Structuring SOCCA Models with UML-like Packages

P.J. 't Hoen* L.P.J. Groenewegen J.H.M. Dassen
I.G. Sprinkhuizen-Kuyper P.W.M. Koopman G. Engels

September 27, 1999

Abstract

This document defines UML-like packages for SOCCA (Specification of Coordinated and Cooperative Activities). The packages structure and encapsulate the classes of a (SOCCA) model.

*Please use the primary author's contact address: hoen@wi.LeidenUniv.nl

Contents

1 Introduction

Object-oriented models become more difficult to manage and develop as the number of classes increases. A well known solution to this problem is to add some form of modules to the modeling language [RBP$^+$91, GJM91, Mey97]. Modules are used for:

Structuring Modules are used to group conceptually related model elements, i.e. classes, together. In this way, the model can be cut into individual chunks. If done well, it makes a model simpler to understand and manage. As modules can themselves become quite large, modules are often hierarchical.

Encapsulation Modules can encapsulate (some of) the model elements. Encapsulation is used to enforce that certain classes of the model are only visible to a restricted set of classes of the model. By encapsulating a class in a module, it is explicitly documented and enforced that a class is accessible within the module.

Abstraction Modules offer support for abstraction by hiding/encapsulating some of their contained elements from the rest of the model. Ideally, this makes a module of classes easier to understand than the complete set of classes contained by the module. Only those classes intended for use outside of the module are visible and need to be understood.

Coordinated development of models by multiple teams Modules are a step toward support for coordinated development of a model by several teams working together on a large model. Different teams can be assigned to separate modules where the interactions between the separate modules of the model are clearly defined. Modules can be developed locally as long as these interactions do not change.

SOCCA [EG94] (Specification of Coordinated and Cooperative Activities) is a graphical formalism and associated method for object oriented modelling which is under active development at Leiden University. The main aim of the SOCCA project is to extend object oriented modelling with means to describe precisely communication in a well-integrated fashion. Over time, SOCCA models have however become increasingly large, and even cumbersome, as bigger and more complex problems were tackled. We extend SOCCA with modules to alleviate this problem by achieving the benefits of modules listed above.

SOCCA is closely related to UML [UML97a] which is becoming more and more a de facto standard for (graphically based) OO formalisms. We follow this development by choosing UML-like packages as modules for SOCCA models. We add *UML-like packages* to SOCCA and not *UML packages*. This has two reasons. First of all, packages in UML can be used to manage any type of model elements. We only intend packages to structure and contain the classes and associated relationships of a model. Secondly, SOCCA packages use a simpler form of export and import but sufficient for the purpose of encapsulation. The defined packages are also sufficient for the purpose of future work (see Section 8.2).

2 Z

[DGSK$^+$99] defines, using Z [Spi92], the core of the SOCCA language. We reuse this formalisation in our work to define SOCCA extended with UML-like packages.

The four perspectives in SOCCA that are currently mature and formalised in [DGSK$^+$99] are:

The data perspective which focuses on the static, structural aspects of models.

The behaviour perspective which focuses on dynamic aspects of individual classes and objects which are made available to other classes and objects.

The functionality perspective which focuses on dynamic aspects of individual classes and objects that are internal to them.

The communication perspective which focuses on the communication between individual classes and objects.

The main additions and changes to the Z formalisation occur for the data perspective as the packages structure the classes of the model and impose restrictions on the possible relations between the classes defined in this perspective. Packages in SOCCA are in this document only defined in the data perspective. We postpone the definition of a package for the other perspectives to [tHDG$^+$b] where we define a class-like description (CLD) of a package of a model. The CLD of a package gives a representation of a package in SOCCA for all perspectives. The remaining three perspectives show only minor changes as compared to their original definition in [DGSK$^+$99]. These three perspectives do not play a significant role until the definition of the CLD. Their Formalisations are however included in the appendices B and C for the sake of completeness and as basis for [tHDG$^+$b] and [tHDG$^+$c].

We have slightly simplified the formalisation of [DGSK$^+$99] to restrict the size of our formalisation. We have left out the attributes of classes and we have left out the signature of methods. These two items are left out as we mainly focus on the communication between classes and how these communications can be structured and restricted by the addition of packages to SOCCA.

Furthermore, we have ignored the names of methods, classes, relationships in the formalisation and have worked exclusively with the *identity* of these constructs. This greatly simplifies the formalisation, avoids having to work with both of the related concepts of name and identity and simplifies references to parts of hierarchial structures, i.e. classes or packages in a hierarchial package.

We have made some further minor changes to the original formalisation:

- We have replaced all sets in the formalisation by finite sets. We have redefined all $x : \mathbb{P} X$ to $x : \mathbb{F} X$. This ensures that we do not work with classes with an infinite number of methods or packages with an infinite number of contained classes.

- We handle the definition of inheritance in several small steps. By splitting the definition of inheritance in two parts, we cut the definition of inheritance into more manageable parts. We first define an abstract notion of inheritance in Section 3.4. In Section 3.5.7 we define which classes are actual specialisations of other classes of a package.
- We handle the definition of visibility and import of methods in two separate parts. We split the original formalisation because we go into more detail and putting everything in one part becomes cumbersome. We first define method import in Section 5 and then define how method visibility restricts the possible imports.

In the course of the formalisation, we will need some small extensions to Z and to the type checker Z/EVES used to type check the formalism (see Appendix A). We will also encounter some notions from discrete mathematics. We will define these here as a "toolkit" or "library" for later use.

We must introduce symbols to Z/EVES as syntactic elements:
Syntax $\supset$ *inrel*
Syntax $\supseteq$ *inrel*
For these newly introduced syntactic elements, we supply the following schema to allow Z/EVES to reason about them.

$$[X]$$

$$\_ \supseteq \_, \_ \supset \_ : \mathbb{P}\, X \leftrightarrow \mathbb{P}\, X$$

$$\forall A, B : \mathbb{P}\, X \bullet A \supseteq B \Leftrightarrow B \subseteq A$$
$$\forall A, B : \mathbb{P}\, X \bullet A \supset B \Leftrightarrow B \subset A$$

$$\mathit{partial\text{-}order}[X] == \{R : X \leftrightarrow X \mid R = R^* \land \neg\, (\exists\, x, y : X \bullet x \neq y \land (x, y) \in R \land (y, x) \in R)\}$$

$$\begin{aligned}\mathit{covering}[X] ==\ & \{R : X \leftrightarrow X \mid R^* \in \mathit{partial\text{-}order}[X] \land \\ & (\forall\, x : X \bullet (x, x) \in R) \land (\forall\, x, y, z : X \mid \\ & (x, y) \in R \land (y, z) \in R \land x \neq y \land y \neq z \bullet (x, z) \notin R)\}\end{aligned}$$

The reader interested in some additional remarks on Z and type checking tools used is referred to the Appendix A. We continue with the actual formalisation.

3 SOCCA with Packages

The *data perspective* in SOCCA describes the static structural aspects of the model. This perspective is graphically represented by a class diagram.

For classic SOCCA, as opposed to SOCCA extended with packages, this diagram consists of the classes and relationships of the model. Relationships between classes in SOCCA are the inheritance and the special uses (method import, see page 22) relationships. There may also be other "general" relationships relevant to the problem domain at hand. For the extension of SOCCA, we also include the packages of the model in this diagram as the packages serve to structure and encapsulate the classes of a classic model.

The formalisation of the data perspective takes place in several steps. We first introduce some general types and the basic definition of SOCCA classes and relationships. Next we define the basic structure of a SOCCA package. We then define our basic model which is defined by one package which contains all the classes and relationships. We continue with the definition of import and export for classes and packages.

The approach in [DGSK$^+$99] in formalising the type level is first to model the meta level, i.e. the entities of which the type level concepts are concrete instances of the meta level concepts. The introduced meta level concepts are *MetaClass* and *MetaRelationship*. The definition of a class in this work follows this same approach.

Example See the UML-like diagram in Figure 1. Both *Person* and *Student* are classes, instances of *Metaclass*. The relationship *IsMarriedTo* between *Person* and itself is an instance of the concept of *Metarelationship*. The specialisation of *Person* to *Student*, i.e. the inheritance relationship, is likewise an instance of the concept *Metarelationship*. □

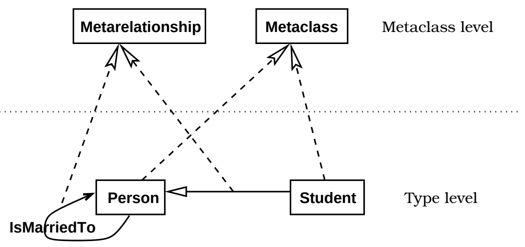


Figure 1: The levels in the SOCCA formalisation

We begin with the definitions of the free types for the identities of the model parts.

3.1 Identity

We define classes, the methods of classes, packages and relationships. All these four items use a separate identity in the formalisation provided by the following types:

$[ClassIdentity, PackageIdentity, MethodIdentity, RelationshipIdentity]$

The next section continues wiht the definition of a class for the data perspective.

3.2 Basic Class Definition

A SOCCA class for the data perspective is defined by an identity, some methods and how these methods are visible to other classes. Furthermore, in OO formalisms, the concept of *encapsulation* is important: aspects of classes may be encapsulated (methods) and, in our case classes as a whole can be hidden from the rest of the model by the packages of the model. We first handle the visibility of the methods of classes. As in [DGSK$^+$99], the categories of visibility of methods are *public*, *private* and *protected*.

The free type definition *MethodVisibility* captures the categories of visibility of class methods:

$$\begin{aligned} \mathit{MethodVisibility} ::= {} & \mathit{public} \\ \mid {} & \mathit{private} \\ \mid {} & \mathit{protected} \end{aligned}$$

With the above types defined, we can make the first step in defining a class. The schema *MetaClass* defines the methods of the class and the visibility of the various methods.

```
── MetaClass ──────────────────────────────
│ methods : 𝔽 MethodIdentity
│ methodvisibility : MethodIdentity ⇸ MethodVisibility
├──────────────
│ dom methodvisibility = methods
└──────────────────────────────────────────
```

Example Figure 2 gives an example class C with methods $M1$ to $M3$. Method $M1$ is visible to all other classes for export, i.e. $C.\mathit{methodvisibility}\ M1 = \mathit{public}$. The method is prefixed with a '+' to indicate the method is "public" following the UML notation of public methods. Method $M2$ is not visible to any other class, i.e. $C.\mathit{methodvisibility}\ M2 = \mathit{private}$. The method is prefixed with a '-' to indicate the method is "private" following the UML notation of private methods. Method $M3$ is only visible to other classes for specialisation, i.e. $C.\mathit{methodvisibility}\ M3 = \mathit{protected}$. The method is prefixed with a '#' to indicate the method is "protected" following the UML notation of protected methods. □

The default method (class and package) visibility in the notation is *public*. The omission of any prefix visibility specifier indicates the default "+" should be inferred.

Individual classes are instances of *MetaClass*: they derive their structure from *MetaClass* and have identity.

```
── Class ──────────────────────────────────
│ MetaClass
│ identity : ClassIdentity
└──────────────────────────────────────────
```

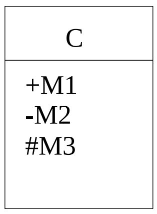


Figure 2: Example of method visibility

We continue with the definition of the possible (meta) relationships between classes.

3.3 Meta Relationships

Relationships between classes can be either a uses, inheritance or general relationship. The uses relationship is a special relationship of SOCCA which defines the import of methods from (other) classes. The possible relationships are captured at the meta level by the free type *MetaRelationship*:

$$MetaRelationship ::= UsesRelationship \mid IsARelationship \mid GeneralRelationship$$

A relationship type defined in [DGSK$^+$99] which is missing from this list is the aggregation relationship. We omit the aggregation relationship in this document for three reasons.

First of all, packages in SOCCA take the place of aggregates with respect to the (hierarchical) grouping of the classes of the model. Each package groups the contained classes and possibly further structures the classes through containment in subpackages.

Secondly, a package does a better job than using aggregation as:

- A package additionally offers encapsulation of its grouped classes. The classes of a package are not simply structured, but also (partially) hidden from the rest of the model.
- A package offers a local namespace for its classes. The names of the classes can be chosen taken only into consideration the names of the other classes of the package.
- A package, through the explicit import and export of the contained model parts, documents the dependencies of the contained parts with the rest of the model.

Lastly, we do not define aggregation at this point as in [tHDG$^+$b] we present a class-like description (CLD) of a package. We define how a class can replace a package of a model. This class precisely captures the communication and synchronisation of the contained classes of the original package with the rest of the model at the precise level of detail offered by SOCCA classes. The CLD of a package replaces the set of classes contained in the package. The CLD thus offers a strong basis for a complete definition of an aggregate for all perspectives and not only for the data perspective and so we postpone the topic of aggregates to [tHDG+b].

We now define the properties of one relationship of a model. A relationship in the model is an instance of a *MetaRelationship*. An actual relationship in a model is characterised by its identity, its type and its participants.

Relationship

$$\begin{array}{l}
\mathit{identity} : \mathit{RelationshipIdentity} \\
\mathit{type} : \mathit{MetaRelationship} \\
\mathit{participants} : \mathrm{seq}\ \mathit{Class} \\
\hline
\mathit{type} \in \{\mathit{UsesRelationship}, \mathit{IsARelationship}\} \Rightarrow \#\ \mathit{participants} = 2
\end{array}$$

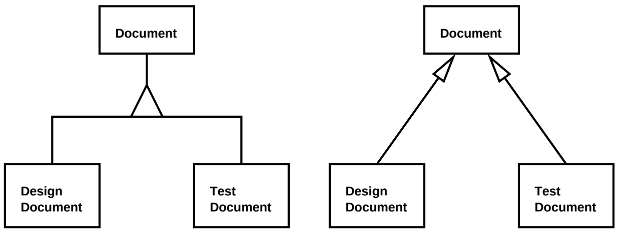
- All relationships of the model are identified by a unique identity.
- All relationships are of a certain *type*, i.e. a uses, inheritance (*IsARelationship*) or general type of relationship.
- All relationships have a number of *participants*. These are the classes which participate in the relationship.
- The uses and inheritance are directed, binary relationships.

In [DGSK+99] as well as in this text, we treat the inheritance relationship as a straightforward *binary* relationship which is relatively simple to formalise. In the past, SOCCA class diagrams used the tree-like generalisation symbol from OMT ([RBP+91]) as depicted in figure 3a. This generalisation symbol might lead one to assume that generalisation is a relationship between a set of classes (children) and a class (parent), instead of simply a relationship between classes. [RBP+91] is unclear in this regard, but [Rum96, p. 326] is not: "Generalization is an n-ary relationship, not a binary relationship. In this we differ from most other authors". In [RTF99], which can be seen as the proper successor notation to OMT's notation, generalisation is however a binary relationship. [UML97b, sect. 4.24.2] explains that in UML the tree-structure is only a display variation of class to class relationships; the other being the separate target style depicted in figure 3b. Generalisation (inheritance) is likewise a binary relationship in SOCCA.

With the relationships between classes defined, we can define when a *Class* is potentially a specialisation of another *Class*.

3.4 Potential Inheritance

When formalising inheritance between classes there are two important things which need to be defined. When class C is a specialisation of class D, then class C inherits a subset of the methods of D depending on the visibility of the methods of D. Furthermore, class C, according to the principle that a child can take the place of its parent class, inherits the relationships to and from D. In [DGSK+99] these two aspects of inheritance are put in one schema. We however handle these two aspects, inheritance of methods and inheritance of relationships, separately as we



(a) Tree-like symbol (b) Separate target style

Figure 3: Different notations for inheritance

have a more complex notion of visibility which leads to larger and more complex schemas. Furthermore, the inheritance of relationships implies the existence of the precise relationships existing between the classes of a package, which we, at this point, have not yet defined. We can however already define when a class C is a potential specialisation of class D. With this we mean that class C has the appropriate methods defined so that it could play the role of a child of D. We introduce the relationship *is-potentially-a* to capture when a class C is a potential specialisation of D, i.e. C $\underline{\textit{is-potentially-a}}$ D. Note that we underline a relationship when written in infix notation for the sake of legibility. The schema for *is-potentially-a* follows.

$$\textit{is-potentially-a} : \mathit{Class} \leftrightarrow \mathit{Class}$$

$$\begin{aligned}
&\textit{is-potentially-a} = \\
&\{r : \mathit{Class} \times \mathit{Class} \mid (\exists\, p, c : \mathit{Class} \mid r = (c, p) \bullet \\
&\quad (\forall\, m : p.\mathit{methods} \mid p.\mathit{methodvisibility}\ m \neq \mathit{private} \bullet \\
&\qquad m \in \operatorname{dom}\ c.\mathit{methods} \land \\
&\qquad (p.\mathit{methodvisibility}\ m = \mathit{public} \lor p.\mathit{methodvisibility}\ m = \mathit{protected}) \Rightarrow \\
&\qquad p.\mathit{methodvisibility}\ m = c.\mathit{methodvisibility}\ m))\}
\end{aligned}$$

Class c must have as methods the public and protected methods of p if c is to be a potential specialisation of p.

Example In Figure 4, class C' is a potential specialisation of class C i.e. C' $\underline{\textit{is-potentially-a}}$ C. Class C' has the required public method $M1$ and the protected method $M3$ but not the private method $M2$ from C. Method $M4$ is added to C' to make it a nontrivial specialisation of C. In practice, inherited methods of classes are often not explicitly indicated by the modeller. The inherited methods are implied by the inheritance relationships between the classes. □

We continue with the definition of the packages, the first substantial deviation from the formalisation of SOCCA in [DGSK$^+$99].

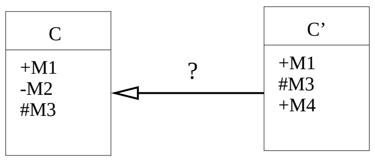


Figure 4: Example of potential inheritance

3.5 Packages

This Section defines the structure of a SOCCA *Package*. This is done in several small steps. We first introduce the free type *ProtoPackage* to define the general structure of packages and restrict in several small steps the possible packages of type *ProtoPackage* until they meet all the constraints we want.

3.5.1 The Basic Package Definition

A package in [RTF99] is defined as a grouping of *any* model parts. Furthermore, a package in UML may contain subpackages. We however restrict the task of SOCCA packages to not just group any model parts but just to group classes and relationships. Additionally, like in UML, we choose for a SOCCA package to group (sub)packages.

The grouping of subpackages in a package is useful as at a point in time a package may become too large to manage. We are extending SOCCA with packages to structure and manage the classes of large models. In the same way we allow subpackages to structure the classes of a single package. By using hierarchical packages (packages within packages) we allow a model to be split up into parts where each part can be further split up as desired.

The use of hierarchical packages is however not essential for managing the complexity of packages. A valid alternative is to put parts of a large package into several new packages. These new packages are placed outside of the original package and are then imported. In this approach, a model only consist of a collection of classes, relationships and a collection of non-hierarchical packages.

Both approaches to managing the complexity of packages are equivalent in the sense that models with the same functionality can be developed. We have a preference for using hierarchical packages. In [tHDG$^+$c] we consider a package as a part of the model which can be developed separately by one of the teams working concurrently on a model. A single package is developed by one team within the static context of the rest of the model. A team can internally restructure its own package using subpackages without influencing the structure of the rest of the model. Adding new packages outside of the package being developed by the team is not desirable as it can influence the context for other teams working on other packages of the model.

Note that a hierarchical package can be reused in various ways. One option is to reuse the package as a whole. A second possibility is to reuse only a subpackage of the original package. The structuring of a model into hierarchical packages as is done for the sake of development in [tHDG$^+$c] does not imply that parts of package can not be reused separately or that a model can not be restructured after completion of the development to only include non-hierarchical packages.

The *ProtoPackage* free type is introduced to define the structure of a package:

$$
\begin{array}{l}
\mathit{ProtoPackage} ::= \\
\mathit{PackageNode}\langle\!\langle \mathit{PackageIdentity} \times \mathbb{F}\,\mathit{Class} \times \mathbb{F}\,\mathit{ProtoPackage} \times \mathbb{F}\,\mathit{Relationship} \rangle\!\rangle
\end{array}
$$

A *ProtoPackage* is defined by:

- An identity of type *PackageIdentity*. The identity of a package serves to identify uniquely a package of the model.
- The set of classes defined by the *ProtoPackage*.
- The set of (sub)packages defined by the *ProtoPackage*.
- The set of relationships defined by the *ProtoPackage*.

We introduce the functions *ProjIdentity* to *ProjRelationships* to isolate one part of the definition of a *ProtoPackage*, i.e. project the identity, contained classes, contained (sub) packages and relationships of a package respectively.

$$
\begin{array}{|l}
\mathit{ProjIdentity} : \mathit{ProtoPackage} \rightarrow \mathit{PackageIdentity} \\
\mathit{ProjClasses} : \mathit{ProtoPackage} \rightarrow \mathbb{F}\,\mathit{Class} \\
\mathit{ProjPackages} : \mathit{ProtoPackage} \rightarrow \mathbb{F}\,\mathit{ProtoPackage} \\
\mathit{ProjRelationships} : \mathit{ProtoPackage} \rightarrow \mathbb{F}\,\mathit{Relationship} \\
\hline
\forall\, p : \mathit{ProtoPackage};\ \mathit{identity} : \mathit{PackageIdentity}; \\
C : \mathbb{F}\,\mathit{Class};\ P : \mathbb{F}\,\mathit{ProtoPackage};\ R : \mathbb{F}\,\mathit{Relationship} \mid \\
\quad p = \mathit{PackageNode}(\mathit{identity}, C, P, R) \bullet \\
\qquad \mathit{ProjIdentity}\ p = \mathit{identity} \land \\
\qquad \mathit{ProjClasses}\ p = C \land \\
\qquad \mathit{ProjPackages}\ p = P \land \\
\qquad \mathit{ProjRelationships}\ p = R
\end{array}
$$

The next Section introduces some useful operations on a package which collect all the contained classes, relationships and (sub)packages of a package.

3.5.2 Working with the Package Parts

A package is a recursive tree-like structure where the nodes of the tree are either classes, packages or relationships. We first define the classes and (sub)packages contained in a package:

- *ClassElements* returns the classes of a package which are defined by the package itself and not by a subpackage contained by the package.

- *PackageElements* Likewise as for *ClassElements* but now for packages.
- *ContainedClasses* Returns all the classes contained by a package. These are the class elements of the package and the classes contained by the package elements of the package.
- *ContainedPackages* Likewise as for *ContainedClasses* but now for packages.

The *ClassElements* and *PackageElements* of a package are called the "elements" of the package.

Example A package in UML is shown as a large rectangle with a small rectangle (a "tab") attached on one corner (usually the left side of the upper side of the large rectangle). The small rectangle contains the identity of the package. The contents of the package, i.e. the elements of the package, are shown within the large rectangle. If the contents of the package are not shown, then the identity of the package is placed within the large rectangle.

In figure 5, a package P is shown where:

- *ClassElements* $P = \{C1\}$ and also *ClassElements* $P2 = \{C4, C5\}$.
- *PackageElements* $P = \{P1, P2\}$.
- *ContainedClasses* $P = \{C1, C2, C3, C4, C5\}$.
- *ContainedPackages* $P = \{P1, P2, P3\}$.

$\square$

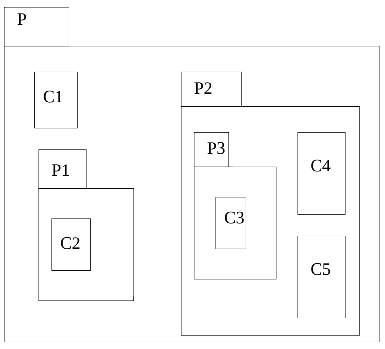


Figure 5: Example Package P

The definition of *ClassElements* and *PackageElements* is basically another name for the *ProjClasses* and *ProjPackages* functions:

$$\mathit{ClassElements} == \mathit{ProjClasses}$$
$$\mathit{PackageElements} == \mathit{ProjPackages}$$

The definition of *ContainedClasses* and *ContainedPackages* are more complex and recursive:

$$\begin{array}{|l}
ContainedClasses : ProtoPackage \rightarrow \mathbb{F}\, Class \\
ContainedPackages : ProtoPackage \rightarrow \mathbb{F}\, ProtoPackage \\
\hline
\forall\, p : ProtoPackage \bullet \\
(ContainedClasses\ p = \\
ClassElements\ p\ \cup \\
\{c : Class \mid \exists\, np : ProtoPackage \mid np \in (PackageElements\ p) \bullet c \in ContainedClasses\ np\}) \\
\wedge \\
(ContainedPackages\ p = \\
PackageElements\ p\ \cup \\
\{p_1 : ProtoPackage \mid (\exists\, np : ProtoPackage \mid np \in PackageElements\ p \bullet p_1 \in ContainedPackages\ np)\})
\end{array}$$

We introduce the function, $AllPackages : ProtoPackage \rightarrow \mathbb{F}\, ProtoPackage$ where $AllPackages\ p$ serves as a shorthand for the frequently used expression $ContainedPackages\ p \cup\ \{p\}$.

$$\begin{array}{|l}
AllPackages : ProtoPackage \rightarrow \mathbb{F}\, ProtoPackage \\
\hline
\forall\, p : ProtoPackage \bullet \\
\quad AllPackages\ p = ContainedPackages\ p \cup \{p\}
\end{array}$$

Example In Figure 5, $AllPackages\ P$ equals $\{P, P1, P2, P3\}$. □

Lastly, we define relationship elements and contained relationships of a package:

$$RelationshipElements == ProjRelationships$$

$$\begin{array}{|l}
ContainedRelationships : ProtoPackage \rightarrow \mathbb{F}\, Relationship \\
\hline
\forall\, p : ProtoPackage \bullet \\
\quad ContainedRelationships\ p = \\
\quad RelationshipElements\ p\ \cup \\
\quad \{r : Relationship \mid (\exists\, p' : ContainedPackages\ p \bullet \\
\qquad r \in ContainedRelationships\ p')\}
\end{array}$$

We use the terminology *classes of a package p* to refer to the set $ContainedClasses\ p$. We likewise use the terminology packages and relationships of a package p to refer to the sets $ContainedPackages\ p$ and $ContainedRelationships\ p$. We use the terminology *package parts* of a package to refer to the classes, relationships and (sub)packages of a given package. The next Section defines when the package parts have the proper identities.

3.5.3 Indentities of the Package Parts

The free type definition of *ProtoPackage* in Section 3.5.1 is very unrestricted. An identity should identify a unique class, relationship or (sub)package contained by a given package and this is not enforced by the free type definition. The set *PackagePartIdentities* contains those packages p which do meet these identity constraints.

$$PackagePartIdentities : \mathbb{F}\, ProtoPackage$$

$$\begin{array}{l}
PackagePartIdentities = \\
\{p : ProtoPackage \mid \\
\#(AllPackages\ p) = \\
\#\{i : PackageIdentity \mid (\exists\, p' : ProtoPackage \mid p' \in AllPackages\ p \bullet ProjIdentity\ p' = i)\} \\
\wedge \\
\#(ContainedClasses\ p) = \\
\#\{i : ClassIdentity \mid (\exists\, c : Class \mid c \in ContainedClasses\ p \bullet c.identity = i)\} \\
\wedge \\
\#(ContainedRelationships\ p) = \\
\#\{i : RelationshipIdentity \mid (\exists\, r : Relationship \mid r \in ContainedRelationships\ p \bullet r.identity = i)\}\}
\end{array}$$

We continue with the definition of the tree-like structure of Packages.

3.5.4 The Tree-Like Structure of a Package

As yet, we allow within the definition of *PackagePartIdentities* for (sub)packages and classes to be an element of more than one package.

Example In Figure 6, both packages P_1 and P_2 which are defined so that they are elements of *PackagePartIdentities* have class C as class element, i.e. $P_1 \neq P_2 \wedge ClassElements\ P_1 \cap ClassElemtents\ P_2 \neq \varnothing$. $\square$

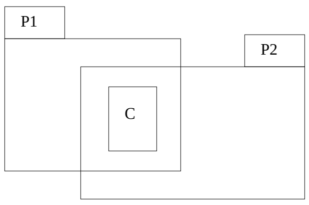


Figure 6: Example of a non strict tree Package

According to the UML notation guide [RTF99]: "Each element is directly owned by a single package, so the package hierarchy is a strict tree." Thus, the UML packages, if they are not nested, are disjoint and do not overlap. This means that every class or package of the model is directly owned by exactly one (owner) package. We use the terminology *strict* package for a package for which the containment structure of the package forms such a strict tree.

We likewise ensure that SOCCA packages form a strict tree. This is done for three reasons.

First of all, a package is less modular if a class may belong to two packages like in Figure 6. We can not study P_1 without considering the use of class C in P_2. A package becomes less of a black box if it shares some of its contained classes with another non nested package.

As a consequence, the semantics of the import of a package become less precise. The import of a package should document the dependencies of the contained classes and packages with the rest of the model. For Figure 6, the shared class C may however require different imported functionality for its role in package P_1 and P_2. The import required by class C can not then be deduced by only studying the import of one of the two packages.

Furthermore, in [tHDG$^+$c] we allow multiple teams to work concurrently on one model. Each team works on a package of the model which does not share any overlapping classes and subpackages with the packages being developed by other teams. This makes it straightforward to ensure that only one team at a time can modify a class of the model. The distribution of packages over the teams is relatively simple if we use packages which are all strict trees.

We introduce the set *StrictPackages* to capture the packages from *PackagePartIdentities* that are strict:

$$\begin{array}{|l}
\mathit{StrictPackages} : \mathbb{F}\,\mathit{PackagePartIdentities} \\
\hline
\mathit{StrictPackages} = \\
\{p : \mathit{PackagePartIdentities} \mid \\
\quad (\forall\, c : \mathit{ContainedClasses}\ p \bullet (\exists_1\, p' : \mathit{AllPackages}\ p \bullet c \in \mathit{ProjClasses}\ p')) \land \\
\quad (\forall\, p' : \mathit{ContainedPackages}\ p \bullet (\exists_1\, p'' : \mathit{AllPackages}\ p \bullet p' \in \mathit{ProjPackages}\ p''))\}
\end{array}$$

Example In Figure 5, the package P is an element of *StrictPackages*. However, in Figure 6, the package P_1 and P_2 are not elements of *StrictPackages* as both have class C as element. □

We continue with the definition of the *owner* of a package element.

3.5.5 The Owner of a Package Element

With respect to the strict tree-like structure of packages in Section 3.5.4, a class of a package is directly owned, i.e. is a class element of, exactly one package. We call this package the *owner* of the class. For example, in Figure 5, the owner of class $C3$ is package $P3$. We can likewise identify the owner of a package or relationship. The concept of a owner package of a class, (sub)package or relationship recurs frequently in the formalisation.

Note that in the complete formalisation we make a distinction between *ownership* of model parts and *use* of model parts. A class or subpackage is owned by *one* unique package of the model but may be imported and used by *many* other parts of the model. The ownership relations in a model form a tree-like structure. The (uses) relationships between the classes and packages of the model form a graph with as nodes the classes and packages of the model and as edges (uses) relationships between the classes and packages.

We introduce the function *OwnerofClass* and *OwnerofPackage* to return respectively the owner of a class or package of a model. The function *OwnerofClass* is however not simply of type $\mathit{Class} \rightarrow \mathit{ProtoPackage}$ such that *OwnerofClass* c is the package which owns a given class c. The owner of a class is only defined for classes contained by packages which are a strict tree. Furthermore, at this point in the formalisation, we have characterised the packages which can contain a class. For a given class, we can design an infinite number of strict packages to contain the class. But, if we take one such strict package, within that package we can find the owner of the class. We call such a package the *context* in which the owner of a class can be found.

Keeping this in mind, we give the following two functions:

- The function *OwnerofClass* returns for a given strict package p and a class c contained by p the package in *AllPackages* p of which the class c is an element.
- The function *OwnerofPackage* returns for a given strict package p and a package p' contained by p the package in *AllPackages* p of which the package p' is an element.

The package p is the context in which the owner of the class or package is sought.

Example In Figure 5, the owner of class $C3$ is package $P3$. Or, otherwise put, *OwnerofClass* $P2\ C3 = P3$. In the same figure, the owner of package $P3$ is package $P2$. Thus, *OwnerofPackage* $P\ P3 = P2$. Packages P and $P2$ are the contexts in which the owners of $C3$ and $P3$ are sought. □

$$\begin{array}{l}
\mathit{OwnerofClass} : \mathit{StrictPackages} \rightarrow\!\!\!\!\!\!+\; (\mathit{Class} \rightarrow\!\!\!\!\!\!+\; \mathit{StrictPackages}) \\
\hline
\forall\, p : \mathit{StrictPackages} \bullet \\
\quad \mathrm{dom}\,(\mathit{OwnerofClass}\ p) = \{c : \mathit{Class} \mid c \in \mathit{ContainedClasses}\ p\} \land \\
\quad (\forall\, c : \mathit{Class} \mid c \in \mathrm{dom}\,(\mathit{OwnerofClass}\ p) \bullet \\
\quad \mathit{OwnerofClass}\ p\ c = \\
\quad (\mu\, np : \mathit{StrictPackages} \mid (np \in \mathit{AllPackages}\ p) \land (c \in \mathit{ClassElements}\ np) \bullet np))
\end{array}$$

$$\begin{array}{l}
\mathit{OwnerofPackage} : \mathit{StrictPackages} \rightarrow\!\!\!\!\!\!+\; (\mathit{StrictPackages} \rightarrow\!\!\!\!\!\!+\; \mathit{StrictPackages}) \\
\hline
\forall\, p : \mathit{StrictPackages} \bullet \\
\quad \mathrm{dom}\,(\mathit{OwnerofPackage}\ p) = \{tp : \mathit{StrictPackages} \mid tp \in \mathit{ContainedPackages}\ p\} \land \\
\quad (\forall\, tp : \mathit{StrictPackages} \mid tp \in \mathrm{dom}\,(\mathit{OwnerofPackage}\ p) \bullet \\
\quad \mathit{OwnerofPackage}\ p\ tp = \\
\quad (\mu\, np : \mathit{StrictPackages} \mid (np \in \mathit{AllPackages}\ p) \land (tp \in \mathit{PackageElements}\ np) \bullet np))
\end{array}$$

Likewise, all relationships are owned by exactly one package. For a given package p from the set *StrictPackages* and a relationship r from *ContainedRelationships* p, we define the owner of the relationship as the unique package p' from *AllPackages* p which owns r.

$$\begin{array}{|l}
\mathit{OwnerOfRelationship} : \mathit{StrictPackages} \nrightarrow \mathit{Relationship} \nrightarrow \mathit{StrictPackages} \\
\hline
\forall p : \mathit{StrictPackages} \bullet \\
\quad \mathrm{dom}\ (\mathit{OwnerOfRelationship}\ p) = \mathit{ContainedRelationships}\ p \land \\
\quad (\forall r : \mathit{Relationship} \mid r \in \mathrm{dom}\ (\mathit{OwnerOfRelationship}\ p) \bullet \\
\qquad (\mathit{OwnerOfRelationship}\ p\ r) = \\
\qquad (\mu\, p' : \mathit{AllPackages}\ p \mid r \in \mathit{RelationshipElements}\ p' \bullet p'))
\end{array}$$

We continue with the constraints placed on ownership by packages of relationships.

3.5.6 Packages and Ownership of Relationships

In [DGSK$^+$99] the relationships of a model are defined in one set. This set contains all the relationships defined between the classes of the model. In this document we distribute the relationships over the packages which structure the model.

In UML every model part must belong to exactly one package. For relationships, this means that every relationship of the model must be owned by one package of the model. Furthermore, these relationships have to be properly defined in the sense that a package may not own a relationship where one of the participating classes is not a class contained by the package.

For SOCCA each package of a model is intended to be defined as locally/modular as possible. We want the smallest possible encompassing package to contain a relationship between a set of classes. This makes the definition of a package as small as possible, which is useful when splitting up packages of a model during development as defined in [tHDG$^+$c] or if a package is to be reused. We therefore add the constraint that a package may only own a relationship if the package is the smallest package which contains all the classes participating in the relationship.

Example In Figure 7, there is a relationship defined from class $C1$ to $C2$. This relationship is owned by package P' as all the participants of the relationship are contained by P' and there is no smaller package which meets this criteria and thus has at least one of the participants of the relationship as one of its class elements. The relationship is however *not* owned by package P as P' is contained by P and P' is large enough to contain all the participants of the relationship. □

The set *ProperPackageRelationships* contains the packages from *StrictPackages* which meet the criteria for ownership of relationships:

$$\begin{array}{|l}
\mathit{ProperPackageRelationships} : \mathbb{F}\ \mathit{StrictPackages} \\
\hline
\mathit{ProperPackageRelationships} = \\
\{p : \mathit{StrictPackages} \mid \\
(\forall r : \mathit{RelationshipElements}\ p \bullet \\
\quad (\mathrm{ran}\ (r.\mathit{participants}) \subseteq \mathit{ContainedClasses}\ p) \land \\
\quad (\exists c : \mathit{ClassElements}\ p \bullet c \in \mathrm{ran}\ (r.\mathit{participants}))) \land \\
(\forall p' : \mathit{ContainedPackages}\ p \bullet p' \in \mathit{ProperPackageRelationships})\}
\end{array}$$

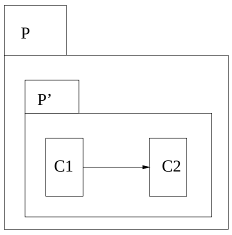


Figure 7: Illustration of ownership of relations

- Relationships defined by each package p are only defined for participating classes contained by p.
- Each relationship is owned by the smallest package which contains all the classes participating in the relationships.
- The contained packages of a package from *ProperPackageRelationships* are also elements of *ProperPackageRelationships*.

We continue with the definition of inheritance for the classes of a package.

3.5.7 Inheritance of Classes

In Section 3.4 we have defined when a class is a potential specialisation of another class. We say a class $D \underline{\mathit{IsPotentiallyA}}\ C$ if class D has the right methods to be a specialisation of C. With the classes of packages defined, we can now define which classes of the package are actual specialisations of other classes of the package. Furthermore, with the parent-child relationships (i.e. inheritance) in a package defined, we can define inheritance of relationships. With inheritance of relationships we mean that a specialised child inherits the relationships its parent is a participant in. The function *is-directly-a* captures, for a given package P, which classes of the package are direct specialisations of other classes. Basically, $(D, C) \in$ $(\mathit{is\text{-}directly\text{-}a}\ P)$ if $D\ \underline{\mathit{IsPotentiallyA}}\ C$, $\{C, D\} \subseteq \mathit{ContainedClasses}\ P$ and there is a relationship of type *IsARelationhip* defined from D to C. The relationship $(\mathit{is\text{-}a}\ P)$ is the closure of the $(\mathit{is\text{-}directly\text{-}a}\ P)$ relationship.

Example In Figure 4, we have given classes C and C' where C' is a potential specialisation of C,
i.e. $C'\ \underline{\mathit{is\text{-}potentially\text{-}a}}\ C$. If C and C' are both in $\mathit{ContainedClasses}\ p$ and $(\exists\, r : \mathit{ContainedRelationships}\ p \mid r.\mathit{type} = \mathit{IsARelationship} \bullet r.\mathit{participants} = \langle C', C\rangle)$, then indeed $C'\ \underline{(\mathit{is\text{-}a}\ p)}\ C$. $\square$

$$
\begin{array}{|l}
\mathit{is\text{-}a} : \mathit{ProperPackageRelationships} \rightarrow (\mathit{Class} \leftrightarrow \mathit{Class}) \\
\mathit{is\text{-}directly\text{-}a} : \mathit{ProperPackageRelationships} \rightarrow (\mathit{Class} \leftrightarrow \mathit{Class}) \\
\hline
\forall\, p : \mathit{ProperPackageRelationships} \bullet \\
\mathit{is\text{-}directly\text{-}a}\ p = \{ r : \mathit{Class} \times \mathit{Class} \mid \\
\quad \exists\, g : \mathit{ContainedRelationships}\ p;\ c, d : \mathit{Class} \mid \\
\qquad \{c, d\} \subseteq (\mathit{ContainedClasses}\ p) \land c\ \underline{\mathit{is\text{-}potentially\text{-}a}}\ d \bullet \\
\qquad\quad r = (c, d) \land \\
\qquad\quad g.\mathit{type} = \mathit{IsARelationship} \land \\
\qquad\quad g.\mathit{participants} = \langle c, d \rangle \} \land \\
\mathit{is\text{-}a}\ p = ((\mathit{is\text{-}directly\text{-}a})\ p)^{*} \land \\
\mathit{is\text{-}a}\ p \in \mathit{partial\text{-}order}[\mathit{Class}]
\end{array}
$$

- $\mathit{is\text{-}directly\text{-}a}\ p$ captures which of the potential specialisations in $\mathit{is\text{-}potentially\text{-}a}$ is actually realised for a specific package p.
- $\mathit{is\text{-}a}\ p$ is the transitive closure of $\mathit{is\text{-}directly\text{-}a}\ p$.

The set *PackageClassRelationshipInheritance* captures the constraints on the possible relationships of the package imposed by the substutivity principle for specialised classes. The substutivity principle states that every specialised class must be able to take the role of the parent class of which it is a specialisation.

Example Consider the class diagram (fragment) in figure 8: as *Design* is a *Document*, and *Manager* monitors *Document*, one can infer that *Manager* monitors *Design*. In the example, we have drawn it as dashed line. It is customary not to draw the relationships induced by inheritance in order not to clutter the class diagram. □

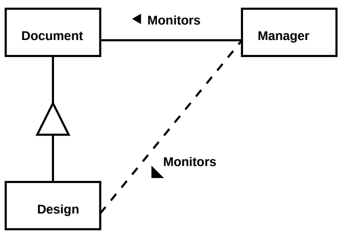


Figure 8: Inheritance of relationships

$$\begin{array}{|l}
\mathit{PackageClassRelationshipInheritance} : \mathbb{F}\, \mathit{ProperPackageRelationships} \\
\hline
\mathit{PackageClassRelationshipInheritance} = \\
\{ p : \mathit{ProperPackageRelationships} \mid \\
\forall\, c, d : \mathit{Class} \mid (c, d) \in (\mathit{is\text{-}a}\ p) \bullet \\
\quad (\forall\, R : \mathit{ContainedRelationships}\ p \mid R.\mathit{type} \neq \mathit{IsARelationship} \bullet \\
\qquad \forall\, i : 1 \mathinner{..} \#(R.\mathit{participants}) \mid R.\mathit{participants}\ i = d \bullet \exists\, S : \mathit{ContainedRelationships}\ p \bullet \\
\qquad\quad S.\mathit{type} = R.\mathit{type} \land \\
\qquad\quad \#\ (S.\mathit{participants}) = \#\ (R.\mathit{participants}) \land \\
\qquad\quad S.\mathit{participants}\ i = c \land \\
\qquad\quad (\forall\, j : (1 \mathinner{..} \#(R.\mathit{participants})) \setminus \{i\} \bullet S.\mathit{participants}\ j = R.\mathit{participants}\ j))\}
\end{array}$$

A specialised class can fulfill the role of its owner class(es) in a relationship.

We continue with the complete definition of a SOCCA *Package*.

3.5.8 Package Definition

Since the definition of the free type *ProtoPackage* we have imposed several restrictions on the packages allowed by this definition. We have ensured that these packages along with their contained elements have proper identities and have defined the ownership of the contained elements. Furthermore, we have defined inheritance between the contained classes. The set *Package* captures the packages which meet the structure criteria of SOCCA packages along with the final constraint that every SOCCA package contains at least one class:

$$\begin{array}{|l}
\mathit{Package} : \mathbb{F}\, \mathit{PackageClassRelationshipInheritance} \\
\hline
\forall\, p : \mathit{Package} \bullet \\
\quad (\forall\, p' : \mathit{AllPackages}\ p \bullet \#(\mathit{ContainedClasses}\ p') > 0)
\end{array}$$

We consider only packages which contain at least one class as in SOCCA only (the objects instantiated from) classes show behaviour. For the remainder of this document we use the term “package” to refer to an element from *Package* exclusively.

We continue with the definition of the package which contains all the classes of a model.

4 Model Definition

We have now defined all constraints for the desired type of package so that we can define the classes, relationships and packages of a basic model for the data perspective. We use the term *basic model* as at this point we have not yet defined the import, export and visibility of model parts.

A model at the type level is essentially represented by one package. We call this package the *TopPackage*. The schema *BasicModel* defines a basic model by defining this one package.

BasicModel

TopPackage : *Package*

We use the terminology *model part* to refer to either the *TopPackage* itself or to a part of this package. Recall that a package part is either a class, relationship or (sub)package contained by the package. A class of the model is a class of the *TopPackage* of the model. Relationships and Packages of a model are likewise defined.

The *TopPackage* which represents the whole model is often not drawn by the modeler. The *TopPackage* is represented by the piece of paper or program window in which the model is being drawn.

In [DGSK$^+$99], the schema *DataPerspectiveDomain* defines the variable *classes* : $\mathbb{P}$ *Class*. This variable is the complete set of classes of the model. The link with this formalisation is that *classes* is equal to *ContainedClasses TopPackage*. The classes of a model can however now be structured and encapsulated by packages.

The variable *relationships* in the schema *DataPerspectiveDomain* is introduced as an abbreviation for all the relationships defined between the classes of the model, i.e. the classes in *ContainedClasses TopPackage*. This abbreviation is used extensively in the rest of the formalism.

DataPerspectiveDomain

BasicModel
relationships : $\mathbb{F}$ *Relationship*

relationships = *ContainedRelationships TopPackage*

The next Section continues with the constraints on the use of methods.

5 The Import of Methods

In many formalisms there is not much attention to import within (an object instantiated from) a class: methods within a class are automatically available for use by other methods within the class.

In a SOCCA class diagram, the *uses* relationship between classes describes method import. The uses relationships are drawn as arcs from the class which imports the method to the class from which the methods are imported. The arcs are labeled with the identities of the imported methods.

Example In Figure 9, the class *client* imports the methods $M1$ and $M2$ from the class *server*. The uses relationship is depicted as a directed edge from *client* to *server* and the imported methods are added as labels to the edge. □

In SOCCA, there is no implicit import within a class; methods within a class are *not* automatically available for use by other methods within the class. Thus, even within a class or an object, one method can only

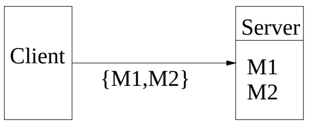


Figure 9: The uses relationship

call another method if it has a *uses* relationship to that class with that method in the labelling set.

The reason for making this import within a class explicit is that this import has consequences for other parts of a SOCCA model: namely, those parts that deal with coordination. Furthermore, the uses relationship documents the possible communication between *two* classes.

For SOCCA, method invocations of methods within one object can in principle be executed concurrently (i.e. SOCCA objects can be multi-threaded). Thus, intra-object method use necessitates coordination between threads. To emphasise the consequences of intra-object method use, it is made explicit in the data perspective through the uses relationship. The schema *Uses* captures the basic definition of the uses relationships:

Uses

$$
\begin{array}{l}
DataPerspectiveDomain \\
uses : Class \leftrightarrow Class \\
useslabel : (Class \times Class) \nrightarrow \mathbb{F}\, MethodIdentity \\
\hline
uses = \{ r : Class \times Class \mid \exists\, g : relationships;\ c, d : ContainedClasses\ TopPackage \bullet \\
\qquad r = (c, d) \land g.type = UsesRelationship \land g.participants = \langle c, d \rangle \} \\
\mathrm{dom}\, useslabel = uses \\
\forall\, c, d : ContainedClasses\ TopPackage;\ M : \mathbb{F}\, MethodIdentity \mid (c, d) \mapsto M \in useslabel \bullet \\
\qquad M \neq \varnothing \land \\
\qquad M \subseteq d.methods
\end{array}
$$

The relation *uses* captures the classes (c, d) of the model for which there is a uses relationship from c to d. The function *useslabel* defines which method identities of d adorn the uses relationship from c to d when $(c, d) \in uses$. The schema *Uses* as presented here is a shortened version of the schema *Uses* as presented in [DGSK$^+$99] as we postpone the constraints imposed on import by the visibility of methods to the end of this Section.

Note that the uses relationship is defined such that a class may import its own methods, i.e $(\exists\, c : Class \bullet (c, c) \in uses)$. For the functionality perspective in Appendix B.3, we define that a class may not call a method unless this method was previously imported. A class thus has to import its own methods previously to being able to call them. This may seem extreme. We however use import (and export) to document dependencies between model parts. An import of a class of one of its own methods documents that the class is one of its own clients, which is not always the case.

Example In Figure 10, the class $client{-}server$ imports its own method $do - it$. This is necessary if a method of the class is to call the method. $\square$

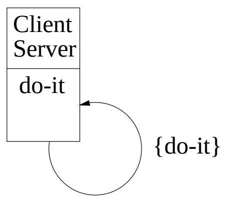


Figure 10: Importing your own methods

A modeler will however in may cases not draw this reflexive import as he is often more interested in dependencies with other model parts.

A method has to be visible to a class in order to import it through a uses relationship. The schema *UsesMethodRestrictions* captures the restrictions on possible import of methods by classes.

$$
\begin{array}{|l}
\textit{UsesMethodRestrictions} \\
\hline
\textit{Uses} \\
\hline
\forall\, c, d : \textit{ContainedClasses}\ \textit{TopPackage};\ M : \mathbb{F}\,\textit{MethodIdentity} \mid (c, d) \mapsto M \in \textit{useslabel} \bullet \\
\quad \forall\, m : \textit{MethodIdentity} \mid m \in M \bullet \\
\quad\quad c = d \lor \\
\quad\quad (d.\textit{methodvisibility}\ m) = \textit{public} \lor \\
\quad\quad (d.\textit{methodvisibility}\ m = \textit{protected} \land ((c, d) \in (\textit{is-a}\ \textit{TopPackage}))) \\
\hline
\end{array}
$$

All methods of the used class must be visible to the using class:

- Any class may import its own methods.
- Any class may import the public methods of another class.
- A class may import protected methods of its parent class. This constraint was omitted in [DGSK$^+$99].

The next Section continues with the export and import of classes by packages.

6 Export and Import of Classes

In the previous Section, we have defined how the visibility of methods regulates the possible import of methods by classes. In this Section we define how the visibility of classes and subpackages of a package determines the export of the package and thus the possible import by other packages of a model. We begin by defining the export of a package of a model by defining the visibility of the contained classes and packages of the package.

6.1 Export of Classes by Packages

Like in UML, we use a simple type of visibility for package elements. An element of an UML package is either *public*, i.e. can be exported by the package, or *private* and is not exported by the package.

We introduce the free types *ClassVisibility* and *PackageVisibility* to define the possible visibilities of classes and packages respectively:

$$\begin{aligned} \mathit{ClassVisibility} ::= {} & \mathit{publicclass} \\ \mid {} & \mathit{privateclass} \end{aligned}$$

$$\begin{aligned} \mathit{PackageVisibility} ::= {} & \mathit{publicpackage} \\ \mid {} & \mathit{privatepackage} \end{aligned}$$

In the schema *PackageExport* we define the visibility of the classes and packages of a model. The function *export* defines the export of a SOCCA package. The export of a UML package is defined as a subset of its public parts. These include the public classes and public (sub)packages of a package. We simplify this for SOCCA packages by expressing the export of a SOCCA package solely in terms of exported classes. The export of a class of a subpackage implicitly defines the export of the corresponding subpackage.

PackageExport

$$\begin{array}{l}
\mathit{DataPerspectiveDomain} \\
\mathit{packagevisibility} : \mathit{Package} \mathrel{⇸} \mathit{PackageVisibility} \\
\mathit{classvisibility} : \mathit{Class} \mathrel{⇸} \mathit{ClassVisibility} \\
\mathit{export} : \mathit{Package} \mathrel{⇸} \mathbb{F}\ \mathit{Class} \\
\hline
\mathrm{dom}\ \mathit{classvisibility} = \mathit{ContainedClasses}\ \mathit{TopPackage} \\
\mathrm{dom}\ \mathit{packagevisibility} = \mathit{ContainedPackages}\ \mathit{TopPackage} \\
\mathrm{dom}\ \mathit{export} = \mathit{ContainedPackages}\ \mathit{TopPackage} \\
\forall p : \mathrm{dom}\ \mathit{export} \bullet \\
\quad \mathit{export}\ p \subseteq \\
\quad \{c : \mathit{ContainedClasses}\ p \mid \\
\quad\quad (c \in \mathit{ClassElements}\ p \land \mathit{classvisibility}\ c = \mathit{publicclass}) \lor \\
\quad\quad (\exists p' : \mathit{PackageElements}\ p \mid \mathit{packagevisibility}\ p' = \mathit{publicpackage} \bullet c \in \mathit{export}\ p')\}
\end{array}$$

- The functions *classvisibility* and *packagevisibility* return the visibility of a class and package of a model.
- The export of a package is a subset of the public class elements of the package and the classes exported by the public package elements of the package.

The next Section defines the import the packages of a model, i.e. which of the classes exported by packages are imported for possible use.

6.2 Import of Classes by Packages

The schema *PackageImport* with the function *import* documents which classes exported by packages are imported by (other) packages. Like for the export of a SOCCA package, we define the import of a SOCCA package as a set of classes. The import of a class implicitly determines the import of the package(s) which exported the class.

PackageImport

$$\begin{array}{l}
\mathit{Uses} \\
\mathit{PackageExport} \\
\mathit{import} : \mathit{Package} \pfun \mathbb{F}\ \mathit{Class} \\
\hline
\mathrm{dom}\ \mathit{import} = \mathit{ContainedPackages}\ \mathit{TopPackage} \\
\forall p : \mathrm{dom}\ \mathit{import} \bullet \forall c : \mathit{ClassElements}\ p \bullet \\
\quad \forall d : \mathit{ContainedClasses}\ \mathit{TopPackage} \setminus \mathit{ClassElements}\ p \mid \\
\quad\quad (c, d) \in (\mathit{is\text{-}a}\ \mathit{TopPackage}) \lor \\
\quad\quad (c, d) \in \mathit{uses} \lor \\
\quad\quad (\exists r : \mathit{relationships} \mid r.\mathit{type} = \mathit{GeneralRelationship} \bullet \\
\quad\quad\quad (\exists i, j : 1 \ldots \#\ (r.\mathit{participants}) \bullet r.\mathit{participants}\ i = c \land r.\mathit{participants}\ j = d)) \bullet \\
\quad\quad\quad (d \in \mathit{import}\ p) \\
\forall p : \mathrm{dom}\ \mathit{import} \bullet \\
\quad \mathit{import}\ p \subseteq (\mathit{ContainedClasses}\ \mathit{TopPackage} \setminus \mathit{ClassElements}\ p) \land \\
\quad (\forall d : (\mathit{import}\ p) \bullet \\
\quad\quad (\forall p' : \mathit{ContainedPackages}\ \mathit{TopPackage} \mid \\
\quad\quad\quad p \neq p' \land \\
\quad\quad\quad p \notin \mathit{ContainedPackages}\ p' \land \\
\quad\quad\quad d \in \mathit{ContainedClasses}\ p' \bullet \\
\quad\quad\quad\quad d \in \mathit{export}\ p'))
\end{array}$$

- The necessary classes are imported as required. Class d not contained by a package p needs to be imported if there is a class c contained by p which has a uses relationship to class d or if there is a general relationship with as participants the classes c and d.
- All imported classes are actually exported by the packages which contain the imported packages.

The next Section presents the final schema for the data perspective.

7 The Complete Data Perspective

The schema *DataPerspective* defines a SOCCA model with packages for the data perspective.

```
┌─ DataPerspective ───────────────────────────
│ DataPerspectiveDomain
│ UsesMethodRestrictions
│ PackageExport
│ PackageImport
└─────────────────────────────────────────────
```

- The schema *DataPerspectiveDomain* defines SOCCA classes and packages for the data perspective.
- The schema *UsesMethodRestrictions* defines the import of methods.
- The schema *PackageExport* defines the export of classes by packages.
- The schema *PackageImport* defines the import of classes by packages.

8 Conclusion

8.1 Results

We have formalised (using Z) how the classes of SOCCA models are structured and encapsulated by UML-like Packages. The architecture of a SOCCA model is no longer a set of classes but a collection of hierarchial packages. A large model can be split up into smaller, more comprehensible chunks.

8.2 Future Work

With SOCCA packages defined, we will formalise [tHDG$^+$b] the concept of a class-like description (CLD) of a package. We investigate how one class, the CLD of the package, replaces a package of a model. The CLD of a package unifies the concepts of class and packages/modules of classes.

In [tHDG$^+$a] we discuss/investigate special CLDs of packages which capture the interactions/communication of the classes of a package with the rest of the classes of a model. We use the special CLDs of a package to define when a change to a package is local. i.e. the change of the package does not affect the interactions of the classes of the package with the other classes of the model. We use the definition of local change of a package to prevent conflict in configuration management during the merging of separately developed parts/versions of a model.

APPENDICES:

A Z

In this Section we discuss:

- In Section A.1 the formalisation language used.
- In Section A.2 the tool used to check the formalisation.
- In Section A.3 the style of our formalisation.

A.1 Why Z

Several factors positively influenced the choice of Z as the specification language in [DGSK$^+$99] and this document.

Abstraction level Z is suitable for specifications at a high level of abstraction, as it focuses on mathematical description of systems, rather than expressing systems in terms of a particular machine model. This allows one to focus on the "what" rather than the "how".

Widespread use Z has been applied successfully in a large number of diverse projects in both industry and academia, and there are numerous publications about it available [00bds].

Mathematical foundation Z is founded in set theory and predicate logic, both branches of mathematics that are familiar to computer scientists. The particular set-theory underlying Z is non-exotic.

Standardisation Z is currently undergoing standardisation by the international standards body ISO.

Tool support There are a number of tools available [ZZads] that provide support for the development of Z specifications, including pretty-printers, type checkers and theorem provers.

Continuity Continued use of Z allows the reuse of the work done in [DGSK$^+$99].

A.2 The Type Checker

The Z parts of the document were checked using Z/EVES [Saa95], a theorem prover for Z. Information about it can be found at `http://www.ora.on.ca/z-eves/welcome.html`. Although the theorem prover mainly used as type checker was a very valuable tool we experienced some difficulties during the use of Z/EVES:

- This document was written using LaTeX. The type checker had some difficulties in properly parsing the formatting information inserted into the definition of a mathematical set. The straightforward definition of the set would not present difficulties and would check for type. Inserting a newline or spacing information would in many cases lead to parsing errors. We have modified the layout until it was readable and did not produce unwarranted errors.
- Functions likewise presented some problems with formatting information. Inserting any layout information in or between the parameters of a function lead to errors. Thus, all function (calls) in schemas must fit on one source line if they are to check for type.
- Use of cross products lead to long expressions. Expressions of the form $\{(c, d) : X \times X \mid \mathit{Assertion}(c, d)\}$ were not accepted and had to be rewritten to the equivalent expression of the form $\{r : X \times X \mid (\exists\, c, d : X \mid (c, d) = r \bullet \mathit{Assertion}(c, d)\}$. This is annoying as this expression distracts by its inelegant complexity. Furthermore this second form is especially annoying when taking into account the property that formatting information in sets often leads to typing

errors as the second expression grows expansively so as to not fit on one line. We have thus avoided cross products in our work when they were not required from a legacy viewpoint. For example, we have worked with $f : X \rightarrow X \rightarrow Y$, the curried version of f, rather than with f defined as $f : (X \times X) \rightarrow Y$.

- We avoided use of relations where possible as relations lead to cross products. We have used functions to circumvent relations as demonstrated with a small example. Instead of $R : X \leftrightarrow Y$ we chose to work with $Rf : X \rightarrow \mathbb{F}\, Y$ where $y \in Rf\ x \Leftrightarrow (x, y) \in R$. The expression $(x, y) \in R$ is avoided by working with the equivalent expression $y \in Rf\ x$.

A.3 The Style of Z Used

The problems with the type checker listed in Appendix A.2 influenced the style of our Z specification. We often used curried functions as they allow complex domain restrictions of partial functions to be defined step by step. This can make domain restrictions of partial functions easier to understand. Take, for example, $f : (A \times B \times C) \nrightarrow D$. The domain of f is expressed as constraints on the parameters of type A, B and C. This can lead to large complicated expressions. The curried version of f is of the type $A \nrightarrow B \nrightarrow C \nrightarrow D$. The domain constraints of the curried version of f can now be defined separately step by step for each type A to C.

Lastly, we encountered several other small problems using the type checker:

- The right side of a free type expression (i.e. after the ::=) cannot be split over several lines. Newlines in the LaTeX source file result in error messages.
- Text in a comment block is not ignored by Z-eves. Comment blocks therefore cannot be used to (temporarily) easily remove parts of the formalisation from the text. This may however be a useful feauture to include Z parts into a document which you need for the type checking but which you do not want to display in the document. For example, this feauture will be very for the writing of [tHDG$^+$b] where the Z parts of this document have to be included for the type checker while the reader is not interested in seeing all of this Z code once more.
- No formatting information can be inserted inbetween the closing sequence of brackets (inbetween)))))) in a schema without the type checker reporting errors.
- For a function $f : A \nrightarrow B \nrightarrow C$, for $a : A$, $b : B$ and $c : C$, the expression $(a, (b, c)) \in f$ does not type. You have to work with $f : (A \times B) \nrightarrow C$ as $((a, b), c) \in f$ does not type. we however wanted to avoid cross products as much as possible!
- The expressions $(p, (c, d)) \in \mathit{is\text{-}a}$, $(p \mapsto (c, d)) \in \mathit{is\text{-}a}$, $c\ \underline{(\mathit{is\text{-}a}\ p)}\ d$ and $(c, d) \in (\mathit{is\text{-}a}\ p)$ are all equivalent. Only the last one makes it through the type checker.

B The behaviour and functionality perspectives

This Appendix defines a class for the behaviour perspective. The formalisation of the rest of the document does not deviate much from the original material as presented in [DGSK$^+$99]. The only change occurs for the functionality perspective for the definition of the free type *SYMBOL*. Calls to methods of classes are no longer of the form $\mathit{ClassName} \times \mathit{MethodName}$ but of the form $\mathit{ClassIdentity} \times \mathit{MethodIdentity}$. We work with *ClassIdentity* instead of *ClassName* as the name of a class no longer uniquely identifies a class of the model. *MethodIdentity* is used instead of *MethodName* as the names of methods here play a secondary role. The appendices B.2 to C.2 are thus included for the sake of completeness.

In Section 7 we have described the data perspective of a SOCCA model, which describes a static structure of classes and their relationships. Now we describe two more perspectives of SOCCA, which describe dynamic aspects of SOCCA classes.

The *behaviour perspective* deals with visible behaviour (behaviour that is visible to other classes); whereas the *functionality perspective* describes hidden behaviour (which describes the functionality of the various methods). Later on, in the communication perspective, we describe the coordination between the behaviours of objects.

The behavioural aspects of SOCCA models are specified through State Transition Diagrams (STDs): graphical diagrams containing states and labeled transitions between them. As we see further on, our means of expressing communication is based on STDs too.

In using STDs for the functionality perspective, SOCCA clearly differs from OMT, revised OMT and UML. Originally OMT ([RBP$^+$91]) used data flow diagrams for it's "functional model". In revised OMT ([Rum96, p. 353]), "the functional model consists of use cases and operation descriptions, as well as object interaction diagrams, pseudo code designs, and actual code to specify how they work." In UML, there is no clear equivalent for the functionality perspective. UML is only a notation; it offers several ways of expressing some perspectives; there is no associated method that clarifies which language elements are to be used for what perspective. For instance, UML still has data flow diagrams, but for the description of behaviour perhaps statecharts can be used as an alternative.

B.1 STDs

In Computer Science, behaviour is often expressed through abstract machines from Formal Language Theory (such as Turing machines, finite state machines or stack automata; see e.g. [HU79]). In these abstract machines, there is a finite control operating on a possibly potentially infinite storage structure. In the PARADIGM formalism ([Gro88]), which is the basis of the communication perspective of SOCCA, behaviour was expressed through semi-Markov decision processes, a formalism well-known in Operational Research which can express stochastic behaviour.

STDs are used in SOCCA because they are a mid-way compromise between semi-Markov decision processes and computer science automata models. They are quite close to the finite state machines (FSMs) familiar to computer scientists, but are allowed to have an infinite state space (of the control — there is no additional storage structure), they can express non-determinism (as can many other automata models), but they lack expressive power for describing true stochastics which semi-Markov decision processes have. Because STDs are quite similar to FSMs, which are common throughout computer science, we describe them in an FSM-like manner. To emphasise the distinction between the type level and the instance level, we distinguish between STDs (on the type level) and STMs (state transition machines, "STDs in action" on the instance level).

The way in which we use STDs in the formalisation of SOCCA is different from that in Formal Language Theory. In Formal Language Theory, STDs are primarily devices for generating languages, whose internal structure doesn't matter much (often STDs are considered equivalent when they generate the same language, which means no attention is given to the exact sequence(s) of states involved in generating or recognising a particular word). In SOCCA the precise structure of STDs is highly relevant, as we focus on communication between STMs, STDs in action. The possible behaviours allowed by a SOCCA model result from the interaction between STMs.

An STD consists of a set of states (some marked as initial and/or final) and a transition relation between states marked with symbols; a function doesn't suffice as an STD may be non-deterministic. It provides a static description of behaviour on the type level, i.e. it describes all possible behaviours, rather than any particular behaviour that is actually occurring in an instance.

We introduce a type for the states of STDs.

$[STATE]$

Transitions can in general be labeled with plain method identities (in external STDs), "act" labels (in internal STDs; they indicate the activation of the STDs behaviour), or "call" labels (in internal STDs). It is often desirable to have the option not to label transitions; for this we include ϵ.

$$\begin{aligned} SYMBOL ::= {} & ml\langle\!\langle MethodIdentity\rangle\!\rangle \\ \mid {} & \epsilon \mid act\langle\!\langle MethodIdentity\rangle\!\rangle \mid call\langle\!\langle (ClassIdentity \times MethodIdentity)\rangle\!\rangle \\ \mid {} & OTHER \end{aligned}$$

Once more, note that as opposed to [DGSK$^+$99] we define calls to be of type *ClassIdentity* $\times$ *MethodName* as opposed to *ClassName* $\times$ *MethodName*. This is because class names no longer uniquely identify a class for a model since the addition of packages. We remedy this by working with class identity. Furthermore, note that we continue to work with calls on internal stds of the form *call ClassIdentity x MethodIdentity* while only the *MethodIdentity* would suffice. This is done as otherwise this document would deviate too far from [DGSK$^+$99] and secondly this approach would not be in the spirit of the rest of the work on SOCCA.

With $SYMBOL$ defined, we describe the structure of an STD in general (on the meta class diagram level):

$$
\begin{array}{|l}
\textit{MetaSTD} \\
\hline
states : \mathbb{F}\, STATE \\
labels : \mathbb{F}\, SYMBOL \\
transrel : (STATE \times SYMBOL) \leftrightarrow STATE \\
initial : \mathbb{F}\, STATE \\
final : \mathbb{F}\, STATE \\
\hline
initial \cup final \subseteq states \\
states \neq \varnothing \Rightarrow initial \neq \varnothing \\
transrel \subseteq (states \times labels) \times states \\
labels = \{ l : SYMBOL \mid \exists\, s_1, s_2 : states \bullet ((s_1, l), s_2) \in transrel \} \\
\hline
\end{array}
$$

And adding identity to $MetaSTD$, we get regular STDs.

$[STDIdentity]$

$$
\begin{array}{|l}
\textit{STD} \\
\hline
Identity : STDIdentity \\
MetaSTD \\
\hline
\end{array}
$$

- $STATE$ is the type of states in STDs; $states$ is the set of actual states in a specific STD. Therefore, both $initial$ and $final$ have to be in $states$.
- STDs may have multiple initial and final states.
- Often, but not always $states \neq \varnothing$, $initial \neq \varnothing$, $final \neq \varnothing$.

It is sometimes useful to be able to work with the edges directly, disregarding their labels.

$$
\begin{array}{|l}
edges : STD \rightarrow (STATE \leftrightarrow STATE) \\
\hline
\forall\, std : STD \bullet \\
\quad edges\ std = \{ e : STATE \times STATE \mid \exists\, p, q : STATE \mid \\
\qquad e = (p, q) \bullet (\exists\, sym : SYMBOL \bullet (p, sym) \mapsto q \in std.transrel) \}
\end{array}
$$

To describe the realisation of behaviour of objects on the instance level, we define State Transition Machines (STMs): abstract processors that run exactly one program; this program is described by an STD. Like for the STDs they are instances of, we do not require STMs to be finite (although they almost always are finite in practice). As we shall see, a particular object may have multiple STMs running simultaneously, allowing it to be multi-threaded.

B.2 The behaviour perspective: External behaviour STDs

With each class, we associate an STD that specifies the *external behaviour*. The external behaviour STD of a class describes behaviour that is visible to other classes, namely the order in which calls to methods the class exports are accepted. Note that a class may also export methods to itself (e.g. if one object of a class may call the method of another object of the same class, or for when one method of an object calls another method of the same object).

Edges in the external behaviour STD of a class are unlabeled or labeled with the identities of operations exported by that class to other classes or itself:

$$ExternSTD : \mathbb{F}\, STD$$

$$\forall\, s : ExternSTD \bullet s.labels \subseteq \operatorname{ran} ml \cup \{\epsilon\}$$

BehaviourPerspective

$$DataPerspective$$
$$externalbehaviour : Class \leftrightarrow STD$$

$$externalbehaviour \subseteq ContainedClasses\ \ TopPackage \times ExternSTD$$
$$\forall\, c : ContainedClasses\ \ TopPackage;\ m : MethodIdentity \bullet \#\{estd : STD \mid (c, estd) \in externalbehaviour \land ml$$
$$\forall\, c : ContainedClasses\ \ TopPackage \bullet \{m : MethodIdentity \mid \exists\, estd : STD \mid (c, estd) \in externalbehaviour \bullet ml$$

- Note that not all the method identities need to occur in the external behaviour STD.
- Note that even when a method name is used in the external behaviour STD, there is no guarantee a call to it will ever be handled. An external behaviour STD merely constrains the order in which calls may be accepted.
- We allow for multiple external STDs. This feature has already proven useful in thesis projects [Wil95, vdZ96]. This feauture proves useful for the definition of the CLD of a package.

B.3 The Functionality perspective: Internal behaviour STDs

With each method of a class, we can associate an *internal behaviour STD* that describes how that method is realised.

An internal behaviour STD's transitions are labeled with method calls to methods of the class it belongs to and methods exported to that class by other classes (or itself). Unlike other formalisms, in SOCCA methods within a class are not automatically available for use within other methods of the same class; there has to be a suitably labeled *uses* relation from the class to itself.

The transitions in an internal behaviour STD may be labeled with "*act methodidentity*" (indicating activation of the execution of *methodidentity*) or "*call class.methodidentity*" (indicating a request to start the execution of *methodidentity*).

$$InternSTD : \mathbb{F}\,STD$$

$$\forall s : InternSTD \bullet s.labels \subseteq \{\epsilon\} \cup (\mathrm{ran}\ act) \cup (\mathrm{ran}\ call)$$
$$\forall s : InternSTD \bullet \forall l : s.labels;\ i : s.initial;\ st : s.states \mid (i, l) \mapsto st \in s.transrel \bullet st \notin s.initial$$
$$\forall s : InternSTD \bullet (\forall s_1, s_2 : s.states;\ l : s.labels \mid (s_1, l) \mapsto s_2 \in s.transrel \land l \in \mathrm{ran}\ act \bullet s_1 \in s.initial)$$

- Initial states do not connect to each others.
- All transitions from an initial states are labeled with an "act" label.

With STDs defined, we can now define the *FunctionalityPerspective*:

FunctionalityPerspective

$$BehaviourPerspective$$
$$internalbehaviour : (Class \times MethodIdentity) \mathrel{+\!\!\!\rightarrow} STD$$

$$\mathrm{dom}\ internalbehaviour =$$
$$\{r : Class \times MethodIdentity \mid$$
$$\quad (\exists c : Class;\ m : MethodIdentity \mid (c \in ContainedClasses\ \ TopPackage) \land (m \in c.methods) \bullet r = (c, m))\}$$
$$\{i : InternSTD \mid$$
$$\quad (\exists c : Class;\ m : MethodIdentity \mid$$
$$\qquad (c \in ContainedClasses\ \ TopPackage) \land (m \in c.methods) \bullet internalbehaviour(c, m) = i)\} \subseteq$$
$$InternSTD$$
$$\forall c : ContainedClasses\ \ TopPackage \bullet \forall m : c.methods;\ std : STD$$
$$\quad \mid std = internalbehaviour(c, m) \bullet$$
$$\qquad (\forall i : std.initial;\ s : std.states;\ l : std.labels \mid ((i, l) \mapsto s) \in std.transrel \bullet l = act\ \ m)$$
$$\forall c, d : ContainedClasses\ \ TopPackage \bullet$$
$$\quad useslabel(c, d) =$$
$$\quad \{n : MethodIdentity \mid \exists m : c.methods;\ std : STD \bullet$$
$$\qquad internalbehaviour(c, m) = std \land$$
$$\qquad call\ \ (d.identity, n) \in std.labels\}$$
$$\forall c : ContainedClasses\ \ TopPackage;\ m : MethodIdentity;\ s : STD \mid s = internalbehaviour(c, m) \bullet$$
$$\quad ml\ \ m \in (externalbehaviour\ \ c).labels$$

- The transition(s) in the internal STD of a method m starting at an initial node are labeled with "*act m*", indicating activation of the method invocation. Usually, there is only one initial node, but we haven't ruled out multiple initial nodes.
- Invocations of other methods are indicated by
  "*call* ClassIdentity.MethodIdentity".
  This import is precisely what the uses relationship describes.
- Methods for which an implementation (STD) is provided, must occur as labels in an external STD of the class they belong to.

When specifying the instance level, we show invocations of methods of particular objects are done. At that point, we also see how the visibility restrictions dealing with objects (calls to private members of other objects are disallowed, are implemented. For now, we restrict ourselves to indicating only the class of objects whose methods are invoked.

C The communication perspective

The *communication perspective* in SOCCA expresses how communication between instances of classes occurs. It is based on PARADIGM [Gro88]. As with the behaviour and functionality perspectives, we use concepts based on STDs in our description, rather than ones based on semi-Markov decision processes. We need to introduce several notions before we can address the communication perspective.

C.1 Intuitive description

The communication perspective is where SOCCA differs the most from other object oriented modeling languages. If presented in a purely factual or formal way, it can be quite daunting. Therefore we'll give you a rough sketch of the intuition underlying it first.

A fundamental observation about communicating processes is that their behaviour can be viewed as having two levels. The first is the level of *local behaviour* which describes the pieces of behaviour that the process may have which do not require communication with other processes. Such local behaviour has

parts in which no communication is desired, and no coordination is necessary, and

parts

in which communication is desired to arrange coordination to prepare the way for another piece of local behaviour. Until this communication has taken place, the process is restricted to the current piece of local behaviour. The second, more abstract, level is that of *global behaviour* which describes how the processes' behaviour through coordination by communication may be switched from one piece of local behaviour to another.

As we have seen earlier, in SOCCA we describe the global behaviour of classes through an external STD, and the local behaviour of methods through internal STDs. The different parts of local behaviour we describe

by *subprocesses* and *traps*. A subprocess describes a temporary restriction of behaviour, a piece of local behaviour. A *trap* defines the part of a subprocess where coordination is desired.

Example In figure C.1, a simple STD is shown (labels are left out to keep things simple), together with two possible subprocesses and their traps. The traps are shown as shaded areas. When more than one trap is presented with a subprocess, they are often given numbers. The subprocesses are partial versions of the original STD (disregarding initial and final states).

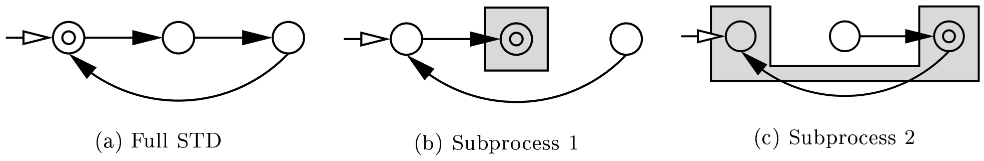

(a) Full STD (b) Subprocess 1 (c) Subprocess 2

Figure 11: An STD with two subprocesses

□

In light of communication, we distinguish two roles of STDs: *employee* and *manager*. An employee is an STD augmented by a structure of subprocesses and traps known as a *partition and trap structure*. An employee is managed by a manager (meaning the manager prescribes when and which transitions the employee may make between its subprocesses). The manager is an STD augmented with two functions. One, the *state interpreter*, which maps its states to prescribed subprocesses, and one, the *action interpreter*, which labels its transitions with traps that its employee(s) must have reached for the transition to be allowed.

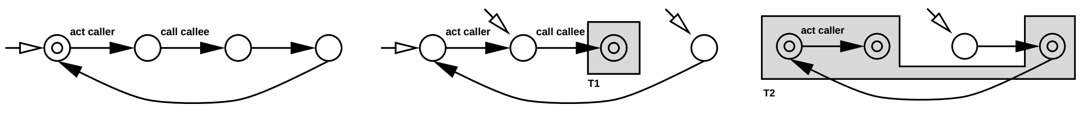
In SOCCA, the external STDs form the basis for the managers, and the internal STDs for the employees. This imposes more structure than in PARADIGM, where the choice of managers and employees was up to the modeler.

The notions of employee and manager are dual: an equally valid view on a given model is that the employees manage their manager. For PARADIGM, this has been proved in [Mor93]. Using this duality, the concepts of employee and managers can be formalised more symmetrically; we don't do this, as this view is somewhat less natural.

There is a behavioural consistency that works in both directions: an employee's behaviour obeys the restriction imposed by the current subprocess prescribed by the manager, while the manager's behaviour obeys the restrictions imposed by subprocesses (not making a transition labeled with a trap that hasn't been reached yet).

By itself, PARADIGM lacks the structure provided object orientation of SOCCA and thus allows the modeler very large degrees of freedom in modeling. In SOCCA this freedom has been restricted through the object oriented structure, making it more manageable. In SOCCA, the modeler no longer has the freedom of choosing employee and manager roles arbitrarily: a class' external STD(s) gets the role of manager of the internal STD(s): the external STDs receive messages (calls) and start up behaviours of internal STDs to handle them.

Example As an illustration of how the communication perspective in SOCCA is used, consider the following situation: we have two classes, *A* and *B*. Method *A.Caller* needs to perform a synchronised call to method *B.Callee*, i.e. it calls *Callee* and has to wait until that call has been handled completely.

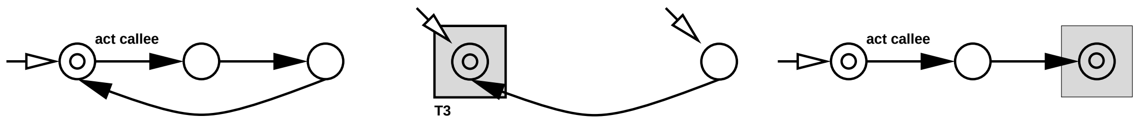


(a) STD (b) First subprocess R_1 (c) Second subprocess R_2

Figure 12: Caller

Caller's STD is depicted in figure 12(a). It has a fairly simple structure: activation, call *Callee*, some internal stuff, and repeat when desired.

The handling of the call to *Callee* induces two subprocesses: one, R_1 (depicted in figure 12(b)) in which the actual call is allowed and in which the trap $T1$ expresses the waiting for the call to finish; the other, R_2 (depicted in figure 12(c)) in which permission to perform the call is temporarily revoked; its big trap $T2$ indicating its willingness to regain that permission as soon as possible.



(a) STD (b) First subprocess E_1 (c) Second subprocess E_2

Figure 13: Callee

Callee's structure is more simple than *Caller*'s: activation, and internal stuff (see figure 13(a)). Like in *Caller*, the synchronised way we want to call it induces two subprocesses: the first, E_1 (in figure 13(b)) in which *Callee* waits to perform its activities; the second, E_2 (in figure 13(c)) in which performs them.

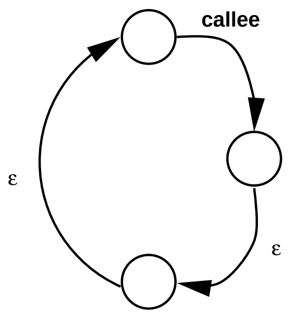
In this example, there is just one designated trap for each subprocess; this need not be in the general case: there can be more than one designated trap.

In figure 14(b), a suitable manager is depicted. The state and transition interpreters are indicated by an appropriate labelling of the states and transitions respectively.

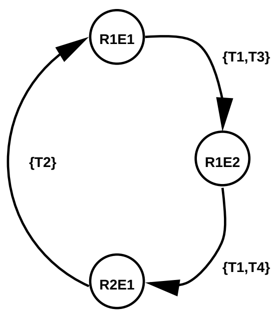
□

C.2 Formal description

Subprocess The basic idea is that a process's full behaviour is described by an STD, but that most of the time it is useful to view a process



(a) External STD of A



(b) Corresponding manager

as being in a *subprocess* of that STD. A subprocess functions as a temporary restriction on what behaviour the process is allowed to exhibit. It is an STD too.

Communication between processes is required to make a switch between subprocesses.

$$\begin{array}{|l}
\mathit{isSubProcessOf} : \mathit{STD} \leftrightarrow \mathit{STD} \\
\hline
\forall\, \mathit{std}, \mathit{subp} : \mathit{STD} \bullet \mathit{subp} \ \underline{\mathit{isSubProcessOf}} \ \mathit{std} \Leftrightarrow \\
\qquad \mathit{subp.states} \subseteq \mathit{std.states} \land \\
\qquad \mathit{subp.labels} \subseteq \mathit{std.labels} \land \\
\qquad \mathit{subp.transrel} \subseteq \mathit{std.transrel} \cap ((\mathit{subp.states} \times \mathit{subp.labels}) \times \mathit{subp.states})
\end{array}$$

- Note that the initial and final states of a subprocess of an STD are not required to come from that STD: a subprocess has its own initial and final states, unrelated to those of the STD it relates to.
- It is easy to see that S *isSubProcessOf* S. An STD is its own *trivial subprocess*.

Trap A *trap* is a set of states within a particular subprocess that, once reached, cannot be left while the behaviour restriction expressed by the subprocess holds. The traps a modeler chooses indicate that a process is ready to switch from one subprocess to another. That a subprocess has reached a trap, does not mean that it is idle. It can still perform useful actions. That it has reached a trap merely means that it has entered a final phase of the behaviour restriction imposed by the subprocess the trap belongs to.

We introduce a relation to check if a particular set of states is a trap of an STD:

$$\begin{array}{|l}
\mathit{isTrapOf} : \mathbb{F}\,\mathit{STATE} \leftrightarrow \mathit{STD} \\
\hline
\forall\, S : \mathbb{F}\,\mathit{STATE} \bullet \forall\, \mathit{std} : \mathit{STD} \bullet S\ \underline{\mathit{isTrapOf}}\ \mathit{std} \Leftrightarrow \\
S \neq \varnothing \land \qquad (\forall\, s, t : \mathit{std.states} \bullet \\
\qquad\qquad \forall\, l : \mathit{std.labels} \mid \\
\qquad\qquad\qquad s \in S \land ((s, l), t) \in \mathit{std.transrel} \bullet t \in S)
\end{array}$$

- It is easy to see that *std.states isTrapOf std*. This is known as the *trivial trap*: the trap consisting of all states of a subprocess.

A trap can lead from a subprocess to another subprocess.

$$\begin{array}{|l}
\mathit{isTrapConnectionOf} : \mathit{STD} \times \mathbb{F}\,\mathit{STATE} \times \mathit{STD} \leftrightarrow \mathit{STD} \\
\hline
\forall\, \mathit{std}, \mathit{subp}_1, \mathit{subp}_2 : \mathit{STD};\ \mathit{trap} : \mathbb{F}\,\mathit{STATE} \bullet \\
\quad (\mathit{subp}_1, \mathit{trap}, \mathit{subp}_2)\ \underline{\mathit{isTrapConnectionOf}}\ \mathit{std} \Leftrightarrow \\
\qquad (\mathit{subp}_1\ \underline{\mathit{isSubProcessOf}}\ \mathit{std} \land \\
\qquad \mathit{subp}_2\ \underline{\mathit{isSubProcessOf}}\ \mathit{std} \land \\
\qquad \mathit{trap}\ \underline{\mathit{isTrapOf}}\ \mathit{subp}_1 \land \\
\qquad ((\mathit{subp}_1 = \mathit{subp}_2) \lor (\mathit{trap} \subseteq \mathit{subp}_2.\mathit{initial} \land \mathit{trap} \subseteq \mathit{subp}_1.\mathit{final})))
\end{array}$$

Partition and Trap Structure Often we consider an STD with a particular set of associated subprocesses that "cover" the STD. Such a set of subprocesses, each with its own set of traps is known as a *partition and trap structure* of that STD.

Such a structure shows how the behaviours of the STD are partitioned in the light of communication. The modeler has degrees of freedom in choosing the subprocesses, and within them, in choosing the relevant traps.

We define a relation to check if a set of STDs and set of states is indeed a partition and trap structure of a given STD:

$$\begin{array}{|l}
\mathit{isPartitionAndTrapStructureOf} : (\mathbb{F}(\mathit{STD} \times \mathbb{F}(\mathbb{F}\,\mathit{STATE}))) \leftrightarrow \mathit{STD} \\
\hline
\forall\, \mathit{part} : \mathbb{F}(\mathit{STD} \times \mathbb{F}(\mathbb{F}\,\mathit{STATE}));\ \mathit{std} : \mathit{STD} \bullet \\
\quad (\mathit{part}, \mathit{std}) \in \mathit{isPartitionAndTrapStructureOf} \Leftrightarrow \\
\qquad \{\mathit{state} : \mathit{STATE} \mid \exists\, \mathit{partstd} : \mathit{part} \bullet \mathit{state} \in (\mathit{first}\ \mathit{partstd}).\mathit{states}\} = \mathit{std.states} \land \\
\qquad \bigcup\{\mathit{trans} : (\mathit{STATE} \times \mathit{SYMBOL}) \leftrightarrow \mathit{STATE} \mid \\
\qquad\quad \exists\, \mathit{partstd} : (\mathit{STD} \times \mathbb{F}(\mathbb{F}\,\mathit{STATE})) \bullet \mathit{trans} = (\mathit{first}\ \mathit{partstd}).\mathit{transrel}\} = \mathit{std.transrel} \land \\
\qquad (\forall\, \mathit{subp} : \mathit{STD};\ \mathit{traps} : \mathbb{F}(\mathbb{F}\,\mathit{STATE}) \mid (\mathit{subp}, \mathit{traps}) \in \mathit{part} \bullet \\
\qquad\quad \mathit{subp}\ \ \underline{\mathit{isSubProcessOf}}\ \mathit{std} \land \\
\qquad\quad (\forall\, \mathit{trap} : \mathbb{F}\,\mathit{STATE} \mid \mathit{trap} \in \mathit{traps} \bullet \\
\qquad\qquad \mathit{trap}\ \underline{\mathit{isTrapOf}}\ \mathit{subp} \land \\
\qquad\qquad (\exists\, \mathit{subp}_2 : \mathit{STD};\ \mathit{traps}_2 : \mathbb{F}(\mathbb{F}\,\mathit{STATE}) \mid \\
\qquad\qquad\quad (\mathit{subp}_2, \mathit{traps}_2) \in \mathit{part} \bullet \\
\qquad\qquad\qquad (\exists\, \mathit{ctrap} : \mathit{traps}_2 \bullet (\mathit{subp}, \mathit{ctrap}, \mathit{subp}_2)\ \underline{\mathit{isTrapConnectionOf}}\ \mathit{std}))))
\end{array}$$

- The subprocesses in the partition and trap structure cover the STD in both states and labels.
- The subprocesses are connected via traps.

- In the graphical notation, traps are named; in the abstract Z syntax there is no need to name them, as they are uniquely identified within their STDs.
- The connection constraint is local only; we do not impose a reachability constraint between subprocesses in a partition and trap structure in general.

Employee process An STD with a partitioning into subprocesses, each with a set of traps that connects it to the others, is known as an *employee (process).*

$$\begin{array}{|l} \textit{employee} \\ \hline \textit{std} : \textit{STD} \\ \textit{pts} : \mathbb{F}(\textit{STD} \times (\mathbb{F}(\mathbb{F}\, \textit{STATE}))) \\ \hline \textit{pts}\ \ \underline{\textit{isPartitionAndTrapStructureOf}}\ \ \textit{std} \\ \hline \end{array}$$

Often, the employees follow the pattern of having two subprocesses, one containing the "act" label(s) (corresponding to "starting"), one without (corresponding to "functioning"). Discussion of such patterns is outside the scope of this paper; we refer you to [Bru98].

Manager process A *manager (process)* is an STD that describes the coordination that takes place when employee processes change subprocess. The states of a manager are used to prescribe the subprocesses of its employees; the transitions of a manager are labeled with the traps (of its employees) that need to be reached before the transition is possible.

With each manager, for each of his employees, comes a *state and transition interpreter* which describes how the manager relates to the employee: it maps each state of the manager STD to the subprocesses it prescribes to the employee and maps each transition label of the manager STD to the traps of this particular employee that have to be reached in order for the transition to be allowed.

Example See figure 14(b). The state interpreter labels each state of the manager STD with the subprocesses the manager in that state prescribes to its employees. Similarly, the transition interpreter labels each transition of the manager STD with the set of traps of its employees that have to be reached for the transition to be allowed. □

In earlier SOCCA publications, this was termed the *state action interpreter.* The term "action" originates in decision process theory; in light of our use of STDs, the term "transition" is clearer. Also, originally the state action interpreter described the relation between a manager and all its employees. In the formalisation, it is more convenient to split this out for each employee, and distinguish the state and transition parts of the state and transition interpreter.

We use abbreviation definitions to make state and transition interpreters more visible:

$stateint == STATE \pfun STD$
$transint == (STATE \times SYMBOL) \times STATE \pfun (\mathbb{F}\, STATE)$

$$
\begin{array}{l}
\textit{manager} \\
\hline
std : STD \\
empsti : \mathrm{seq}(employee \times stateint \times transint) \\
\hline
(\forall i : 1 \mathrel{..} \#empsti \bullet \\
\quad (\forall e : employee;\ si : stateint;\ ti : transint \mid (e, si, ti) = empsti\ i \bullet \\
\quad\quad \mathrm{dom}\, si = std.states \land \\
\quad\quad \mathrm{dom}\ ti = \{t : (STATE \times SYMBOL) \times STATE \mid \\
\quad\quad\quad \exists s_1, s_2 : std.states;\ sym : SYMBOL \bullet \\
\quad\quad\quad\quad t = ((s_1, sym), s_2) \land t \in std.transrel\} \land \\
\quad\quad (\forall s_1, s_2 : std.states;\ sym : std.labels \mid ((s_1, sym), s_2) \in std.transrel \bullet \\
\quad\quad\quad si\ s_1\ \underline{isSubProcessOf}\ e.std \land \\
\quad\quad\quad si\ s_2\ \underline{isSubProcessOf}\ e.std \land \\
\quad\quad\quad ti\ ((s_1, sym), s_2)\ \underline{isTrapOf}\ si\ s_1 \land \\
\quad\quad\quad\quad (si\ s_1, ti\ ((s_1, sym), s_2), si\ s_2)\ \underline{isTrapConnectionOf}\ e.std)))
\end{array}
$$

- The state interpreters map states of the manager to appropriate subprocesses of the employee at hand.
- The action interpreters map transitions of the manager to appropriate traps of the employee at hand.
- All transitions in the manager's STD, interpreted to any of them employees involved, corresponds to a proper connection between two (possibly identical) subprocesses via a relevant trap (possibly the trivial one).

And we define some auxiliary functions to handle managers more easily; one to get a manager's employees:

$$
\begin{array}{l}
HasEmployees : manager \rightarrow \mathbb{F}\, employee \\
\hline
\forall m : manager \bullet \\
\quad HasEmployees\ m = \\
\quad\quad \{i : employee \mid \exists si : stateint;\ ti : transint \bullet \\
\quad\quad\quad (i, si, ti) \in \mathrm{ran}\ m.empsti\}
\end{array}
$$

and one for the reverse:

$$
\begin{array}{l}
HasManagers : employee \rightarrow \mathbb{F}\, manager \\
\hline
\forall e : employee;\ m : manager \bullet \\
\quad m \in HasManagers\ e \Leftrightarrow e \in HasEmployees\ m
\end{array}
$$

Now we can put these concepts together to express the communication perspective. The internal STDs of the various methods are employees of the external STD of their class acting as manager.

CommunicationPerspective

$$
\begin{array}{l}
\mathit{DataPerspective} \\
\mathit{BehaviourPerspective} \\
\mathit{FunctionalityPerspective} \\
\mathit{managers} : \mathbb{F}\, \mathit{manager} \\
\mathit{employees} : \mathbb{F}\, \mathit{employee} \\
\mathit{externalstds} : \mathbb{F}\, \mathit{STD} \\
\mathit{internalstds} : \mathbb{F}\, \mathit{STD} \\
\mathit{asmanager} : \mathit{STD} \nrightarrow \mathit{manager} \\
\mathit{asemployee} : \mathit{STD} \nrightarrow \mathbb{F}\, \mathit{employee}
\end{array}
$$

$$
\begin{array}{l}
\mathit{externalstds} = \mathrm{ran}\, \mathit{externalbehaviour} \\
\mathit{internalstds} = \mathrm{ran}\, \mathit{internalbehaviour} \\
\forall\, m : \mathit{managers} \bullet \exists\, \mathit{estd} : \mathit{externalstds} \bullet m.\mathit{std} = \mathit{estd} \\
\forall\, m : \mathit{managers} \bullet \mathit{HasEmployees}\ m \neq \varnothing \\
\forall\, e : \mathit{employees} \bullet \mathit{HasManagers}\ e \neq \varnothing \\
\forall\, e : \mathit{employees} \bullet \exists\, \mathit{istd} : \mathit{internalstds} \bullet e.\mathit{std} = \mathit{istd} \\
\mathrm{dom}\ \mathit{asmanager} = \mathit{externalstds} \wedge \mathrm{ran}\, \mathit{asmanager} = \mathit{managers} \\
\forall\, s : \mathit{externalstds} \bullet (\mathit{asmanager}\ s).\mathit{std} = s \\
\mathrm{dom}\ \mathit{asemployee} = \mathit{internalstds} \\
\mathit{employees} = \{ e : \mathit{employee} \mid (\exists\, \mathit{istd} : \mathrm{dom}\ (\mathit{asemployee}) \bullet e \in (\mathit{asemployee}\ \mathit{istd}))\} \\
\forall\, s : \mathit{internalstds} \bullet \forall\, e : (\mathit{asemployee}\ s) \bullet e.\mathit{std} = s \\
\forall\, c : \mathit{ContainedClasses}\ \mathit{TopPackage} \bullet \mathit{HasEmployees}\ (\mathit{asmanager}\ (\mathit{externalbehaviour}\ c)) = \\
\quad \{ e : \mathit{employees} \mid \exists\, \mathit{mn} : \mathit{MethodIdentity} \bullet \\
\qquad ((\mathit{mn} \in c.\mathit{methods} \wedge e \in \mathit{asemployee}\ (\mathit{internalbehaviour}(c, \mathit{mn}))) \vee \\
\qquad (\exists\, d : \mathit{ContainedClasses}\ \mathit{TopPackage} \mid \\
\qquad\quad \mathit{mn} \in \mathit{useslabel}\ (c, d) \bullet \\
\qquad\qquad e \in \mathit{asemployee}(\mathit{internalbehaviour}(d, \mathit{mn}))))\}
\end{array}
$$

- The external STDs are the basis for the managers; the managers manage the employees based on the internal STDs.
- The managers manage the employees corresponding to their class' internal STDs and the internal STDs of methods the class uses.

Discussion Communication Perspective In this description of the communication perspective, we have identified the manager STDs and the external STDs, as well as the employee STDs and the internal STDs. This is a simplification of the reality of modeling. In the reality of modeling, one starts with a simple external STD which is later refined in light of communication. The resulting STD is also an external STD, but one which is suited for the manager role. Similarly, the internal STDs are refined to form the employee STDs. The precise notion of refinement/extension/compatibility involved is currently understood in an intuitive fashion only; we hope to formalise it in the future.

References

[00bds] Z bibliography. URL: `http://www.comlab.ox.ac.uk/archive/z/bib.html`, 1990 onwards.

[Bru98] H.G. Brugman. Software process modeling in SOCCA. Master's thesis, Department of Computer Science, Leiden University, May 1998.

[DGSK$^+$99] J.H.M. Dassen, L.P.J. Groenewegen, I.G. Sprinkhuizen-Kuyper, P.W.M. Koopman, P.J. 't Hoen, and G. Engels. A formalisation of SOCCA using Z; part 1: the type level concepts. Technical Report 1999–03, Leiden Institute of Advanced Computer Science, February 1999. Available on the web as `http://www.wi.leidenuniv.nl/TechRep/1999/tr99-03.ps.gz`.

[EG94] Gregor Engels and Luuk P.J. Groenewegen. SOCCA: Specifications of coordinated and cooperative activities. In A. Finkelstein, J. Kramer, and B.A. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 71–102. Research Studies Press Ltd. / John Wiley & Sons Inc., 1994. Taunton 1994.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice-Hall, Inc., Englewood Cliffs, 1 edition, 1991.

[Gro88] L.P.J. Groenewegen. Parallel phenomena, 1986–88. A series of technical reports, consisting of [Gro86, Gro87c, Gro87g, Gro87h, Gro87i, Gro87b, Gro87e, Gro87d, Gro87f, Gro88a, Gro88b, Gro87a].

[Gro86] L.P.J. Groenewegen. Processes. Technical Report 86-20, Department of Computer Science, Leiden University, 1986. Part of [Gro88].

[Gro87a] L.P.J. Groenewegen. Changing managing cooperation in a hierarchy. Technical Report 88-18, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87b] L.P.J. Groenewegen. A critical section model. Technical Report 87-18, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87c] L.P.J. Groenewegen. Decision processes. Technical Report 87-01, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87d] L.P.J. Groenewegen. Dijkstra's semaphore solution. Technical Report 87-29, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87e] L.P.J. Groenewegen. Goeman's solution and a stochastic solution. Technical Report 87-21, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87f] L.P.J. Groenewegen. Lamport's bakery problem. Technical Report 87-32, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87g] L.P.J. Groenewegen. Modelling. Technical Report 87-05, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87h] L.P.J. Groenewegen. Parallel processes. Technical Report 87-06, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro87i] L.P.J. Groenewegen. Two examples of a parallel control process. Technical Report 87-11, Department of Computer Science, Leiden University, 1987. Part of [Gro88].

[Gro88a] L.P.J. Groenewegen. Trap process hierarchy: an almighty manager. Technical Report 88-15, Department of Computer Science, Leiden University, 1988. Part of [Gro88].

[Gro88b] L.P.J. Groenewegen. Trap process hierarchy: cooperating managers. Technical Report 88-17, Department of Computer Science, Leiden University, 1988. Part of [Gro88].

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Mass., USA, 1979.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction.* Prentice-Hall, Inc., New York, N.Y., second edition, 1997.

[Mor93] P.J.A. Morssink. *Behaviour Modelling in Information Systems Design: Application of the PARADIGM Formalism.* PhD thesis, Department of Computer Science, Leiden University, 1993. Co-promoter: L. Groenewegen.

[RBP$^+$91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., 1991.

[RTF99] UML specification v. 1.3. Technical report, OMG Unified Modeling Language Revision Task Force (UML RTF), June 25 1999.

[Rum96] James Rumbaugh. *OMT Insights: perspectives on modelling from the Journal of Object-Oriented Programming.* Prentice-Hall, Inc., 1996.

[Saa95] Mark Saaltink. The Z/EVES system. `ftp://ftp.ora.on.ca/pub/doc/z-eves-draft.ps.Z`, September 1 1995.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[tHDG$^+$a] P.J. 't Hoen, J.H.M. Dassen, L.P.J. Groenewegen, I.G., P.W.M. Koopman, and G. Engels. Restricted class like descriptions of modules. Technical report, liacs. To Appear.

[tHDG$^+$b] P.J. 't Hoen, J.H.M. Dassen, L.P.J. Groenewegen, I.G. Sprinkhuizen-Kuyper, P.W.M. Koopman, and G. Engels. Class like descriptions of SOCCA packages. Technical report, liacs. To Appear.

[tHDG$^+$c] P.J. 't Hoen, J.H.M. Dassen, L.P.J. Groenewegen, I.G. Sprinkhuizen-Kuyper, P.W.M. Koopman, and G. Engels. Coordinated model development in SOCCA˙ Technical report, liacs. To Appear.

[UML97a] Unified modeling language 1.0. Technical report, Rational Software Corporation, January 13 1997.

[UML97b] UML notation guide 1.0. Technical report, Rational Software Corporation, January 13 1997.

[vdZ96] Jeroen van der Zon. Evolutionary change, the evolution of change management. Master's thesis, Department of Computer Science, Leiden University, April 1996. Internal Report IR–96–06. `ftp://ftp.wi.LeidenUniv.nl/pub/CS/MScTheses/vdzon.96.ps.gz`.

[Wil95] R.F. Willemsen. TEMPO and SOCCA: Concepts, modelling and comparison. Master's thesis, Department of Computer Science, Leiden University, May 1995. Internal Report IR–95–09. `ftp://ftp.wi.LeidenUniv.nl/pub/CS/MScTheses/willemsen.95.ps.gz`.

[ZZads] Z archive. URL: `http://www.comlab.ox.ac.uk/archive/z.html`, 1994 onwards.