCCA model itself).

Apart from diagrams, chunk can contain other chunks as well. This way a hierarchical structure exists, which could be used for browsing. Browsing is discussed in section 3.5.2.

It is important that the chunks only contain references to the diagrams and other chunks they group in order to make it possible for one diagram or chunk to be part of different chunks. These references can be compared to the grouping relationships in the meta model. Deletion of some chunk results in deletion of the references (the grouping relationships), but not the diagrams themselves.

Apart from grouping diagrams for the specific groups mentioned above, the chunks can also be used for analysing, simulating and browsing. The concept can also be used for resolving clashes regarding version management or multiple users. In version management, it could be possible to give each chunk some version number and allow newer versions of the same chunk to contain the old version and new versions of some of the diagrams. With multiple users it is possible to give each user working on the same chunk or diagram, a copy of the original chunk. If at the end both copies clash, that is: they cannot be resolved in one chunk, then both copies would stay into existence. Meanwhile, a so-called 'choice-chunk' is used to let users who use the changed chunk make a choice between the different versions. This way we get multiple versions for some part of the SOCCA model. These versions could also split up themselves and so on. A tree-like versioning structure can be recognized. This, however, is outside the scope of this thesis. More on version management and environments in which multiple users can work together can be found in [C$^+$93].

Functionality of the chunk editor

The chunk editor must allow the user to create, edit and maintain chunks. So it should provide the user with the possibility to group the diagrams in particular chunks. It should have user friendly interface in which it is possible to manage all kinds of chunks. Important functions for this editor are:

- **Creation of a chunk:** It must be possible to create all special types of chunks. Also, it must be possible to create general chunks. The chunk editor must also be involved in the creation of the choice-chunks, although this must not be an explicit option for the user.
- **Change of a chunk:** The user must be able to add and remove diagrams and chunks to and from each user-defined chunk. In the specific chunks, the chunk editor must check if the diagrams are allowed for that kind of chunk.
- **Removal of a chunk:** Removal of chunks must also be possible.

3.4.3 Link editor

With the diagram editors and the chunk editor we have provided the user with the possibility to create and manipulate diagrams and groups of diagrams (chunks). In the database for SOCCA models, we still need a system component to manage the inter-diagram relationships between (parts of) diagrams. This system component, which we will also call an editor, is responsible for the dependencies between the different diagrams and thus also responsible for the dependencies between the different groups (which is actually the same thing). In this editor it must be possible to relate (parts of) diagrams which are meant to be dependent on each other and also tell the system in what manner these dependencies must be enforced on the database. This editor will be called the link editor and is responsible for the inter-diagram relationships and the constraints in the meta model. Therefore, it is responsible that parts of SOCCA models actually become related in the database.

Let us first introduce the concept of links, before discussing the functionality of the link editor

The link concept

Like plugs are optional connections between building blocks, links are optional connections between different diagrams and/or building blocks of different diagrams. Like with plugs, links allow only parts of SOCCA models to be related, which are allowed to be related according to the meta model. Just like a plug, a link can be connected or not connected. A connected link corresponds to an instance of an inter-diagram relationship. It indicates which parts of the SOCCA model are related to each other in the SOCCA model. Therefore, it implies that some consistency should exist between these parts (just like plugs do with building blocks within one diagram). This consistency can involve label equivalence between two building blocks, but also large constraints, like every operation from some class having to occur as a label of some transition from the corresponding external STD, are involved.

Like with plugs, there are parts which need to be linked to other parts, and there are parts which can be linked to other parts. This can be useful while checking the consistency of a (part of a) SOCCA model: required, unconnected links are always errors in consistency.

The idea behind links and plugs is thus largely the same, but with links linking larger objects and forcing larger consistency demands across different diagrams. Therefore, the links can be seen as a more general version of the plugs.

There are six types of links. This number corresponds to the number of inter-diagram relationships which exist in the meta model. For each type of link, there are different dependencies which need to hold in order for the parts to be consistent with each other.

Although a connected link implies that two parts of some SOCCA model are dependent on each other in some way, it might be useful to allow the two linked parts to be temporarily inconsistent with each other. This makes the system more flexible. If consistency between linked parts were enforced at all times, most changes to some diagram would imply errors, because temporarily some label is non-existent or different. On the other hand is must also be possible to tell the system that the dependencies between the two parts need to be satisfied. In that case, the system must not allow inconsistencies to exist between the linked parts any longer.

The manner in which the system allows inconsistencies to exist, we will call the level of consistency enforcement. There should at least be the following levels of consistency enforcement possible with every link:

- **Enforce consistency:** At all times, the dependencies between the two parts of the SOCCA model must be hold. This enforcement can thus only be set if the constraints are fulfilled to begin with.

  How the constraints are being kept fulfilled is a matter of change propagation. This is explained later.

- **Warn for violation of consistency:** Consistency is allowed to be violated, but every time a (part of a) dependency is violated, the system warns the user.

- **Allow violation of consistency:** Consistency can be violated. Note, that although it is not necessary for the two parts to be consistent with each other, the two parts are linked. This means, that at some moment in creating the SOCCA model, the consistency must hold.

Like mentioned, change propagation is an important concept here. Change propagation states if an object linked to another object must change if the first changes. If the link is set to 'enforce consistency' then a change on one side would inevitably imply a change on the other side.

If we want to enforce consistency, it is important to know in what way the system can keep the parts consistent with each other. This, because it needs to know how to change the connected parts in order to keep the link consistent. If the system cannot do this automatically, the user will have to do it or it might be that the system disallows changes at both sides of the link.

Here are some possible settings for change propagation:

- **Enforce change propagation:** A change in one of the link objects propagates through the link. The object on the other side of the link is thus also changed. This is of course only true if the changes violate the consistency demands.

  Such propagation can only be done, if the system can automatically change the other side of the link. For example, with links that link two labels that must be equivalent. It is more difficult, maybe impossible, for links involving difficult constraints to propagate changes. It could be possible to disallow the change or let the user change the other object so it is consistent with the first one.

  Important here too, is that a change on the other side of the link must be a allowed change and a change wished for. It could be possible, that we would not want to change the other side, because it represents a finished part of the model. Or that the change is not allowed, because it is not permitted by security (more on security in subsection 3.6.6). If this is the case, it must not be allowed to set the system to enforce change propagation for this link.

- **Disallow changes:** No changes are allowed that violate the constraints.

Change propagation can be set independently from the constraint enforcement for each link. This way the system could try to propagate changes, but does not need to to keep the constraints fulfilled.

Change propagation involves two directions for each link. It can be so that one direction for a link enforces change propagation, whereas the opposite direction does not. Constraint enforcement is set for a link in general and not for each direction in the link, because a violation of the constraints always involves both sides of the link: one side is not consistent with the other and vice versa.

Links are used for analysis of chunks and browsing too. Refer to 3.5.1 and 3.5.2 for these topics.

Functionality of the link editor

The link editor must provide a user-friendly interface for creating and maintaining links. Therefore, it needs to be able to visualize links. Also, the interface must provide a way to change the settings for the links. Default settings can be used for different types of links.

The link editor must only allow the linking of objects which can be linked according to the meta model (this is already stated in the definition of links). Furthermore, it should use the analysis engine to check whether the linked objects are consistent or not. If constraint enforcement is set for the linked objects, then the system or the user should immediately make the link consistent, if it is not consistent already.

A separate link editor window can be implemented and could be very useful to keep the links organized, but it must also be possible to link and un-link building blocks and diagrams from the diagram editors and chunk editor, and maybe even from the database browser.

3.5 Other system components

Apart from the editors which are responsible for the contents of the internal representation, we can think of some useful system components which only examine the contents of the database. These components of the system are presented in sections 3.5.1 till 3.5.3.

Like with the editors, the descriptions of the functionality of the other system components is limited. This, because in the following chapters concerning the use cases, this functionality and ideas on possible implementation are discussed in more detail.

3.5.1 Analysis engine

Another very important component of the system is the analysis engine. It is used for checking whether diagrams and chunks and the dependencies between them are correct. Therefore, it must be capable of determining if the syntax of each part of a SOCCA model is correct or not. For this, the analysis engine needs to know about every dependency which should hold between parts of a SOCCA model. These dependencies have been modelled in the meta model by means of relationships, cardinalities and constraints. So, what really needs to be checked is if the (part of the) SOCCA model 'fits' in the meta model presented.

The analysis engine gets its information from the internal representation where all information on every part of the SOCCA models is stored. The contents of the internal representation is not changed by the analysis engine. It only 'reads' information from it.

An important responsibility of the analysis engine is to inform the user what is wrong with the (part of the) SOCCA model, so the user can correct the problems the analysis engine has encountered. In this context, a possible advise on how to correct the problem is useful too.

The following notions are important for the analysis engine:

- **Input:** The user must be able to define which chunk or diagram is to be checked.

  Maybe, it is useful to allow the user to check multiple chunks together. This may require some additional functionality. However, such a check corresponds to a check of a chunk containing only these multiple chunks and thus could be implemented as such.

- **Output:** The user must be able to define how the output is presented.

  Possible ideas:

  - list of errors/warnings;

  - visualisation of errors/warnings: for example, it is possible to light up the conflicting parts of the model;
  - ...

- **Settings:** The user must tell the system what to check for.

  Normally, all dependencies will be checked. But there could be situations in which the user desires only a partial check: some dependencies should not be checked in that case.

To provide the user with the possibility to change these options, a user interface is needed in which the options can be set. Although the *output* and the *settings* can be of some default value, the *input* must be specified by the user.

Apart from such an explicit user interface, it must also be possible to start an analysis from other components in the system, for example from the diagram editors or the database browser.

3.5.2 Database browser

The database browser is another important component of the system. It is responsible for making the contents of the database visible. Besides this, the browser is also used for allowing the user to make selections from the database. These selections can be used for reusing information, like parts of diagrams, names of classes and operations, etc.

The database browser does not change the contents of the database, but only 'reads' information from it.

The database browser must provide a nice interface for looking at the database. In this interface it must be possible to look at the contents of chunks, diagrams and perhaps building blocks. The part-of relationships are important in this context. Furthermore, the other relationships must be made visible as well.

The contents of chunks consists of diagrams and/or other chunks. The database browser could cooperate with the chunk editor, because both the browser and the chunk editor must provide a nice view on the hierarchy of chunks and diagrams present in the database.[2]

The contents of diagrams consists of building blocks. The database browser could cooperate with the diagram editors to make these building blocks and the (intra-diagram) relationships between these diagrams visible. A view on some diagram in the editor is just a view on the contents of some diagram. The browser could thus be made responsible for the displaying of the building blocks and the relationships in the diagram editor. If that is done, the diagram editors and the database browser should work together.

The links (or inter-diagram relationships) must also be made visible by the database browser. Just like they must be made visible in the link editor]. Here too, an intensive cooperation between the link editor and the database browser is required.

In general it can be said that the database browser is responsible for the visualisation of the contents of the database. It does not matter for which component the contents must be made visible. If it must, the database browser does it.

3.5.3 Simulator/enactor

The simulator/enactor is needed for simulating or enacting the model. For simulation and enacting, information is needed on how to make 'runs' through (part of) the model. The interface for this component must provide the user with the possibility to give this information.

[2]Note, that it is still useful to separate the different components for browsing and chunk management. Although both might use (part of) the same interface. If you would not allow different components to make use of the same interface, you would end up with a number of similar looking interfaces in which only a little functionality is implemented. If you would not distinguish the different components, it would be harder to identify the functionality.

Apart from information on how to make runs, other information for simulating and enacting is needed too. (For example, for enacting some information about the context is needed). This is however beyond the scope of this thesis and will thus not be examined.

3.6 Additions to the meta model towards implementation

Although a database build according to the current meta model could contain SOCCA models, there are still some ideas which are useful (if not necessary) to include in a newer version of the meta model. These ideas are not part of SOCCA models, they are involved with using the SOCCA models in the future system. Therefore, if these ideas are added to a newer version of the meta model, then this newer meta model would rather be a meta model for the system (including SOCCA models) than a meta model for SOCCA models (as it is now). Such a meta model is needed for the system, but was not the scope of this thesis. Therefore, we only present the ideas and a few solutions on how to change the meta model for some of them.

3.6.1 Graphical information

Up till now, the meta model is not suitable for representing graphical information for the diagrams. Although it is possible to store which building blocks belong to which diagrams, it is not possible to store where the building blocks should be displayed in the diagrams. It is on purpose, that this has not been done in chapter 2. The focus was mainly on the parts of SOCCA models an the dependencies between them. We had no use for graphical information on the diagrams. However, if the meta model is used for an internal representation, this information must be included in it. For this, the meta model must be adapted.

The first idea was to include the layout information as attributes for the building block classes. This way every building block would have attributes for their x- and y-coordinates, their width and height. Other optional attributes like color and style could also be added. It is possible to let all building block classes inherit from some class called BUILDING BLOCK with all of these attributes. See figure 3.8

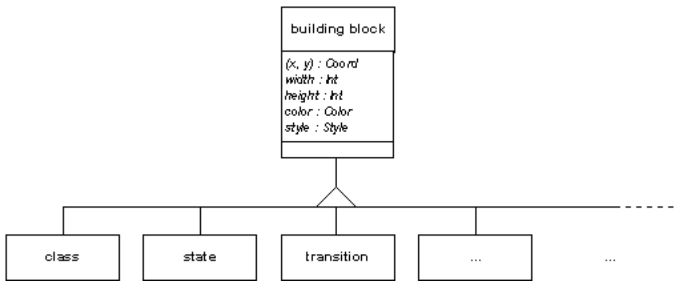


Figure 3.8: Inheritance from the class BUILDING BLOCK

This solution would be correct, were it not true that we can have the same building block existing in two or more diagrams. Take for example a state from some internal STD/employee. The same state exists in several subprocesses. Although the state is the same, the graphical information associated with it might not. It could be possible that the state could have different positions or other sizes in different diagrams. In the solution presented, it can have only one position and one size in all of the diagrams, which is not acceptable.

For a building block it must thus be possible to have different graphical information associated with it for every diagram it appears in. The solution is to make the graphical information attributes of the part-of relationship from the diagram classes to the building block classes (see figure 3.9.

This way, every building block has it s own graphical information at every diagram, as for every building block part of some diagram, there exists a part-of relationship from the diagram class object to it.

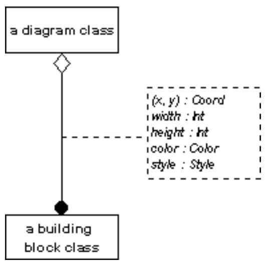


Figure 3.9: The graphical information with the part-of relationship

How about the building block classes ATTRIBUTE and OPERATION? These building block classes are not directly part of some diagram class. This is not a problem, because these classes are part of the class CLASS. If any graphical information is needed, it can be stored in those part-of relationships. Note, that in that case, it is not any different to store the information in the classes themselves or with the relationships.

So, changes to the meta model for supporting graphical information is nothing more than adding attributes to some relationships.

3.6.2 Chunks

Although there is room for the specific types of chunks in the meta model (as they correspond with the grouping classes), general chunks cannot be stored in it. Therefore the class CHUNK is introduced. The class CHUNK represents chunks and thus from it part-of relationships must be created to the different diagram classes. Because the grouping classes represent specific chunks, they should inherit from CHUNK. After this, there is no need for the grouping relationships anymore, so they can be erased. To allow chunks to contain other chunks a part-of relationship from CHUNK to itself is created. The result is displayed in figure 3.10

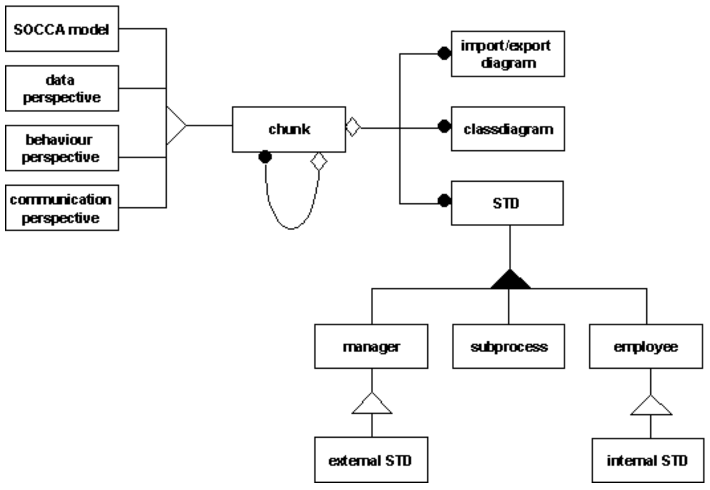


Figure 3.10: First try to add chunks to the meta model

Although this change to the meta model allows for all types of chunks to exist, it also limits the expressiveness of the meta model, because now it is possible for any types of diagrams to be part of the specific types of chunks. For example, it is possible to add STDs to a data perspective chunk or three class diagrams to one SOCCA model chunk. It is even possible to let a partition chunk contain a SOCCA model chunk. We do not want these possibilities and we thus need to look for another solution.

A better solution might be to leave the original grouping relationships intact and discard the part-of relationships from CHUNK to the diagram classes. But because chunks can contain chunks, this would not solve the problem.

We could also add another class GENERAL CHUNK as another special type of chunk, like in figure 3.11. The problem of the special chunks containing diagrams which are not allowed, is solved here. The inheritance is total and disjoint. This way all chunks can be classified in one special type (also general) of chunk.

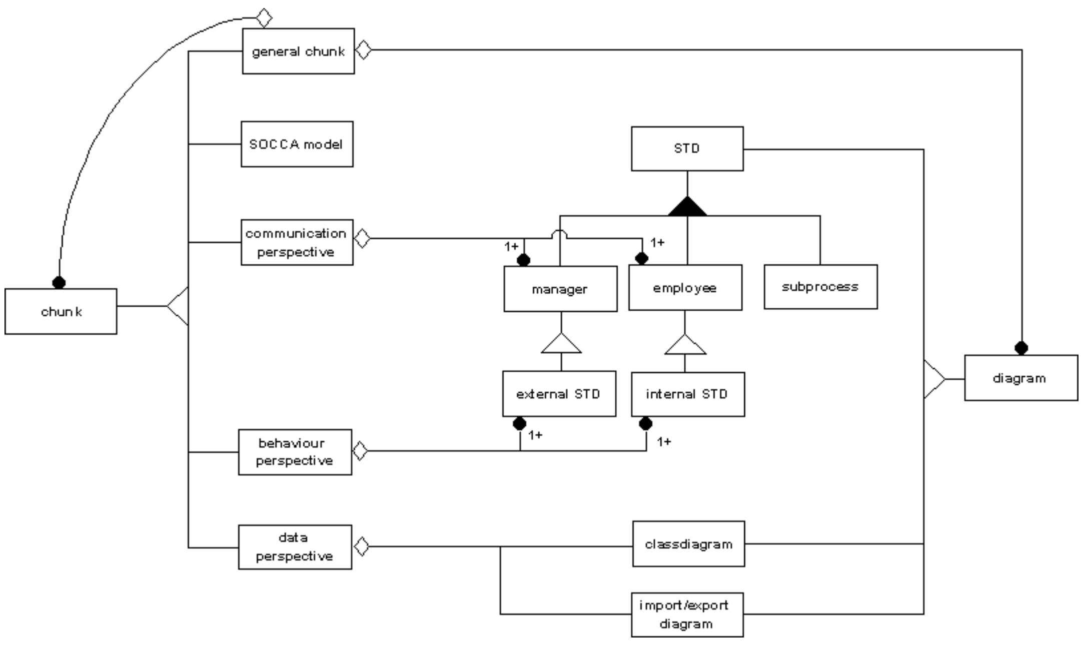


Figure 3.11: Adding chunks to the meta model

Although the solution presented is acceptable, it still does not allow for a SOCCA model chunk to immediately contain diagrams. This is done via three perspective chunks. Also, it is not possible to define a SOCCA model chunk to contain several general chunks, which together could span the model intended. The first solution (presented in figure 3.10) was more suitable for that purpose. In that case, the problem of the special types of chunks containing diagrams which are really not allowed in such a chunk, must be solved by constraints.

3.6.3 Links

Although the inter-diagram relationships can be identified with links, the meta model does not show any possibility to store settings on constraint enforcement and change propagation. This is easily solved by adding attributes to the inter-diagram relationships which would represent these settings. One attribute for constraint enforcement and two attributes for change propagation (one for each direction) should be sufficient. Other future settings regarding links can be stored as attributes for the relationships as well.

Note, that the ternary relationship *manages* involves six attributes for change propagation: one for each direction between two classes.

3.6.4 Version management

Version management cannot be distinguished in the meta model either. The question is whether version management needs to be present in the meta model. Newer versions of diagrams and chunks are normal diagrams and chunks too, so they 'fit' in the meta model as it is now.

It is possible to add attributes to the diagram classes and grouping classes (now including the classes CHUNK and GENERAL CHUNK) regarding version numbers, but this is not the only thing involved in version management. In version management it is also important to keep track of changes: which part is a revision of which other part and what has changed regarding the earlier version. Also important are the dependencies the newer versions are involved in. Say a newer version A2 of some diagram A1 is created. If A2 is changed, it is possible that a dependent diagram B1 must be changed as well. This dependent diagram B1 could now violate dependencies with A1. In this case, should a new version of B1 be made or should all dependencies regarding earlier version be disregarded?

This problem and a lot of other problems are part of version management. If the future system should include all possible functionality regarding version management, this must be studied. That is however beyond the scope of this thesis. For now, we think that a simple part of version management can be included in the meta model by adding some attributes to the diagram classes and grouping classes. For more complicated version management it is suspected that larger changes are necessary.

More on version management can be found in [C+93].

3.6.5 Multi user environment

Like with version management, if a multi user environment is to be part of the future system, it needs to be studied separately. Implementation of a multiple user environment would definitely change the meta model. It might be possible to include a small part of it as attributes in the meta model as it is now, but addition of classes and relationships are most likely needed as well.

More on multi user environments can be found in [C+93] as well.

3.6.6 Security

If the system is going to be used by multiple users, either at once or one at a time, security is an important issue. If part of a model is correct, you would not want another person to change it or maybe even see it. So in the system it should be possible to secure parts of models.

Although it's definitely useful to allow a user to secure diagrams, it might be also useful to allow a user to secure building blocks, chunks and even inter-diagram relationships. For example, some class might be finished modelling, but not the whole class diagram. A user then must be allowed to change the class diagram, but not the name, operations and attributes of the specific class. In general, it is preferable that some stable part of a model, whether that is a chunk, a diagram or a building block, cannot be changed by a person who should not do that.

The meta model does not contain any notions of security, it only presents a way to store a SOCCA model in a data structure and the relationships which should hold between parts of that SOCCA model. To add security to the meta model, you would have to add the concept of users to it (with a class called USER for example). Furthermore, you would need to add attributes to all classes indicating which users are allowed to view or change a model.

3.7 From system components to the use cases

In the following chapters, use cases are provided for the expected components in the system. For an introduction to these use cases refer to chapter 4.

These use cases have been created to present the expected functionality for the future system. As the future system consists of the various system components introduced in this chapter, the use cases present the functionality of these components. Also notes on possible implementation and user interface are given. The description of the system components is therefore far more detailed then in sections 3.4 and 3.5.

The system components are identified with the use cases as follows:

- diagram editors $\leftrightarrow$ use case "specify diagram" for DiagramBuilder
- chunk editor $\leftrightarrow$ use case "manage chunks" for ModelIntegrator
- link editor $\leftrightarrow$ use case "manage links" for ModelIntegrator
- analysis engine $\leftrightarrow$ use cases "analyse diagram" for DiagramBuilder and "analyse chunk" for ModelIntegrator
- database browser $\leftrightarrow$ use case "browse SOCCA model" for all actors
- simulator/enactor $\leftrightarrow$ no use cases yet

Chapter 4

Introduction on the use cases

4.1 Use cases

In [JEJ95], use cases are defined as "sequences of transactions in a system whose task is to yield a measurable value to an individual actor of the system". To better understand this definition, we have taken the following definitions from [Ber96]:

- a use case is "a specific flow of events through the system, that is, an instance" ([JEJ95]). Using the concept of a class as the set of all items which share a collection of similar characteristics, it is suggested that many similar courses of events be grouped into a "use-case class." (Note that this definition, i.e., a class is a set of instances, is not the same definition of class that is used in a Smalltalk, C++, or Eiffel context.)
- an actor is "a role that someone or something in the environment can play in relation to the business" ([JEJ95]). Further, the same person (or other item) can assume more than one role.
- "transactions in a system" implies that the system will make available to its actors a set of capabilities that will both allow the actors to communicate with the system and to accomplish some meaningful work (i.e., meaningful value).
- "a measurable value" implies that the performance of the task has some visible, quantifiable, and/or qualifiable impact on those things which lie outside of the system, and, in particular, the actor who initiated the task.
- a transaction is defined as "an atomic set of activities that are performed either fully or not at all. It is invoked by a stimulus from an actor to the system or by a point in time being reached in the system. A transaction consists of actions, decisions and transmission of stimuli to the invoking actor or to some other actor(s)." (See [JEJ95].)

Three main reasons for creating use cases presented at [Ber96] are:

- gaining an understanding of the problem,
- capturing an understanding of the proposed solution, and
- identifying candidate objects.

In our case, we will indeed use the use cases to gain an understanding of the problems involved in the process of creating and using SOCCA models in an environment. The use cases (in our case too) provide a proposed solution to these problems. Because of the informal description, it not so hard to understand the solutions. The last reason lies outside the scope of this thesis, but will certainly be used if the system as proposed here is going to be implemented.

4.2 The use cases for our future system

For our future system, we have defined the following actors:

- **DiagramBuilder:** A DiagramBuilder builds the diagrams that eventually will make up a SOCCA model.
- **ModelIntegrator:** A ModelIntegrator integrates diagrams and submodels (represented by chunks) in larger submodels. This integration also includes linking of diagrams.
- **EnactmentCoordinator:** An EnactmentCoordinator uses some complete model to set up a system in which the model is used (enacted).

The use cases for the different actors can be seen in the use case diagram in figure 4.1.

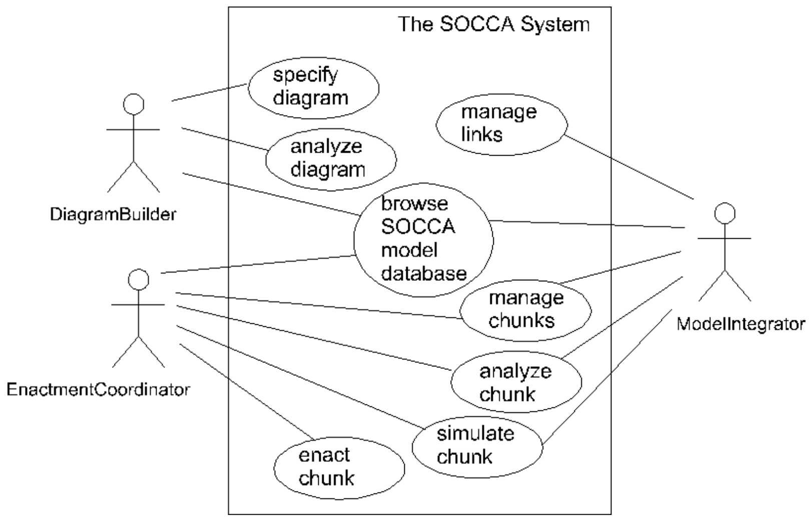


Figure 4.1: The use case diagram for our future system

A short description of these use cases:

- **Specify diagram:** (Used by DiagramBuilder) Allows for a diagram to be edited. Either a new diagram or an existing diagram can be edited.
- **Analyse diagram:** (Used by DiagramBuilder) Allows for a diagram to be analysed. Analysis is limited to one diagram only.
- **Browse SOCCA model database:** (Used by DiagramBuilder, ModelIntegrator and EnactmentCoordinator) Allows for the database to be browsed and make all information visible, either graphical (e.g. for diagrams) or text.
- **Manage links:** (Used by ModelIntegrator) Allows links to be created and removed between diagrams. Also allows the settings for the links to be changed.
- **Manage chunks:** (Used by ModelIntegrator and EnactmentCoordinator) Allows chunks to be created and removed.
- **Analyse chunk:** (Used by ModelIntegrator and EnactmentCoordinator) Allows for chunks to be analysed.
- **Simulate chunk:** (Used by ModelIntegrator and EnactmentCoordinator) Allows for a chunk to be simulated. Not discussed in the next chapters as it falls outside the scope of this thesis.

- **Enact chunk:** (Used by EnactmentCoordinator) Allows a chunk to be enacted as a working model. Also not discussed in the next chapters.

It is possible that research in the future might show the need for more use cases. For example, the EnactmentCoordinator might need a use case for setting the context for some enactment. These use cases will then have to be added. In our opinion, no other actors will be needed, as these actors cover all phases in creating in a SOCCA model.

In the next two chapters the use cases are described in detail and alternatives and notes on possible implementation are given. We have described the use cases to the point of the functionality wished for by the user. We have not modeled in the use cases, what the user exactly has (or wants) to do in order to receive that functionality. For most cases, the system notes for the use cases involved describe this. The border between what is and what is not modeled in the use cases is made more clear in the following example:
A DiagramBuilder can create building blocks in the diagram. This is indeed modeled in the use case "Specify diagram" (see step B in use case U1 in the next chapter). However, this is the lowest level in the use case, which is involved with the creation of building blocks. It is not described in the use case, what the user should do in order to create a building block. The system notes on use case U1 describe this.

4.3 Notation used in describing the use cases

4.3.1 Steps in the use cases

In [JEJ95] the identified use cases were described by a simple list descriptions of events. These description of events are called: steps. An example taken from [JEJ95] is provided in figure 4.2.

Serving Dinner

A. The use case begins when the actor Guest enters the restaurant.

B. The actor Guest has the possibility of leaving his/her coat in the cloakroom, after which he/she is shown to a table and given a menu.

C. When the actor Guest has had sufficient time to make up his/her mind, he/she is asked to state his/her order. Alternatively, Guest can attract the waiter's attention so that the order can be placed.

D. When the Guest has ordered, the kitchen is informed what food and beverages the order contains.

E. In the kitchen, certain basic ingredients, such as sauces, rice, and potatoes, have already been prepared. Cooking therefore involves collecting together these basic ingredients, adding spices and so on and sorting out what needs to be done just before the dish is served. Also, the required beverages are fetched from the refrigerator.

F. When the dish is ready, it is served to the actor Guest. When it has been eaten, the actor is expected to attract the waiter's attention in order to pay.

G. Once payment has been made, Guest can fetch his/her coat from the cloakroom and leave the restaurant. The use case is then complete."

Figure 4.2: Example of the original way to describe the use cases

The descriptions are very informal and could possibly lead to more interpretations. For example, is it in the above case possible for the customer to order two entrees?

4.3.2 Additional constructs in the descriptions

As described in the previous section, use cases were originally described by a sequence of informal descriptions of steps. In order to describe the use cases in a more systematic manner and make them easier to read, we have enhanced the original notation somewhat. These enhancements introduce some extra notation to replace often reoccurring concepts which originally would have been described in words only.

Our new notation involves: sub use cases, choices, options which can be repeated and preconditions. By giving these concepts their own notation, the use cases are easier to read and their structure becomes much clearer. The following four sections introduce the additional notation used in our description of the use cases.

Note, that our additions replace the need to describe these concepts in text. So, the additions do not add to the expressiveness of the descriptions of the use cases. They merely make the descriptions much more easier to read.

Splitting up the description of a use case

The description of a use case is usually divided in steps. Most of the time, it is possible to find a useful split up of the events in steps. Sometimes these steps can also be divided in steps and so on, indicating some kind of refinement. In the original descriptions of the use cases, only one level of steps was possible. If a step could be divided in multiple smaller steps, it just had to be described informally in the text or a new use case for the step had to be introduced. This new use case could then be 'used' by the original use case. This results in use cases which are defined on a different level and are only used by other use cases and not by actors.

In order to avoid these use cases on different levels, we have called these use cases which are only used by other use cases: sub use cases. As sub use cases are only used by 'real' use cases or other sub use cases and not by actors immediately, they are nothing more then steps within steps in a use case. We now do not have to refer to these sub use cases as 'normal' use cases and we do not have to describe steps within steps as this can be done in the sub use cases.

The notation used with the description of these sub use cases is shown in figure 4.3.

U1 Actor use case: 'Real' use case

A. First step in the use case;

B. 'Name of sub use case'; (**reference to the sub use case and page**)

C. ...

U1.1 'Name of sub use case'

A. First step in the sub use case (really first step within step B. of the 'real' use case);

B. Second step in the sub use case;

C. ...

Figure 4.3: Sub use cases

Note, that sub use cases can contain references to sub use cases as well (sub-sub use cases).

Choices in flow of events

In trying to describe the flow of events through parts of the system, we encountered the problem that we often needed to describe very many alternatives within one step in a (sub) use case. The traditional way of doing this, was by specifying the alternatives separately at the bottom of the description of the use case. This can easily be done with just one or a few alternatives, but with multiple alternatives possible to an actor it became unclear. Therefore, we have introduced a notation to give different alternatives within one step. It is presented in figure 4.4.

X. General description on the kind of choice;

- First choice;
- Second choice;
- ...
- ...

Y. ...

Figure 4.4: Choices in the use case descriptions

It means the actor can do either "First choice", or either "Second choice" and so on. After having finished one choice (which can mean finishing a sub use case), the following step in the (sub) use case is regarded. Thus, it is not possible to do multiple choices within one step in a run through a (sub) use case.

Repeating choices

Although we have now a useful notation to represent choices, we still have no way to present an actor with an option, which he/she can do zero or as many times as requested. For example, while editing, the actor could want to change the zoom of the diagram several times, between editing parts of the diagram. In the original use case model, this could be done by telling the use case to "proceed at" some point in the use case. If, we would have told the use case to proceed at a step, before the current step, we had modeled some kind of iteration. With this iteration and alternatives like described in the previous subsection, we could have modeled such repeating choices in the original descriptions. However, it would have resulted in a description which would have been much more difficult to read than a description with the notation presented here.

The notation introduced in figure 4.5 makes repeatable options much more easier to describe.

X. General description on the kind of repeating choice;

- • First choice;
- • Second choice;
- • ...
- • ...

Y. ...

Figure 4.5: Repeating choices in the use case descriptions

The notation represents the following. When entering step X the actor can do “First choice” as many times as he/she likes. The same goes for “Second choice” and all other choices. It is also possible to do “First choice”, then “Second choice” and then again “First choice” and so on. When the actor decides that he does not want to do any more options, the (sub) use case proceeds to step Y.

Preconditions

Sometimes, it is useful to know whether some condition has been fulfilled, before entering doing a step. For example, a trap can only be entered to a diagram if the diagram is a subprocess. These preconditions could only be described in the original descriptions by sentences like: “If a subprocess is being edited, then...”. By introducing a new notation, such preconditions can be distinguished immediately.

The notation is introduced in figure 4.6.

X. {*this is a precondition*} Only do this when pre-condition is met;

Y. ...

Figure 4.6: Preconditions in use case descriptions

So, step X is only done if the precondition is true. If it is not, step X is simply not possible and in this case the model would proceed with step Y.

4.4 Structure of the chapters on the use cases

Besides the descriptions of the use cases for our future system, the following chapters also contain discussions and system notes on these use cases. The discussions describe possible alternatives for the use cases and motivation for the choice of the current use cases. The system notes on the use cases describe implementation issues, which we encountered while creating the use cases.

By letting the parts start on an empty page on the right side, it is possible to separate the three parts and read them simultaneously. This is particularly useful if the use cases are studied in detail and discussion on them and issues regarding implementation are needed as well.

The same numbering of the use cases and sub use cases has been used in each of the three parts. This way, it is easy to reference to the same (sub) use case in another part. Furthermore, the numbering of the (sub) use cases is done in a depth-first way: so if a refinement of a (sub) use case occurs at some level, then the refinement will occur one level deeper and follow the original (sub) use case as soon as possible. Figure 4.7 will make this clear. With this numbering it is easiest to track refinements of (sub) use cases. To make it even easier, the page number is included in the references to the refinements.

U1	**First use case**
U1.1	**First refinement within first use case**
U1.2	**Second refinement within first use case**
U1.2.1	First refinement within second refinement ...
U1.2.2	Second refinement within second refinement ...
U1.2.3	...
U2	**Second use case**
U2.1	...

Figure 4.7: The numbering used in the chapter on the use cases

Chapter 5

The DiagramBuilder use cases

5.1 Introduction

This chapter contains the use cases for the actor DiagramBuilder. As mentioned in chapter 4, this chapter contains four sections: the introduction, the use cases, the discussion on the use cases and the system notes.

5.1.1 The use cases

The actor DiagramBuilder is responsible for creating and maintaining diagrams for the SOCCA models. In order to do this, the DiagramBuilder will at least need a use case for specifying diagrams. This use case should provide the DiagramBuilder with the possibility to load and save diagrams and do all sorts of editing operations on them. In order to check diagrams, the DiagramBuilder should be provided with a use case for analysing a diagram. Also a use case for browsing the database is to be provided. This to examine other parts of the model and possibly reuse those parts in the current diagram.

A short description of the use cases is given in the following sections.

5.1.1.1 The use case: Specify diagram

This use case provides the DiagramBuilder with all functionality regarding the creation and editing of diagrams. All types of diagrams used in SOCCA can be created. Copies and new versions of existing diagrams can be made. For editing the diagram, the DiagramBuilder can create and manipulate building blocks. Automatic generation of diagram parts is also included. Finally, the diagram can be stored in the database.

5.1.1.2 The use case: Analyse diagram

With this use cases, the DiagramBuilder can analyse a diagram. This analysis is limited to the diagram only. Thus, it does not check consistency with other diagrams. The diagrams can thus only be checked on their own syntax.

5.1.1.3 The use case: Browse SOCCA model database

This use case provides all functionality concerning viewing information from the SOCCA model database. It provides ways to 'walk' through the database, search it, and make all concepts from SOCCA models visible to the actor. Furthermore, it allows the actor to change views on the diagrams.

5.1.2 List of use cases and sub use cases

With the help of the following list, it is easy to find the use cases and sub use cases in this document. After each (sub) use case, three page numbers are listed. The first page number is that of the (sub) use case. The second page number is that of the discussion on the particular (sub) use case. And the third is that of the system notes on the particular (sub) use case. Sometimes no page number is listed. In that case no discussion or system notes are given on the (sub) use case.

number	**name**	use case page	dis-cus-sion page	system notes page
1	**DiagramBuilder use case: Specify diagram**	63	67	73
1.1	**Open new (empty) diagram**	63	68	-
1.2	**Generate diagram part(s)**	64	68	74
1.3	**Apply generic editing operation**	64	68	-
2	**DiagramBuilder use case: Analyse diagram**	64	69	75
3	**DiagramBuilder use case: Browse SOCCA model database**	64	69	75
3.1	**Change view on diagram**	64	69	75
3.2	**Search within chunks**	65	70	76
3.2.1	Specify range of search	65	70	76
3.2.2	Specify type of search	65	70	76
3.3	**Browse through chunks**	65	70	77
3.4	**Follow links**	65	71	77
3.4.1	Narrow link choices	65	71	77

Section 5.2

DiagramBuilder use cases

The use cases for the DiagramBuilder actor are the following:

- Specify diagram; (**See U1**)
- Analyse diagram; (**See U2 on the next page**)
- Browse SOCCA model database. (**See U3 on the next page**)

U1 DiagramBuilder use case: Specify diagram

A. Open diagram;

   - Open new (empty) diagram; (**See U1.1**)
   - Load specific diagram;
   - Open copy of specific diagram;
   - Open new version of specific diagram.

B. Change diagram.

   - Create building block;
   - {*current diagram type is 'subprocess'*} Specify traps;
   - Generate diagram part(s); (**See U1.2 on the next page**)
   - Apply generic editing operation. (**See U1.3 on the next page**)

C. Save diagram.

U1.1 Open new (empty) diagram

- Open new (empty) EER diagram;
- Open new (empty) external STD;
- Open new (empty) internal STD;
- Open new (empty) manager;
- Open new (empty) employee;
- Open new (empty) subprocess;
- Open new (empty) general STD;

U1.2 Generate diagram part(s)

A. Select building block(s) to generate from;

B. Initiate generating of diagram part(s).

U1.3 Apply generic editing operation

- Move building block;
- Delete building block;
- Scale building block;
- Stretch building block;
- Rotate building block;
- Exchange building block;
- Connect plugs;
- Disconnect plugs.

U2 DiagramBuilder use case: Analyse diagram

A. Select diagram;

B. Initiate analysis.

U3 DiagramBuilder use case: Browse SOCCA model database

A. Types of browsing;

- Change view on diagram; (**See U3.1**)
- Search within chunks; (**See U3.2 on the next page**)
- Browse through chunks; (**See U3.3 on the next page**)
- Follow links. (**See U3.4 on the next page**)

U3.1 Change view on diagram

A. Select diagram;

B. Select view operation;

- Zoom diagram;
- Move the view port;
- Select parts to view and hide.

U3.2 Search within chunks

A. Specify search options;

- Specify text to search for;
- Specify range of search; (**See U3.2.1**)
- Specify type of search; (**See U3.2.2**)

B. Initiate search.

U3.2.1 Specify range of search

- Specify chunks to search in;
- Specify diagram types to search in;
- Specify building block types to search in.

U3.2.2 Specify type of search

- Toggle 'match case' on/off;
- Toggle 'match complete string' on/off;
- Toggle 'fuzzy search' on/off.

U3.3 Browse through chunks

- Choose current chunk;
- Expand chunk hierarchy;
- Collapse chunk hierarchy.

U3.4 Follow links

A. Select diagram;

B. Narrow link choices; (**See U3.4.1**)

C. Select link to follow;

D. Initiate following of link.

U3.4.1 Narrow link choices

- Select building block;
- Select target type;

Section 5.3

Discussion on DiagramBuilder use cases

The comments on the different use cases for the DiagramBuilder actor can be found on the following pages:

- Specify diagram; (**See D1**)
- Analyse diagram; (**See D2 on page 69**)
- Browse SOCCA model database. (**See D3 on page 69**)

D1 Discussion on: Specify diagram

A. **Open diagram:** To (further) specify a diagram, you first have to pick the diagram. A diagram will also have to be picked in the use cases "Analyse diagram" and "Animate diagram". In those use cases there is no need for copies or new diagrams, so only the "Load specific diagram" is used there.

Because there are several types of diagrams used in SOCCA, the DiagramBuilder will have to specify what type of diagram he or she wishes to create when choosing the option "Open new (empty) diagram". Refer to D1.1 on the next page for the different types possible.

The difference between "Open copy ... " and "Open new version ... " is that in the former a diagram is copied *without* taking the links with other diagrams into consideration. When opening a new version however, the links to other diagrams are maintained. Note that this introduces the problem of links with more than one source or destination. This problem will be handled by the version management component of the SOCCA environment (see section 3.6.4).

The choices "Open copy ... " and "Open new version ... " can functionally be handled in terms of "Open new (empty) diagram" followed by "Generate diagram part(s)". However, from a user's point of view, this is not an obvious thing to have to do. The current solution is much clearer.

B. **Change diagram:** The term building block has a very specific meaning. It is introduced in [HW96].

- **Create building block:** Building blocks can be created in the editor. What building blocks can be created depends on the type of diagram which is being edited.
- ***{current diagram type is 'subprocess'}* Specify traps:** This option should provide for the creation, editing and removal of traps in a subprocess. Because a trap does not

meet the current definition of a building block, it cannot be created with the previous option. A building block has a fixed layout, whereas a trap does not. This might not be a problem, if the definition of a building block is adjusted.

- **Generate diagram part(s):** Refer to D1.2 for discussion.
- **Apply generic editing operation:** Refer to D1.3 for discussion.

C. **Save diagram:** The diagram must be saved explicitly.

D1.1 Open new (empty) diagram

The following types of diagrams can be created:

- EER diagram:
- external STD;
- internal STD;
- manager;
- employee;
- subprocess;
- general STD;

The option to create a general STD is added for flexibility towards the user.

D1.2 Generate diagram part(s)

The idea about generating diagram parts is that you first specify particular building blocks. These building blocks should have a particular relationship with the part that you want to generate. Therefore, some diagram parts can be generated by the system itself.

A. **Select building block(s) to generate from:** The building blocks can be chosen from the SOCCA model browser as described in the use case "Browse SOCCA model". The selected building blocks must have some relationship to the diagram worked on, so generation is possible.

B. **Initiate generating of diagram part(s):** The parts of the diagram are generated. The links from the generated part to the part generated from will be created too. These links will all be of some default type (possibly the identity relationship) and some default permissions will be set for it. This way, the DiagramBuilder does not have any control over the links. The ModelIntegrator is allowed to change link-types and thus the responsibility is for that role only.

D1.3 Apply generic editing operation

This list of choices is based on the generic, graphic editor which uses building blocks and plugs. It is important to appreciate the exact meanings of those terms. See [HW96] for this.

D2 Discussion on: Analyse diagram

Analysis of the diagram is limited to the diagram itself. No links will be followed to check other consistencies. That kind of analysis is done in in the use case “Analyse chunk” discussed in sub use caseU3 of chapter 6.

All things that could make a general diagram faulty can be checked here. Most of these possible checks, have to do with whether some building blocks are connected or not. Connections between building blocks can be done via plugs. The concept of plugs is introduced in section 3.4.1 and in [HW96]. It is through this concept, that it is not possible to make connections between building blocks, which are not allowed in a diagram. Therefore, the analysis engine in the system will not have to check the connections between the building blocks. It only needs to check if the plugs which should be connected are indeed connected.

Other checks which could be possible are the check for cycles in inheritance and part-of relationships in EER diagrams. Also, warnings could be generated if for example an inherited operation has the same name as an operation from the class itself. For STDs, the analysis engine could also check for parts which cannot be reached from every state.

A. **Select diagram:** To analyse a diagram, it first has to be selected.

B. **Initiate analysis:** Starts the analysis of the diagram.

D3 Discussion on: Browse SOCCA model database

Browsing a SOCCA model involves viewing operations on different types of diagrams used in SOCCA, search operations within the SOCCA model and exploring it in several ways. The viewing operations are explained in **D3.1**, the discussion involving the search operations is at **D3.2** and exploring SOCCA models is discussed at the end of this section.

A. Types of browsing:

- **Change view on diagram:** Refer to D3.1 for discussion on this subject.
- **Search within chunks:** Refer to D3.2 on the next page for discussion on this subject.
- **Browse through chunks:** Refer to D3.3 on the next page for discussion on this subject.
- **Follow links:** Refer to D3.4 on page 71 for discussion on this subject.

D3.1 Change view on diagram

A. **Select diagram:** First a diagram must be selected. This diagram is not opened by this use case, so it should already be open. We do not allow an DiagramBuilder to open a diagram, because this is no browsing as we see it. Opening a diagram could for example be done in **Open diagram**.

B. **Select view operation:**

- **Zoom diagram:** Zooming is supported in most editors. The zooming option for this editor must supply a method to change the zooming-factor of the diagram.
- **Move the view port:** If the diagram is larger than the screen, there should be a way to move to parts of the diagram, which cannot be seen.
- **Select parts to view and hide:** This is not really a standard option for editors, but it could prove rather useful for our purpose. Building blocks and types of building blocks can be hidden and made to reappear as wished. This is particularly useful if the diagram is clutters with building blocks and relationships.
- **...:** Other operations on the view on diagrams (so no operations which actually change the diagrams) could be added.

D3.2 Search within chunks

Searching for all kinds of objects, like chunks, diagrams, building blocks and even links must be possible.

A. **Specify search options:**
   - **Specify text to search for:** The text (which is usually a name from some building block or diagram) can be specified. It is possible to search for nothing, which would list all objects specified by the range of search.
   - **Specify range of search:** See D3.2.1 for discussion.
   - **Specify type of search:** See D3.2.2 for discussion.

B. **Initiate search:** Starts the search.

D3.2.1 Specify range of search

- **Specify chunks to search in:** In these chunks the object(s) specified will be searched for. With this option, the search can also be limited to one diagram. This because one diagram is also a chunk.
- **Specify diagram types to search in:** The types of diagrams specified here will be searched for the object(s) specified. Something like: 'in all external STDs' can be specified here.
- **Specify building block types to search in:** Here, the types of building blocks which will be searched in are defined. This way you could search just the names of classes or common relationships, for example.

D3.2.2 Specify type of search

- **Toggle 'match case' on/off:** If 'match case' is on, the search engine searches for objects with exactly the same case as the search string specified.
- **Toggle 'match complete string' on/off:** If 'match complete string' is on, the search engine will not look for a string in which the search string occurs, but only those strings which match the whole search string are seen as successful.
- **Toggle 'fuzzy search' on/off:** Fuzzy search means that the search engine will search for strings which match the search string closely, but not totally. For example, if one character is different, the search engine will still accept the discovered string as successful.

D3.3 Browse through chunks

Browsing through chunks means that the DiagramBuilder can explore the database using the ideas of chunks. This can be useful if some particular diagram or chunk is searched for, or if an impression of the database is wanted.

Because chunks can group chunks, a hierarchical structure is available. At the bottom of this structure the diagrams are situated, at the top the SOCCA models. The DiagramBuilder can now browse through the hierarchy of chunks by means of the following operations. Together with the option to follow links in the **Browse SOCCA model database** use case, this would result in a very flexible browser.

In this browser, the concept of the current chunk is used. This 'current chunk' is the chunk which the browser is focussed on; The browser displays the contents of it.

- **Choose current chunk:** Make selected chunk the current chunk (leave the previous current chunk) From this chunk the browsing can continue (compare: the current directory in a file-browser).

- **Expand chunk hierarchy:** Show all chunks and diagrams one level deeper (compare: expanding a directory-tree).
- **Collapse chunk hierarchy:** Show all chunks and diagram one level less deep (compare: collapsing a directory-tree).

D3.4 Follow links

This part of the use case allows the DiagramBuilder to follow links between diagrams. Together with the browsing through chunks it allows for a flexible browser within the SOCCA environment. The DiagramBuilder can now explore the SOCCA models through the links, because they can be followed to reach other diagrams and from these diagrams other chunks can be reached.

A. **Select diagram:** First, a diagram from which the link has to be followed has to be specified. This can be done by clicking its window, but also by selecting the name in the model browser.

B. **Narrow link choices:** See D3.4.1 for comments.

C. **Select link to follow:** There still could be several links left to choose from, this must be done here.

D. **Initiate following of link:** Follows the link. If this is done from the model browser, the diagram on the other side of the link will become the new current diagram in the browser; browsing will resume from another node. If this operation is done from some open diagram, this would result in opening the diagram on the other side of the link.

D3.4.1 Narrow link choices

These are optional choices to narrow down the possible links.

- **Select building block:** Some links are from building blocks to other building blocks, others are from building blocks to diagrams and there are also links from diagram to diagram. Therefore, sometimes a building block can be specified. This to select from the links, which are from that particular building block.

  If no building block is specified, all links involving the diagram (and its building blocks) are available for following.

- **Select target type:** There are several types of diagrams and building blocks involved in links. If the type of the diagram or building block at the other side of the link is specified, this narrows down the choices of links to follow.

  Note, that with such a target choice (together with the source), the type of link is almost specified. There will not be much links with the same source and target (possibly none).

- . . . : Other options might follow.

Section 5.4

System notes on DiagramBuilder use cases

The system notes on the different use cases for the DiagramBuilder actor can be found on the following pages:

- Specify diagram; (**See S1**)
- Analyse diagram; (**See S2 on page 75**)
- Browse SOCCA model database. (**See S3 on page 75**)

S1 System notes on: Specify diagram

A. **Open diagram:** Opening a diagram is something that will have to be done for analysis and animation of a diagram as well. In an actual information system this part will be handled by the familiar windowing interface. In such a system traditionally there is a file menu with options such as **New**, **Open** and **Close**. The system will have to maintain the notion of *current diagram/window*.

   Note that a command like **Close** does not have a direct counterpart in the use case specification. The **Close** command just ends a particular sequence of actions, corresponding to a use case. In the actual system it is necessary because several use case sequences interact; in the use case specification it is implicit.

   To specify a diagram the user will have to enter some kind of identification. A proper identification consists of a name string and a version number. In a lot of cases the user will not have to enter a version number, as the latest version can often be used for a default here. Of course, specifying a diagram's name may be accomplished by picking a name from a list.

   - **Open new (empty) diagram:** This corresponds to the command **New**. When opening a new diagram, the user will have to supply a name. The system may either ask for this on opening the new diagram, or when closing the diagram. It is common practice nowadays to specify file names for new files on saving. In the meantime, generic, unique names are used. From that perspective, this may also be the right thing to do for the SOCCA environment. It is also the view that fits nicely with the idea of making the system as inobtrusive as possible. However, this may interfere with database management issues. A decision should be taken on this in detailed design.

     Apart from a name, the diagram type must be specified as well. This must happen before editing the diagram, because the editor will contain commands specific for the different diagram types. Therefore, it might be nicer to define both name and type when opening the diagram. The type of the diagram should of course be chosen from a list of possible types.

- **Load specific diagram:** This corresponds to the command **Open**.
- **Open copy of specific diagram:** This corresponds to the command **Open Copy**. Links to other diagrams are not taken into consideration.
- **Open new version of specific diagram:** This corresponds to the command **Open New Version**. Links to other diagrams are taken into consideration. Creating of a new version thus adds a choice to the process of integrating a SOCCA model from its components: either choose the new version or one of the old.

B. **Change diagram:** Changing an diagram closely follows the ideas that formed the basis of the prototype for the structured, generic, graphical editor ([HW96]). This editor does not contain functionality for the generation of diagram parts.

- **Create building block:** Building blocks can be created using special-purpose buttons that have been generated from a user's description of diagrams.

  It must also be possible to drag labels from other windows onto the canvas. This way the labels are used as building blocks as well. For example, operation names can be dragged to a class, to add them to it.
- **{*current diagram type is 'subprocess'*} Specify traps:** If the diagram being edited is a subprocess, there must be a way to create, edit and remove traps.

  Creation of traps would most preferably be done by selecting the states and transitions wanted in the trap and then select an option 'create trap' from some menu or button. The selection could be made by simply clicking on the states and transitions in the diagram. The trap should be provided with a name, which could be prompted for.

  Editing a trap would involve adding and removing states and transitions from it. Again a selection of the states and transition wished to be removed or added is needed. If this selection is made, an option for removal from or addition to the trap must be chosen from some menu or button.

  Removal of a trap could be done by selecting it (by a mouse click or a name) and issuing a remove command.
- **Generate diagram part(s): (See S1.1)**
- **Apply generic editing operation:** Once building blocks have been created, they are handled like any other graphical primitive. The plug semantics take care of the coupling of different building blocks. Apart from the commands to (dis)connect plugs, the commands are thus very simple and can be found in any graphical editor.

  It is important to note that in the existing prototype, the integration of these commands and the building block and plug concepts is not yet fully accomplished.

C. **Save diagram:** When closing the diagram, it should be saved explicitly. When changes to a diagram have been made, the system could inform the user if the changes should be committed to the underlying database. This way, it is possible to dismiss all changes.

S1.1 System notes on: Generate diagram part(s)

The use case here describes how one diagram part can be generated. In an actual system there may also be a need of features that allow the user to generate several diagram parts at the same time. To do this it is necessary to have some kind of mechanism to specify *bulk generation*. For each part that is generated it is necessary to be able to specify *source*, *link* and *destination*.

Careful thought should be given to the decision whether features for bulk generation are necessary, how strong these features need be, and what form the specification should take.

Because, there are only a limited number of types of generations possible, it might be possible to provide a step by step interface for all of them. Such an interface should provide the same steps presented here, but it would be rather strict on the selections of building blocks: only building blocks for the specified generation-type would be available for selection.

A. **Select building block(s) to generate from:** To select building blocks (or even whole diagrams) the browser part of the system should be used to find the building block(s) and select them in some way.

   The system could check if it is possible to generate from the selected building blocks. Maybe only useful selections should be allowed.

B. **Initiate generating of diagram part(s):** The diagram parts that are generated should always be placed in the current diagram, unless the user specifically specified otherwise.

S2 System notes on: Analyse diagram

This use case is used to analyse a diagram.

A. **Select diagram:** Selecting a diagram is done in the familiar windowing interface as described in S1 or by simply clicking in its window (if its already open).

B. **Initiate analysis:** The system analyses the specified EER diagram.

   The system can have several possibilities to feed the information acquired from the analysis of the diagram back to the user. We have come up with the following notions:

   - Some kind of graphical feedback: building blocks which are involved in some problem light up;
   - A textual description of all the errors and the building blocks involved in them put together in one file;
   - A user-interface in which all errors can be displayed and from which the involved building blocks can be reached for edit.

   Each error will have some static information linked to it, which will describe the possible problem and what could be done to fix the error. This information should be available from the different feedback options.

S3 System notes on: Browse SOCCA model database

A. There are three kinds of browsing operations:

   - **Change view on diagram:** Refer to S3.1 for system notes on this.
   - **Search within chunks:** Refer to S3.2 on the next page for system notes.
   - **Browse through chunks:** Refer to S3.3 on page 77 for system notes.
   - **Follow links:** Refer to S3.4 on page 77 for system notes.

S3.1 Change view on diagram

The view operations should be embedded in the editors for the different diagrams. No special interface is needed for them. Browsing in this sense is nothing more than adding some functionality to the editor.

A. **Select diagram:** A diagram is selected by making the window active.

B. A few notes on the different view operations:

- **Zoom diagram:**: This can be done in several ways (all known from well known design programs), like specifying a zoom-percentage, specifying an area on which to zoom, etc. For all there exist well known interfaces.
- **Move the view port:** Again, there are several ways to accomplish this. The one most used is the implementation by scrollbars, but also the dragging of the view port can be done.
- **Select parts to view and hide:** This is no standard option for editors. A window could be provided to select the types of building blocks to hide and to view. If it should be possible to select the building blocks individually, this should be done in the editor directly, for example as a right mouse click option.

S3.2 Search within chunks

The search engine used for searching the SOCCA model must recognize all objects by means of strings. So, diagrams and chunks are recognized by their name and states and transitions by their labels, etc. Its interface should include all options specified in the following sections.

A. **Specify search options:** The search options must be specified in an easy to comprehend interface.

   - **Specify text to search for:** No comments.
   - **Specify range of search:** Refer to S3.2.1 for system notes.
   - **Specify type of search:** Refer to S3.2.2 for system notes.

B. **Initiate search:** The search engine will be fed with the options specified and will try to come with results. The results can be presented in several ways. This is an option for the user. It has not been added to the use case, because it is just a way of presenting results, not a way of getting them. Apart from a textual result, graphical feedback could be provided: diagrams and chunks searched for could lit up in the exploring-interface described earlier and building blocks could lit up in diagrams.

S3.2.1 Specify range of search

- **Specify chunks to search in:** To specify the range of the search some kind of exploring interface can be used. This time it must be possible to select branches of the chunk-structure. For example, by means of check-boxes.
- **Specify diagram types to search in:** There are only several types of diagrams used in SOCCA, so it cannot be a problem to allow the user to select or deselect the types.
- **Specify building block types to search in:** A useful interface for selecting or deselecting types of building blocks searched for must be created. Note, that if some diagram types are deselected in the previous option, the building block types, which are unique for that diagram type will not have to be included in this interface.

S3.2.2 Specify type of search

The options in for the types of search are all toggle options. They could be include in an interface as simple check boxes, which can be on or off.

S3.3 Browse through chunks

A user-interface should be supplied for exploring the SOCCA-model. This user-interface must supply a clear way to go from one part of the SOCCA-model to another. Because the chunk-mechanism provides a kind of hierarchical structure within a SOCCA-model, it can be used for an exploring interface resembling a file-browser, with the chunks resembling directories and the diagrams resembling files. This way all options specified can be implemented as mouse-clicks and buttons on the browser.

S3.4 Follow links

Because the diagrams are linked to each other by means of links, these links could be used for exploring to. In the file-like browsing interface described in S3.3, but also from within other diagrams, when the user wants to look at a linked diagram.

A. **Select diagram:** In the model browser this can be accomplished simply by clicking on the name of the diagram, if it is visible in the browser interface. If it is not, the chunks should be browsed and expanded to make the name visible.

   For open diagrams (which have their own window) the selection is merely making their window active. For example, by a mouse click.

B. **Narrow link choices:** Refer to S3.4.1 for system notes.

C. **Select link to follow:** In both the model browser and open diagrams, a list of possible links to follow should be displayed. These links are the links linked to the selected building block. If no building block is selected, all links regarding the diagram are displayed.

   A selection can be made by a simple mouse click on the link which should be followed. If only one link can be associated with a building block or diagram, a simple double click on it can be used to select the link.

D. **Initiate following of link:** This is done automatically by the choice of the link.

S3.4.1 Narrow link choices

- **Select building block:** In the model browser it could be possible to display all the building block names, if the current node is a diagram. So, if a diagram is clicked on in the model browser, it is made the current node in the browser and the building blocks are displayed. This way, the building blocks from the diagram can be selected.

  In an open diagram the building blocks are visible, so a click with the mouse to select the building block could be enough.

- **Select target type:** Because there are a limited number of possible types, it is possible to make these selections from some kind of list.

Chapter 6

The ModelIntegrator use cases

6.1 Introduction

This chapter contains the use cases for the actor ModelIntegrator. As mentioned in chapter 4, this chapter contains four sections: the introduction, the use cases, the discussion on the use cases and the system notes.

S1 The use cases

The actor ModelIntegrator is responsible for chunking and linking diagrams in order to create sub models: small pieces of SOCCA models which contain all of the three perspectives. To do this, use case for managing links and chunks are provided. The ModelIntegrator must also be able to analyse and check these sub models (chunks), for these purposes other use cases need to be provided. Like, the DiagramBuilder, the ModelIntegrator must also be able to browse through the SOCCA model database. The same use case as the one presented in the previous chapter can be used for this.

A short description of the use cases is given in the following sections.

S1.1 The use case: Manage chunks

This use case offers all functionality regarding the creation, editing and deletion of chunks. General chunk as well as special chunks (referring to special groups of diagrams) can be created. Diagrams but also chunks can be added and removed from the chunks.

S1.2 The use case: Manage links

With this use case, the ModelIntegrator can create and remove links between chunks, diagrams and building blocks. Only some types of links can be created. The use case also provides the possibility to specify permissions and propagations with each link.

S1.3 The use case: Analyse chunk

This use case provides the ModelIntegrator with the possibility to analyse chunks. In the analysis of the chunks, all possible and required links are taken into consideration. Therefore, this analysis must be able to check whether the chunk is a correct (part of a) SOCCA model or not.

S1.4 The use case: Simulate chunk

The use case “Simulate chunk” should be able to do a simulation run on the chunk. Simulation was not in the scope of this thesis, so the use case is not presented here. It is only mentioned in order to make the picture somewhat more complete.

S1.5 The use case: Browse SOCCA model database

This use case must be the same as the use case “Browse SOCCA model database” for the DiagramBuilder. Refer to section 5.1.1.3 for a short description.

S2 List of use cases and sub use cases

With the help of the following list, it is easy to find the use cases and sub use cases in this document. After each (sub) use case, three page numbers are listed. The first page number is that of the (sub) use case. The second page number is that of the discussion on the particular (sub) use case. And the third is that of the system notes on the particular (sub) use case. Sometimes no page number is listed. In that case no discussion or system notes are given on the (sub) use case.

number	**name**	use case page	dis-cus-sion page	system notes page
1	**ModelIntegrator use case: Manage chunks**	81	85	91
1.1	**Specify chunk**	81	85	91
1.1.1	Open new (empty) chunk	81	86	-
1.1.2	Add diagram(s) to chunk	82	86	92
1.1.3	Remove diagram(s) from chunk	82	86	92
2	**ModelIntegrator use case: Manage links**	82	87	92
2.1	**Specify link**	82	87	93
2.1.1	Specify node for link	82	87	93
2.1.2	Specify link permissions	82	88	93
2.1.3	Specify link propagations	83	88	94
3	**ModelIntegrator use case: Analyse chunk**	83	88	94
	ModelIntegrator use case: Browse SOCCA model database	64	69	75

Section 6.2

ModelIntegrator use cases

- ModelIntegrator use case: Manage chunks; (**See U1**)
- ModelIntegrator use case: Manage links; (**See U2 on the next page**)
- ModelIntegrator use case: Analyse chunk; (**See U3 on page 83**)
- ModelIntegrator use case: Browse SOCCA model database. (**See U3 on page 64**)

U1 ModelIntegrator use case: Manage chunks

- Specify chunk; (**See U1.1**)
- Remove chunk.

U1.1 Specify chunk

A. Open chunk.

  - Open new (empty) chunk; (**See U1.1.1**)
  - Open specific chunk;
  - Copy specific chunk.

B. Change chunk.

  - Add diagram(s) to chunk; (**See U1.1.2 on the next page**)
  - Remove diagram(s) from chunk; (**See U1.1.3 on the next page**)

U1.1.1 Open new (empty) chunk

A. Specify type of chunk;

  - Open new general chunk;
  - Open new SOCCA model chunk;
  - Open new data perspective chunk;
  - Open new behaviour perspective chunk;
  - Open new partition chunk;
  - ...

B. Initiate creation.

U1.1.2 Add diagram(s) to chunk

- Add diagram to chunk;
- Add chunk to chunk;
- Add chunk as diagrams to chunk.

U1.1.3 Remove diagram(s) from chunk

- Remove diagram from chunk;
- Remove chunk from chunk.

U2 ModelIntegrator use case: Manage links

- Specify link; (**See U2.1**)
- Remove link.

U2.1 Specify link

A. Open link.

- Open new link;
- Open existing link;

B. Change link;

- Specify node for link; (**See U2.1.1**)
- Specify link type;
- Specify link permissions; (**See U2.1.2**)
- Specify link propagations. (**See U2.1.3 on the next page**)

C. {*Link fully specified*} Initiate link.

U2.1.1 Specify node for link

A. Select which node of link to specify;

B. Narrow down choices.

- Select chunk;
- Select diagram;
- Select link type;

C. Select node;

U2.1.2 Specify link permissions

- Set analysis allowance;
- Set browse allowance;
- ...

U2.1.3 Specify link propagations

- Set change propagation;
- ...

U3 ModelIntegrator use case: Analyse chunk

A. Select chunk;

B. Initiate analysis.

Section 6.3

Discussion on ModelIntegrator use cases

- **ModelIntegrator use case: Manage chunks:** See D1 for discussion.
- **ModelIntegrator use case: Manage links:** See D2 on page 87 for discussion.
- **ModelIntegrator use case: Analyse chunk:** See D3 on page 88 for discussion.
- **ModelIntegrator use case: Browse SOCCA model database:** See D3 on page 69 for discussion.

D1 Discussion on: Manage chunks

The definition of chunks and the concept of chunking is explained in section 3.4.2. The choice-chunks explained there will not return in these use cases. These choice chunks are used to resolve clashes in a multi-user environment and version management. The choice has been made to leave out these problems till a later date.

Chunks are the responsibility of the ModelIntegrator. So, he or she must be able to create, remove and maintain them.

- **Specify chunk:** Refer to D1.1 for the discussion on this.
- **Remove chunk:** A chunk can be deleted. This does not imply that the diagrams and chunks, that are present in the chunk are deleted. This should never be done by removing chunks. Diagrams and chunks must be removed explicitly themselves.

D1.1 Specify chunk

A. **Open chunk:** To change a chunk it first has to be opened. If a new chunk is created, it is handled as a chunk with no diagrams in it and then opened. For a new chunk a name has to be provided.

   An option to open a copy of some specific chunk is also provided. A new name has to be provided for this chunk, but the hierarchy is exactly the same as in the original chunk.

   There are some groups of diagrams, which have a special purpose in SOCCA models. These groups all have their own type of chunk. Such a chunk can only contain diagrams which fulfill some requirements. Refer to D1.1.1 on the next page for the types of chunks available.

B. Change chunk.

- **Add diagram(s) to chunk:** Refer to D1.1.2 for discussion on this.
- **Remove diagram(s) from chunk:** Refer to D1.1.3 for discussion on this.

D1.1.1 Open new (empty) chunk

There are several types of chunks available, these are:

- *General chunks:* General chunks can contain all types of diagrams and any number of them. This is the most flexible type of chunk.
- *SOCCA model chunks:* SOCCA model chunks should be used for grouping all diagrams of a SOCCA model together. A useful way to do this, is by grouping the data perspective chunk and the behaviour perspective chunk for that SOCCA model.

  A SOCCA model chunk can also contain all types of diagrams and any number of them. Except, that only one classdiagram is allowed in such a chunk.
- *Data perspective chunks:* Data perspective chunks should be used for grouping the diagrams from some data perspective of some SOCCA model. Only one classdiagram and any number of import/export diagrams can be grouped in such a chunk.
- *Behaviour perspective chunks:* Behaviour perspective chunks are chunks for grouping diagrams from behaviour perspectives of SOCCA models. Any number and any type of STDs can be grouped in a chunk of this type. Also, subprocesses can be (or rather should be) added to this type of chunks. For the subprocesses, several partition chunks could be included in the chunks.
- *Partition chunk:* Partition chunks group subprocesses of one partition in SOCCA.

  Partition chunks are different from all the other types of chunks. This, because they are also used as being partitions and thus links are created to these chunks. Therefore, whenever a partition is mentioned in the rest of this document, we are talking about a partition chunk.
- *...:* Maybe other types of chunks could be useful too. If discovered, they will be added here later on.

D1.1.2 Add diagram(s) to chunk

To an open chunk, diagrams can be added. Apart from diagrams, chunks can be added to other chunks as well. This way a hierarchy of chunks can be created. This hierarchy can then be used for browsing. An example for a useful hierarchy could be the following: a SOCCA model chunk grouping a data perspective chunk and a behaviour perspective chunk which group the diagrams of that SOCCA model. The system does not enforce such a hierarchy, it is just a possibility.

To allow the ModelIntegrator to add diagrams and chunks to some chunk, the system must provide a possibility to select them. The SOCCA model browser could be used for this.

To make the interface more flexible, the ModelIntegrator is allowed to include a chunk as diagrams into a chunk. All the diagrams from the selected chunk are then added directly to the chunk. So, the hierarchy of the selected chunk is not copied.

To chunks of some types only some types of diagrams can be added. The ModelIntegrator must not be able to add diagrams of types, which are not allowed (anymore).

D1.1.3 Remove diagram(s) from chunk

It should be possible to remove diagrams or whole chunk hierarchies from chunks. Compare this with the possibilities to remove files and directories from some directory in a file browser. If a diagram is removed from a chunk, the diagram does not disappear from the database. Chunks only provide links to diagrams, not diagrams themselves.

D2 Discussion on: Manage links

This use case involves the managing of links. Links are also the responsibility of the ModelIntegrator. All things involved with the managing of links should be done from here.

- **Specify link:** For discussion, see D2.1.
- **Remove link:** The selected link will be removed from the database.

D2.1 Specify link

A. **Open link:** A link has to be selected in order to further specify or remove it. A new link can be created by the option "**Open new link**". An existing link can be selected by the option "**Open existing link**".

The choice of the term 'open' has been made to show the correspondence between links and the other concepts, like chunks and diagrams. All these concepts have a similar step in their use cases, namely that of opening a new or existing one.

B. Change link;

- **Specify node for link:** To specify a link, the system first needs to know what will be linked. Links can involve diagrams, building blocks and even groups of diagrams. A node of a link is thus either a diagram, a building block or a group of diagrams.
  To specify a link, all nodes of it need to be specified. Almost all links involve two nodes, but at least one involves three nodes. This is why we have made this a repeatable option. Note, that because of this it is possible to skip this step, which could result to no nodes being defined. This should not be possible for a new link, but an existing link does not have to have its nodes specified again.
  See D2.1.1 for more discussion on this topic.
- **Specify link type:** The type of link is the type of dependency between the nodes of the link. For each type of link, some kind of dependency between its nodes exists. This dependency can involve equality of labels, but also more complex dependencies can be expected.
  If the selection of the nodes does not fully specify the type of link, it has to be provided by the user. However, in almost all cases the selection of the nodes specify the type of link.
- **Specify link permissions:** See D2.1.2 on the next page for discussion.
- **Specify link propagations:** See D2.1.3 on the next page for discussion.

C. {*Link is fully specified*}**Initiate link:** Link is actually created (or changed) by this option. In order to do this, the type of link and the nodes of the link need to have been provided.

A possible check of consistency could follow after this.

D2.1.1 Specify node for link

A. **Select which node of link to specify:** First, the ModelIntegrator has to select which node of the link he or she wants to specify. If the type of nodes are available, such a selection is a selection of the node type. If it is not, the selection only states if some node needs to be overwritten or if a new one should be added.

B. **Narrow down choices:** The system should assist the user in linking two (or more) nodes to the fullest extent. Therefore it should assist the ModelIntegrator in selecting the nodes. This can be done by showing: the links needed, the links possible, the links created, etc. It should also provide a user-friendly way to choose the nodes involved. This is done in this step. Several options to limit the choice of node are given below.

- **Select chunk:** The choice is limited to the chunk selected.
- **Select diagram:** The choice is limited to the diagram selected; Only building blocks of the selected diagram can be chosen from.
- **Select link type:** By explicitly selecting a link type, the node types are defined too. So, if a link type is provided, the system can limit the choice to the allowed nodes only.

C. **Select node:** The node still needs to be selected.

D2.1.2 Specify link permissions

Some sorts of permissions can be set for links. Although it is not yet clear what types of permissions should be possible, some ideas are presented below. What permissions should be implemented is a decision, which still has to be made.

The general idea behind the permissions is, that some users are allowed to do some operations on or via the links, and some are not.

- **Set analysis allowance:** Allow or disallow analysis with this link.
- **Set browse allowance:** Allow or disallow the use of this link with browsing.
- ...

D2.1.3 Specify link propagations

Links always denote a dependency between two or more parts of a SOCCA model. Therefore, a change at one side of the link could (and perhaps should) enforce a change at the other side(s) of a link. However, such a change is not wished for in every case. Therefore, we need to provide the ModelIntegrator with options to configure what happens if one side of the link is altered.

- **Set change propagation:** The ModelIntegrator must be able to define if a change is propagated through the link. Maybe a warning is wished for, or should this change be forbidden. Note, that a such a propagation setting for a link can be different for different directions. It might be useful to have a link propagate changes from one side to the other, but not the other way around. For example, if a piece of model is added to a large part, changes in the little piece should not enforce changes in the large part. The other way around could however be useful.
- ...

D3 Discussion on: Analyse chunk

Analysis of chunks is like checking if the chunk is a correct SOCCA model itself. So, the analysis engine checks for all dependencies which should exist between the diagrams in the chunk. These dependencies can be extracted from the meta model for SOCCA models and involve links. For some diagram, it is possible that some required link does not exist in the chunk. This can be for two reasons:

- It was indeed forgotten. This is indeed a real error.
- The link was created between two diagrams of which only one is present in the chunk. This is a special kind of error: although the chunk is not a SOCCA model on it's own, it could be useful to check all possible dependencies between only the diagrams in the chunk. Therefore, this error must be handled differently.

When analysing whether a chunk is correct, the diagrams of that chunk must also be checked for layout. This can be done with the help of the use case “Analyse diagram” presented in section U2 of chapter 5.

A. **Select chunk:** To analyse a particular chunk, it first must be selected.

B. **Initiate analysis:** Starts the analysis.

Section 6.4

System notes on ModelIntegrator use cases

- **ModelIntegrator use case: Manage chunks:** See S1 for system notes.
- **ModelIntegrator use case: Manage links:** See S2 on the next page for system notes.
- **ModelIntegrator use case: Analyse chunk:** See S3 on page 94 for system notes.
- **ModelIntegrator use case: Browse SOCCA model database:** See S3 on page 75 for system notes.

S1 System notes on: Manage chunks

Because the chunk hierarchy is used in the SOCCA model browser, it could be useful to use the same browser to add, remove and edit chunks. Compare this with the idea of managing directories in a file-browser.

Some kind of button or menu option from the browser could allow the user to add a new chunk in the current one or at the top of the hierarchy. The same goes for removing some chunk. To edit a chunk, diagrams and chunks can be dragged to it (to add them) or deleted from it. The diagrams and chunks could be displayed in another window of the browser or in a whole new browser all together. Remember that a chunk only contains links to diagrams and not the diagrams themselves. If a diagram is added to a chunk, in reality a link to that diagram is added to it. This way the diagrams can all be stored at one place (or in one database) and no careless behaviour will accidentally change them.

- **Specify chunk:** (**See S1.1**)
- **Remove chunk:** No further comments.

S1.1 Specify chunk

A. Open chunk.

- **Open new (empty) chunk:** A new chunk can be created in the current one by pressing a 'new chunk'-button or by a menu option. The chunk is created in the current chunk, if the SOCCA model browser is open. The browser should make the new chunk the current node in the hierarchy. If the browser is not open, the chunk will be added at the top level of the hierarchy and the browser will be opened and make the new chunk the current node.
  There are several types of chunks. The type for a new chunk must be specified as well.

- **Open specific chunk:** A new chunk can be opened by making it the current node in a browser. This should also be possible by typing the name of the chunk.
- **Copy specific chunk:** If a specific chunk has been selected, an option to copy it must be available. This option could be implemented as a button or a menu-item. A new name has to be provided for the new chunk.

B. **Change chunk:** Chunks can be specified by adding (links to) diagrams and chunks to them or by removing (links to) diagrams and chunks from them. Refer to the following sections for more specific system notes.

- Add diagram(s) to chunk; (**See S1.1.1**)
- Remove diagram(s) from chunk; (**See S1.1.2**)

S1.1.1 Add diagram(s) to chunk

- **Add diagram to chunk:** Can be done by dragging the name of the diagram to the chunk. The name of the diagram can come from another part of the browser or a whole other window. Compare this with dragging a file to some directory in a file browser. Although in a file browser this would result in a copy or a move most of the time, in the SOCCA model browser, this would have to result in a link to the specified diagram.

  Only diagrams which are admitted by the chunk type of the chunk can be added to the chunk and only in the numbers in which they are allowed. The system thus has to check whether the type of diagram is allowed for this chunk and if there are not to many of the diagram type in the chunk already (this is only a problem with classdiagrams, for now).

- **Add chunk to chunk:** Like with the diagrams, the selected chunk can just be dragged to the chunk in order to add it. Here too, only a link to the chunk is used. Original chunks always resite in some database or at the highest level of the hierarchy. This too is for safety.

  The system should check if the chunk which is added, does not contain diagrams which are not admissible to the chunk edited as its leaves. If this is the case, the chunk cannot be added as a whole. It could be useful then, to add only the diagrams and hierarchy admissible. Although it should also be possible to dismiss the operation totally.

- **Add chunk as diagrams to chunk:** As with both previous options, the diagrams cannot be added, if the chunk type won't allow it. The ModelIntegrator might choose to add only the diagrams allowed or none at all.

S1.1.2 Remove diagram(s) from chunk

Diagram and chunks, or rather: links to diagrams and chunks, can be removed from chunks by dragging them from the chunk (the current node in the browser) to outside the browser. Another way to remove diagrams and chunks from chunks is to select them and choosing an option: delete. This option could be in some menu from the browser or associated with a combination of keys on the keyboard.

S2 ModelIntegrator use case: Manage links

- **Specify link:** See S2.1 on the next page for more specific system notes.
- **Remove link:** The system should communicate the consequences of the removal of the selected link to the ModelIntegrator. Confirmation should be required for this action.

S2.1 Specify link

A. **Open link:** The system should provide ways to select existing links and creating new ones. Like similar concepts, the interface must be able to display the links in order to allow the ModelIntegrator to select from them. For a new link a separate button could be created in this interface.

If a useful way for displaying links can be produced, such an interface would again resemble the usual interface for opening, creating and saving. Only instead of files, diagrams or chunks, the issue here is links. A useful way to display links is to display the type of link and the nodes connected to it.

B. Change link.

- **Select which node of link to specify:** If the types of nodes, which need to be specified are known, the system can present a view of the link involved. This view would involve the nodes. A click on one of them, could indicate which node is going to be specified. Such a view can also be created for a general link.

  How the view on one link looks exactly is not important. It must provide, however, a nice view on the link and its nodes and a way to select these nodes.
- **Specify node for link:** See S2.1.1 for more specific system notes.
- **Specify link type:** There are only a limited number of link types. So, it is possible to select the link type from a list.

  If nodes have already been specified for the link, only a limited number of link types will be possible. This, because the link type and the type of its nodes are strongly related. The system should support this relation to the fullest extent. For example, by limiting the choices for a link type as a result of the specified nodes for some link. Limiting the choice for nodes of the link as a result of a defined link type is possible as well.
- **Specify link permissions:** See S2.1.2 for more specific system notes.
- **Specify link propagations:** See S2.1.3 on the next page for more specific system notes.

C. {***Link fully specified***} **Initiate link:** A link is fully specified, if all of its nodes and its type have been specified. The permissions and propagations must also be set for each link, but some default value for both concepts could be used, if no permissions or propagations have been specified.

S2.1.1 Specify node for link

The types of nodes of a link are strongly related to the type of the link. For a specific type of link, the types of the nodes are already determined. Therefore, if for a link the type has already been specified, the choices for the nodes can be limited drastically. However, if such a type has not been specified, the step “**Narrow down choices**” allows for the user to limit the possible choices.

The interface used for the selection of the nodes would be similar (or perhaps equal) to the browser interface. This interface provides way to focus on chunks and diagrams and allows the user to select chunks, diagrams and building blocks. Building blocks and diagrams can of course also be selected from an open diagram on screen. This can be useful for linking the diagram (and its building blocks), while the user is working on it.

S2.1.2 Specify link permissions

These are nothing more than options which can be set for each link in particular. For every type of allowance only a limited number of possibilities exists. A well-known way to choose such a possibility is by radio buttons. These radio buttons should be available on a properties sheet for each link.

S2.1.3 Specify link propagations

Like in system notes on sub use case S2.1.2 on the preceding page.

S3 System notes on: Analyse chunk

As discussed in section D3, there should be support for various kinds of errors and warnings. Some ideas on presenting these errors and warnings to the user are:

- **Graphical indication:** Show the diagram(s) involved and points out where something is missing. This by coloring the constructs involved.
- **Textual indication:** Tell the user what is wrong textually. This can more clear as the system can tell you what the problem is.
- **Advise:** Advise the user on how to correct the problem, if that is necessary. Indicate how severe the problem is.

A. **Select chunk:** A chunk must be selected. This can be accomplished by selecting it in the browsing interface or chunk editor.

B. **Initiate analysis:** Accomplished by selection of a menu-item or a button.

Chapter 7

Results and future work

This thesis has lead to various results. These results are described in following sections.

7.1 Inventarory of the functionality

The goal of this thesis was to make an inventory of the required functionality for a future environment for creating and using SOCCA models, with the emphasis on the creation of SOCCA models (see section 1.1). This has resulted in a description of the required functionality in several use cases presented in chapters 4 till 6 and a meta model describing the dependencies between certain model fragments and diagrams in SOCCA. The use cases represent all of the major components which can be expected in the future system. This has been achieved by our working method which is presented in section 7.2.

The use cases describe the required functionality to the level on what a user must be able to do, not on how he/she should do it. Ideas on how to implement some of the expected functionality are presented as well. The description of the required functionality is rather complete, but decisions on some issues should still be made when the environment is going to be implemented. We have tried to point out these issues and sometimes we have also provided some possible ideas and solutions.

Together with the meta model, the use cases provide a basis for the a future meta model for the complete SOCCA environment. That meta model will be probably a SOCCA model. Refer to section 7.5 for more discussion on this.

7.2 Working method

Another result of this thesis is an idea for a working method for creating an inventory of the functionality of systems that will be build. Most of the time, when a system's expected functionality is inventorized, the process is started with an informal method of modelling, such as use cases. This is not strange, as most of the time nothing formal on such an environment can be stated.

In our case, however, something formal could be said about the environment. As the environment should be responsible for creating (and using) SOCCA models, it was decided to first describe accurately what dependencies exist between model fragments and diagrams in SOCCA. An informal description of SOCCA models was already at hand. If this informal description could be converted to a formal description, it was likely to give us more insight in the required components of the system. So, we first created a meta model for SOCCA models in chapter 2. This meta model was in the form of a class diagram with separate constraints written in Z.

As expected, the meta model provided us with some concepts which would have to be supported by the future environment. Some components could be identified in order to offer this support. These components were introduced in chapter 3. Other expected components were also introduced in that chapter.

As the expected components were known, it was now easier to make an inventory of the functionality of the environment with the use cases. Because some components were identified with some special concepts from the meta model, their functionality also depended largely on these concepts. The ideas for the other components' functionality were largely fed by comparable components of other environments. The functionality was described in the use cases in chapters 4 till 6. Also ideas on possible implementation were presented there.

The idea of starting with a formal way of modelling in order to have a starting point for describing the required functionality informally, might seem strange. On the other hand, if it is possible to define some particular concepts which the environment should support with the help of a formal model, then the required functionality can be described more accurately and the informal description is more precise. A more accurate informal description of the required functionality of a system is likely to lead to a more efficient implementation phase for that system. Simply, because it is more accurate.

7.3 Structured use case description

In the effort to describe the functionality of the system in the use cases, we stumbled upon some difficulties in expressing some often reoccurring constructs. In the original way to describe the use cases, all these constructs would normally be described in text only. As this resulted in description which were difficult to read and understand, we have developed a notation to structure use case descriptions and facilitate the use of some often reoccuring constructs. This notation was introduced in chapter 4.

7.4 Basis for implementation

The ideas on how to implement the functionality in the use cases are provided in the system notes sections of chapters 5 and 6. Besides these ideas, the meta model provides a basis for the internal representation and database of the future system. Therefore, this document provides a basis for implementation.

However, at the moment only ideas on implementation and a basis for the internal representation is given. These ideas and the meta model should be expanded in a much more detailed description of the requirements for the system. Such a description is usually called a Requirements Definition Document. Look at section 7.5 for more on this topic.

7.5 Future work

This document only provides an inventory of the required functionality for a SOCCA environment. It does not provide an exact description of the requirements of a future system, but it presents ideas on the possible functionality and concepts which will be important in the system. In order to describe the requirements exactly, first some decisions on the required functionality should be made: there are points which need discussion, but also it must be decided that the presented functionality is indeed the functionality wished for. It is plausible that some functionality will have to be added and other functionality is not needed (or temporarily discarded).

If the functionality is precisely bounded, a meta model which describes the future system can and should be made. It would be logical to use SOCCA for this purpose.

This results in fully specified requirements for the system (a Requirements Definition Document), after which implementation can be started.

Appendix A

The meta model

$$\begin{aligned} \forall u : \text{USES};\ c : \text{CLASS};\ o : \text{OPERATION}\ \mid \\ (u\ \textbf{calls}\ o) \bullet \\ (u\ \textbf{to}\ c) \Leftrightarrow (o\ \textbf{part-of}\ c) \end{aligned} \tag{A.1}$$

$$\begin{aligned} \forall cl : \text{CLASS};\ cd : \text{CLASS DIAGRAM};\ ie : \text{IMPORT/EXPORT DIAGRAM}; \bullet \\ (\exists dp : \text{DATA PERSPECTIVE} \bullet (cd\ \textbf{part-of}\ dp \wedge ie\ \textbf{part-of}\ dp)) \Leftrightarrow \\ ((cl\ \textbf{part-of}\ cd) \Leftrightarrow (cl\textbf{part-of}\ ie)) \end{aligned} \tag{A.2}$$

$$\begin{aligned} \forall o : \text{OPERATION};\ c : \text{CLASS} \bullet \\ (o\ \textbf{part-of}\ c) \Leftrightarrow \exists es : \text{EXTERNAL STD};\ t : \text{TRANSITION} \bullet \\ t\ \textbf{part-of}\ es \wedge es\ \textbf{specifies}\ c \wedge t\ \textbf{refers-to}\ o \end{aligned} \tag{A.3}$$

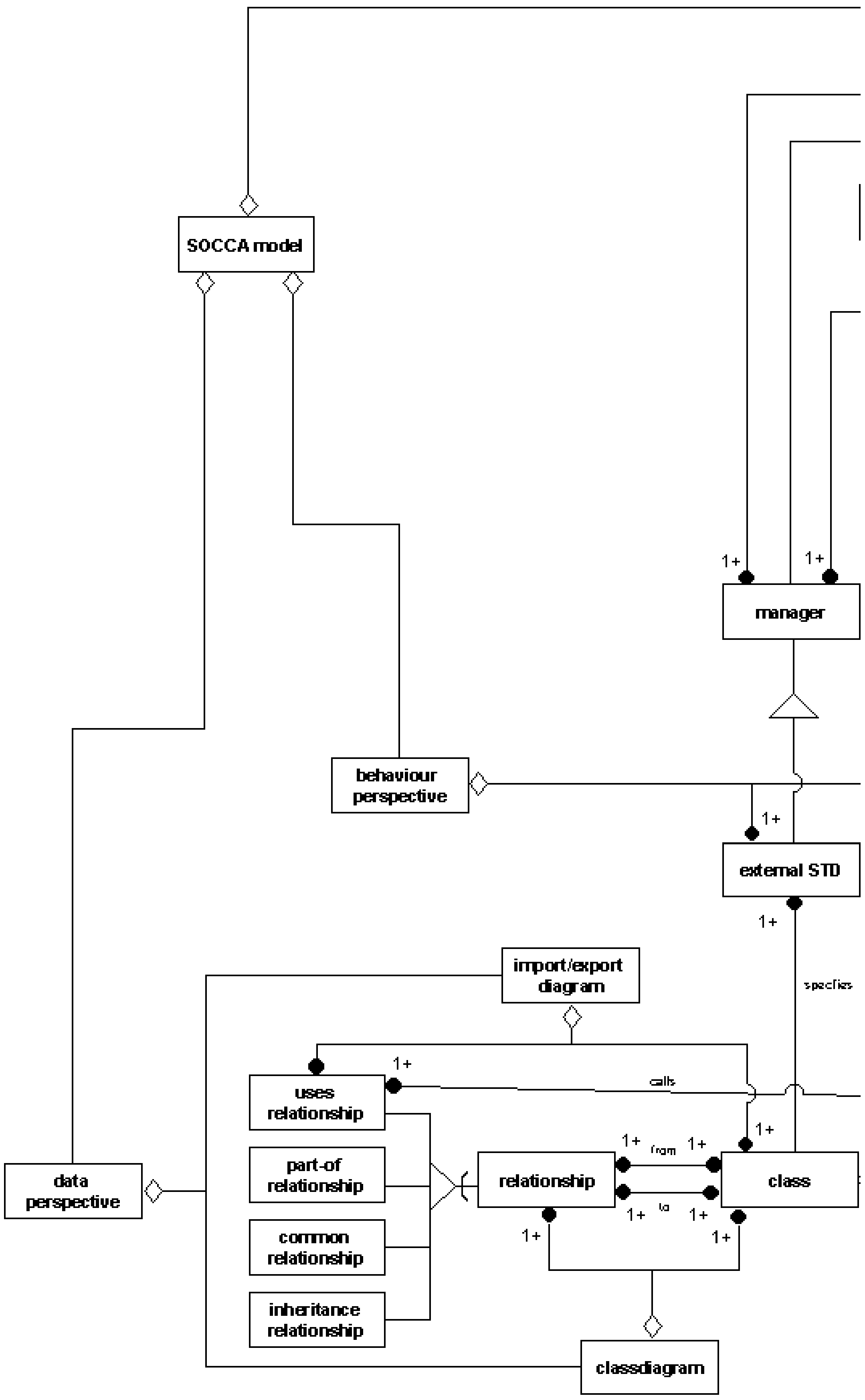
$$\begin{aligned} \forall o_1, o_2 : \text{OPERATION};\ c_1, c_2 : \text{CLASS} \mid (o_1\ \textbf{part-of}\ c_1 \wedge o_2\ \textbf{part-of}\ c_2) \bullet \\ (\exists u : \text{USES RELATIONSHIP} \bullet (u\ \textbf{from}\ c_1 \wedge u\ \textbf{to}\ c_2 \wedge u\ \textbf{imports}\ o_2) \Leftrightarrow \\ (\exists is : \text{INTERNAL STD};\ t : \text{TRANSITION} \bullet \\ (is\ \textbf{specifies}\ o_1 \wedge ct\ \textbf{part-of}\ is \wedge t\ \textbf{refers-to}\ o_2))) \end{aligned} \tag{A.4}$$

$$\forall p : \text{PARTITION};\ \exists_1 e : \text{EMPLOYEE} \bullet e\ \textbf{manages}\ p \tag{A.5}$$

$$\begin{aligned} \forall m : \text{MANAGER};\ s : \text{STATE} \mid \\ s\ \textbf{part-of}\ m \bullet (\exists sp : \text{SUBPROCESS} \bullet s\ \textbf{prescribes}\ sp) \end{aligned} \tag{A.6}$$

$$\begin{aligned} \forall s : \text{STATE};\ m : \text{MANAGER};\ p : \text{PARTITION} \mid \\ (s\ \textbf{part-of}\ m \wedge m\ \textbf{manages}\ p) \bullet \\ \exists_1 sp : \text{SUBPROCESS}(sp\ \textbf{part-of}\ p \wedge s\ \textbf{prescribes}\ sp) \end{aligned} \tag{A.7}$$

$$\begin{aligned} \forall s : \text{STATE};\ m : \text{MANAGER};\ sp : \text{SUBPROCESS} \mid \\ (s\ \textbf{part-of}\ m \wedge s\ \textbf{prescribes}\ sp) \bullet \\ \exists p : \text{PARTITION}(m\ \textbf{manages}\ p \wedge sp\ \textbf{part-of}\ p) \end{aligned} \tag{A.8}$$


SOCCA model
manager
1+
1+
behaviour perspective
1+
external STD
1+
specifies
import/export diagram
uses relationship
1+
calls
part-of relationship
common relationship
inheritance relationship
relationship
1+
from
1+
1+
to
1+
class
1+
1+
1+
data perspective
classdiagram

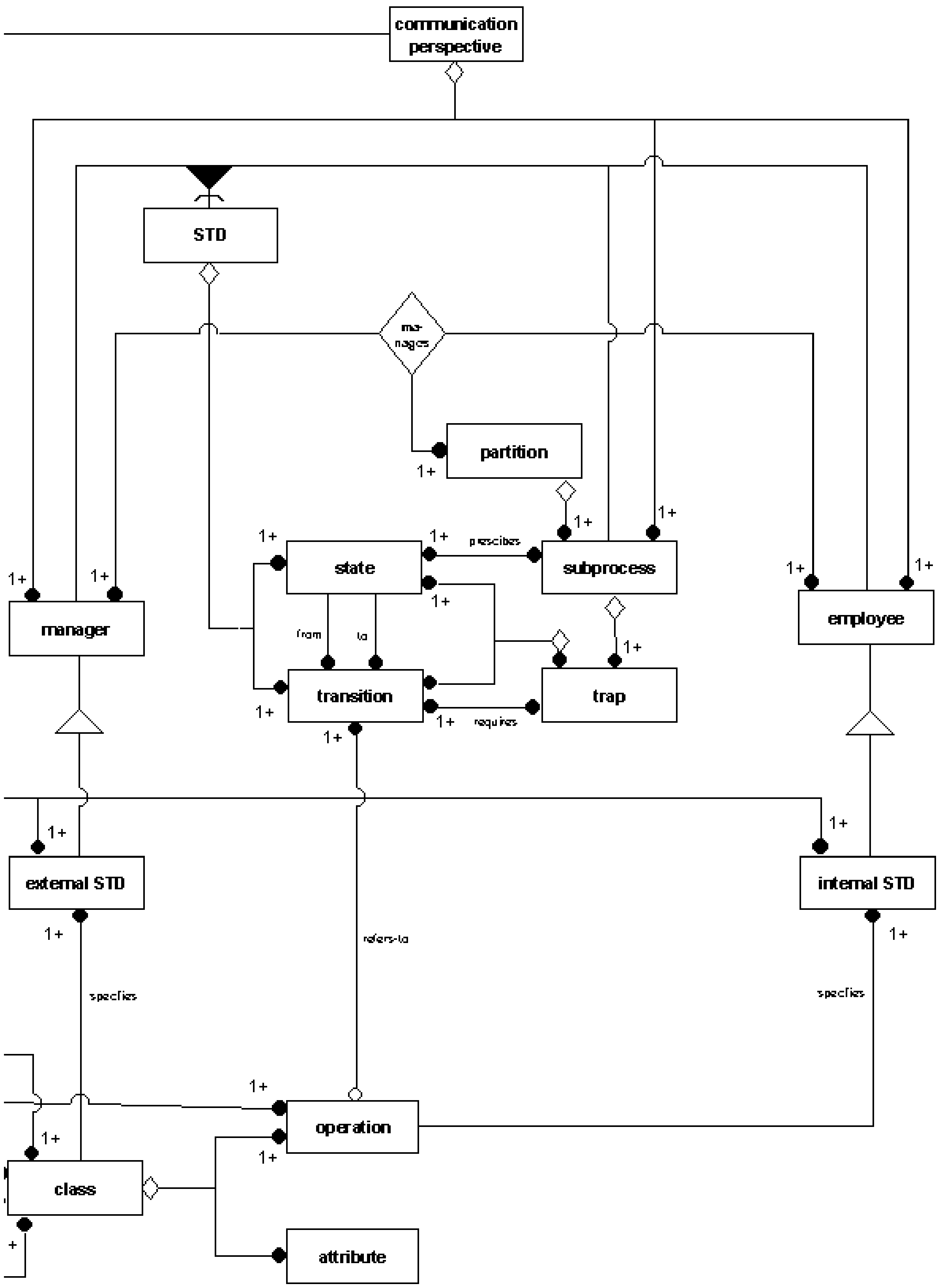


Figure A.1: The meta model.

$$\begin{aligned}&\forall m : \textsc{manager};\ t : \textsc{state};\ | \\ &\qquad t\ \textbf{part-of}\ m \bullet (\exists tp : \textsc{trap} \bullet t\ \textbf{requires}\ tp)\end{aligned} \tag{A.9}$$

$$\begin{aligned}&\forall s : \textsc{state}; t : \textsc{transition};\ tp : \textsc{trap}\ | \\ &\qquad (t\ \textbf{from}\ s \wedge t\ \textbf{requires}\ tp) \bullet \\ &\qquad\quad \exists_1 sp : \textsc{subprocess}(tp\ \textbf{part-of}\ sp \wedge s\ \textbf{prescribes}\ sp) \\ &\qquad)\end{aligned} \tag{A.10}$$

$$\begin{aligned}&\forall sp : \textsc{subprocess};\ p : \textsc{partition};\ e : \textsc{employee};\ s : \textsc{state}\ | \\ &\qquad (p\ \textbf{manages}\ e \wedge sp\ \textbf{part-of}\ p \wedge s\ \textbf{part-of}\ sp) \bullet (s\ \textbf{part-of}\ e)\end{aligned} \tag{A.11}$$

$$\begin{aligned}&\forall sp : \textsc{subprocess};\ p : \textsc{partition};\ e : \textsc{employee};\ t : \textsc{transition}\ | \\ &\qquad (p\ \textbf{manages}\ e \wedge sp\ \textbf{part-of}\ p \wedge t\ \textbf{part-of}\ sp) \bullet (t\ \textbf{part-of}\ e)\end{aligned} \tag{A.12}$$

Bibliography

[Ber96] Edward V. Berard. *Be careful with use cases*, The Object Agency, Inc., 1996, *http://www.toa.com/pub/html/use_case.html.*

[BRJ97] Booch, Rumbauch and Jacobson. *UML v1.0 Notation Guide.* Rational Software Corporation, 1997. *http://www.rational.com.*

[C$^+$93] Per Cederqvist et al. *Version Management with CVS.* Signum Support AB. 1992-1993, ftp://ftp.nluug.nl/vol/1/linux-debian/unstable/sources/rcs*.tar.gz.

[Dil94] Antoni Diller. *Z: an introduction to formal methods.* John Wiley & Sons Ltd., 1994, ISBN 0-471-93973-0.

[EG94] Gregor Engels and Luuk P.J. Groenewegen. *SOCCA: Specifications of Coordinated and Cooperative Activities*, pages 71–102. Research Studies Press Ltd. / John Wiley & Sons Inc., 1994, ISBN 0863801962 / 0471952060. Taunton 1994. Also: department of Computer Science, Leiden University technical report 94-10.

[GE95] Luuk P.J. Groenewegen and Gregor Engels. *Coordination by Behavioural Views and Communication Patterns.* In W. Schäfer, editor, *Proceedings of the Fourth European Workshop on Software Process Technology (EWSPT '95)*, number 913, pages 189–192, Noordwijkerhout, The Netherlands, April 1995. *ftp://ftp.wi.LeidenUniv.nl/pub/CS/SEIS/ewspt95.ps.gz.*

[Gro91] L.P.J. Groenewegen. *Parallel Phenomena*, 1986–91.
A series of technical reports: 86-20, 87-01, 87-05, 87-06, 87-11, 87-18, 87-21, 87-29, 87-32, 88-15, 88-17, 88-18, 90-18 and 91-19.

[Höp94] J.J. Höppener. *The Merlin Process Transactions, Specified with SOCCA.* Master's thesis, Department of Computer Science, Leiden University, 1994. *ftp://ftp.wi.LeidenUniv.nl/pub/CS/MScTheses/hoppener.95.ps.gz.*

[HW96] Antoinette Hartog and Michael Wijnakker. *The syntax of a visual language generates its editor.* Master's thesis, Department of Computer Science, Leiden University, August 1996. Internal Report 96-31.

[JEJ95] I. Jacobson, Ericsson and A. Jacobson. *The object advantage: Business process reengineering with object technology.* Addison-Wesley, 1995, ISBN 0-201-42289-1.

[Rij95] M. Rijnbeek. *Modelling a Software Process Using SOCCA.* Master's thesis, Department of Computer Science, Leiden University, 1995.

[Wul95] Alex P. Wulms. *Adaptive Software Process Modelling with SOCCA and PARADIGM.* Master's thesis, Department of Computer Science, Leiden University, April 1995, *ftp://ftp.wi.LeidenUniv.nl/pub/CS/MScTheses/wulms.95.ps.gz.*