A formalisation of SOCCA using Z

Part 1: the type level concepts

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

5 February 1999

Abstract

This document starts the formalisation of the modelling language of the SOCCA OO method in the formal specification language Z. It captures the "static" aspects of SOCCA, i.e. the meaning of SOCCA language elements at the type level, rather than "dynamic" aspects (dealing with the instance level (objects in execution)). It is restricted to core SOCCA; proposed extensions are not formalised.

Contents

[*] Please use the primary author's contact address: <jdassen@wi.LeidenUniv.nl>.

1 Introduction

With this document, we start the formalisation of SOCCA in Z. In this introduction, we briefly summarise SOCCA and the formal specification language Z, and we explain the goals of the formalisation, we discuss the part of the formalisation addressed in this document, and the structure of that part.

1.1 SOCCA

SOCCA [EG94](Specification of Coordinated and Cooperative Activities) is a graphical formalism and associated method for object oriented modelling of software systems and software development processes (see e.g. [DKW98]) which is under active development at Leiden University. The main aim of the SOCCA project is to extend object oriented modelling with means to precisely describe communication in a well-integrated fashion.

SOCCA shares a lot of elements and concepts with other graphical notations in widespread use in the area of object orientation, especially OMT [RBP$^+$91] and UML [UML97a]. This reduces the learning curve for SOCCA.

1.2 Goals of the formalisation

Like many other visual languages for expressing object oriented models, SOCCA is semi-formal: while SOCCA allows the modeller to express herself clearer than by using natural language only, there is no formal definition of the semantics of SOCCA models yet. With this document, we start a proper formalisation of SOCCA by means of using a formal specification language.

The benefits of applying formal methods in software engineering and computer science in general are well-known [NAS95]. In addition to attempting to realise these benefits, there can be several goals specific to the development of a formal specification of a modelling language like SOCCA:

- Explaining the structure of models for teaching purposes.
- Preparing for building tools to support model development such as graphical editors, analysis tools and a process centered software engineering environment.
- Exploring the underlying structure of models, i.e. improving the understanding of the language elements and their interaction.

For teaching purposes, the focus of a formalisation should primarily be on explaining the concepts of the modelling language, and showing how these apply in practice. A tutorial-like approach supporting learning by example is probably the most suitable in this case.

In building tools to support model development, a number of issues arise that are outside of the scope of the modelling language itself, such as user interface design, support for group-ware model development, and more basic decisions like choosing efficient representations. While having a well-defined semantics is very useful in building a tool and handling these issues, the development of a formal semantics should not be influenced by issues pertaining only to tool development, as addressing them would increase the size of the semantics and lessen its generality and understandability.

The approach we take here aims primarily at achieving a precise description of the underlying structure of SOCCA models, to highlight its concepts (language features) and their interaction. The other goals, though important in themselves, are secondary.

In light of the first goal, we combine our exposition of the specification with several examples, to make it more accessible for those who do not know SOCCA very well yet. We do not employ a "translational approach" ([EA98]) (focusing on formalising individual SOCCA models): we want to formalise SOCCA language elements and their interaction in general. We believe that translational approach would have resulted in more direct and more easily understandable formalisations, but that it would not have contributed as much to a precise understanding of the SOCCA language in general. Others believe differently, and a translational approach is employed in some UML formalisation efforts (e.g. [BR98, SF97]).

SOCCA is still evolving, both as a method and as a language. For the language, most of the evolution is in the development of extensions; there is a core SOCCA language which by now has become fairly stable. The evolution of SOCCA since its original publication has for the most part been driven by practical experience. The development of this formalisation has forced us to reexamine the core language from a different perspective, which led to new insights and made us modify some aspects of the core language. This documents reflects both the evolution of SOCCA concepts and the evolution of SOCCA notation.

It is not a goal of this formalisation to halt the evolution of SOCCA as a language. Rather, we wish to provide a baseline for extensions and changes to build upon.

1.3 Z

The formalisation is done in the formal specification language Z [Spi92].

Several factors positively influenced our choice of Z as the specification language.

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, typecheckers and theorem provers.

Z unfortunately also has some drawbacks for our purpose.

Notation The Z notation is an acquired taste. It can be highly compact and often allows for several ways to express something. Unfortunately, the compactness is achieved by the use of a plethora of symbols.

Not executable Z is by its nature not a language for writing executable specifications; most Z users rightly feel that aiming for executable specifications results in specifications that are not abstract and general enough and that are too large. As one of the intended uses of this specification effort is in the development of tool support for SOCCA, we have chosen to keep the specification style constructive wherever we felt a constructive style would not be awkward. It should be possible to translate large parts of the specification into a suitable language or animate it (see e.g. [Dil94]) without great effort.

During the writing of this document, we have regularly checked its Z parts using Z/EVES [Saa95, MS97, Saa97], a theorem prover for Z. Information about it can be found at `http://www.ora.on.ca/z-eves/welcome.html`.

Where we judged it necessary, the notation in this document exceeds that of regular Z [Spi92]. In such cases, we incorporate instructions to introduce these extensions to Z/EVES.

1.4 Scope of this document

In this document, we formalise the "static", "structural" concepts of the SOCCA language, i.e. the class level. In a forthcoming document we will formalise the "dynamic" aspects (individual objects on the instance level).

1.5 Conventions used in this document

We use some conventions to make the structure of this document more clear:

- Formal Z text is preceded by a natural language explanation of the topic covered. When aspects of Z text merit comments, these comments directly follow the Z text; they are given in a bulleted list.
- Examples are set apart; they start with "**example**" in bold typeface, and end with the □ symbol at the right margin.

2 Z extensions and toolkit

In the course of the formalisation, we will need some small extensions to Z. We will also encounter some notions from discrete mathematics. We will define these here as a "toolkit" or "library" for later use.

[Spi92] does not include the "is strict superset" ($\supset$) and "is superset or equal" ($\supseteq$) infix relations.

Thus, we must introduce theses 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.

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

2.1 Partial orders

In the formalisation, we need the notion of *partial orders*, for instance to describe the nature of the inheritance relationship.

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

2.2 Covering relation

The covering relation (terminology from [DP90, 1.8, p. 7]) is the relation depicted in a Hasse diagram (a "minimal" depiction of a partial order, i.e. one without transitive edges), with all reflexive edges added.

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

3 Graphical notation and examples

In the course of the formalisation, we will use examples to illustrate the relationship between the formalisation of SOCCA in Z and actual SOCCA models.

The prevalent notation in OO has been shifting since the original SOCCA paper [EG94]. In our examples we have updated the notation to as compatible with UML (which is considered to be the emerging standard) as possible.

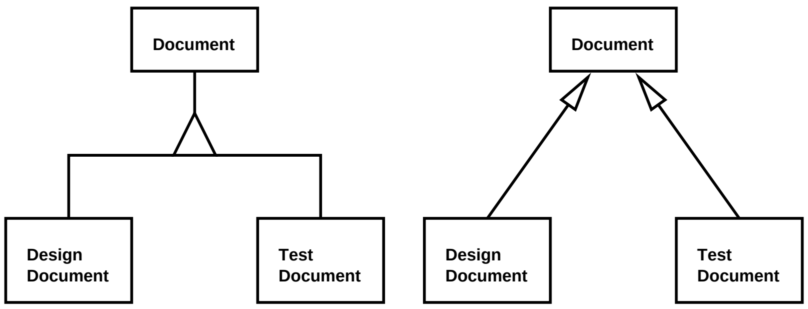
Our definition of the semantics of SOCCA is in terms of an abstract syntax for SOCCA, expressed as Z schemata rather than in terms of SOCCA's visual syntax, as this makes the task somewhat more manageable. The precise visual syntax of SOCCA in most cases does not matter. In a few cases however, the change in graphical notations is significant for interpreting SOCCA models.

Originally, SOCCA's class diagrams were depicted in EER style (see e.g. [EN94]). We will use UML-style class diagrams here (see [UML97a]).

Example In the past, SOCCA class diagrams used the tree-like generalisation symbol from OMT ([RBP$^+$91]) as depicted in figure 1a.

This generalisation symbol might lead one to assume that generalisation is a relationship between a set of classes (children) and classes (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 UML, which can be seen as the proper successor notation to OMT's notation, generalisation is 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 1b. As we shall see, generalisation (inheritance) is a binary relationship in SOCCA.



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

Figure 1: Different notations for inheritance

In practice, no SOCCA models relied on the OMT interpretation of the generalisation, so this change in interpretation does not affect existing models. □

The examples will serve several purposes besides showing SOCCA's updated graphical notation. They will be used to make the subject more concrete without losing abstraction, and to establish the relationship between the mathematical structures in the formalisation and the visual language in which we express SOCCA models.

4 Structure of the formalisation

The structure of the formalisation will follow the structure of SOCCA models closely. SOCCA models consist of several *perspectives*. [EG94] details and motivates this *eclectic* approach to modelling. The structure is also influenced by the way schemas in Z specifications are ordered: they never contain forward references.

4.1 Perspective covered

In this formalisation, we address the four perspectives in SOCCA that are currently mature. Earlier material on SOCCA also discusses a *process perspective*, but no definite decisions have been made on its precise role and formalism. The perspectives we will formalise 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 perspectives are presented in this order, which turns out to be nicely suited for a formalisation in Z, as no forward references will be necessary in the formal text, while the number of forward references in the informal text will be fairly small.

4.2 Levels

The SOCCA formalisation as a whole covers three distinct levels:

The metaclass diagram level This level contains metaclasses like *Metaclass* and *Metarelationship* which capture SOCCA concepts like "class" and "relationship".

The type level This is the level at which SOCCA concept instances including concrete classes (e.g. *Person*) and concrete relationships (e.g. *isMarriedTo*) reside. A SOCCA model deals mainly with this level.

The instance level This is the level of model instances, including objects (class instances) and links (relationship instances), e.g. *John* and *Mary* as instances of the class *Person*, and the link *(John,Mary)* as instance of the relationship *isMarriedTo*.

Example See the UML-like diagram in figure 2. Both `Person` and `Student` are classes, instances of `Metaclass`. The relationship `IsMarriedTo` between `Person` and itself is an instance of the concept of `Metarelationship`. `John` and `Mary` are both `Person`s (`Mary` is a `Student`, which is a special kind of `Person`); the unnamed link `John IsMarriedTo Mary` is an instance of `IsMarriedTo`.

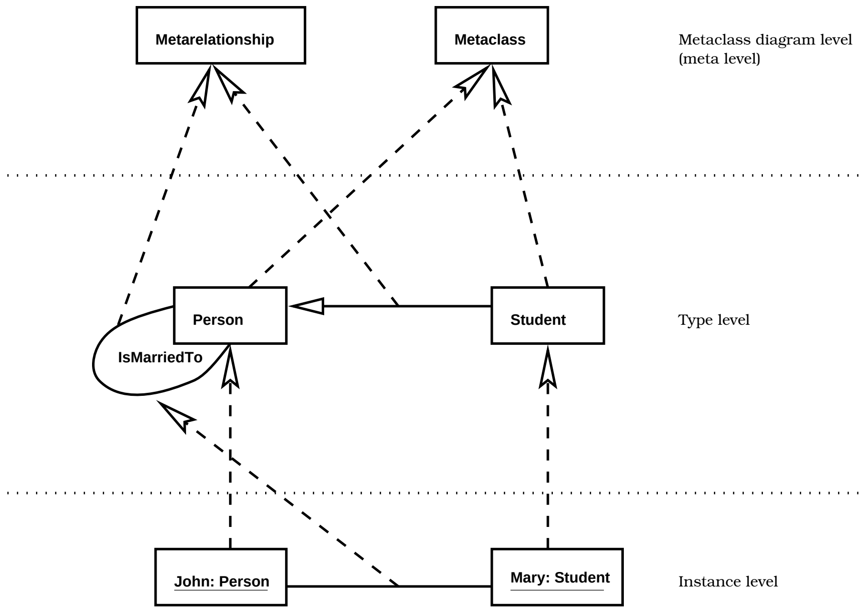


Figure 2: The levels in the SOCCA formalisation

□

In this document, we focus primarily on the type level. Aspects of the metaclass diagram level are treated insofar as they affect the formalisation of the type level; it has not been discussed in prior work on SOCCA.

In other OO research, there is interest in providing a level between class and instance level, the prototypical object level [JBAG97], which describes prototypical behaviour of sets of objects (a kind of universal quantification over sets of objects). While we are interested in extending SOCCA to address modelling at this level, there are currently no concrete proposals for it. Therefore, the prototypical object level is not addressed in this formalisation.

In order to properly formalise this complex structure, we must take into account that Z's notion of equality differs from that in object orientation. In object orientation the notion of *identity* is important: two instances that have the same values for all their constituents need not be the same, because they may differ in their identity. Z's notion of equality is mathematical: two instances (of e.g. schemata) are the same if and only if all their constituents (schema variables) are equal.

This means that wherever we want to work with object orientation's notion of equality in Z, we need to make the identity of instances explicit, via a schema member.

5 The data perspective

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

The data perspective describes the classes and the relationships.

Classes have attributes and methods. Special binary relationships between classes are the [z]*is-a* (inheritance, see page 13), [z]*part-of* (aggregation, see page 17) and [z]*uses* (import, see page 17) relationships; there may also be other "general" relationships relevant to the problem domain at hand; these need not be binary. These general relationships are sometimes known as *associations*.

Note that often the term *relation* is used rather than *relationship*. In the context of this document, that term might lead to confusion with the mathematical concept of relation ($\_ \leftrightarrow \_$), so we will not use it. We will use *relationship* to refer to connections between classes (i.e. something we model).

Because of the amount of information involved, the class diagram is often split into several subdiagrams, such as import/export diagrams, class diagrams without relationships, inheritance diagrams and aggregation diagrams. Such a split often has an informal meaning to the modeller, but is for the purposes of this formalisation merely a matter of convenient representation; it has itself no formal semantics.

Example In figure 3 a fairly typical classdiagram is given that models part of an email system: a mailer uses an external editor (to load a skeleton message, edit it and save it); messages are a kind of text, and consist of a header and a body (each of which are texts in their own regard).

Notation-wise, we follow UML for classes, general relationships, direction in which to read a relationship's name, inheritance and aggregation. For the uses relationship, we use arrows (to indicate the direction) with filled heads (to differentiate them from inheritance arrows).

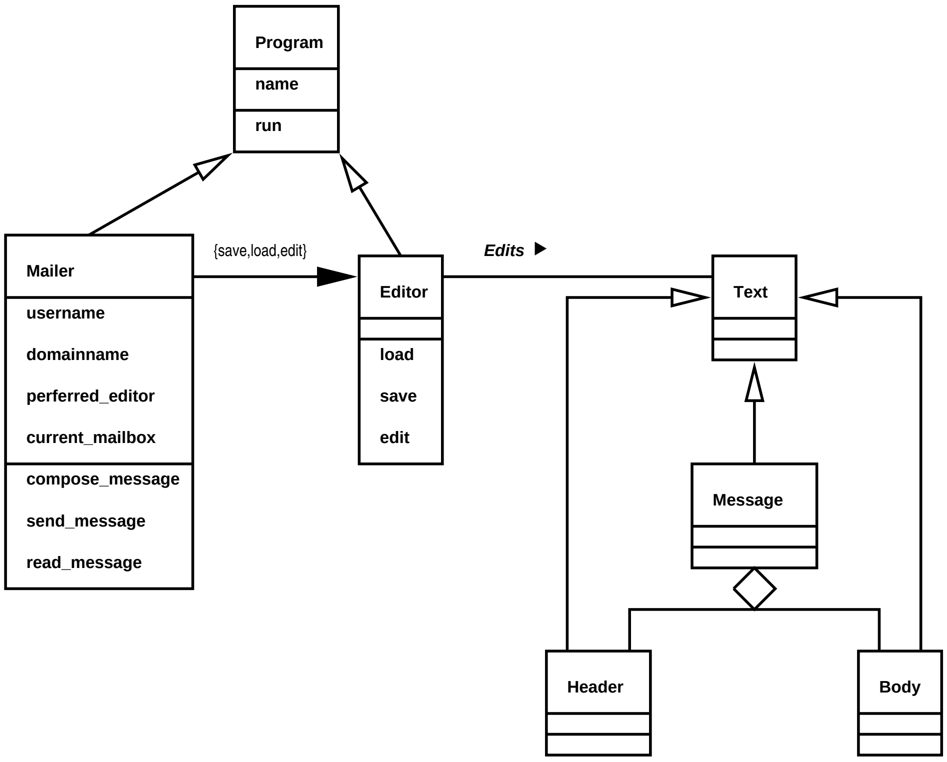


Figure 3: A class diagram.

□

In formalising the data perspectives, we start with the notion of types, then deal with classes: methods and attributes, their visibility and their polymorphism. Then we proceed with relationships (including the special relationships like inheritance), and put the full data perspective together.

5.1 Types

In formalising the data perspective we need a number of types that we deal with in an abstract manner, as given sets.

These include [z]*Identity* for identifiers that provide identity to entities, [z]*Identifier* for generic identifiers, [z]*FeatureName* for names of methods and attributes, [z]*MethodName* for names of methods, [z]*AttributeName* for names of attributes, [z]*ClassName* for class names, and [z]*RelationshipName* for relationship names. The various subtypes of *Identity* will be used to clarify the relationship between concepts on the metaclass diagram level and the type level. Essentially, we will have an identity type for each concept in the metaclass diagram level of which there are multiple instances on the type level.

Note that for some concepts, like *Class*, there is a field like *name* that can be used to uniquely identify an instance of that concept. Nonetheless, for these concepts too we have a separate *Identity*. We have several reasons for this.

- Name and identity are different concepts; in a formalisation they should be separated. Identity is an important concept, that should not be hidden.

- Identifying identity with a particular name (or other attribute) makes modifications more difficult ("Does this change in name have other consequences?").
- Some extensions of SOCCA will cause particular names no longer to be unique. For example, while in this formalisation, names of classes are unique, in an extension of this formalisation that would address the modularisation mechanisms of [Hoe99], this would no longer be true, as names are required to be unique there only within a module.

We declare the types as given sets:

$$[\mathit{Identity}, \mathit{Identifier}, \mathit{FeatureName}, \mathit{ClassName}, \mathit{RelationshipName}]$$

Within feature names, we distinguish between method and attribute names:

$$\begin{array}{|l} \mathit{AttributeName}, \mathit{MethodName} : \mathbb{P}\,\mathit{FeatureName} \\ \hline \langle \mathit{AttributeName}, \mathit{MethodName} \rangle \text{ partition } \mathit{FeatureName} \end{array}$$

For concrete applications (especially with a software tool for developing SOCCA models), it will be necessary to be (much) more specific about these types (e.g. "*FeatureName* consists of the strings of length 255 and less over the ISO 8859-15 character set"). Incorporating this specificity in the formalisation we undertake here would have been possible. However, this would have been distracting, increasing the size of the formalisation while reducing its generality and clarity.

Within SOCCA models, there is a notion of a *type system* for the types of attributes, method arguments and return values etc. The details of this type system are irrelevant to this specification; they may even vary from model to model, allowing one to use a type system that is appropriate for the problem domain. We will therefore treat the type system in an abstract fashion: we assume there is a set of basic types (e.g. $\{\mathrm{Int}, \mathrm{Nat}\}$), a set of type constructors (e.g. $\{\times, \rightarrow, \mathbb{P}\}$) that can be used to introduce new types (like $\mathbb{P}\,\mathrm{Int} \rightarrow \mathrm{Int}$), and a type compatibility relation $\preceq$.

The basic types and type constructors we represent by given sets:

$$[\mathit{BasicType}, \mathit{TypeConstructor}]$$

Types are either basic types, or constructed from types by applying type constructors to them:

$$\begin{array}{lll} \mathit{Type} & ::= & \mathit{basic}\langle\langle \mathit{BasicType} \rangle\rangle \\ & | & \mathit{constructor}\langle\langle \mathit{TypeConstructor} \times \operatorname{seq} \mathit{Type} \rangle\rangle \end{array}$$

And the type relation $\preceq$ expresses compatibility:
Syntax $\preceq$ *inrel*

$$\begin{array}{|l} \_ \preceq \_ : \mathit{partial\text{-}order}[\mathit{Type}] \end{array}$$

5.2 Class

The concept of *class*, a unit of data and behaviour, is at the core of object orientation in SOCCA. A class has a name, some attributes, describing pieces of data, and some methods, describing pieces of behaviour and functionality: actions which it can perform that can be viewed as one unit in its behaviour. We introduce it in several steps.

Example In figure 4 the data perspective part of an example class (taken from figure 3) is depicted: a rectangle, partitioned in three parts, the first listing the class name (*Mailer*), the second containing the attributes (data) (e.g. *username*), and the third listing the methods (operations). □

At first, we will focus only on the concept of class, i.e. the metaclassdiagram level. Once we have that concept described properly, we will derive from it the description of classes at the type level.

method An operation (of a class or object). Also known as a *member function* or procedure or function of a class.

attribute Attributes are (types of) data associated with a class.

feature A method or attribute. In the C++ community, the term *member* is used.

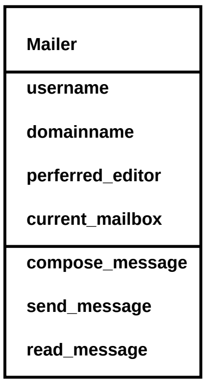


Figure 4: A class.

*ProtoMetaClass*1

$name : ClassName$

$features, methods, attributes : \mathbb{P}\, FeatureName$

$features = methods \cup attributes$

$methods \subseteq MethodName$

$attributes \subseteq AttributeName$

5.3 Visibility

In OO formalisms, the concept of *encapsulation* is important: aspects of classes (like methods and attributes), and sometimes classes as a whole, have a restricted visibility for other elements of a model.

The precise choice of visibility restriction mechanism is not really relevant for SOCCA's goal of combining OO and precise behaviour/communication/coordination description.

In this formalisation, we will describe a basic visibility restriction mechanism, inspired by the one used in C++. We expect current research on SOCCA ([Hoe99]) to result in a more sophisticated visibility restriction mechanism that will address the visibility between classes and objects.

We distinguish between:

public A public feature may in principle be exported to every class. Note that public attributes are discouraged; we recommend to make them private and manipulate them through explicit "get" (read access) and "set" (write access) methods.

protected A protected method may not be exported for use by another class in a call, but is available for use in defining specialised classes. Similarly, a protected attribute is accessible by specialised classes, but not to unrelated ones.

private A private feature may not be exported.

These visibilities influence what features a class will inherit (namely the public and protected ones of ancestors) and what features other classes and objects can access (the public ones) (e.g. for methods: what methods objects will be able to call; we will detail this when formalising the uses relationship (see page 17).

$$Visibility ::= public \mid protected \mid private$$

This way of specifying visibility is fairly crude, but seems to be sufficient for most practical purposes in small to medium sized models. If need be, the public/private/protected scheme can be replaced by a more sophisticated visibility/access control mechanism. This scheme is not fundamental to SOCCA, but we have to formalise it of course.

*ProtoMetaClass*2

*ProtoMetaClass*1

$visibility : FeatureName \rightarrow\!\!\!\!\!\!+ \; Visibility$

$\mathrm{dom}\, visibility = features$

5.4 Binding

At this point, we are near a complete schema for *Class*. What is still missing has to do with support for inheritance. Due to the existence of inheritance in object orientation, identifiers do not always refer to the same things in all contexts (this is sometimes known as *polymorphism* or *overloading*). We will look at a simple example of this first.

Example Consider the situation in figure 5 (this is not a syntactically correct classdiagram: we have used an informal annotation to indicate how `Age` might be implemented). At the level of `Person`, an implementation of `Age` is given. `Manager`, a which inherits from `Person` does not redefine this binding, and thus inherits it. □

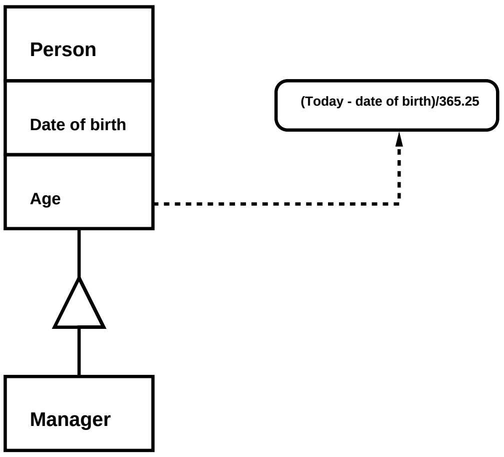


Figure 5: Inheritance of bindings

In general, as a result of polymorphism by inheritance, identifiers (names of methods) do not in all contexts refer to the same thing (implementation). There is a context-dependent *binding* between names and what they refer to.

We make polymorphism explicit by expressing *binding*: the coupling of identifiers to the entities they refer to. For inheritance, we need the binding of feature names to the actual features: feature names are inherited, but their implementation may be overridden.

For binding, we give features an identity (to be able to distinguish between them even when they have the same values). While this is not strictly necessary (we could derive a feature's identity from that of the class(es) it is bound to), it makes things more manageable.

Conceptually, we will extend the types we are about to define with additional information during the course of the formalisation. On the practical level, rather than changing their schemata and restating work, we will add the new information to them via coupling through functions and relationships.

Without a notion of identity, this type of coupling has undesirable side effects: without identity, schemata that have the same values are identical, resulting in couplings "collapsing into each other". With a concept of identity, entities that have the same values otherwise, need not be identical, so this collapse does not occur.

$FeatureIdentity : \mathbb{P}\, Identity$

We use this identity in defining actual members:

FEATURE

$id : FeatureIdentity$
$signature : Type$

$METHOD, ATTRIBUTE : \mathbb{P}\, FEATURE$

$\langle METHOD, ATTRIBUTE \rangle \text{ partition } FEATURE$

A class can declare explicitly to which actual feature a particular feature identifier is bound. However, usually, most of the binding pertaining to a particular class will have been acquired through the inheritance mechanism (see page 15);

the only cases in which a class ought to declare a binding explicitly are when a feature is new, redefined (methods only: attributes cannot be redefined) or is inherited through multiple inheritance in an ambiguous manner.

We explicitly represent the bindings a class declares through two partial functions. These functions are partial for two reasons: not every *MethodName* has to have a binding in the classdiagram and the binding can be acquired by inheritance. After we have formalised inheritance, we will show how the *localmethodbinding*s and *localattributebinding*s are combined to determine bindings within the class diagram as a whole (see page 15).

$$\begin{array}{|l}
\textit{ProtoMetaClass3} \\
\hline
\textit{ProtoMetaClass2} \\
\textit{localfeaturebinding} : \textit{FeatureName} \pfun \textit{FEATURE} \\
\hline
\operatorname{dom} \textit{localfeaturebinding} \subseteq \textit{features} \\
\hline
\end{array}$$

This contains all we need for a schema to express the concept of class. The only thing we add to it are *localmethod-binding* and *localattributebinding* which provide us with convenient abbreviations for aspects of *localfeaturebinding*; they do not add anything new.

$$\begin{array}{|l}
\textit{MetaClass} \\
\hline
\textit{ProtoMetaClass3} \\
\textit{localmethodbinding}, \textit{localattributebinding} : \textit{FeatureName} \pfun \textit{FEATURE} \\
\hline
\textit{localmethodbinding} \cup \textit{localattributebinding} = \textit{localfeaturebinding} \\
\operatorname{dom} \textit{localmethodbinding} \subseteq \textit{methods} \\
\operatorname{ran} \textit{localmethodbinding} \subseteq \textit{METHOD} \\
\operatorname{dom} \textit{localattributebinding} \subseteq \textit{attributes} \\
\operatorname{ran} \textit{localattributebinding} \subseteq \textit{ATTRIBUTE} \\
\hline
\end{array}$$

ClassIdentity is the second subtype of *Identity* we need; more will follow later. All the subtypes are mutually disjoint.

$$\begin{array}{|l}
\textit{ClassIdentity} : \mathbb{P}\,\textit{Identity} \\
\hline
\operatorname{disjoint}\langle \textit{ClassIdentity}, \textit{FeatureIdentity} \rangle
\end{array}$$

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

$$\begin{array}{|l}
\textit{Class} \\
\hline
\textit{identity} : \textit{ClassIdentity} \\
\textit{MetaClass} \\
\hline
\end{array}$$

5.5 Relationships

The data perspective consists of a number of classes, and some relationships between them.

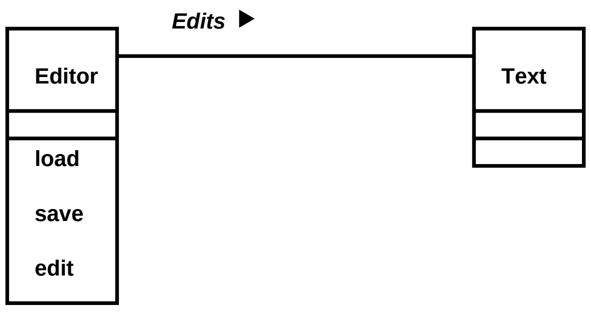


Figure 6: A relationship.

Example In figure 6 a relationship (from the earlier example in figure 3) is depicted. The relationship is the *Edits* relationship between *Editor* and *Text*. An example instance could be the ordered pair (vim, report.tex). □

On the metaclassdiagram level, SOCCA has a few special relationships: [z]*uses* (import (e.g. [GJM91, sect 4.2.1.1, the USES relation])), [z]*part-of* (aggregation, known from EER) and [z]*is-a* (inheritance), each with special properties which we will describe later on. That these relationships are on the metaclassdiagram level, rather than the type level, can be seen as follows: we tend to speak about them as if they were singular ("the uses relationship" etc.), but on the type level, they are plural ("the uses relationship between class A and class B", "the uses relationship between class B and class C" etc.); only on the metaclassdiagram level they are singular; the plural ones on the type level are instances of the singular ones on the higher level.

On the meta classdiagram level, SOCCA has general relationships, that are similar to associations in (E)ER modelling. These relationships are defined by the modeller to capture relationships that are specific to the domain being modelled.

Meta relationships On the metaclassdiagram level, we distinguish four types of relationships: uses, partof, inheritance, and general.

$$\mathit{MetaRelationship} ::= \mathit{UsesRelationship} \mid \mathit{PartOfRelationship} \mid \mathit{IsARelationship} \mid \mathit{GeneralRelationship}$$

Each of these can have instances on the type level; these instances have identity.

$$\begin{array}{|l}
\mathit{RelationshipIdentity} : \mathbb{P}\,\mathit{Identity} \\
\hline
\mathrm{disjoint}\langle \mathit{ClassIdentity}, \mathit{FeatureIdentity}, \mathit{RelationshipIdentity} \rangle
\end{array}$$

The actual instances are named, have an arity, and an ordered set of participants.

$$\begin{array}{|l}
\textit{Relationship} \\
\hline
\mathit{identity} : \mathit{RelationshipIdentity} \\
\mathit{name} : \mathit{RelationshipName} \\
\mathit{type} : \mathit{MetaRelationship} \\
\mathit{arity} : \mathbb{N}_1 \\
\mathit{participants} : \mathrm{seq}\,\mathit{Class} \\
\hline
\#\mathit{participants} = \mathit{arity} \\
\mathit{type} \in \{\mathit{UsesRelationship}, \mathit{PartOfRelationship}, \mathit{IsARelationship}\} \Rightarrow \mathit{arity} = 2 \\
\hline
\end{array}$$

- A modeller may choose to use a lax interpretation of the ordering of the participants in a particular relationship.
- We do not require a namespace for the names of relationships, as this would conflicts with inheritance of relationships.
- Usually, the name is indicated in the graphical notation only for general relationships.

Data perspective domain In a model's data perspective, we describe classes and relationships between them.

$$\begin{array}{|l}
\textit{DataPerspectiveDomain} \\
\hline
\mathit{classes} : \mathbb{P}\,\mathit{Class} \\
\mathit{relationships} : \mathbb{P}(\mathit{Relationship}) \\
\hline
\#\mathit{classes} = \#\{i : \mathit{Identity} \mid \exists\, c : \mathit{classes} \bullet c.\mathit{identity} = i\} \\
\#\mathit{relationships} = \#\{i : \mathit{Identity} \mid \exists\, r : \mathit{relationships} \bullet r.\mathit{identity} = i\} \\
\hline
\end{array}$$

- The constraints express that each class and each general relationship has a unique identity.
- With this schema, we highlight the differences between the intensional and extensional meanings (terminology of [JBAG97]) of "class" and "relationship": *Class* and *Relationship* express the intensional meaning (a type of sets of objects / links); *classes* and *relationships* express the extensional meaning (the particular set of objects / links in a particular model).

On top of this "data structure", we will add constraints to fully describe the data perspective.

Constraints on the domain Throughout the data perspective, we focus only on *classes*, the concrete set of classes involved, rather than *Class* (the set of all potential classes).

Thus, we do not consider all potential relationships, but only relationships between the concrete classes (*classes*).

DataPerspectiveDomainConstraints

DataPerspectiveDomain

$$\forall\, rel : relationships \bullet \\ \quad \forall\, c : \operatorname{ran}\ rel.participants \bullet c \in classes$$

Namespace The classes form a namespace: a set of elements with a name within which an element's name is sufficient to identify it.

DataPerspectiveNameSpaces

DataPerspectiveDomain

$$\forall\, c, d : classes \mid c.name = d.name \bullet c = d$$

- Note that we do not have a similar namespace constraint for relationships.

5.6 Inheritance

The inheritance relation *is-a* is fundamental to the data perspective. It is a partial order; thus it forms a hierarchy (there are no cycles in the inheritance, except for the trivial (reflexive) ones). As a partial order, it is reflexive and transitive, but the reflexive and transitive edges are seldomly drawn in the class diagram (i.e. the class diagram usually depicts its covering relation).

Note that the *is-a* relations allows for multiple inheritance (classes having more than one parent class).

We also introduce an *is-directly-a* relation derived from *is-a* to refer to a parent-child relation, rather than a ancestor-descendant one.

IsA

DataPerspectiveDomain
$is\text{-}a : Class \leftrightarrow Class$
$is\text{-}directly\text{-}a : Class \leftrightarrow Class$

$$is\text{-}directly\text{-}a = \{ r : Class \times Class \mid \exists\, g : relationships;\ c, d : classes \bullet \\ \quad r = (c, d) \wedge g.type = IsARelationship \wedge g.participants = \langle c, d \rangle \}$$

$$is\text{-}a = is\text{-}directly\text{-}a^{*}$$

$$is\text{-}a \in partial\text{-}order[Class]$$

Our notion of inheritance is based on *substitutivity*: a descendant class can occur and should be usable anywhere any of its ancestor classes can. This means that it has all the public and protected methods and attributes its parents have, and that it participates in the general, *uses* and *part-of* relationships they participate in: inheritance applies to the non-*is-a* special relationships as well.

We define inheritance on the data perspective only; inheritance plays a role in the other perspectives, but in the other perspectives, inheritance imposes no constraints above what is necessary for a consistent model. While it is possible to have a notion for inheritance on another perspective, say, the behaviour perspective, there are often quite different, but equally valid notions of inheritance possible; [EE94] for instance identifies two different, but equally valid, notions of inheritance of behaviour.

Example Consider the classdiagram (fragment) in figure 7: as `Design` is a `Document`, and `Projectmanager` *monitors* `Document`, one can infer that `Projectmanager` *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.

This example also shows why we did not impose a namespace constraint on the names of relationships. If we did, we would have to find a different name for the relationship between *Manager* and *Design*. Also note that `monitors` is a directed relationship.

□

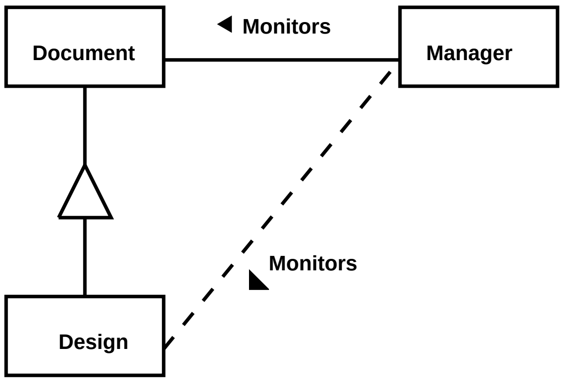


Figure 7: Inheritance of relationships

DataPerspectiveInheritance
IsA

$$
\begin{array}{l}
\forall p, c : classes \mid c \ \underline{is\text{-}a}\ p \bullet \\
\quad (\forall f : p.features \mid p.visibility\ f \neq private \bullet f \in \operatorname{dom} c.visibility \wedge p.visibility\ f = c.visibility\ f) \wedge \\
\quad (\forall R : relationships \mid R.type \neq IsARelationship \bullet \\
\qquad \forall i : 1 \mathinner{..} R.arity \mid R.participants\ i = p \bullet \exists S : relationships \bullet \\
\qquad\quad S.type = R.type \wedge \\
\qquad\quad S.arity = R.arity \wedge \\
\qquad\quad S.participants\ i = c \wedge \\
\qquad\quad (\forall j : (1 \mathinner{..} R.arity) \setminus \{i\} \bullet S.participants\ j = R.participants\ j))
\end{array}
$$

- Private attributes and methods cannot be inherited; protected and public ones are always inherited. We will formalise this in the *DataPerspectiveBinding* schema (page).
- Features that exist in both parent and child, have the same visibility.
- If a parent participates in a particular rôle (position) in a relationship, its children can fulfill the same role.

Example For example, say we have a relationship *sale(Person,Good,Person)* to describe a person selling a good to another person, and we have specialised persons *Salesman* and *Client* then the *sale* relationship also encompasses *sale(Person,Good,Client)*, *sale(Salesman,Good,Person)* and *sale(Salesman,Good,Client)*. □

As with *is-a*, in our diagrams we tend to leave out details that can be easily inferred: usually we only draw a relationship between the "highest" classes in the *is-a* hierarchy it pertains to. In the previous example, we would draw *sale* between *Person*, *Good* and *Person*, but we would not draw all the implications like between *Salesman*, *Good*, *Client*.

Inheritance of binding Now that we have formalised the inheritance relationship, we can deal with polymorphism by inheritance formally. As discussed earlier, this is the property that a particular method or attribute name need not always correspond to the same entity. Rather, in a particular context (class), a particular identifier is bound to a particular entity. We express this *binding* through functions *methodbinding* and *attributebinding*.

These bindings are determined by inheritance and local overriding (*localmethodbinding* and *localattributebinding*).

In the constraints on *methodbinding* and *attributebinding*, we deal with the potential ambiguity resulting from multiple inheritance: if two parents of a class have a different binding for a particular method or attribute name, the child class has to disambiguate by providing an explicit local binding for that name.

methodbinding is partial, as it is defined only for the concrete methods within the concrete *classes*, rather than all potential methods (*METHOD*) of all potential classes (*Class*). Likewise for *attributebinding*. As a side effect, this allows for method names that do not have an implementation attached to them: *abstract methods*. These methods, also known as pure virtual methods or deferred methods, are useful because they allow the modeller to introduce an operation at a suitable level high up in a class hierarchy at which the commonality can be expressed, but at which no reasonable implementation can be specified.

- *attributebinding* is total on the concrete (class, attributename) pairs.

- *methodbinding* is not necessarily total on the concrete (class, methodname) pairs: it can be partial to indicate abstract methods.

Example Multiple inheritance introduces a complication: how does binding work when two (or more) parents of a class have different bindings for a particular feature?

Consider for example, the informal classdiagram (fragment) in figure 8. A `.c file` is both a `File` and `C code`;

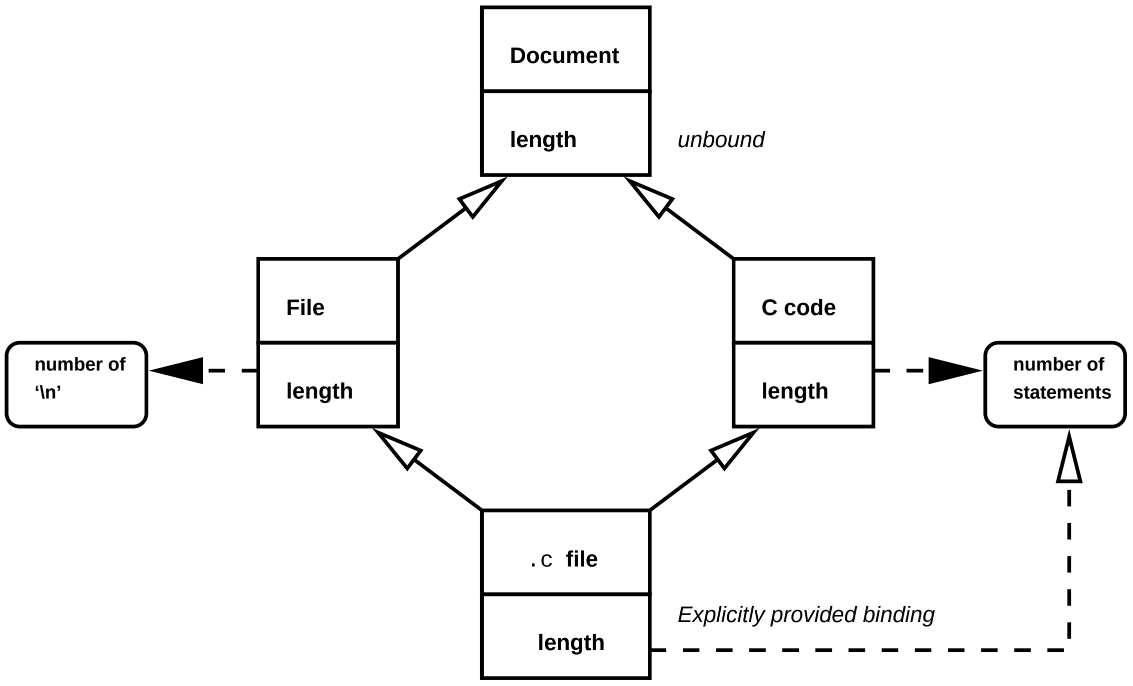


Figure 8: Binding and multiple inheritance

for `File`, `length` is defined as the line count; for `C code`, `length` has a different definition: the number of statements (as used in some definitions of (K)LOC).

We resolve this complication by requiring that, when two or more parents (here, `File` and `C code`) have different (non-empty) bindings for a particular feature (`length`), the child class (`.c file`) explicitly specifies a (potentially new) binding (to `number of statements`). When all parents that have a non-empty binding to the same feature it is inherited (binding takes precedence over lack of binding). □

DataPerspectiveBinding

$$\begin{array}{l}
DataPerspectiveInheritance \\
featurebinding, methodbinding, attributebinding : Class \times FeatureName \pfun FEATURE \\
\hline
featurebinding = methodbinding \cup attributebinding \\
\operatorname{dom} methodbinding \subseteq Class \times MethodName \\
\operatorname{ran} methodbinding \subseteq METHOD \\
\operatorname{dom} attributebinding \subseteq Class \times AttributeName \\
\operatorname{ran} attributebinding \subseteq ATTRIBUTE \\
\forall p, c : Class \mid c \ \underline{is\text{-}a} \ p \bullet \forall fn : FeatureName \mid (p, fn) \in \operatorname{dom} featurebinding \land p.visibility \ fn \neq private \bullet \\
\qquad (c, fn) \in \operatorname{dom} \ featurebinding \\
\forall p_1, p_2, c : Class \mid p_1 \neq p_2 \land c \ \underline{is\text{-}directly\text{-}a} \ p_1 \land c \ \underline{is\text{-}directly\text{-}a} \ p_2 \bullet \\
\qquad \forall fn : FeatureName \mid \\
\qquad\qquad featurebinding \ (p_1, fn) \neq featurebinding \ (p_2, fn) \land \\
\qquad\qquad p_1.visibility \ fn \neq private \land p_2.visibility \ fn \neq private \bullet \\
\qquad\qquad\qquad fn \in \operatorname{dom} \ c.localfeaturebinding \\
\forall c : Class \bullet \forall a : FeatureName \bullet \forall i : FEATURE \bullet \\
\qquad (c, a) \mapsto i \in featurebinding \Leftrightarrow \\
\qquad\qquad (a \mapsto i \in c.localfeaturebinding \lor \\
\qquad\qquad (a \mapsto i \notin c.localfeaturebinding \land \\
\qquad\qquad\qquad (\forall p : Class \mid c \ \underline{is\text{-}directly\text{-}a} \ p \bullet (p, a) \mapsto i \in featurebinding))) \\
\forall c : classes \bullet \forall a : c.features \bullet \forall p : classes \mid c \ \underline{is\text{-}a} \ p \land (p, a) \in \operatorname{dom} featurebinding \bullet \\
\qquad (featurebinding \ (p, a)).signature \preceq (featurebinding \ (c, a)).signature \\
\forall c : Class; \ fn : FeatureName \mid (c, fn) \in \operatorname{dom} \ featurebinding \bullet \\
\qquad c.visibility \ fn = private \Rightarrow \\
\qquad\qquad \neg \ (\exists d : Class; \ fn_2 : FeatureName \bullet \\
\qquad\qquad\qquad ((c, fn) \neq (d, fn_2) \land (featurebinding(c, fn) = featurebinding(d, fn_2)))) \land \\
\qquad c.visibility \ fn = protected \Rightarrow \\
\qquad\qquad (\forall d : Class; \ fn_2 : FeatureName \mid featurebinding(c, fn) = featurebinding(d, fn_2) \bullet \\
\qquad\qquad\qquad fn = fn_2 \land (\exists p : Class \bullet c \ \underline{is\text{-}a} \ p \land d \ \underline{is\text{-}a} \ p \land featurebinding(p, fn) = featurebinding(c, fn)))
\end{array}$$

Note

$$\begin{array}{l}
DataPerspectiveBinding \vdash \\
\qquad \operatorname{dom} \ featurebinding \subseteq \{p : Class \times FeatureName \mid \exists c : classes \bullet \exists f : c.features \bullet p = (c, f)\}
\end{array}$$

- Once a member has become bound, it cannot become unbound in a specialised class.
- If there is no explicit *localmethodbinding*, a binding can be obtained through inheritance.
- A *methodbinding* entry is generated either by an explicit *localmethodbinding*, or by unambiguous inheritance; ambiguity due to multiple inheritance must be resolved by an explicit *localmethodbinding*.
- Signatures of members can only be changed through inheritance in a fashion consistent with the type system employed in the model.
- All attribute names that actually occur are bound.
- When a binding is private, it is not shared.
- Protected bindings can only be shared by classes related to an ancestor with the same binding.

The *uses* relationship In a class diagram, there is a *uses* relationship between classes which describes import (arcs are labelled with method names through *useslabel*; associated information like the method's signature can be accessed using the method name).

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

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 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.

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 *public, private, protected* visibility mechanism was popularised by the C++ programming language [Str97]. In that language, the accessibility/visibility specified by these keywords regulates access control on the basis of classes only. For example, an object of a particular class can access private features of another object of that class. This is not the case in SOCCA, where this kind of access will be regulated too (see figure 9). We will specify this when formalising the concepts dealing with the instance level of SOCCA models.

callee		member	
caller	public	protected	private
same object	Y	Y	Y
other object, same class	Y	N	N
other object, descendant class	Y	N	N
other unrelated object	Y	N	N

Figure 9: Potential access.

Thus, there is no difference in accessibility between *protected* and *private* members in a caller–callee situation. The only difference between *protected* and *private* is that *protected* members are used in defining members of a descendant class.

Uses

$DataPerspectiveInheritance$
$uses : Class \leftrightarrow Class$
$useslabel : (Class \times Class) \rightarrow\!\!\!\!\!+\, \mathbb{P}\, MethodName$

$$uses = \{r : Class \times Class \mid \exists g : relationships;\ c, d : classes \bullet \\ \quad r = (c, d) \wedge g.type = UsesRelationship \wedge g.participants = \langle c, d \rangle\}$$

$$\mathrm{dom}\ useslabel = uses$$

$$\forall c, d : classes;\ M : \mathbb{P}\, MethodName \mid (c, d) \mapsto M \in useslabel \bullet \\ \quad M \subseteq \{m : d.methods \mid d.visibility\ m = public\} \cup \\ \qquad (\textbf{if}\ c = d\ \textbf{then}\ \{m : d.methods \mid d.visibility\ m \in \{protected, private\}\}\ \textbf{else}\ \varnothing)$$

- We restrict *uses* and *useslabel* to the concrete classes in our model.
- *useslabel* is used to annotate edges with the names of methods that are used. The labelling is not inherited; there need not be a relation between a parent class' *useslabel* and an child class'. A particular *useslabel* may be the empty set; this can occur between wholly unrelated classes, but also when two classes formally have a *uses* relationship, due to inheritance, which is not used.
- This schema only captures the constraints on the class level. This is not sufficient to describe whether or not a particular method of a particular object can be called. See the merchant example in [Hoe99]. On the instance level, we will add additional constraints that prevent different objects from using each other's protected and private methods.

The *part-of* relationship The *part-of* relationship describes aggregation between classes. We regard aggregation as a binary relationship (a (single) part *is part of* a whole) rather than an n-ary relationship (a set of parts *forms* a whole).

PartOf

$DataPerspectiveInheritance$
$part\text{-}of : Class \leftrightarrow Class$

$$part\text{-}of = \{r : Class \times Class \mid \exists g : relationships;\ c, d : classes \bullet \\ \qquad r = (c, d) \wedge g.type = PartOfRelationship \wedge g.participants = \langle c, d \rangle\}$$

- We restrict *part-of* to the concrete classes (*classes*) in our model, rather than all potential classes (*Class*).
- *uses* and *part-of* are derived components ([Spi92, p. 3]); they do not impose additional constraints, but are merely aliases (abbreviation definitions) for the binary relations underlying the relationships within this scheme. Unfortunately, Z has no syntactical construct to indicate this type of use explicitly.
- There are no constraints on *part-of*. There are some constraints that are often, but not always reasonable, for example that *part-of* should be a forest (thus, something can not be *part-of* itself, nor be *part-of* two different classes). One can also argue that it should be transitive, but that is a matter of preference.

The full data perspective At this point, we can put the data perspective together.

$$
\begin{array}{|l}
\hline
\textit{DataPerspective} \\
\textit{DataPerspectiveDomain} \\
\textit{DataPerspectiveDomainConstraints} \\
\textit{DataPerspectiveNameSpaces} \\
\textit{DataPerspectiveInheritance} \\
\textit{DataPerspectiveBinding} \\
\textit{Uses} \\
\textit{PartOf} \\
\hline
\end{array}
$$

As an illustration of inheritance of relationships, we can now see

$$
\begin{array}{l}
\textit{DataPerspective} \vdash \forall p, c, q : \textit{classes} \mid c \ \underline{\textit{is-a}}\ p \bullet \\
\qquad (p \ \underline{\textit{part-of}}\ q \Rightarrow c \ \underline{\textit{part-of}}\ q) \wedge (q \ \underline{\textit{part-of}}\ p \Rightarrow q \ \underline{\textit{part-of}}\ c)
\end{array}
$$

5.7 Discussion

In the process of formalising SOCCA a large number of choices have been made; we will motivate some of them, and point out alternative choices.

We have chosen to model the identity of SOCCA entities separate from their names. This provides flexibility in dealing with issues of unique names. The identities of an entity is unique globally within a model; its name may be unique, or unique only within a particular namespace. With this approach it is easy to modify the formalisation for example to add a namespace constraint for relationships.

The formalisation does not address the actual type system to be used in SOCCA models. We believe this to be a good thing. The important concepts of SOCCA are independent of any particular type system, so there is no need to formalise a particular type system. Furthermore, a modeller should be free to work with a type system that is natural for the subject area at hand.

The visibility/hiding mechanism is to some degree similar. However, unlike the type system, it is quite difficult to treat the visibility/hiding mechanism in an abstract fashion. Rather than treat it abstractly, we have chosen to use a concrete mechanism that is small, but sufficient for showing the issues involved. Like with the type system, the concrete mechanism is not crucial to SOCCA and should be left to the modeller.

The section on binding shows how we integrate entities into more complete model fragments: via functions. This allows us to some freedom in organising the formal material, and, more importantly, allows us to extend prior material without restating it (recall that plain Z does not have the features for reuse that OO versions of Z have).

6 The behaviour and functionality perspectives

So far we have described the data perspective of a SOCCA model, which describes a static structure of classes and their relationships. Now we will 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 will 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 labelled transitions between them. As we will see in the next section, 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 its “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.

6.1 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 will describe them in an FSM-like manner. To emphasise the distinction between the type level and the instance level, we will 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 does not 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 does not 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.

$[\mathit{STATE}]$

Transitions can in general be labelled with plain method names (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.

$$\begin{aligned} \mathit{SYMBOL} ::= {} & \mathit{ml}\langle\!\langle \mathit{MethodName} \rangle\!\rangle \\ \mid {} & \epsilon \mid \mathit{act}\langle\!\langle \mathit{MethodName} \rangle\!\rangle \mid \mathit{call}\langle\!\langle (\mathit{ClassName} \times \mathit{MethodName}) \rangle\!\rangle \end{aligned}$$

With these, we describe the structure of an STD in general (on the meta classdiagram level):

MetaSTD

$$\begin{aligned} & \mathit{states} : \mathbb{P}\,\mathit{STATE} \\ & \mathit{labels} : \mathbb{P}\,\mathit{SYMBOL} \\ & \mathit{transrel} : (\mathit{STATE} \times \mathit{SYMBOL}) \leftrightarrow \mathit{STATE} \\ & \mathit{initial} : \mathbb{P}\,\mathit{STATE} \\ & \mathit{final} : \mathbb{P}\,\mathit{STATE} \end{aligned}$$

$$\begin{aligned} & \mathit{initial} \cup \mathit{final} \subseteq \mathit{states} \\ & \mathit{states} \neq \varnothing \Rightarrow \mathit{initial} \neq \varnothing \\ & \mathit{transrel} \subseteq (\mathit{states} \times \mathit{labels}) \times \mathit{states} \\ & \mathit{labels} = \{ l : \mathit{SYMBOL} \mid \exists s_1, s_2 : \mathit{states} \bullet ((s_1, l), s_2) \in \mathit{transrel} \} \end{aligned}$$

And adding identity to it, we get regular STDs.

$STDIdentity : \mathbb{P}\, Identity$

$\mathrm{disjoint}\langle ClassIdentity, FeatureIdentity, RelationshipIdentity, STDIdentity\rangle$

STD

$Identity : STDIdentity$
$MetaSTD$

Notes:

- *STATE* is the type of states in *STD*s; *states* is the set of actual states in a specific *STD*. Therefore, both *initial* and *final* have to be in *states*.
- *STD*s 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.

$edges : STD \rightarrow (STATE \leftrightarrow STATE)$

$$\forall std : STD \bullet \\ \quad edges\ std = \{e : STATE \times STATE \mid \exists p, q : STATE \mid \\ \qquad e = (p,q) \bullet (\exists sym : SYMBOL \bullet (p, sym) \mapsto q \in std.transrel)\}$$

To describe the realisation of behaviour of objects on the instance level, we will 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.

6.2 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 unlabelled or labelled with the names of operations exported by that class to other classes or itself:

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

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

BehaviourPerspective

$DataPerspective$
$externalbehaviour : Class \leftrightarrow STD$

$externalbehaviour \subseteq classes \times ExternSTD$

$\forall c : classes;\ m : MethodName \bullet \#\{estd : STD \mid (c, estd) \in externalbehaviour \wedge ml\ m \in estd.labels\} \leq 1$

$$\forall c : classes \bullet \\ \quad \{m : MethodName \mid \exists estd : STD \mid (c, estd) \in externalbehaviour \bullet ml\ m \in estd.labels\} \subseteq c.methods$$

- Note that not all the method names need to occur in the external behaviour STD. For instance, it depends on the particular model whether or not it is useful to include labels for abstract methods in it.
- 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]. In [Hoe99] this feature is being used to provide class-like descriptions of sets of classes. For now, multiple external STDs for the same class are disjoint in that a method name can occur as a label in at most one of them; possible extensions in which this requirement is weakened may prove useful in modelling certain multithreaded systems. It is useful for cases where one can distinguish a number of distinct "facets" to an object. For example, take a (composite) object representing a multi-windowed, multi-threaded application. In such an application, each window can have its own functionality (methods), and its own state. It is rather natural to model this by several disjoint external STDs.

6.3 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. Methods with which we do not associate an internal behaviour STD are termed *abstract methods*.

An internal behaviour STD's transitions are labelled 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 labelled *uses* relation from the class to itself.

Example In figure 10 you find a typical internal STD (belonging to a method *Foo*), which makes two calls (one, *B.bar* to a class *B*; the other to *Baz* within the class) and does some internal stuff (an unlabelled (ϵ) transition).

Rather than using separate end states, an ϵ transition from what is effectively an end state to the initial state, which is also an end state, is provided. This convention is used in several SOCCA publications. The underlying intuition is that of a process that in some sense becomes dormant after handling a call and is woken up by a new call.

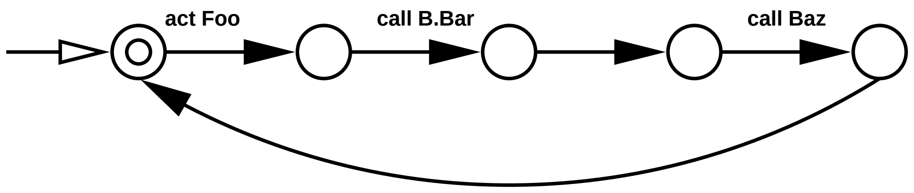


Figure 10: A typical internal STD

□

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

$$
\begin{array}{|l}
InternSTD : \mathbb{P}\,STD \\
\hline
\forall s : InternSTD \bullet \\
\quad s.labels \subseteq \{\epsilon\} \cup (\operatorname{ran} act) \cup (\operatorname{ran} call) \land \\
\quad (\forall l : s.labels;\ i : s.initial;\ st : s.states \mid (i, l) \mapsto st \in s.transrel \bullet st \notin s.initial) \land \\
\quad (\forall s_1, s_2 : s.states;\ l : s.labels \mid (s_1, l) \mapsto s_2 \in s.transrel \land l \in \operatorname{ran} act \bullet s_1 \in s.initial)
\end{array}
$$

- Initial states do not connect to each other.
- All transitions from an initial state are labelled with an "act" label.

It is customary to leave out classnames in calls in the graphical notation of STDs where this does not introduce ambiguity.

To describe the functionality perspective, we need to attach internal STDs to *METHOD*s and ensure consistency with the *uses* relationship and the behaviour perspective.

FunctionalityPerspective

$$
\begin{array}{l}
\mathit{BehaviourPerspective} \\
\mathit{internalbehaviour} : \mathit{METHOD} \pfun \mathit{STD} \\
\hline
\operatorname{dom} \mathit{internalbehaviour} \subseteq \\
\quad \{M : \mathit{METHOD} \mid \exists\, c : \mathit{classes};\ m : \mathit{MethodName} \mid m \in c.\mathit{methods} \bullet \\
\qquad M = \mathit{methodbinding}\ (c, m)\} \\
\operatorname{ran} \mathit{internalbehaviour} \subseteq \mathit{InternSTD} \\
\forall\, c : \mathit{classes} \bullet \forall\, m : c.\mathit{methods};\ M : \mathit{METHOD};\ \mathit{std} : \mathit{STD} \\
\quad \mid (c, m) \mapsto M \in \mathit{methodbinding} \wedge (M, \mathit{std}) \in \mathit{internalbehaviour} \bullet \\
\qquad (\forall\, i : \mathit{std}.\mathit{initial};\ s : \mathit{std}.\mathit{states};\ l : \mathit{std}.\mathit{labels} \mid ((i, l) \mapsto s) \in \mathit{std}.\mathit{transrel} \bullet l = \mathit{act}\ m) \\
\forall\, c, d : \mathit{classes} \bullet \\
\quad \mathit{useslabel}(c, d) = \\
\quad \{n : \mathit{MethodName} \mid \exists\, m : c.\mathit{methods};\ \mathit{std} : \mathit{STD};\ M : \mathit{METHOD} \bullet \\
\qquad (c, m) \mapsto M \in \mathit{methodbinding} \wedge (M, \mathit{std}) \in \mathit{internalbehaviour} \wedge \\
\qquad \mathit{call}\ (d.\mathit{name}, n) \in \mathit{std}.\mathit{labels}\} \\
\forall\, c : \mathit{classes};\ f : \mathit{FeatureName};\ m : \mathit{METHOD};\ s : \mathit{STD} \mid (c, f) \mapsto m \in \mathit{featurebinding} \wedge (m, s) \in \mathit{internalbehaviour} \bullet \\
\quad \exists\, e : \mathit{ExternSTD} \bullet (c, e) \in \mathit{externalbehaviour} \wedge \mathit{ml}\ f \in e.\mathit{labels}
\end{array}
$$

- The transition(s) in the internal STD of a method *m* starting at an initial node are labelled with "*act m*", indicating activation of the method invocation. Usually, there will only be one initial node, but we have not ruled out multiple initial nodes.
- Invocations of other methods are indicated by "*call* classname.methodname". 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 will see how invocations of methods of particular objects are done. At that point, we will also see how the visibility restrictions dealing with objects (calls to private or protected members of other objects are disallowed, as described earlier) are implemented. For now, we restrict ourselves to indicating only the class of objects whose methods are invoked.

6.4 Discussion

We have described the behaviour and functionality perspectives of SOCCA on the type level. In SOCCA these perspectives are closely related in that they are both described using one concept: the State Transition Diagram. In the next section, we will formalise the communication perspective of SOCCA at the type level. As we will show, the communication perspective builds on the behaviour and functionality perspectives in two ways: the formalism in which communication is expressed in SOCCA is based on extensions of the notion of STD, and the communication structures will be closely coupled to the external and internal STDs that make up the behaviour and functionality perspectives.

7 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.

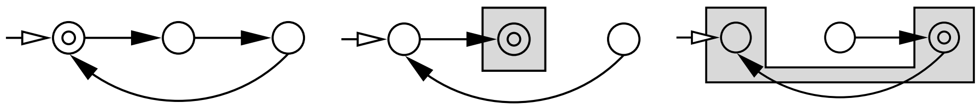
7.1 Intuitive description

The communication perspective is where SOCCA differs the most from other object oriented modelling languages. If presented in a purely factual or formal way, it can be quite daunting. Therefore we will 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 may be switched from one piece of local behaviour to another through coordination by communication.

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 7.1, 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. The *state interpreter*, which maps its states to prescribed subprocesses, and the *action interpreter*, which labels its transitions with traps that its employee(s) must have reached for the transition to be allowed.

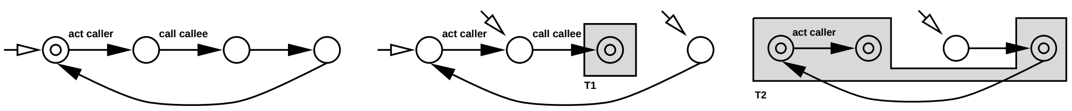
In SOCCA, the external STDs form the basis for the managers, and the internal STDs for the employees. This imposes more structure than in PARADIGM, where the choice of managers and employees was up to the modeller.

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 manager can be formalised more symmetrically; we do not 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 the subprocesses of its employees (not making a transition labelled with a trap that has not been reached yet).

By itself, PARADIGM lacks the structure provided object orientation of SOCCA and thus allows the modeller very large degrees of freedom in modelling. In SOCCA this freedom has been restricted through the object oriented structure, making it more manageable. In SOCCA, the modeller 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.

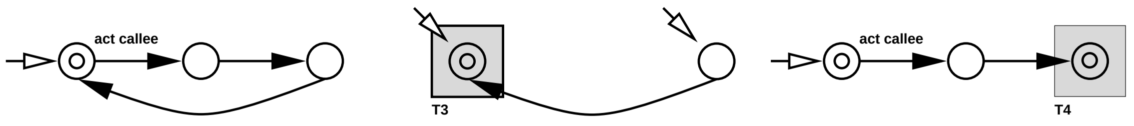


(a) STD (b) First subprocess R_1 (c) Second subprocess R_2

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, **R1** (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, **R2** (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.

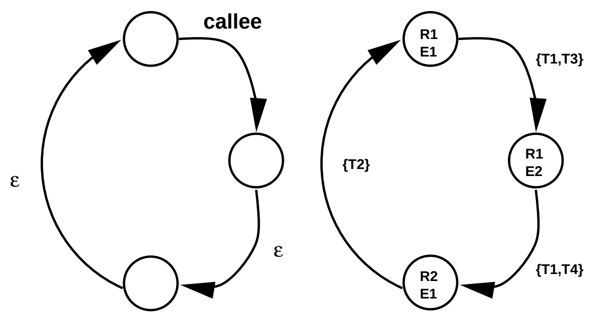


(a) STD (b) First subprocess E_1 (c) Second subprocess E_2

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, **E1** with trap **T3** (in figure 13(b)) in which *Callee* waits to perform its activities; the second, **E2** with trap **T4** (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.



(a) External STD of *A* (b) Corresponding manager

Figure 14: External STD and corresponding manager

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

7.2 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 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.

$$isSubProcessOf : STD \leftrightarrow STD$$

$$\begin{aligned}&\forall std, subp : STD \bullet subp \ \underline{isSubProcessOf} \ std \Leftrightarrow \\ &\quad subp.states \subseteq std.states \land \\ &\quad subp.labels \subseteq std.labels \land \\ &\quad subp.transrel \subseteq std.transrel \cap ((subp.states \times subp.labels) \times subp.states)\end{aligned}$$

- 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*.

7.3 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 modeller 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 its current subprocess.

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

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

- 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.

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

7.4 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 modeller 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.

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

- 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 subprocesses.
- The connection constraint is local only; we do not impose a reachability constraint between subprocesses in a partition and trap structure in general.

7.5 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)*.

```
employee
std : STD
pts : ℙ(STD × (ℙ(ℙ STATE)))
──────────
pts  isPartitionAndTrapStructureOf  std
```

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].

7.6 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 labelled 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 the manager in figure 14. Its state interpreter, which labels each state of the manager STD with the subprocesses the manager in that state prescribes to its employees, is given in figure 15 Similarly, its transition interpreter, which 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, is given in figure 16.

State	Subprocess of Caller	Subprocess of Callee
top	$R1$	$E1$
right	$R1$	$E2$
bottom	$R2$	$E1$

Figure 15: The state interpreter

Transition	Trap(s) of Caller	Trap(s) of Callee
top → right	$T1$	$T3$
right → bottom	$T1$	$T4$
bottom → top	$T2$	*none*

Figure 16: The transition interpreter

□

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.

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

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

- 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.

```
  $HasEmployees : manager \fun \mathbb{P}\, employee$
  ---
  $\forall m : manager \bullet$
      $HasEmployees\ m =$
          $\{i : employee \mid \exists si : stateint;\ ti : transint \bullet$
              $(i, si, ti) \in \mathrm{ran}\, m.empsti\}$
```

An another for the reverse.

```
  $HasManagers : employee \fun \mathbb{P}\, manager$
  ---
  $\forall e : employee;\ m : manager \bullet$
      $m \in HasManagers\ e \Leftrightarrow e \in HasEmployees\ m$
```

7.7 The full communication perspective

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.

$$
\begin{array}{l}
\textit{CommunicationPerspective} \\
\hline
\textit{DataPerspective} \\
\textit{BehaviourPerspective} \\
\textit{FunctionalityPerspective} \\
\textit{managers} : \mathbb{P}\,\textit{manager} \\
\textit{employees} : \mathbb{P}\,\textit{employee} \\
\textit{externalstds} : \mathbb{P}\,\textit{STD} \\
\textit{internalstds} : \mathbb{P}\,\textit{STD} \\
\textit{asmanager} : \textit{STD} \pfun \textit{manager} \\
\textit{asemployee} : \textit{STD} \pfun \textit{employee} \\
\hline
\textit{externalstds} = \operatorname{ran} \textit{externalbehaviour} \\
\textit{internalstds} = \operatorname{ran} \textit{internalbehaviour} \\
\forall m : \textit{managers} \bullet \exists\, \textit{estd} : \textit{externalstds} \bullet m.\textit{std} = \textit{estd} \\
\forall m : \textit{managers} \bullet \textit{HasEmployees}\ m \neq \varnothing \\
\forall e : \textit{employees} \bullet \textit{HasManagers}\ e \neq \varnothing \\
\forall e : \textit{employees} \bullet \exists\, \textit{istd} : \textit{internalstds} \bullet e.\textit{std} = \textit{istd} \\
\operatorname{dom} \textit{asmanager} = \textit{externalstds} \wedge \operatorname{ran} \textit{asmanager} = \textit{managers} \\
\forall s : \textit{externalstds} \bullet (\textit{asmanager}\ s).\textit{std} = s \\
\operatorname{dom} \textit{asemployee} = \textit{internalstds} \wedge \operatorname{ran} \textit{asemployee} = \textit{employees} \\
\forall s : \textit{internalstds} \bullet (\textit{asemployee}\ s).\textit{std} = s \\
\forall c : \textit{classes} \bullet \\
\quad \{e : \textit{employee} \mid \exists\, \textit{extstd} : \textit{STD} \bullet ((c, \textit{extstd}) \in \textit{externalbehaviour}) \wedge (e \in \textit{HasEmployees}\ (\textit{asmanager}\ \textit{extstd}))\} = \\
\quad \{e : \textit{employees} \mid \exists\, \textit{mn} : \textit{MethodName};\ M : \textit{METHOD} \bullet \\
\qquad ((\textit{mn} \in c.\textit{methods} \wedge \textit{methodbinding}(c, \textit{mn}) = M) \vee \\
\qquad (\exists\, d : \textit{classes} \bullet \textit{mn} \in \textit{useslabel}\ (d, c) \wedge \textit{methodbinding}\ (d, \textit{mn}) = M)) \wedge \\
\qquad\quad \textit{asemployee}\ (\textit{internalbehaviour}\ M) = e\}
\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.

7.8 Discussion

We have illustrated SOCCA's communication perspective by giving an intuitive description and an example. Then we rephrased the PARADIGM concepts which lie at the core of SOCCA's description of communication in terms of STDs. Lastly, we have shown how the flexibility and power of the communication concepts from PARADIGM is made manageable by tightly coupling the PARADIGM structures of manager and employee to the concepts of internal and external STDs (from the behaviour and functionality perspectives) that were themselves structured through the principles of object orientation in the data perspective.

We have identified the manager STDs with the external STDs, as well as the employee STDs with the internal STDs. This is a simplification of the reality of modelling. In the reality of modelling, 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.

8 Lessons learned

Working on the formalisation has made us focus on aspects of SOCCA that we were not as sharply aware of until now.

- The concept of binding which captures the meaning of polymorphism by inheritance.
- The similarities between methods and attributes, which we unified through the concept of feature.

- The possibility of explaining the concepts of SOCCA with as few forward references as possible (Z's "no forward references" nature forced us to write the formal text without forward references; the order this imposed allowed us to structure the informal text (natural language description of SOCCA's concepts) so as to contain but a few forward references.

Also, it forced us to make some decisions about the SOCCA core language.

- We decided to make the core language have multiple external STDs per class.
- We have chosen a visibility mechanism (admitedly a crude one, but one which practical experience has shown to be quite powerful).

9 Future work

In this document we have focussed on the more syntactical aspects of SOCCA models, describing the structure of SOCCA models on the class level. Currently, this work is being extended to formalise the instance level of SOCCA models, including the concepts of object, link and State Transition Machine. Such a description of the instance level of SOCCA models will hopefully provide the basis for extending SOCCA to encompass a prototypical instance level between the type and instance levels which will give modellers more expressive power.

10 Discussion

10.1 Related work on SOCCA

The structure of the specification here can be viewed as implicitly defining a meta class diagram of SOCCA. This implicitly defined meta class diagram is quite similar to the one developed in [Sch97]. Some noteworthy differences are

- No self-referential approach. In [Sch97] a SOCCA class diagram is used to chart the relationships between the various SOCCA concepts. In this document, SOCCA is described through Z, which is a quite different, more mathematically oriented, specification language.
- In our Z approach, the concepts of attributes and methods are unified through the feature concept.

The most important difference is that Z's no forward references nature has forced us to focus on a no forward references exposition of the concepts on SOCCA's class/type level. This has provided us with a natural order in which SOCCA's concepts can be introduced.

10.2 Related work on other OO formalisms

Work has been done on the formalisation of other graphical object oriented formalisms, like OMT and UML.

Self-referential approach The UML's authors have chosen to describe UML's semantics largely by means of UML itself through using a metaclass diagram, augmented with a fairly low-level logical notation, the Object Constraint Language.

We believe that describing the semantics of an OO formalism by means of a different, not object oriented, language is a more fruitful approach, as it forces one to step outside the framework of concepts employed in the OO formalism and translate those concepts themselves.

Translational approach [SF97] reports about a formalisation of UML using Z. There is a key difference between their approach and ours.

[SF97] shows how a particular given UML model can be translated to Z and argues that the approach used can be extended to an algorithm to translate UML models to Z in general (assuming syntactic validity). This has been termed a "translational approach" ([EA98]).

Our work does not focus on translating individual models to Z. Rather, we show how SOCCA concepts, and from there SOCCA models, are translated to Z. We do not assume that models have been determined to be syntactically valid by an external algorithm, but give an abstract syntax of SOCCA in Z by means of which syntactic correctness can be determined.

References

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

[BR98] J-M. Bruel and R.B.France. Transforming UML models to formal specifications. In *UML'98 - Beyond the notation*, LNCS. Springer, 1998.

[Bru98] H.G. Brugman. Software process modeling in SOCCA. Master's thesis, Department of Computer Science, Leiden University, May 1998, http://www.wi.leidenuniv.nl/MScThesis/IR98-04.tar.gz.

[Dil94] A. Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, 2nd edition, 1994, ISBN 0-471-93973-0.

[DKW98] Jean-Claude Derniame, Badara Ali Kaba, and David Wastell, editors. *Software Process: Principles, Methodology and Technology*, number 1500 in Springer Lecture Notes in Computer Science. Springer Verlag, Berlin, Heidelberg, New York., 1998.

[DP90] D.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, 1990, ISBN 0 521 36766 2.

[EA98] A. S. Evans and A.N.Clark. Foundations of the unified modeling language. In *2nd Northern Formal Methods Workshop, Ilkley*, electronic Workshops in Computing. Springer-Verlag, 1998.

[EE94] J. Ebert and G. Engels. Observable or Invocable Behaviour - You Have to Choose! Technical Report 94-38, Department of Computer Science, Leiden University, December 1994, ftp://ftp.wi.LeidenUniv.nl/pub/CS/TechnicalReports/1994/tr94-38.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. 1994, Research Studies Press Ltd. / John Wiley & Sons Inc., year. Taunton 1994. Also available as Technical report 94-10, Department of Computer Science, Leiden University.

[EN94] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, 2nd edition, 1994.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, Inc., Englewood Cliffs, 1 edition, 1991, ISBN 0-13-818204-3.

[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].

[Hoe99] Pieter Jan 't Hoen. Distributed model behaviour and development for (extended) SOCCA, 1999. Work in progress.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Mass., USA, 1979, ISBN 0-201-02988-X.

[JBAG97] Stefan Joos, Stefan Berner, Martin Arnold, and Martin Glinz. Hierarchische zerlegung in objektorientierten spezifikationsmethoden. *Softwaretechnik-Trends*, 17(1):29–37, February 1997, `http://www.ifi.unizh.ch/groups/req/ftp/papers/oo_hierarchien.ps`.

[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.

[MS97] Irwin Meisels and Mark Saaltink. The Z/EVES Reference Manual (for Version 1.5). Technical Report TR-97-5493-03d, ORA Canada, September 1997, `ftp://ftp.ora.on.ca/pub/doc/97-5493-03d.ps.Z`.

[NAS95] NASA Office of Safety and Mission Assurance, Washington, DC. *NASA Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion*, 1995.

[RBP$^+$91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., 1991, ISBN 0-13-620941-9.

[Rum96] James Rumbaugh. *OMT Insights: perspectives on modelling from the Journal of Object-Oriented Programming*. Prentice-Hall, Inc., 1996, ISBN 0-13-846965-2.

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

[Saa97] Mark Saaltink. The Z/EVES Mathematical Toolkit Version 2.2 for Z/EVES Verision 1.5 . Technical Report TR-97-5493-05a, ORA Canada, September 1997, `ftp://ftp.ora.on.ca/pub/doc/97-5493-05a.ps.Z`.

[Sch97] E.J. Schuitema. Towards a SOCCA environment: a meta class diagram for SOCCA and use cases for the environment. Master's thesis, Department of Computer Science, Leiden University, 1997, `ftp://ftp.wi.LeidenUniv.nl/pub/CS/MScTheses/schuitema.97.ps.gz`. Internal Report IR–97-02.

[SF97] Malcolm Shroff and Robert B. France. Towards a formalization of UML class structures in Z. Technical Report TR-CSE-97-4, Department of Computer Science & Engineering, Florida Atlantic University, Boca Raton, FL33431, USA, 1997, `http://www.cse.fau.edu/~robert/publication/compsac97.ps.gz`. Also appears in COMPSAC'97.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992, ISBN 013-978529-9.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997, ISBN 0-201-88954-4.

[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, http://www.comlab.ox.ac.uk/archive/z.html.