Class-like Descriptions of Packages

The Behaviour of Packages

P.J. 't Hoen [*] L.P.J. Groenewegen [†] J.H.M. Dassen [‡]

P.W.M. Koopman [§] G. Engels [¶]

I.G. Sprinkhuizen-Kuyper [||]

February 9, 2000

Abstract

Current object-oriented modelling languages provide concepts like packages (or modules) to structure large-scale object-oriented models. These concepts are purely syntactical. They are used to define model-specific name scopes and thus to hide encapsulated features inside a package. But, in order to understand the semantics of what is contained in a package, one still has to zoom into a package and to study its detailed fine-grained structure.

We present a new approach termed class-like description (CLD) of packages. These CLDs allow to view a package as an ordinary class, and to abstract from the detailed (hierarchical) structure of a package. Thus, packages can be treated as ordinary classes on this higher level of abstraction.

We illustrate our approach by extending the object-oriented modelling language SOCCA with CLDs. SOCCA offers UML-like class diagrams and state transition diagrams to model static and dynamic aspects of a system. In addition, enhanced state transition diagram mechanisms are provided to model sophisticated inter-object communications. The CLDs in SOCCA reflect these aspects too.

[*] hoen@liacs.nl
[†] luuk@liacs.nl
[‡] jhm@cistron.nl
[§] pieter@cs.kun.nl
[¶] engels@uni-paderborn.de
[||] kuyper@cs.unimaas.nl

Contents

1 Introduction **2**

2 Formalisation **7**

2.1 Data Perspective 8
2.1.1 Defining the Class 8
2.1.2 Replacing a Package by a Class 10
2.1.3 Isolating the Replaced Class 13
2.1.4 Relationships 13
2.1.5 Labels of the Uses Relationships 16
2.1.6 Visibility 18
2.1.7 Export and Import 20
2.2 Behaviour Perspective 22
2.3 Functionality Perspective 23
2.4 Communication Perspective 26

3 Conclusion **29**

4 Future Work **30**

1 Introduction

The modelling and design of large object-oriented (OO) systems is still a difficult task. Much effort has been put into developing techniques to facilitate the design of such large systems. The behaviour of objects is a continuing topic of research within this effort [ABKR89, Jon93, Dem97, BKS98, Sil99].

It is however essential for the design of large, complex models that the behaviour of a collection of objects of the model can be seen as a black box. This black box captures the behaviour of the individual objects while hiding them. This is essential if the modeller is to have a well defined abstract view of the behaviour of the model. Most current OO languages however require the modeller to examine the behaviour of the individual objects to understand the behaviour of the whole.

This is problematical as the behaviour of a collection of objects is not well defined at an abstract level. The modeller has to compose the behaviour of the collection in an ad-hoc fashion from the behaviour of the individual objects. Thus, for n modellers working on a project, there will be n interpretations of the combined behaviour of a collection of objects.

Additionally, each modeller has to study the behaviour of a collection of objects at the detailed level of the individual objects. This is fine for a modeller who is working on the innards of such a collection. This is however not desirable for the designer of the architecture. An architect is more interested in an abstract overview of the behaviour of the model. Furthermore, most modellers will only work on a small part of the model. Like the architects, they are likewise not interested in figuring out the detailed behaviour of the rest of the model. They are more interested in an overview of the behaviour of the rest of the model.

It is thus a problem if the abstract behaviour of a collection of objects is not defined. There is thus a need for a black box model part with well defined behaviour where the behaviour of the black box does not have to be constructed by the modeller from the behaviour of the objects which make up the part.

The need for such a black box has led to the development of components. This ongoing development focuses on finding appropriate modelling mechanisms of a coarser-grain than single objects. Examples of this research are the Component Object Model [COM99], CORBA [Obj91, Gro96], design components also known as *patterns* [GHJV94], *frameworks* [HP94, Sil99], real-time constraints for components of objects [NA99] and dynamic components [WMB99]. The behaviour of these components is however roughly defined.

In this paper, we define a component at the type level. We define a component as a coarser grained modelling part than single classes. We do not, as yet, define how a set of objects is defined as a component. We instead define the set of classes from which these objects are instantiated as a black box. The behaviour of one class is a template for the behaviour of all the objects instantiated from this one class. The behaviour of a set of classes is thus a template for the behaviour of all the objects instantiated from these classes. A set of classes as a black box is thus an abstract template of the behaviour of the instantiated

objects. This black box representation is thus also a template for the behaviour of the instantiated objects as a component of objects.

In Figure 1 we give example classes which make up a bank and which communicate in various ways as shown by the arrows between the classes. In this paper, we define a representation of a collection of classes as a black box. The objects to which the classes are instantiated, when seen as a component, are an instance of the black box representation of the set of classes from which the objects are instantiated. Future work (see Section 4) defines the instances of such a black box representation of a set of classes. This future work thus defines a component of objects while this text defines a component of classes.

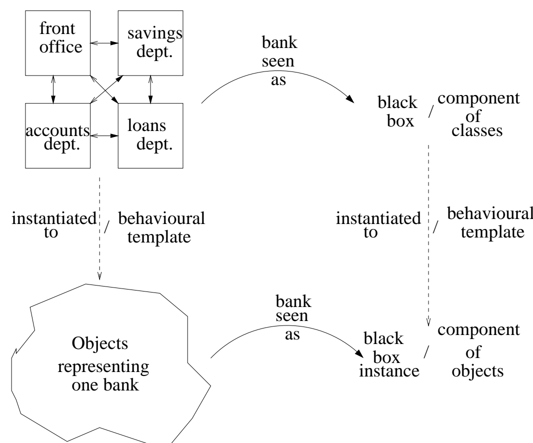


Figure 1: Instances of a black box

Figure 1 is an implementation of a general Model-Instance Viewpoint architecture proposed in [EK99]. Parts of the model can be observed at type (class) and instance level (objects) and the black box representations gives behavioural viewpoints on the model.

[RK99] observes that class diagrams, showing the classes of a model, can be compressed by (temporarily) effacing less essential classes. Non-essential classes are hidden and the modeller can concentrate on the remaining classes. This, however, immediately raises the question of which classes are essential and to whom?

To answer the above questions, most OO modelling/ programming languages use some form of (hierarchical) packages to manage the complexity of the model by structuring and encapsulating classes [RBP+91, GJM91, Mey97, UML99]. Encapsulation of classes by packages hides classes intended only for use within the package from the rest of the model. The exported classes of a package are deemed essential with respect to the rest of the model while the "non-essential" classes which only play a role within the package are hidden. These packages are however syntactical structures and have no behaviour of their own. A modeller has to consider all individual classes which are contained by a package to study

the collective behaviour of the classes of the package. As such, packages are not the sought for black-box with respect to the behaviour.

[MWB99] however observes that in [UML99], a "subsystem" can be seen as an object. The term subsystem is a keyword which refers to a package that represents an independent part of the system being modelled. A subsystem contains a collection of classes which are, conceptually, instantiated to one, complex object. This object is then a black box representation of the package. This approach is promising, but the definition of how the behaviour of the classes in the subsystem is mapped to the behaviour of one object, and visa versa, is not given. [MWB99] however points out the need of modellers to be able to have two views of a subsystem; it can either be seen as complex structure or as a black box.

In this paper, the behaviour of a package is defined in terms of the behaviour of a single class. The techniques for defining the behaviour of a single class are reused to model the behaviour of the set of classes contained in a package. We define the mapping of a package to a single class. This class captures the behaviour of the original classes in the package at the level of detail offered by the OO modelling language in which we apply this technique.

With the above approach, a (hierarchical) package is defined as a black-box where the individual classes contributing to the behaviour of the package are hidden. Each modeller working on the model has the same interpretation of the definition of the behaviour of a package of the model. Secondly, the behaviour of a package/part of the model no longer has to be deduced from the individual classes which make up the package. The behaviour of the package is expressed by *one* class and the modeller can abstract from the internal structure of the package when studying the behaviour of the package.

We introduce a layer above the model with packages where we can observe the behaviour of the packages of the model in terms of the classes which capture the behaviour of the package. We call this class, which defines the behaviour of a package, a class-like description (CLD) of the package. The CLD represents the original package and brings the concepts of packages and classes closer together. All packages of a model can be replaced by their respective CLDs as sketched in Figure 2. We give a model where the behaviour of its packages P_1 to Pn are captured by their corresponding CLDs.

Any package of the model can thus be seen either as a complex structure or as a single class, its CLD. A modeller who is developing a package will want/need to observe all details while other developers working on other parts of the model only observes the non-encapsulated classes of the package. If even less detail is desired, the CLD of the package is used to represent the package as a black box. The concept of a class is reused to represent packages of classes. The above gives strength to the quote of Meyer that "classes are the only modules" [Mey97].

A hierarchical package can even be replaced by its CLD in intermediate steps for different levels of abstraction. We once more return to the example of a bank of Figure 1. In Figure 3, we show three different ways of looking at a bank. We can see the bank as a complex package where each department of the bank, formerly represented by a single class, is represented by a horrendously complex

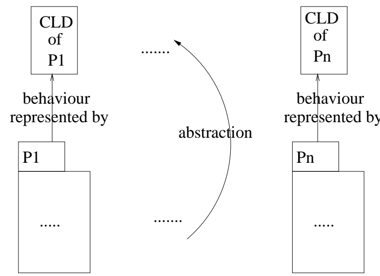


Figure 2: The Levels of Behaviour in a Model

package. We can also observe the entire bank as one class when we use the CLD of the entire bank. Or, we can take the middle road by observing the bank as a moderately complex package where each of its departments is represented by its CLD.

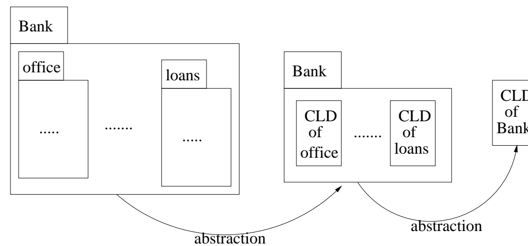


Figure 3: The scalable use of CLDs of packages

The above gives strength to [Mey99] where "All the evidence is there that successful component technology must build on object orientation". We are not sure whether OO is the only viable route to build components. But, packages of classes represented by their CLDs provide a means to scale the concepts developed in OO for a single class to a set of classes. Concepts and modelling techniques can be lifted to apply to a package of classes.

The CLD of a package is a viable concept for any object-oriented modelling language. We however restrain ourselves and only illustrate and formalise the CLD of hierarchical packages in SOCCA (Specification of Coordinated and Cooperative Activities). SOCCA is a graphical formalism and associated method for object-oriented modelling which is under active development at Leiden University. The strength of SOCCA is that it allows the precise and detailed specification of the communication and synchronisation between the modelled classes. The modelling of the behaviour of a class in SOCCA is very sophisticated. As a consequence, the CLD of a SOCCA package captures the behaviour of the package at a high level of precision.

In Section 2, the CLD of a (hierarchical) SOCCA package is formalised. In-

terwoven with this section we discuss how a CLD of a package is a black-box representation of the package. Furthermore, we discuss how the encapsulation of a package is reflected in the CLD of the package. Section 3 gives conclusions. Section 4 lists some future work.

2 Formalisation

The formalisation of the CLD of a package is based on the formalisation of SOCCA, UML-like packages as defined in [tHDG+99]. In this text we have formalised how the classes of SOCCA models are structured and encapsulated by packages. The architecture of a SOCCA model is not a set of classes but a collection of hierarchical packages. A large model can thus be split up into smaller, more comprehensible chunks.

We assume in this text that the reader is familiar with [tHDG+99]. A reader unfamiliar with SOCCA can read [tHZG99] as an introduction. The formalisation, like the previous work, is done in Z [Spi92]. All schemas, axioms, etc... are type checked using Z/EVES [Saa95], a theorem prover for Z [1].

We define the CLD of a package as a Δ operation on a model where a package of the model is replaced by a single class. This single class is as yet defined as part of a model as the behaviour of a class in SOCCA is defined in the context of a concrete model. We thus take one model M and one package p of the model and transform the model to replace the package and all classes it contains by one single class as depicted in Figure 4. A model is depicted by the *TopPackage* which contains all classes and packages of the model. The transformed model M' now contains the CLD of p. We use the Z convention to adorn modified variables with a single ′.

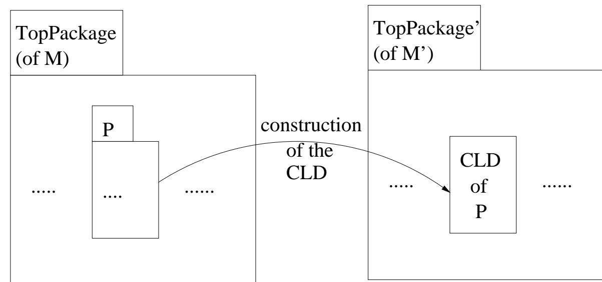


Figure 4: How the CLD of a package is defined

The total of the four perspectives of SOCCA that are formalised in [tHDG+99] and used in this text are:

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

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

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

[1] Further information on the theorem prover can be found at `http://www.ora.on.ca/z-eves/welcome.html`

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

The schema *CommunicationPerspective* in [tHDG+99] defines a SOCCA model with respect to all four perspectives. We introduce the schema *Model* as an synonym for this schema.

Model
CommunicationPerspective

We build the definition of the CLD of packages perspective by perspective. This is done to show how the CLD of a package is defined for the various perspectives. Additionally, this keeps the size of the required schemas in the formalisation manageable. The schemas *DeltaData*, *DeltaBehaviour*, *DeltaFunctionality* and *DeltaCommunication* of Sections 2.1 to 2.4 define the CLD of a package step by step as a Δ operation for each perspective. Each of these schemas is likewise build in small, clear steps.

The next section defines the CLD of a package with respect to the data perspective.

2.1 Data Perspective

The data perspective of SOCCA in [tHDG+99] is basically defined by three aspects: the containment structure of the model parts, the visibility of model parts and the import and export of model parts. We build the definition of the CLD for the data perspective in separate steps for these three aspects and in the order in which they are listed as once we define the changes in structure, we can define the changes in visibility. We can define the changes in import and export only after these first two steps.

We begin the formalisation by defining the changes in the structure of the packages as a result of the introduction of the CLD. We replace one package p of a model by its CLD. We need to restructure the owner of the replaced package as one of its package elements, namely p, is replaced by a class element; the CLD of p. The first step is to define this class which replaces the package with respect to the data perspective.

2.1.1 Defining the Class

A class in SOCCA as defined in [tHDG+99] with respect to the data perspective is defined by an identity, a set of method identifiers and the visibility of the methods. We first define a CLD of a package for the first two of these aspects. We postpone the visibility of the methods of the CLD until we have redefined the containment structure of the model with the CLD. Only then can we define the modified visibility of the methods. We first define the identity of the CLD of a package.

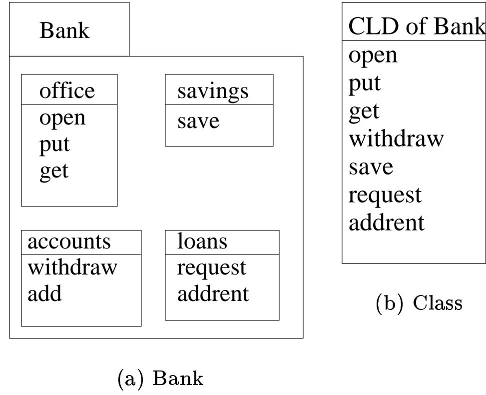
We define the function $CLDidentity : Package \rightarrow ClassIdentity$ where $CLDidentity\ p$ returns a fresh identity for a CLD of a subpackage of p.

$$
\begin{array}{|l}
CLDidentity : Package \rightarrow ClassIdentity \\
\hline
\forall p : Package \bullet \\
\quad CLDidentity\ p \notin \\
\quad \{ i : ClassIdentity \mid (\exists c : ContainedClasses\ p \bullet i = c.identity) \}
\end{array}
$$

We ensure that the identity of the CLD of package of p is not equal to the identity of any contained class of p. The CLD of the subpackage of p is identified by the identity returned by *CLDidentity*.

The CLD of a package is a class which has as methods the methods of the classes of the original package. The CLD of a package has the same functionality as the individual classes of the package.

Example In Figure 5(a) we give as example the methods of the classes of a package *Bank*. The CLD of this package has as methods the collected methods of all these classes. These are the methods of the class in Figure 5(b). □



(a) Bank

(b) Class

Figure 5: The methods of the CLD

We introduce the function *collectmethods* to collect the method identifiers of the classes of one given package.

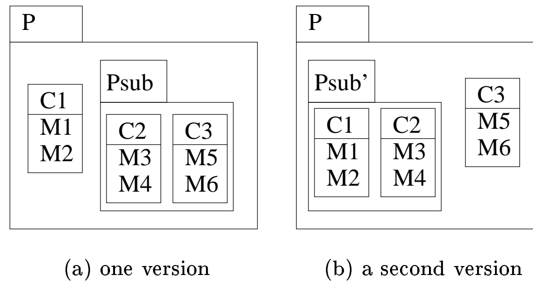
$$
\begin{array}{|l}
collectmethods : Package \rightarrow \mathbb{F}\ MethodIdentity \\
\hline
\forall p : Package \bullet \\
\quad collectmethods\ p = \\
\quad \bigcup \{ m : \mathbb{F}\ MethodIdentity \mid (\exists c : ContainedClasses\ p \bullet m = c.methods) \}
\end{array}
$$

- The methods of the classes of a package p, are the methods of the classes contained by p. These are thus not only the methods of the class elements of p, but also the methods of the classes contained by the subpackages of p. The function *collectmethods* collects *all* the methods of the contained classes.

- We do not have to rename the methods of the CLD of a package when they are collected as methods of one class. We have ensured in [tHDG+99] that each method of the class has a unique identity with respect to the rest of the methods of the model.

The CLD of a package abstracts from the internal structure of a package. The internal structure of a package can be changed without affecting the CLD of the package. If we ignore visibility constraints, we can state that we can change the internal structure of a package as long as we do not change the classes contained by the package.

Example In Figures 6(a) and 6(b) we give two packages P which both have the same CLD. In both cases, the CLDis a class with the methods $M1$ to M6. In the first Figure, the classes $C2$ and $C3$ are contained by the sub package *Psub* of P. In the second Figure, classes $C2$ are instead grouped in a different subpackage $Psub'$. This however does not affect the CLD of P as only the methods of the contained classes matter. □



(a) one version (b) a second version

Figure 6: Two packages with the same CLD

In principle we could even shuffle the methods between the classes in Figures 6(a) and 6(b). For example, we could switch the methods of class $C1$ and $C2$. This would not make a difference for the methods of the CLD of P as these methods are defined by the collection of methods of all the classes contained by the package. It does not matter which class of the package provides the method.

The CLD of a package p is thus partially defined as a class with as identity *CLDidentity* p and as methods the methods of the classes of p,
i.e. *collectmethods* p. The next section defines how we replace a subpackage of a package by its partial CLD.

2.1.2 Replacing a Package by a Class

We now define how we replace one subpackage of a package by its partial CLD. This CLD is partial as we, as yet, ignore the visibility of the methods of the class. The relation $replace : (Package \times Package) \leftrightarrow Package$ holds for the

packages $((in, rep), out)$ if subpackage *rep* of *in* is *rep*laced by its partial CLD in the package *out*.

We however as yet ignore how the relationships of a package are changed by replacing a subpackage by its CLD. The changes in the relationships are complex and are handled separately. We define the relation partial-equality of type $Package \leftrightarrow Package$. We have p_0 $\underline{\text{partial-equality}}$ p_1 if packages p_0 and p_1 are equivalent except for the relationships with respect to their contained classes. We use this relation to ignore the changes of the relationships in the definition of *replace* at this point in time.

Example The example packages of Figures 7(a) and 7(b) are equivalent with respect to the relation partial-equality as only the defined relationships differ. □

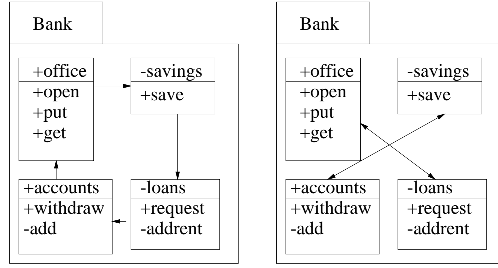


Figure 7: Equivalence with respect to partial-equality

We now define the relation partial-equality:

$$\text{partial-equality} : Package \leftrightarrow Package$$

$$\begin{array}{l}
\forall\, p_0, p_1 : Package \bullet \\
\quad p_0 \ \underline{\text{partial-equality}}\ p_1 \Leftrightarrow \\
\quad ProjIdentity\ p_0 = ProjIdentity\ p_1 \wedge \\
\quad ClassElements\ p_0 = ClassElements\ p_1 \wedge \\
\quad \#\ (PackageElements\ p_0) = \#\ (PackageElements\ p_1) \wedge \\
\quad (\forall\, psub_0 : PackageElements\ p_0 \bullet \\
\qquad (\exists_1\, psub_1 : PackageElements\ p_1 \bullet \\
\qquad\quad psub_0\ \underline{\text{partial-equality}}\ psub_1))
\end{array}$$

We now define the relation *replace* which defines how we replace one subpackage *rep* of package *in* by its partial CLD to build package *out*, i.e. $((in, rep), out)$ is in the relation *replace*. We define the relation *replace* in two small steps. We first define how *rep* is replaced by its CLD is a package element of *in*, i.e. *rep* is directly contained by *in*.

$$
\begin{array}{|l}
replace : (Package \times Package) \leftrightarrow Package \\
\hline
\forall\, in, rep, out : Package \mid \\
\quad ((in, rep), out) \in replace \bullet \\
\quad rep \in ContainedPackages\ in \wedge \\
\quad ProjIdentity\ out = ProjIdentity\ in \wedge \\
\quad rep \in PackageElements\ in \Leftrightarrow \\
\quad ((\exists\, cld : Class \mid \\
\qquad cld.methods = collectmethods\ rep \wedge \\
\qquad cld.identity = CLDidentity\ in \bullet \\
\qquad\quad ClassElements\ out = ClassElements\ in \cup \{cld\}) \wedge \\
\quad \#\ (PackageElements\ out) = (\#\ (PackageElements\ in) - 1) \wedge \\
\quad (\forall\, outsub : PackageElements\ out \bullet \\
\qquad\quad (\exists_1\, insub : PackageElements\ in \mid insub \neq rep \bullet \\
\qquad\quad outsub\ \underline{\text{partial-equality}}\ insub)))
\end{array}
$$

- The element package *rep* of *in* is removed and is replaced by its CLD to produce package *rep*.
- The containment structure for classes and packages of the (sub)packages which are not replaced by a CLD, i.e. the other package elements of *in*, are left unmodified. *rep* is directly replaced by its CLD.
- We ignore constraints on the relationships at this point through use of the relation partial-equality.

In the second case, the package *rep* is not a package element of *in*. As we ensure that *rep* is a contained package of *in*, we know there must be a package element of *in* which contains *rep*. We replace *rep* by its CLD in this package by, once more, using the relation *replace*.

$$
\begin{array}{l}
\forall\, in, rep, out : Package \mid \\
\quad ((in, rep), out) \in replace \bullet \\
\quad rep \in ContainedPackages\ in \wedge \\
\quad rep \notin PackageElements\ in \Leftrightarrow \\
\quad (ClassElements\ out = ClassElements\ in \wedge \\
\quad \#\ (PackageElements\ out) = \#\ (PackageElements\ in) \wedge \\
\quad (\exists_1\, outsub : PackageElements\ out \bullet \exists_1\, insub : PackageElements\ in \bullet \\
\qquad ((insub, rep), outsub) \in replace \wedge \\
\qquad (\forall\, out'' : PackageElements\ out \mid out'' \neq outsub \bullet \\
\qquad\quad (\exists_1\, in'' : PackageElements\ in \mid in'' \neq insub \bullet \\
\qquad\qquad out''\ \underline{\text{partial-equality}}\ in''))))
\end{array}
$$

With the above definitions, we can define, with respect to the data perspective, how we replace one package of a model by its CLD, without yet taking into consideration the rerouting of relationships or the visibility of model parts. The input variable *which*? contains the identity of the package we wish to replace in the model. We introduce the variable p to capture this package. The use of the

variable *which*? is more intuitive for the modeller while use of the variable p is used throughout most of the formalisation.

```
DeltaDataContainment
ΔModel
which? : PackageIdentity
p : Package
―――――――――――――――
p ∈ ContainedPackages TopPackage
ProjIdentity p = which?
((TopPackage, p), TopPackage′) ∈ replace
```

The schema *DeltaDataContaiment* defines how we replace a package p with identity *which* by its partial CLD. Recall that *TopPackage* is the package of a model which contains all the classes, packages and relationships of a model. We use the Z convention of labelling modified variables with an apostrophe.

2.1.3 Isolating the Replaced Class

In the schema *DeltaDataContainment*, one package p of a model is replaced by its CLD. We isolate this class in the variable $cld : Class$ in the modified model, amongst others, in order to be able to define the rerouting of relationships and for the definition of the modified visibility of the model parts.

```
DeltaisolateCLD
DeltaDataContainment
cld : Class
―――――――――――――――
cld ∈ ContainedClasses TopPackage′
cld ∉ ContainedClasses TopPackage
```

The class *cld* is the unique class due to its fresh identity which is contained by the modified *TopPackage*′ and not contained by the original *TopPackage*. The next section defines the changes in the relationships of the model due to the definition of the CLD of a package.

2.1.4 Relationships

In this section we define how the relationships of a model are affected by replacing one of its packages by its CLD. The relationships to and from the classes contained by the package we replace are rerouted to the new CLD.

Example In Figure 8, relationships 1 to 3 to and from the classes of a package P are redirected to the CLD of P, the class P. The relationship from $C1$ to $C2$, a class of P, is rerouted to the CLD of P. The relationship from $C2$ to $C3$ becomes a relationship from the CLD of P to itself. Finally, the relationship from $C3$ to $C4$ becomes a relationship from the CLD of P to $C4$. □

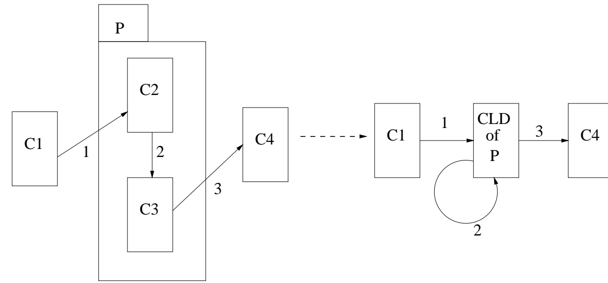


Figure 8: Example of redirecting relationships

This approach does not work with respect to inheritance if we reroute relationships to the CLD of a package in this straightforward manner. In Figure 9, if we replace package p by its CLD, then class $C3$ becomes a specialisation of the CLD. This means that suddenly class $C3$ can inherit methods from $C1$.

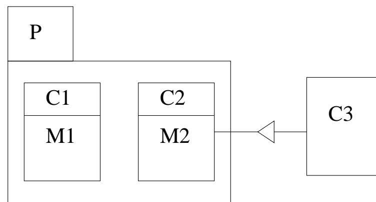


Figure 9: A problem with inheritance

We could solve the problem by defining the notion of selective inheritance. With this notion, we can define that the CLD of the package p only allows for class $C3$ of the above example to inherit the intended methods originating from class $C2$. This approach is however beyond the scope of this work. We, for now, solve the problem in the most simple way by avoiding it. We introduce the function $nonbase : Package \to \mathbb{F}\, Package$ where $nonbase\ p$ returns the (sub)packages $psub$ of p which do not contain any class c which is used as base class for a class d. We define the CLD of a package of a model for those packages which do not have the problem with inheritance, i.e. those packages returned by $nonbase\ TopPackage$.

$$
\begin{array}{l}
nonbase : Package \rightarrow \mathbb{F}\, Package \\
\hline
\forall\, p : Package \bullet \\
\quad nonbase\ p = \\
\quad \{psub : ContainedPackages\ p \mid \\
\qquad (\forall\, c, d : Class \mid \\
\qquad\quad c \in ContainedClasses\ psub \wedge \\
\qquad\quad d \notin ContainedClasses\ psub \bullet \\
\qquad\qquad (c, d) \notin (is\text{-}a\ p))\}
\end{array}
$$

We now ensure that we only work with correct packages, i.e. the package p which we replace is an element of the set *nonbase* which does not suffer from the inheritance problem. Note that inheritance between the classes of p is allowed. A class of p can also be a specialisation of a class not contained by p.

$$
\begin{array}{l}
\textit{DeltaDataCorrect} \\
\hline
DeltaisolateCLD \\
\hline
p \in (nonbase\ \ TopPackage)
\end{array}
$$

It is however not difficult to convert a package p of a model which does not meet the above constraint to a non-base package. It is sufficient to remove all undesired specialisations from the model as given in the operation *Fix*. The existence of a specialisation relationship from class c to d means class d inherits certain methods, relations, etc ... from class c. Removing the inheritance relationship does not change the properties of d itself. It is however no longer documented that if c changes, then d is automatically changed as well. This is however not a problem for the construction of the CLD where we are only interested in the behaviour/abstract representation of a given package.

$$
\begin{array}{l}
\textit{Fix} \\
\hline
DeltaDataContainment \\
\hline
(\forall\, c, d : ContainedClasses\ \ TopPackage' \bullet \\
\quad (c, d) \in (is\text{-}a\ \ TopPackage') \\
\quad \Leftrightarrow \\
\quad ((c, d) \in (is\text{-}a\ \ TopPackage) \wedge \\
\quad \neg\, (c \in ContainedClasses\ p \wedge d \notin ContainedClasses\ \ p)))
\end{array}
$$

We can thus use this operation to "repair" a model if we whish to give the CLD for a package p of which contained classes are specialised to classes not contained by the package.

We can now define the function *changeparticipants* for rerouting the relationships. This function is used to change relationships r which have as participants a class from a given set C. The participants of these relationships are replaced by a given class c; the CLD of a package.

Example The function *changeparticipants* (R, C, c) changes all relationships $r : R$ which use classes in C to use the class c instead. □

The function is used to redirect relationships to classes of a package p by calling *changeparticipants* $(R, \mathit{ContainedClasses}\ p, \mathit{cld})$ where *cld* is the *cld* of p and R are all the relationships of the model.

$$\mathit{changeparticipants} : (\mathbb{F}\,\mathit{Relationship} \times \mathbb{F}\,\mathit{Class} \times \mathit{Class}) \rightarrow \mathbb{F}\,\mathit{Relationship}$$

$$\begin{array}{l}
\forall R : \mathbb{F}\,\mathit{Relationship};\ C : \mathbb{F}\,\mathit{Class};\ c : \mathit{Class} \bullet \\
\quad \mathit{changeparticipants}\ (R, C, c) = \\
\quad \{r' : \mathit{Relationship} \mid \\
\qquad (\exists\, r : R \bullet \\
\qquad\quad r'.\mathit{identity} = r.\mathit{identity} \land \\
\qquad\quad r'.\mathit{type} = r.\mathit{type} \land \\
\qquad\quad \#\ (r'.\mathit{participants}) = \#\ (r.\mathit{participants}) \land \\
\qquad\quad (\forall\, i : 1 \mathinner{..} \#\ (r.\mathit{participants}) \bullet \\
\qquad\qquad (r.\mathit{participants}\ i \notin C \Rightarrow r'.\mathit{participants}\ i = r.\mathit{participants}\ i) \land \\
\qquad\qquad (r.\mathit{participants}\ i \in C \Rightarrow r'.\mathit{participants}\ i = c)) \land \\
\qquad\quad (r'.\mathit{type} = \mathit{IsARelationship} \Rightarrow r'.\mathit{participants} \neq \langle c, c\rangle))\}
\end{array}$$

- The identity, type and the number of participants of individual relationships from R are left unmodified.
- The participants of individual relationships which are in C are replaced by class c.
- Cyclic inheritance, i.e. $r'.\mathit{participants} = \langle c, c\rangle$ and $r'.\mathit{type} = \mathit{IsARelationship}$, which occur as a result of changing the participants of an *is-a* relationship from classes c_1 to c_2 of C to the new class c are omitted. Note this can cause relationships from R to be omitted in the produced set of modified relationships.

Now that we have defined the redirection of relationships to the CLD of a package, we can define the changes of the relationships of the models with the CLD of the package.

DeltaDataRelationships

$$\begin{array}{l}
\mathit{DeltaDataCorrect} \\
\hline
\mathit{ContainedRelationships}\ \mathit{TopPackage}' = \\
\mathit{changeparticipants}\ ((\mathit{ContainedRelationships}\ \mathit{TopPackage}), (\mathit{ContainedClasses}\ p), \mathit{cld})
\end{array}$$

The participants of the relationships formerly classes of p are changed to the CLD of p. The next section defines the changes in labels of the uses relationships.

2.1.5 Labels of the Uses Relationships

We have rerouted the relationships, including the uses relationships, with the schema *DeltaDataRelationships*. We however still need to adjust the labels of

the uses relationships. These labels document which methods are imported from (other) classes. The labels of uses relationships, as documented by the function *useslabel*, may need to be combined as a result of the use of the CLD of a package.

For example, in Figure 10 we replace package p by its CLD. The two uses relationships from class C to contained classes of p become one uses relationship to the CLD of p.

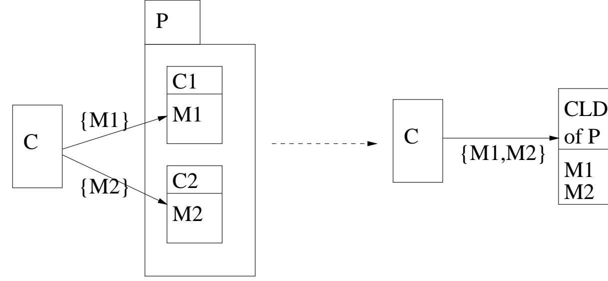


Figure 10: How to Change *useslabel*

When two or more classes of a package of which we construct the CLD have a uses relationship to a class not contained by the package we likewise have to combine the labels of uses relationships. The schema *DeltaDataUsesLabels* combines all these constraints.

DeltaDataUsesLabels

DeltaDataRelationships

$$\begin{aligned}
&\forall c', d' : \mathit{ContainedClasses}\ \mathit{TopPackage}';\ m : \mathit{MethodIdentity} \\
&\quad \mid (c', d') \in \operatorname{dom} \mathit{useslabel}' \bullet \\
&\qquad m \in \mathit{useslabel}'\ (c', d') \Leftrightarrow \\
&\qquad (\exists\, c, d : \mathit{ContainedClasses}\ \mathit{TopPackage} \mid \\
&\qquad\quad (c = c' \lor (c \in \mathit{ContainedClasses}\ p \land c' = \mathit{cld})) \land \\
&\qquad\quad (d = d' \lor (d \in \mathit{ContainedClasses}\ p \land d' = \mathit{cld})) \bullet \\
&\qquad\qquad m \in \mathit{useslabel}\ (c, d))
\end{aligned}$$

A method m is only imported by c' from d' if there is a class c which imports m from d where c and d are classes of the original model which are mapped to the classes c' and d' respectively in the model with the CLD of p.

In SOCCA, a method must be imported by a uses relationship if the method is called. This means the uses relationships between classes documents the static structure of communication between classes in terms of method calls.

The CLD of a package abstracts from the internal static description of the communication in the package. All uses relationships between classes of a package are mapped to uses relationships of the CLD of the package to itself.

Example Figures 11(a) and 11(b) give two packages P with different uses relationships, i.e. communication, between the contained classes. The CLD s

of the packages as given in Figure 11(c) are identical with respect to the data perspective in the sense that in both cases the CLD will import method M of itself. The structure of the internal communication in the packages is hidden and the modified uses relationship abstracts from which classes of the package call each other. □

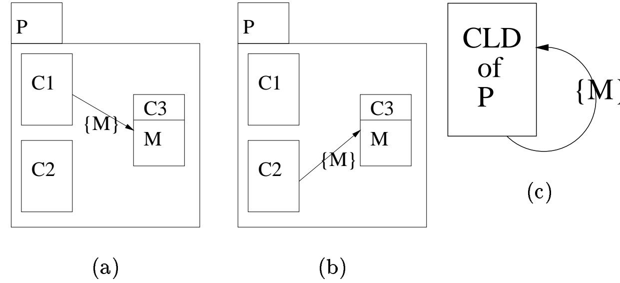


Figure 11: Abstraction from Communication Structure

At this point we have defined how the structuring of the classes, packages and (uses) relationships of the model are affected by the replacement of one package of the model by its CLD. The next section defines how the visibility of the model parts (methods, classes and packages) is affected.

2.1.6 Visibility

In this section we define the changes in the visibility of methods, classes and packages as a consequence of replacing a package by its CLD. The visibility of the classes and packages not contained by the package for which we construct the CLD are not affected as these classes and packages are unchanged. We however have to define the visibility of the methods of the CLD and the visibility of this class. These visibilities are chosen such that they provide an equivalent hiding of the methods of the original classes contained in the package which we replace.

Example In Figure 12(a), the package P is replaced by its CLD of Figure 12(b). The visibility of the methods of the CLD are chosen such that the visibility of the methods M_i with respect to classes not in P are identical. Method $M1$ in package P is not hidden by the package nor by class $C1$ and is thus a public method of the CLD of P. Methods $M2$ to $M4$ are private methods of the CLD. Class $C2$ is a private class of P and is not visible outside of package P and its methods thus become private methods of the CLD. Method $M2$ is a private method of $C1$ and is thus likewise a private method of the CLD. □

We define the visibility of the classes and packages in the new model with the CLD in three steps. We first define the visibility of the packages. We then define the visibility of the classes, except for the CLD. Lastly, we define the visibility

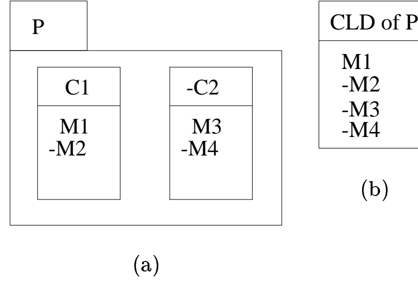


Figure 12: Example of changing method visibility

of the CLD and the visibility of its methods. We begin with the visibility of the packages:

$$
\begin{array}{l}
\textbf{DeltaDataPackageVisibility} \\
\hline
DeltaDataUsesLabels \\
\hline
(\forall p' : ContainedPackages\ \ TopPackage' \bullet \\
\qquad packagevisibility'\ \ p' = \\
\qquad (\mu\, p : ContainedPackages\ \ TopPackage \mid \\
\qquad\quad ProjIdentity\ \ p = ProjIdentity\ \ p' \bullet \\
\qquad packagevisibility\ \ p))
\end{array}
$$

The package visibility of the packages not in *AllPackages p* is unchanged. We continue with the visibility of the classes.

$$
\begin{array}{l}
\textbf{DeltaClassVisibility} \\
\hline
DeltaDataPackageVisibility \\
\hline
\mathrm{dom}\ classvisibility' = \mathrm{dom}\ \ classvisibility \setminus (ContainedClasses\ \ p) \cup \{cld\} \wedge \\
(\mathrm{dom}\ \ classvisibility) \lhd classvisibility' = \\
(\mathrm{dom}\ \ classvisibility \setminus (ContainedClasses\ \ p)) \lhd classvisibility
\end{array}
$$

- The domain of $classvisibility'$ are the classes of of the original model without the classes in *ContainedClasses p* but with the CLD of p added.
- The class visibility of the classes *not* contained by p is unchanged.

We now define the visibility of the CLD and for the visibility of its methods[2]
:

[2] The (**let** $x == expr \bullet body$) local definition is used to associate a variable x with an expression *expr* within the text *body*. The **let** local definition is used to introduce short names for expressions or to avoid writing the same expression more than once in a schema.

$DeltaDataVisibility$

$DeltaClassVisibility$

$$
\begin{array}{l}
classvisibility\ cld = \\
\textbf{if}\ (packagevisibility\ p = publicpackage)\ \textbf{then}\ publicclass \\
\quad \textbf{else}\ privateclass \\
(\forall\, m : cld.methods \bullet \\
(\textbf{let}\ d == (\mu\, du : ContainedClasses\ p \mid m \in du.methods \bullet du) \bullet \\
\quad cld.methodvisibility\ m = \\
\quad \textbf{if}\ d \notin export\ p\ \textbf{then}\ private \\
\qquad \textbf{else}\ d.methodvisibility\ m))
\end{array}
$$

For a method m of the CLD, the class d of package p is the class for which method m is originally defined. The visibility of the CLD of p and the visibility of its methods is defined so as to offer the same visibility of the methods of the classes of p as in the original model. The only public methods of the CLD are the public methods of the classes of p which are exported by p. Protected methods of classes of p are mapped to private methods of the CLD. These methods are not visible to a class not contained by p. These methods are therefore hidden from the rest of the classes in the model with the CLD of p. Furthermore, we have ensured no class inherits the methods of the CLD of p.

The CLD of a package thus not only collects the methods of the package. The encapsulation of the classes by the package is reflected in the construction of its CLD. The CLD of a package is thus more than a straightforward aggregate class which only groups methods and ignores the visibility of the aggregated classes. In this case, only the public methods of the classes exported by the package are public methods of the CLD.

Example Figures 13(a) to 13(c) have three different techniques for specifying the visibility of method M of class C. In all three cases the method is however hidden from the rest of the model. The CLD of P is in all three cases is a class with a private method M. The CLD of a package abstracts from whether the hiding of a method of a class is a result of the encapsulation of an entire subpackage (Figure 13(a)), the encapsulation of the class with the method (Figure 13(b)) or the encapsulation of the method by a class (Figure 13(c)). All three packages give the identical CLD of figure 13(d). □

2.1.7 Export and Import

The schema *DeltaDataExportAndImport* defines the changes in export and import which occur as a result of the use of the CLD of a package. Basically, all import (export) of a class of p is replaced by import (export) of the CLD of p.

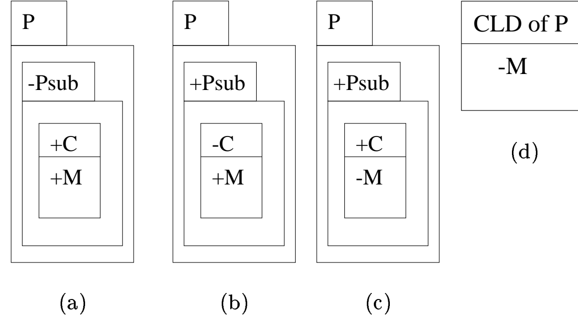


Figure 13: Three ways of encapsulating M

DeltaDataExportAndImport

DeltaDataVisibility

$$
\begin{array}{l}
(\forall p' : ContainedPackages\ TopPackage' \bullet \\
\quad (\mathbf{let}\ op == \\
\quad (\mu\, tempop : ContainedPackages\ TopPackage \mid \\
\qquad ProjIdentity\ tempop = ProjIdentity\ p' \bullet tempop) \bullet \\
\quad (export'\ p' = \\
\quad (export\ op \setminus ContainedClasses\ p) \cup \\
\quad (\mathbf{if}\ (export\ op \cap ContainedClasses\ p \neq \varnothing) \\
\qquad \mathbf{then}\ \{cld\}\ \mathbf{else}\ \varnothing)) \\
\quad \wedge \\
\quad import'\ p' = \\
\quad (import\ op \setminus ContainedClasses\ p) \cup \\
\quad (\mathbf{if}\ (import\ op \cap ContainedClasses\ p \neq \varnothing) \\
\qquad \mathbf{then}\ \{cld\}\ \mathbf{else}\ \varnothing)))
\end{array}
$$

The export (import) of a package p' of model with the CLD of p is equal to the export (import) of package op (*o*riginal *p*ackage) with export (import) of classes of p replaced by the CLD of p.

At this point, we have defined the changes in structure, visibility and import and export with respect to the data perspective. We have thus completely defined the CLD of a package with respect to the data perspective. We introduce the shorthand *DeltaData* for the CLD of packages defined for the data perspective.

DeltaData

DeltaDataExportAndImport

We continue with the definition of a CLD with respect to the behaviour perspective.

2.2 Behaviour Perspective

With respect to the behaviour perspective, we associate, in SOCCA, with each class one or more external STDs that specify the *external behaviour* of the class. These external STDs of a class describe behaviour that is visible to other classes. This is the order in which calls to methods the class exports are accepted. For the definition of the CLD of a package p with respect to the behaviour perspective, we need to migrate the external STDs of classes of p which we replace by its CLD to this newly formed class.

Example In Figure 14(a), we draw the identity of the external STD(s) of a class in a box next to the class. The CLD of the package P of Figure 14(b) groups the external STDs of the classes in P. Note that this notation for external STDs is ad-hoc and not *the* way to display the external STDs. We do not show the actual STDs as these are identical before and after the migration from the individual classes of p to the CLD of p. □

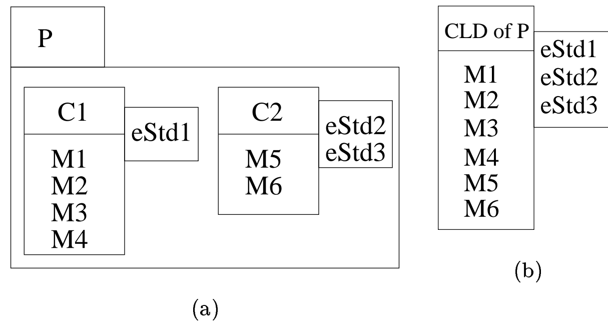


Figure 14: Example of relocating the external STDs

The external STDs of classes not contained by this package p remain unchanged.

The CLD of a package abstracts from the structure of the external behaviour of the classes of the package. All external STDs of the classes are associated to the CLD of the package and the external behaviour of the classes of the package can not be deduced from the external behaviour of the CLD. For example, Figure 14(b) does not give any information on which external STDs of the CLD is contributed by which class of the package without resorting to Figure 14(a).

$$
\begin{array}{l}
\textit{DeltaBehaviour} \\
\hline
\textit{DeltaData} \\
\hline
\textit{externalbehaviour}' = \\
\quad \{e : \textit{Class} \times \textit{STD} \mid (\exists\, c : \textit{Class};\ \textit{estd} : \textit{STD} \mid e = (c, \textit{estd}) \bullet \\
\qquad (\exists\, d : \textit{ContainedClasses}\ \textit{TopPackage} \mid \\
\qquad\quad d \notin \textit{ContainedClasses}\ p \wedge \\
\qquad\quad (d, \textit{estd}) \in \textit{externalbehaviour} \bullet \\
\qquad c = d) \\
\qquad \vee \\
\qquad (\exists\, d : \textit{ContainedClasses}\ p \mid \\
\qquad\quad (d, \textit{estd}) \in \textit{externalbehaviour} \bullet \\
\qquad c = \textit{cld}))\}
\end{array}
$$

- The external STDs of the classes not in p are associated to the same classes.
- The external STDs of the classes in p are associated to the CLD of p.

We continue with the definition of a class with respect to the functionality perspective.

2.3 Functionality Perspective

With respect to the functionality perspective, in SOCCA, we associate with each method of a class an internal STD that describes how that method is realised. We need to move the internal STDs of the classes in the package p of which we form the CLD to this new class.

Example In Figure 15(a), we draw the identity of the internal STDs of the methods of a class in a box next to the class. The CLD of the package P given in Figure 15(b) groups the internal STDs of the classes in P as this class has as methods the methods of the classes in p. Note that this notation for internal STDs is ad-hoc and not *the* way to display the internal STDs. □

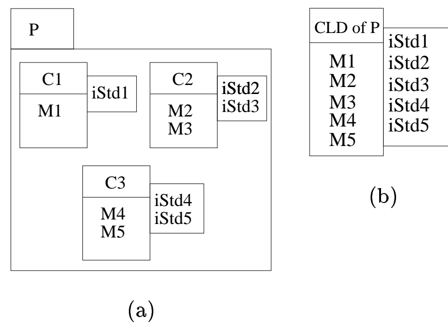


Figure 15: Example of relocating the internal STDs

The CLD of a package abstracts from the structure of the internal behaviour of the classes of the package. All internal STDs of the classes are associated to the CLD of the package and the internal behaviour of the classes of the package can not be deduced from the internal behaviour of the CLD. For example, Figure 15(b) does not give any information on which internal STDs of the CLD is contributed by which class of the package without resorting to Figure 15(a).

Additionally, we need to change calls to methods of the classes of p to into calls to the methods of the CLD of p as the classes of p no longer exist in the new model. For this purpose, we introduce the function *changecalls* which changes all calls in a single internal STD to calls to methods of a given set of classes to calls of the same methods of one given class. So *changecalls* $(I, \mathit{ContainedClasses}\ p, \mathit{cld})$ returns the internal STD I' where all calls to methods of the classes of p are changed to calls of methods of the class *cld*.

Example Figure 16 gives the internal STD I which makes various calls. If classes *Wuz* and *Mu* are classes of the package p of which we want to build the CLD, then *changecalls* $(I, \{\mathit{Wuz}, \mathit{Mu}\}, \mathit{cld})$ gives us the internal STD of Figure 17. □

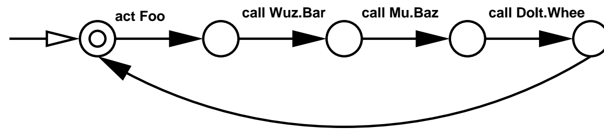


Figure 16: STD I before *changecalls*

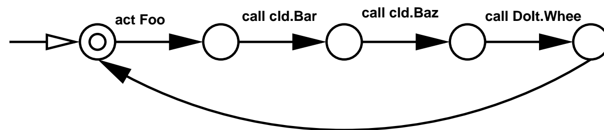


Figure 17: STD I' after *changecalls*

We now define the function *changecalls*:

$$
\begin{array}{l}
changecalls : (STD \times \mathbb{F}\, Class \times Class) \rightarrow STD \\
\hline
\forall\, I : STD;\ C : \mathbb{F}\, Class;\ c : Class \bullet \\
\quad changecalls\ (I, C, c) = \\
\quad (\mu\, I' : STD \mid \\
\qquad I'.Identity = I.Identity \wedge \\
\qquad I'.states = I.states \wedge \\
\qquad I'.transrel = \\
\qquad \{t : (STATE \times SYMBOL) \times STATE \mid \\
\qquad\quad (\exists\, l : SYMBOL;\ s_1, s_2 : I'.states \mid t = ((s_1, l), s_2) \bullet \\
\qquad\quad (l \notin \operatorname{ran}\ call \wedge t \in I.transrel) \vee \\
\qquad\quad (\exists\, l' : SYMBOL;\ ci : ClassIdentity;\ m : MethodIdentity \mid \\
\qquad\qquad l' = call\ (ci, m) \wedge ((s_1, l'), s_2) \in I.transrel \bullet \\
\qquad\qquad\quad (\textbf{let}\ IdentitiesC == \{i : ClassIdentity \mid (\exists\, d : C \bullet d.identity = i)\} \bullet \\
\qquad\qquad\quad ci \notin IdentitiesC \Rightarrow l = l' \wedge \\
\qquad\qquad\quad ci \in IdentitiesC \Rightarrow l = call\ (c.identity, m))))\} \\
\quad \bullet\, I')
\end{array}
$$

In the above definition:

- The set $IdentitiesC$ are the identities of the classes in C.
- Labels of the STD I which are not calls are left unchanged.
- Labels which are calls to methods of classes whose identity does not occur in $IdentitiesC$ are likewise left unchanged.
- Calls to methods of classes whose identities are in $IdentitiesC$ are changed to calls of the same method of c.

The schema *DeltaFunctional* moves the internal STDs of the classes in a package p to the CLD of p. Furthermore, calls to the methods of classes in p are changed to calls to the methods of the CLD.

$$
\begin{array}{l}
\underline{\ DeltaFunctional\ } \\
DeltaBehaviour \\
\hline
(\forall\, c : ContainedClasses\ \ TopPackage \bullet \forall\, m : c.methods \bullet \\
\quad \textbf{let}\ I' == changecalls\ ((internalbehaviour\ (c, m)), (ContainedClasses\ \ p), cld) \bullet \\
\quad c \notin ContainedClasses\ \ p \Rightarrow \\
\quad internalbehaviour'\ (c, m) = I' \\
\quad \wedge \\
\quad c \in ContainedClasses\ \ p \Rightarrow \\
\quad internalbehaviour'\ (cld, m) = I')
\end{array}
$$

- The internal STD I' is an STD in which all calls to methods of the classes of p are changed to calls of the methods of the CLD.
- Internal STDs of classes not in p, the package to be replaced, are associated to the original classes. Calls to methods of the classes of p have become calls of the methods of the CLD of p.

- Internal STDs of classes in p are associated to the CLD of p. Calls to methods of the classes of p likewise have become calls of the methods of the CLD of p.

The next section defines a CLD with respect to the communication perspective.

2.4 Communication Perspective

The communication perspective in SOCCA expresses how regulation of communication between instances of classes is controlled by managers and employees. With respect to the communication perspective we need to define the changes to the managers and employees of a model as a consequence of replacing a package of the model by its CLD. This is fairly straightforward as the formalisation of employees and managers as originally defined in [DGSK+99] and as reused in [tHDG+99] defines the structure and mapping of managers and employees (states and transitions) to classes separately.

We first define the changes in the employees. An employee is defined as a set of subprocesses with traps based on an underlying internal STD. We have only changed some calls in the internal STDs to calls of methods of the CLD. We have not changed the states or edges. We thus only need to change the employees to reflect the changes in calls for the underlying internal STD defined in the functionality perspective.

Example In Figure 18(a) we give internal STD I with one of its subprocesses $S1$. Its call to method *bar* of class B is changed to a call of method *bar* of the class *cld* to give the internal STD I' of Figure 18(b). Subprocess $S1'$ of the modified internal STD has the same structure as the original subprocesses S except for the change in labels. □

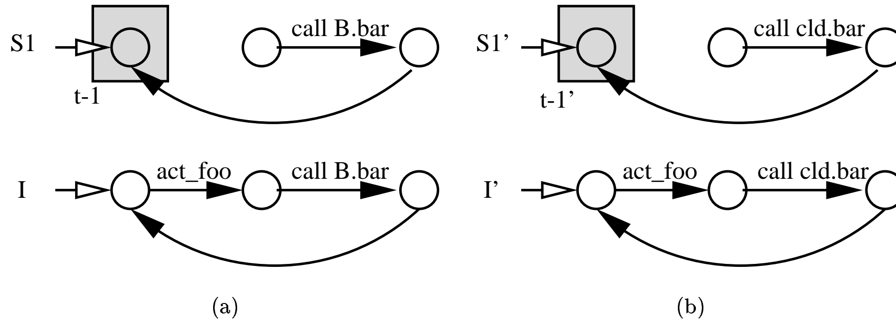


Figure 18: Changes in employees

We define the changes in the employees using the schema *DeltaCommunicationEmployees*.

$$
\begin{array}{l}
\textit{DeltaCommunicationEmployees} \\
\hline
\textit{DeltaFunctional} \\
\hline
(\forall\, intstd : internalstds \bullet \\
\quad (\forall\, intstd' : internalstds' \mid \\
\quad\quad intstd' = changecalls\ (intstd, (ContainedClasses\ p), cld) \bullet \\
\quad\quad (\forall\, e' : employee \bullet \\
\quad\quad e' \in (asemployee'\ intstd') \Leftrightarrow \\
\quad\quad (\exists\, e : asemployee\ intstd \bullet \\
\quad\quad\quad e.std = intstd \wedge \\
\quad\quad\quad (\forall\, sub' : STD;\ traps : \mathbb{F}(\mathbb{F}\, STATE) \bullet \\
\quad\quad\quad\quad (sub', traps) \in e'.pts \Leftrightarrow \\
\quad\quad\quad\quad (\exists\, sub : STD \mid sub' = changecalls\ (sub, (ContainedClasses\ p), cld) \bullet \\
\quad\quad\quad\quad\quad (sub, traps) \in e.pts)))))
\end{array}
$$

- $intstd's$ is an internal STD which is a modified internal STD $intstd$ of the model with the original package p.
- An employee of $intstd'$ has nearly the same subprocess and trap structure compared to an employee of $intstd$. Only calls to methods of the classes of p are changed to calls to methods of the CLD of p.

We next define the changes to the managers. Each manager is defined as an extension of an external STD. We have not changed the structure of the external STDs of a model in the definition of the CLD with respect to the behaviour perspective. We have only "moved" the external STDs of the classes of the package p of which we construct the CLD to this new class. Furthermore, a manager is defined in terms of the names of subprocesses and traps it uses for its employees. These have not changed due to the introduction of the CLD of p. The employees of the classes of p have "moved" to the CLD of p, but are identical in name and structure. The managers of the CLD of a package can thus remain the same. We only have to "move" these managers defined as extensions of the external STDs of classes of p to the CLD of p.

Example In Figure 19, the mapping from the external STDs to the corresponding managers is indicated by a dotted arrow. Both the external STDs of the classes of the package p and the external STDs of the CLD of p map to the same managers. □

The change in managers due to the definition of the CLD of p is trivial:

$$
\begin{array}{l}
\textit{DeltaCommunication} \\
\hline
\textit{DeltaCommunicationEmployees} \\
\hline
asmanager' = asmanager
\end{array}
$$

The function $asmanager$ maps an external STD to a manager. We have already "moved" the external STDs. We therefore automatically associate the

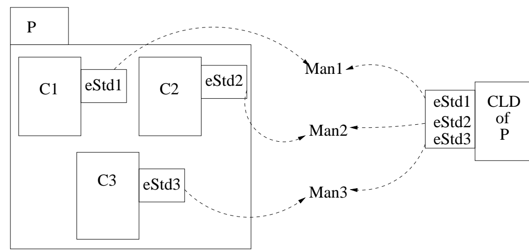


Figure 19: Mapping of External STDs to Managers

managers of the classes of package p to the CLD of p once we have relocated the external STDs.

The above schema not only defines the managers of the CLD. It completes the definition of the CLD. We have completely defined the CLD of a package with respect to all perspectives.

We have formalised how the CLD of a package is a black-box representation of the behaviour of a package. The next section presents some conclusions and future work.

3 Conclusion

Most object-oriented (OO) modelling/ programming languages use some form of (hierarchical) packages to manage the complexity of the model by structuring and encapsulating classes. Encapsulation of classes by packages hides classes intended only for use within the package from the rest of the model. The exported classes of a package are deemed essential with respect to the rest of the model while the "non-essential" classes which only play a role within the package are hidden. These packages are however syntactical structures and have no behaviour of their own. A modeller has to consider all individual classes which are contained by a package to study the collective behaviour of the classes of the package.

We introduce a layer above the model with packages where we can observe the behaviour of the packages of the model in terms of single classes which capture the behaviour of the package. We call each such a class a class-like description (CLD) of a package. The CLD represents the original package and brings the concepts of packages and classes closer together. All packages of a model can be replaced by their respective CLDs as sketched in Figure 20. We give a model where the behaviour of its packages P_1 to Pn are captured by their corresponding CLDs.

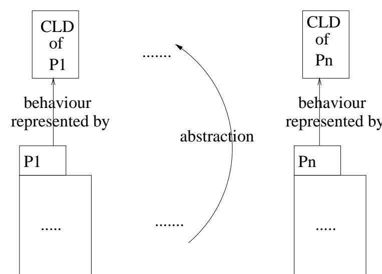


Figure 20: The Levels of Behaviour in a Model

Any package of the model can thus be seen either as a complex structure or as a single class, its CLD. A modeller who is developing a package will want/need to observe all details while other developers working on other parts of the model only observes the non-encapsulated classes of the package. If even less detail is desired, the CLD of the package is used to represent the package as a black box.

We have illustrated and formalised the CLD of hierarchical packages in SOCCA (Specification of Coordinated and Cooperative Activities). SOCCA is a graphical formalism and associated method for object-oriented modelling which is under active development at Leiden University. We have chosen SOCCA as the strength of SOCCA is that it allows the precise and detailed specification of the communication and synchronisation between the modelled classes. The modelling of the behaviour of a class in SOCCA is very sophisticated. As a consequence, the CLD of a SOCCA package captures the behaviour of the package at a high level of precision. In a SOCCA model, as formalised in this text, a package can thus be observed as a single, well defined class.

4 Future Work

We have three main topics of interest for future work.

The integration of behaviour in the interface of a package in most OO languages is often minimal or non-existent. It is therefore not clear whether a change in the behaviour of a package affects the interface of the package. The interface of a package as such does not sufficiently capture the dependencies of the package with the rest of the model. In [tHDG+00], based on the definition of the CLD, we define the interface of a package to include the aspect of behaviour.

Secondly, we wish to express the behaviour of a package in terms of the instantiated objects of the package. In this document, we have expressed the behaviour of a package at the type level in terms of a single class. We wish to develop an abstract object-like description of the behaviour of the objects instantiated from the classes of a package. We define this behaviour in terms of an object instantiated from the CLD of the package. The behaviour of a package can then be seen as a single class at the type level or at the instance level as a single instantiated object from this one class. This work will take place upon completion of [tH00].

Lastly, the concept of a CLD can be used to define some useful concepts. For example, we have several times during the formalisation of the CLD reflected on how two packages were essentially the same. In [tHGKE00], we use the CLD of packages to formally define when packages are equivalent. We also show how the CLD concept can be used during top-down development of a model. We show how the CLD concept can be used, for example, to define decomposition of a class.

References

[ABKR89] P. America, J.W. Bakker, J.N. Kok, and J.J.M.M. Rutten. Denotational semantics of a parallel object-oriented language. In *Information and Computation*, number 83 in 2, pages 152–205, 1989.

[BKS98] M.M. Bonsangue, J.N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. In *Proceedings of the 4th International Conference on Mathematics of Program Constructions (MPC'98)*, pages 68–95, 1998.

[COM99] Microsoft COM Technologies, 1999. http://www.microsoft.com/com/.

[Dem97] C. Demmer. Unified Modeling Language vs. MWOOD- I, 1997. http://stud2.tuwien.ac.at/~e8726711/ummw2.html#MU1.

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

[EK99] Andy Evans and Stuart Kent. Core meta-modelling semantics of UML: The pUML approach. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachussetts, 1994.

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

[Gro96] Object Management Group. Realtime CORBA- a White Paper - issue 1.0. Technical Report ORBOS/96-09-01, Object Management Group, september 1996.

[HP94] Martin Hofmann and Benjamin C. Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 1994.

[Jon93] C.B. Jones. A π-calculus semantics for an object-based design notation. In *Proceedings of CONCUR'93*, volume 715, 1993.

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

[Mey99] Bertrand Meyer. On to components. *Computer (IEEE)*, january 1999. downloadable at `http://www.eiffel.com/doc/manuals/technology/bmarticles/computer/components/page.html`.

[MWB99] Joaquin Miller and Rebecca Wirfs-Brock. How can anything be both a classifier and a package? In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.

[NA99] Brian Nielsen and Gul Agba. Towards reusable real-time objects. *Annals of Software Engineering 7*, pages 257–282, 1999.

[Obj91] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, August 1991.

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

[RK99] Ferenc Racz and Kai Koskimies. Tool-supported compressing of UML class diagrams. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.

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

[Sil99] A.R. Silva. *Concurrent Object-Oriented Programming: Separation and Composition of Concerns using Design Patterns, Pattern Languages, and Object-Oriented Frameworks.* PhD thesis, Technical University of Lisbon, march 1999.

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

[tH00] P.J. 't Hoen. Instance level packages for SOCCA. Technical report, Leiden Institute of Advanced Computer Science, 2000. To Appear.

[tHDG+99] P.J. 't Hoen, J.H.M. Dassen, L.P.J. Groenewegen, I.G. Sprinkhuizen-Kuyper, P.W.M. Koopman, and G. Engels. SOCCA extended with UML like packages. Technical Report 99-06, Leiden Institute of Advanced Computer Science, September 1999.

[tHDG+00] P.J. 't Hoen, J.H.M. Dassen, L.P.J. Groenewegen, I.G. Sprinkhuizen-Kuyper, P.W.M. Koopman, and G. Engels. Behaviour interfaces of uml-like packages. Technical report, Leiden Institute of Advanced Computer Science, 2000. To Appear.

[tHGKE00] P.J. 't Hoen, L.P.J. Groenewegen, P.W.M. Koopman, and G. Engels. Behaviour interfaces of uml-like packages. Technical report,

Leiden Institute of Advanced Computer Science, february 2000. To Appear.

[tHZG99] P.J. 't Hoen, V. Zweije, and L.P.J. Groenewegen. SOCCA Basics. Technical Report 99-14, Leiden Institute of Advanced Computer Science, 1999. downloadable as draft at `http://www.wi.LeidenUniv.nl:/~hoen/publications/soccabasics.ps`.

[UML99] Unified modeling language 1.3. Technical report, Rational Software Corporation, 1999.

[WMB99] Axel Wienberg, Florian Matthes, and Marko Boger. Modeling dynamic software components with UML. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.