Universität Dortmund
Fachbereich Informatik
Lehrstuhl für Software-Technologie

MEMO Nr. 129

An Approach to Algebraic Semantics of Object-Oriented Languages

Alexander Fronk

Oktober 2002

Internes Memorandum des
Lehrstuhls für Software-Technologie
Prof. Dr. Ernst-Erich Doberkat
Fachbereich Informatik
Universität Dortmund
Baroper Straße 301

D-44227 Dortmund

ISSN 0933-7725

An Approach to Algebraic Semantics of Object-Oriented Languages

Alexander Fronk
Software-Technology, University of Dortmund
44221 Dortmund, Germany
fronk@LS10.de

October 17, 2002

Abstract

Studying the semantics of programming languages has a long tradition in computer science. Various approaches use various formalisms with various objectives. In the last two decades, algebraic specifications have frequently been used to study functional as well as imperative languages, and, in particular, object-orientated ones, thereby often focusing on specific aspects and concepts of this programming paradigm. In this paper, we follow this tradition and develop an algebraic semantics of a sample object-oriented language. We thereby distinguish between the object-oriented concepts of the language to structure code, and the imperative ones to implement functionality and thus the algorithmic parts of the language. Therefore, our approach encompasses two steps: first, we develop an algebraic semantics of basic object-oriented principles, into which, secondly, the semantics of the language's imperative parts is embedded. Static semantic aspects are captured by structured algebraic specifications, whereas dynamic ones are reflected by many-sorted algebras. These aspects are treated as "second order" concepts and are thus interpreted within a model class of the underlying specification. The approach elaborated here can be employed to formalize the semantics of "standard" object-oriented languages such as Eiffel, Java, or C++.

1 Introduction

In [22], an object-oriented language for constructively describing hyperdocuments, *DoDL* [14, 15] for short, was given an algebraic semantics. In this thesis, a hyperdocument is understood as a collection of media objects such as texts and graphics connected to each other in a non-linear fashion, i.e. they are *hyperlinked*. It was shown in the thesis that the semantics of *DoDL* can easily be formalized without considering hypermedial aspects. Nonetheless, it is important to remark that the language was especially tailored for the aforementioned application domain. Hence, the language only contains those syntactic constructs necessary for the object-oriented construction of hyperdocuments. Vice versa, concepts such as pointers, threads, or exception handling are not considered in *DoDL* if they do not contribute to describing and implementing hyperdocuments on a conceptual level. This decision keeps the language and its usage simple and comprehensible. Of course, those and other techniques can be put to work within the mentioned domain if technically richer languages such as Java or C++ are used.

Even without considering hypermedial aspects, *DoDL* is rich enough to thereupon develop an approach to formalizing an algebraic semantics of object-oriented languages in general. We use *DoDL* as a sample language to comprehensibly focus on this approach which can be employed to formalize the semantics of any other object-oriented language, and which thus works without loss of generality.

The approach works in two steps. First, we formalize classes, attributes and methods as well as class relations such as aggregation, locality, inheritance and genericity through suitable algebraic specifications. Transformation rules establish semantic functions [29] for each syntactic concept offered by *DoDL*. The integration of simple algebraic specifications, hierarchical specifications, specifications with hidden symbols, and parameterized specifications allows formalizing these concepts as close as possible to the object-oriented paradigm. Thereby, our approach differs from flat specification approaches as usually used in the literature. For example, subclasses with local classes are understood as an integration of hierarchical specifications with hidden symbols. Notions such as *object* and *type*, but also *redefinition*, *late binding* and *polymorphism* can be represented by many-sorted algebras. They serve as a loose semantics. Hence, algebraic specifications can be understood as an intermediate language between a *DoDL*-program and its semantics.

In the second step, we show how the algebraic formalization of imperative features such as *method invocation* and *control structures* can be integrated. Since the algebraic definition of imperative languages and control structures has already been studied (c.f. [7, 13]), we concentrate on object-oriented aspects here. That is, we focus on concepts like *this* and *super*, as well as on object-oriented message invocation and polymorphism, extending the usual concepts found in imperative languages.

The semantics presented here has been encoded within a compiler system [23] where the transformation rules directly determine code generation. The compiler construction, however, is not subject to this paper.

This paper is organized as follows: Section 2 discusses related work; the language under consideration is introduced in Section 3, its semantics is given in Section 4; the paper concludes in Section 5 and suggests further work.

Acknowledgements. We gratefully thank Prof. Dr. E.-E. Doberkat for supervising the author's PhD thesis. He established working conditions under which teaching and research could fruitfully take place. These conditions together with his constructive guidance and detailed remarks were the basis for successfully accomplishing the thesis the present paper is based on. The discussions with Prof. Dr. P. Padawitz as well as his helpful remarks are very much appreciated. They both gave important hints on some subtle points.

2 Related Work

Studying the semantics of languages has a long tradition in computer science. Various approaches use various formalisms with various objectives. On a sheer syntactical level, operational semantics are based on an abstract machine describing the execution of programs by state transitions; denotational semantics assign each instance of a syntactic construct a mathematical object; axiomatic calculi allow to compute assertions on programs formalized through pre- and postconditions, and inference rules are defined on individual syntactic units [53].

In the last two decades, algebraic specification has frequently been used to study denotational semantics of functional [6, 55] and imperative [7] languages. Algebraic specification languages inherently provide algebraic semantics and thus mathematical objects denoting syntactical constructs (c.f. [8, 27, 21, 12, 5, 57, 25]). Algebraic semantics are also used in the field of abstract state machines to formalize the machine model underlying an operational semantics [31]. Based on this approach, Gurevich shows such a semantics for the C programming language [32]. In the context of algebraic specification languages, a variety of object-oriented aspects and concepts have been studied, and many results carry over to object-oriented programming languages as well. We briefly sum up some interesting points to discuss objects, their states, classes, and inheritance to give a short overview on the appealing possibilities algebraic specifications offer for object-oriented issues.

2.1 Objects

In [30], algebraic specifications model instances, that is each specification corresponds to a class instance in a specific state. For each instance obj of a class c a sort of interest [12], c, together with a constant of the form $obj :\to c$ is introduced. An axiom of the form $obj = t$ describes the state of obj by the term t. Changing the object's state as well as creating or deleting objects is reflected by generating new axioms or exchanging existing ones, introducing new constants, or deleting existing ones, respectively. Hence, the algebraic specification and thus the presentation semantics [49] of an object is changed syntactically. This approach aims at modelling declaration and manipulation of objects.

Ehrich follows a similar approach [17]. Objects are modelled by parameterized specifications, where the class designator is used as parameter. These specifications contain operations describing object creation and deletion as well as state change in the context of open, reactive, and distributed systems.

Modelling state changes is discussed in [46] as well. This approach is based on category theory. Object configurations are described by *communities* of objects related to each other. These communities are formalized through *configuration specifications* allowing for suitably representing objects through pairs of attributes and values. State change is modelled through non-homomorphic transformations on the respective algebras. This approach touches issues like persistence and equality of objects.

Objects may trivially be modelled by specifications of the following kind:

$OBJSPEC =$

sorts $ObjVar, S_1, \ldots, S_n$

opns $a_1 : ObjVar \to S_1,$
$\ldots$
$a_n : ObjVar \to S_n,$
$v_1 :\to S_1,$
$\vdots$
$v_n :\to S_n,$
$makeObj : S_1 \times \ldots \times S_n \to ObjVar$

axms $a_1(makeObj(v_1, \ldots, v_n)) = v_1,$
$\ldots$
$a_n(makeObj(v_1, \ldots, v_n)) = v_n$

Here, objects with n attributes are defined. Object identifiers are taken from an arbitrary set of object names, $ObjVar$. Each attribute, a_i, $i = 1 \ldots n$, is formalized as an operation of the form $a_i : ObjVar \to S_i$, where each S_i represents the set of values for attribute a_i. The operation $makeObj$ is used to create objects. It is described semantically by axioms of the form $a_i(makeObj(v_1, \ldots, v_n)) = v_i$. Constants $v_i :\to S_i$ fix the object's state. A model-class semantics is provided in this approach such that each model represents an object's state.

In [26], object states are modelled through specifications with hidden symbols (c.f. [56], Chap. 5). A hidden sort corresponds to a type identifier, and is interpreted through a carrier set reflecting the object's possible states. The current state of the object under consideration can only we observed through suitable methods. This approach aims at checking properties of concurrent systems.

2.2 Classes

In [4], a formal foundation for a framework is provided in which algebraic specifications and object-oriented programming are integrated to work hand in hand. Classes are represented by flat specifications, attributes and methods are both represented by operations. Their semantics are described by axioms. Model classes represent class instances. This approach is also used in [22].

In [44], classes are elaborated in more detailed and defined by five algebraic specifications. They cover a parameter part, an instance interface, a class interface, an import interface, and an implementation part. This definition allows to distinguish between different kinds of inheritance as discussed next.

2.3 Inheritance

Inheritance, in contrast to subtyping, is used as a code structuring mechanism in [4]. The semantics of subtyping is based on partial order-sorted algebras (c.f. [51], Sect. 2.10.4) and fixed through model-class inclusion.

In [45], subclassing is called reuse inheritance. It contrasts from specialization inheritance in such a way that the former allows omission or redefinition of class methods, and that in the latter a subclass obeys the semantics of its superclass. Again, the semantics is fixed through properties laid on model-classes. This process can be found in many approaches, for example in [11], where the specification language Glider is discussed.

2.4 Result

However, these approaches depict certain aspects of object-orientation thereby using individual strategies for their description. In our work, we elaborate a mechanism to uniformly cover basic object-oriented concepts in one formal approach based on the integration of differently structured algebraic specifications.

3 The Language *DoDL*

The aim of this section is to introduce an object-oriented language, *DoDL*, simple yet rich enough to discuss an algebraic semantics. In contrast to [22], slight modifications were made to enhance readability. The language design of *DoDL* regards simple and complex classes, aggregation and use-relation, local classes, a form of inheritance with overloading, redefinition and polymorphism, and generic classes. Classes encompass attributes and methods. Assignment of values to attributes is done by bindings. Finally, scoping rules are mentioned.

3.1 Simple and Complex Classes

A class is given through a class frame as shown in Listing 1.

LISTING 1 The frame of a *DoDL*-class

```
class anyClass is
   declare  class localClass is
            ...
   end localClass;
   ...
   attributes  attrID: attrType;
               anOtherAttrID: list of anOtherAttrType;
               ...
   construct   methType methID(parType parID, ... ){ body }
               ...
end anyClass;
```

The frame defines the name of the class which can be generic (see Sect. 3.4), or may inherit from another class (see Sect. 3.3). The rule *declaration before use* always has to be obeyed. A class provides three optional sections:

the `declare`-section defines a list of local classes (see Sect. 3.2);

the `attributes`-section declares attributes in the form
`attrID`$_1$`, ..., attrID`$_n$`: [`**`list of`**`] attrType` where `attrID`$_i$, $i = 1, \ldots, n$, is an attribute of type `attrType` or a list of this type. The expression `attrID`$_i$`[`j`]` allows to address the j-th element of the list. *DoDL* provides the types `nat`, `bool` and `string`, defined as usual. Further types can be self-defined using classes;

the `construct`-section defines a list of methods of the form
`methType methID(parType parID, ...){ body }`. The empty type `void` may be used both as method type and parameter type. The method bodies use a JAVA-like syntax allowing for variable declaration and assignment, method invocation and the usual control structures such as sequences, alternatives, and `for`- as well as `while`-loops. The identifier `main` is reserved to designate the main method, i.e. the first method invoked at runtime. The class containing this method is called *main class*.

A class without any section is called *empty class*. **Simple classes** are not generic, do not inherit from any class, and do not define local classes. A class is called **complex** if it has at least one of these properties. A ***DoDL*-program** is understood as a collection of either simple or complex classes with exactly one main class.

Aggregation is understood as a part-of relation between an *aggregate* and its *elements*. Elements are classes used in attribute declaration. In contrast, a **use-relation** is deduced by classes used as parameter or method type in method definition. *DoDL* does not provide references. Hence, the notion of association is restricted to aggregation, and both aggregation and use-relations are acyclic.

3.2 Local Classes

Local classes are defined in the **`declare`**-section and may be simple or complex. *DoDL* implements the concept of composition here: using local classes in attribute declaration is only allowed within its *embedding* class, i.e. the class the local class is declared in. The difference between aggregation and composition is illustrated in Listing 2. A declaration of the form `encloseLocalID: encloseID.local` is not allowed in *DoDL*.

LISTING 2 Local class definition with composition and aggregation

```
class enclosing is
   declare  class local is
            ...
            end local;
            ...
   attributes localID: local;        // Composition
   construct ...
end enclosing;

class referToLocal is
   attributes encloseID: enclosing; // Aggregation
   construct ...
end referToLocal;
```

3.3 Inheritance

A subclass is defined using an **is-with** tag. Listing 3 shows its form.

LISTING 3 A subclass in *DoDL*

```
class father is
   ...
end father;

class son is father with
   ...
end son;
```

Each subclass has a unique superclass, that is, multiple inheritance is not allowed. We assume that attributes, methods and local classes of a superclass are replicated within the subclass. New attributes, methods and local classes can be added. If a method occurring in a subclass has the same signature as in its superclass, the method is **redefined**. Different parameter types or a different number of parameters lead to **overloading**. We use **late binding** in case the type of an instance (or object) calling a method cannot be determined during compilation.

We distinguish between **subclass relations** and **subtype relations**. We understand a subclass relation as a partial order on classes induced by subclass declaration, called *class inheritance*. A subtype relation, however, regards class instances. A type is thereby understood as the set of all instances of a class, such that each instance of a subclass is an instance of its superclass (c.f. [54]). This property is called *substitution* and characterizes *type inheritance* (c.f. [1, 9]). **Polymorphism** is characterized by different kinds of substitution (c.f. [1, 54, 9]). In *DoDL*, class inheritance induces type inheritance. Details will be discussed in Sect. 4.4.

In the context of inheritance, the qualifications **this** and **super** have to be discussed. Listing 4 shows abstractly how they are used and understood in *DoDL*.

LISTING 4 The qualifications **this** and **super**

```
class Root is
   attributes ...
   construct
      void m(void) { ... this.n(); ... }
      void n(void) { ... }
end Root;

class Inherit is Root with
   attributes ...
   construct
      void m(void) { ... super.m(); ... }
      void n(void) { ... }
end Inherit;
```

Class `Root` defines two methods, `m` and `n`. Their signature is unimportant, we just have to assume that they are redefined in class `Inherit` which is declared as a subclass of `Root`. The invocation **super**`.m()` in method `m` of class `Inherit` is responsible for calling method `m` of class `Root`. Thereby, the qualification **super** casts the caller of a method into an object of the caller's superclass.

The invocation of **this**`.n()` in method `m` of class `Root`, however, must be interpreted differently. In case `m` is called by an instance `r` of class `Root`, **this**`.n()` refers to method `n` of class `Root`. Hence, **this** correlates to `r`. In case `m` is called by an instance `i` of class `Inherit` via **super**, **this**`.n()` refers to method `n` of class `Inherit`. Then, **this** correlates to `i`. Summing up, **this** always correlates with the caller of a method, and this correlation is kept when superclass methods are called. Accessing attributes is always safe, since attributes cannot be redefined in subclasses. This approach is adapted from `self` in SMALLTALK (c.f. [28], or [1], Sect. 3.2).

3.4 Generic Classes

The syntax of generic classes is shown in Listing 5. The class designator is expanded by a formal parameter.

LISTING 5 A generic class

```
generic class gen [ formPar ] is ...
   ...
   attributes  attrID: formPar;
               ...
   ...
end gen;
```

Substituting each occurrence of the formal parameter by an actual parameter is called *actualization*. *DoDL* provides different kinds of actualization. We show them in Listing 6. Actual parameters can be used in attribute declarations, called *direct actualization*, or in subclass definition. We distinguish between *hierarchical actualization*, *hierarchical actualization with expansion*, and *generic actualization*. It is easy to see that hierarchical actualization (without expansion) of the form **class** `genInst` **is** `gen[actPar]` **with end** `genInst` together with an attribute declaration of the form `attrID: genInst` is mutually exchangeable with direct actualization of the form `attrID: gen[ActPar]`. The advantage of hierarchical actualization, however, is found in respecting the principle of locality and thereby supporting maintenance, since changing an actual parameter is done at exactly one location.

LISTING 6 Forms of actualization

```
class any is
   attributes  attrID: gen[actPar];      // direct actualization
end any;

class genInst is gen[actPar] with        // hierarchical actualization
end genInst;

class subGenInst is gen[actPar] with
   ...                                   // hierarchical actualization with expansion
end subGenInst;

generic class genSubGenInst[anOtherFormPar] is gen[actPar] with
   ...                                   // generic actualization with expansion
end genSubGenInst;
```

3.5 Bindings

A binding assigns values to attributes during compilation. Hence, a binding is responsible for creating instances (or objects) at compile-time. Further instances can be created at runtime using a simple **new**-operator. Changing an attribute's value means changing the belonging object's state and should only be done by suitable `set`-methods. Starting at the main class, the attributes of this class as well as the attributes of its superclass have to be assigned values. Superclasses are inspected recursively. The binding is structured and typed, i.e. attributes are assigned values of a given type by *simple assignments* or by *in-assignments* (see Listing 7). Element classes define a new scope. This protects from attribute name clashes. Lists use the same concept, since their type can be a self-defined class in which further attributes have to be bound. List items are separated by a vertical bar.

LISTING 7 A binding

```
binding mainclass is
   attrID: attrType = value;
   ...
   superClassAttrID: superClassAttrType = anOtherValue;
   ...
   in element: elemtType assign
      anOtherAttrID: anOtherAttrType = yetAnOtherValue;
      ...
   end;
   ...
   in listID: listType assign
      listItemValue;
      |
      anOtherListItemValue;
      |
      ...
   end;
   ...
end;
```

3.6 Scoping

DoDL defines a *top-level scope*, that is, each class if not local is placed on top-level. Top-level classes can use each other regarding *declaration before use*. For the time being, forward-declaration is not allowed, and mutual recursion is not possible. The local classes of an embedding class follow this rule on their *local level*, and their visibility is restricted to this level. Since local classes are defined in advance of attributes and methods, the latter cannot be used in local classes. Further, a class cannot be instantiated within its local classes. This avoids infinite embedding. Scoping carries over to inheritance: each class visible to a superclass is visible to its subclasses.

The scope of classes, attributes, and methods cannot be restricted by qualifiers like **`public`** or **`private`**. Hence, an **interface** encompasses all attributes and methods defined in a class and its superclass, recursively.

4 Algebraic Semantics of *DoDL*

This section follows the structure of Sect. 3. We develop formal mappings, i.e. transformation rules for the syntactic units presented there.

Our semantic approach works as follows. We use transformation rules to convert classes and their relations into structured algebraic specifications. Thereby, we define semantic functions. For technical convenience, we assume that *DoDL*-classes are syntactically correct and free of name clashes.

Interpretation rules thereupon establish model classes such that each algebra serves as a mathematical object denoting a syntactical construction (see Sect. 4.2 to 4.6). Algorithmic parts are embedded into this semantics in a second step (see Sect. 4.7).

Both transformation and interpretation rules can be formulated as an inference system. This leads, for example, to a formal type system for *DoDL*, or to the possibility to argue formally about properties of transformation and interpretation (c.f. [47]). For the time being, we do not exploit these advantages and prefer a natural language notation for simplicity.

For the readers convenience, we introduce some preliminaries first and fix the notation used in this paper. The reader familiar with algebraic specification may proceed to Section 4.2. Further details on algebraic specifications can be found, for example, in [56].

4.1 Algebraic Preliminaries

A **signature** is a tuple $\Sigma = \langle S, \Gamma \rangle$ where S is a set of **sorts**, $S = sorts(\Sigma)$, and Γ is a set of **operation symbols**, $\Gamma = opns(\Sigma)$. Variables are **S-sorted**. $X = \{X_s\}_{s \in S}$ denotes a **S-indexed** family of sets X_s of **variables** for each $s \in S$. The set of **S-sorted** Σ**-terms** over s, $\mathcal{T}(\Sigma, \mathrm{X})_s$ for short, is defined as usual.

An **algebraic specification**, SP, is a tuple $\langle \Sigma, E \rangle$ consisting of a signature, $\Sigma = sig(SP)$, and a set of formulas over Σ, $E = axms(SP)$, called **axioms**. A specification $\langle \Sigma', E' \rangle$ is called **subspecification** of $\langle \Sigma, E \rangle$, $\langle \Sigma', E' \rangle \subseteq \langle \Sigma, E \rangle$ for short, if $\Sigma' \subseteq \Sigma$ and $E' \subseteq E$ hold. **Terms** and **formulas** are defined as usual. We denote the **set of all well-formed formulas** by $WFF(\Sigma)$.

Convention: The collection of all specifications forms a class in set theory. In order not to conflict both with the notions of class and object used in object-orientation, as well as with fundamental mathematical questions, we restrict ourselves to a fixed universe of specifications, where the model classes defined in the sequel can be understood as sets.

A Σ-**algebra**, $\mathcal{A}$, is a pair $(\{A_s\}_{s \in S}, \{f^{\mathcal{A}}\}_{f \in \Gamma})$ consisting of a family $\{A_s\}_{s \in S}$ of non-empty **carrier-sets**, A_s, for each $s \in S$, and a set $\{f^{\mathcal{A}}\}_{f \in \Gamma}$ of **operations** $f^{\mathcal{A}} : A_{s1} \times \ldots \times A_{sn} \rightarrow A_s$ for each $f : s_1 \times \ldots \times s_n \rightarrow s \in \Gamma$. $\mathcal{A}$ is a **model** of a specification, $\langle \Sigma, E \rangle$, if $\mathcal{A}$ satisfies each formula $e \in E$. The **set of all models** of $\langle \Sigma, E \rangle$ is denoted by $Alg(\Sigma, E)$. The **loose semantics** of a specification $SP = \langle \Sigma, E \rangle$, $Mod(SP)$ for short, is defined as the set $Alg(\Sigma, E)$.

Let $\Sigma = \langle S, \Gamma \rangle$ and $\Sigma' = \langle S', \Gamma' \rangle$ be two signatures with $\Sigma \subseteq \Sigma'$, and let $\mathcal{A}'$ be a Σ'-algebra. The Σ-algebra $\mathcal{A}'|_{\Sigma}$ is called Σ**-reduct** of $\mathcal{A}'$, if for each $s \in S$ the carrier-set $(\mathcal{A}'|_{\Sigma})_s$ is defined as A'_s, and for each $f \in \Gamma$ the operation $f^{\mathcal{A}'|_{\Sigma}}$ is defined as $f^{\mathcal{A}'}$.

Let $\Sigma = \langle S, \Gamma \rangle$ and $\Sigma' = \langle S', \Gamma' \rangle$ be two signatures. A **signature morphism**, $\sigma : \Sigma \rightarrow \Sigma'$, is a pair $\sigma = (\sigma_S, \sigma_\Gamma)$ of functions $\sigma_S : S \rightarrow S'$ and $\sigma_\Gamma : \Gamma \rightarrow \Gamma'$, such that for each operation $f : s_1 \times \ldots \times s_n \rightarrow s \in \Gamma$ holds that $\sigma_\Gamma(f) : \sigma_S(s_1) \times \ldots \times \sigma_S(s_n) \rightarrow \sigma_S(s)$ is in Γ'. The **domain** of σ, $\mathsf{dom}(\sigma)$ for short, is defined as the set $\{x \in \Sigma \mid \sigma(x) \neq x\}$. The **set of all signature morphisms** is denoted by $SIGMORPH$.

Let SP be a specification, and let σ be a bijective signature morphism. The specification-building operation $rename\ \_\ by\ \_ : SPEC \times SIGMORPH \rightarrow SPEC$ is called **renaming** and is defined as follows:

$$sig(rename\ SP\ by\ \sigma) := \sigma(sig(SP))$$
$$Mod(rename\ SP\ by\ \sigma) := \{\mathcal{A} \in Alg(\Sigma_{rename\ SP\ by\ \sigma}) \mid \mathcal{A}|_{\sigma} \in Mod(SP)\}$$

We write $SP_{[x_1/y_1,\ldots,x_n/y_n]}$ as an abbreviation for $rename\ SP\ by\ \sigma$, if x_i is mapped to y_i by σ, $i = 1, \ldots, n$; then, $\mathsf{dom}(\sigma) = \{x_1, \ldots, x_n\}$.

Let SP and SP' be two specifications, such that all pairwise equal operation symbols have the same characteristics. The specification-building operation $import\ \_\ into\ \_ : SPEC \times SPEC \rightarrow SPEC$ is defined as follows:

$$sig(import\ SP\ into\ SP') := sig(SP) \cup sig(SP')$$
$$Mod(import\ SP\ into\ SP') := \{\mathcal{A} \in Alg(\Sigma_{import\ SP\ into\ SP'}) \mid \mathcal{A}|_{\Sigma_{SP}} \in Mod(SP)\}$$

We write $import\ SP_1, \ldots, SP_n\ into\ SP$ as an abbreviation for

$$import\ SP_1\ into\ (\ldots(import\ SP_n\ into\ SP)\ldots)$$

We further assume a specification NAT for data type `integer` with a sort nat, and a specification $BOOL$ for data type `bool` with a sort $bool$ together with the usual constants $true$ and $false$ to be given in each specification. Similarly, we assume a ternary operation $\mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ \_ :$ $bool \times s \times s \to s$ to be given for each sort $s \in S$ with the usual semantics:

$$\mathsf{if}\ true\ \mathsf{then}\ x\ \mathsf{else}\ y =_s x \qquad\qquad \mathsf{if}\ false\ \mathsf{then}\ x\ \mathsf{else}\ y =_s y.$$

A standard definition for $BOOL$ and NAT is for example given in [56], on page 699 and 700, respectively.

4.2 Semantics of Simple Classes

Each *DoDL*-class is transformed into a (structured) algebraic specification. Since simple classes are defined through a frame encompassing the class designator and optional sections declaring attributes and defining methods, we can state the following transformation and interpretation rules.

4.2.1 Transformation of Simple Classes

TRANSFORMATION RULE 1 A simple *DoDL*-class is transformed into a flat algebraic specification carrying the name of the class in capital letters.

In the sequel, let `class` be a simple *DoDL*-class and $CLASS$ its transformation. Attributes can be understood as methods with arity zero (c.f. [34]). Hence, we transform each attribute declaration into a suitable operation within a flat specification:

TRANSFORMATION RULE 2

1. An attribute declaration in `class` of the form `attrID: attrType` is transformed into an operation in $opns(CLASS)$ of the form $attrID : class \to attrType$.
2. An attribute declaration in `class` of the form `attrID: `**`list of`**` attrType` is transformed into an operation in $opns(CLASS)$ of the form $attrID : class \to list(attrType)$.
3. The identifiers $class$ and $attrType$ are added to $sorts(CLASS)$.

Remarks:

1. Equally, one could prefer an object-oriented style of notation and write $\_.attrID :$ $class \ldots$. This style produces terms of the form $c.attr$, whereas the one we prefer requests to write $attr(c)$ to refer to an attribute $attr$ of an instance c.
2. We assume that a specification for lists is available as shown, for example, in [42].

Method signatures contained in the **`construct`**-section of a class are transformed as follows:

TRANSFORMATION RULE 3

1. An n-ary method declaration in `class` of the form `methType methID(parType`$_1$` parID`$_1$`, ..., parType`$_n$` parID`$_n$`)` is transformed into a $(n+1)$-ary operation in $opns(CLASS)$ of the form $methID :$ $class \times parType_1 \times \ldots \times parType_n \to methType$.
2. A unary method declaration in `class` of the form `methType methID(void)` is transformed in to a unary operation in $opns(CLASS)$ of the form $methID :$ $class \to methType$.
3. The identifiers $class$, $methType$, $parType_i$, $i = 1, \ldots, n$ and $void$ are added to $sorts(CLASS)$.

Transformation rules for method bodies are discussed in Sect. 4.7. For the time being, axioms are created which give meaning to operations.

In case of aggregation, an attribute has a self-defined type, say, `t`, that is, an instance of class `t` is declared. Both aggregate and element classes can be transformed independently since cyclic aggregation is not allowed. Nonetheless, transformed attributes and methods of the element class have to be accessed in the transformed aggregate. We thus obtain a tree-structure over classes modeling the aggregation relationship. The root of this tree is an aggregate, the inner nodes are element classes which may themselves be aggregates. This yields the following order of transformation:

TRANSFORMATION RULE 4 The tree-structure of the aggregation relationship is transformed in post-order. The transformation of each element class is imported into *CLASS*.

A simple example may illustrate how the transformation rules proposed so far work.

EXAMPLE 1 Listing 8 shows two classes, `grower` and `sponger`. The latter declares an instance, `grw`, of class `grower`.

LISTING 8 Two simple classes in *DoDL*

```
class grower is
   attributes num: nat;     // a natural number num
   construct
      nat get(void){
         return num;        // get value of num
      }
      grower set(nat val){
         num = val;         // set num to val
      }
      grower grow(void){
         num = num + 1;     // increase num by 1
      }
end grower;

class sponger is
   attributes grw: grower; // an agggregation
   construct
      nat sponge(nat val){
         grw.set(val);      // set num to val
         grw.grow();        // invoke method of element class
         return grw.get(); // return num
      }
      nat main(void){
         print(this.sponge(7)); // print a value
      }
end sponger;
```

The methods defined are easy to understand and need no further explanation. The main method of class `sponger` calls method `sponge` by value, here 7. It is easy to see (an intuitive semantics presupposed) that the resulting value is 8.

We apply the transformation rules on class `grower` first, since this class is an element of class `sponger` (rule 4). By rule 1, we yield the flat specification shown in specification 1 on page 12. Sort *grower* is introduced following rule 2. The same rule is responsible for transforming the attribute declaration into an operation $num : grower \rightarrow nat$. The signatures of methods `grow`, `set` and `get` are transformed by rule 3. The axioms reflect the semantics of these operations. For the time being, they are defined "by hand".

SPECIFICATION 1 An explanatory element specification

$GROWER =$	
sorts	$grower$
opns	$num : grower \rightarrow nat,$
	$get : grower \rightarrow nat,$
	$set : grower \times nat \rightarrow grower,$
	$grow : grower \rightarrow grower$
vars	$g : grower, v : nat$
axms	$num(g) = get(g),$
	$get(set(g, v)) = v,$
	$grow(g) = set(g, num(g) + 1)$

Transforming class `sponger` results in specification 2. The **import**-clause is introduced by rule 4. Thereby, sorts and operations of *GROWER* are made available in *SPONGER*.

SPECIFICATION 2 An explanatory aggregate specification

$SPONGER =$ **import** $GROWER$ **into**	
sorts	$sponger$
opns	$grw : sponger \rightarrow grower,$
	$sponge : sponger \times nat \rightarrow nat,$
	$main : sponger \rightarrow nat$
vars	$s : sponger, v : nat$
axms	$sponge(s, v) = get(grow(set(grw(s), v))),$
	$main(s) = sponge(s, 7)$

■

4.2.2 Interpretation of Simple Classes

The specifications obtained by the above transformation rules are interpreted by algebras serving as a mathematical model for *DoDL*-classes. We are not interested in initial or terminal models and use a loose semantics approach. Moreover, a loose semantics allows to leave some carrier-sets uninterpreted, especially those used for technical reasons only; in the above example, sorts *grower* and *sponger* need no specific interpretation. We define the loose semantics for a simple *DoDL*-class as follows:

DEFINITION 1 The **loose semantics** of a simple *DoDL*-class `c`, $[\![\texttt{c}]\!]$ for short, is defined as the set of all models of its transformation, C. We write $[\![\texttt{c}]\!] := Mod(C)$ and assume that $[\![\_]\!]$ is a mapping from the set of all syntactically correct *DoDL*-classes into the set of all model sets.

Two *DoDL*-classes can have the same semantics though they are syntactically different:

DEFINITION 2 Two *DoDL*-classes, `class` and `class'`, are **semantically equal**, $[\![\texttt{class}]\!] = [\![\texttt{class'}]\!]$ for short, if their transformations, $CLASS$ and $CLASS'$, have the same model sets, i.e. if $Mod(CLASS) = Mod(CLASS')$ holds.

If classes are syntactically equal, they are identical and thus have the same semantics. Nonetheless, renaming of identifiers must be respected:

DEFINITION 3 Let `class` and `class'` be two *DoDL*-classes, and let $CLASS$ and $CLASS'$ be their transformations. `class` and `class'` are **equal**, `class = class'` for short, if there exists a bijective signature morphism, σ, such that the following properties hold:

1. $sig(rename\ CLASS\ by\ \sigma) = sig(CLASS')$
2. $Mod(CLASS) = Mod(CLASS')$

With each algebraic specification a set of models is associated. That is, we can obtain "standard" interpretations which satisfy each axiom given in a specification. To establish a loose semantics reflecting the object-oriented paradigm, however, we define a set of interpretation rules and obtain a specific set of models for structured specifications. As an invariant, we can usually employ the axioms themselves to interpret operations. We show this in Example 2. Other interpretation rules need to be defined explicitly. In the sequel, let `class` be a *DoDL*-class, $CLASS$ its transformation, and let $\mathcal{C}$ be a Σ_{CLASS}-algebra.

The sorts obtained by class name transformation are loosely interpreted:

INTERPRETATION RULE 1 Each sort $s \in sorts(CLASS) \setminus \{nat, string, bool, void\}$ is interpreted arbitrarily in $\mathcal{C}$.

The type `void`, however, requires a specific interpretation for all algebras. We use the value ϵ as a reserved value:

INTERPRETATION RULE 2 For each Σ_{CLASS}-algebra $\mathcal{C}$, let $C_{void} = \{\epsilon\}$.

The other predefined types, i.e. `nat`, `string`, and `bool` are interpreted in the usual way, that is, for each Σ_{CLASS}-algebra $\mathcal{C}$ we define $C_{nat} = \mathbb{N}_0$, $C_{bool} = \{true, false\}$, and C_{string} as the free semi-group over letters and digits with concatenation.

As long as bindings are not respected, we can freely interpret transformed attribute declarations:

INTERPRETATION RULE 3 Each operation in $opns(CLASS)$ of the form $f : class \rightarrow s$, where s is a sort in $sorts(CLASS)$ and `f: s` is an attribute declaration in $class$, is interpreted arbitrarily in $\mathcal{C}$.

In case of aggregation, the aggregate class cannot change the semantics of element methods. Hence, each aggregate algebra has to interpret each sort and operation of its elements as in the respective element algebra. Let $AGGR$ be a transformed class aggregating an element class the transformation of which is denoted by $ELEM$. Due to transformation rule 4, $ELEM$ is imported into $AGGR$. We assume that $\mathcal{E}$ is a Σ_{ELEM}-algebra following the above interpretation rules. Let $\mathcal{A}$ be a Σ_{AGGR}-algebra.

INTERPRETATION RULE 4

1. Each operation in $opns(ELEM)$ is interpreted in $\mathcal{A}$ as in $\mathcal{E}$:

$$\forall f : s_1 \times \ldots \times s_n \rightarrow s \in opns(AGGR) \cap opns(ELEM), a \in E_{s1} \times \ldots \times E_{sn} :$$
$$f^{\mathcal{A}}(a) = f^{\mathcal{E}}(a)$$

2. It is required that $E_s \subseteq A_s$ holds for each $s \in sorts(ELEM) \cap sorts(AGGR)$.

Remarks: Note that both $opns(ELEM) \subseteq opns(AGGR)$ and $sorts(ELEM) \subseteq sorts(AGGR)$ hold due to the import of $ELEM$ into $AGGR$.

Moreover, the interpretation rules have to ensure that the semantics of the **import**-clause (for its definition, see Sect. 4.1) is preserved. This can be shown as follows.

OBSERVATION 1 Let $AGGR$ and $ELEM$ be the transformations of two *DoDL*-classes, `aggr` and `elem`, where `elem` is an element class in `aggr`. Let both specifications be obtained by the above transformation rules such that $ELEM$ is imported into $AGGR$. Let $\mathcal{A}$ and $\mathcal{E}$ be a Σ_{AGGR}-algebra and a Σ_{ELEM}-algebra, resp., obtained by the above interpretation rules. $\mathcal{A}$ and $\mathcal{E}$ are thus models for $AGGR$ and $ELEM$, resp. Then, $\mathcal{A}|_{\Sigma_{ELEM}}$ is a model for $ELEM$.

Proof. We have to show that for each $\mathcal{A} \in Mod(AGGR)$ it holds that $\mathcal{A}|_{ELEM} \in Mod(ELEM)$.

1. Following transformation rule 2, part 3, $sorts(ELEM)$ contains a sort $elem$. This sort is interpreted arbitrarily in $\mathcal{A}\,|_{\Sigma_{ELEM}}$ (interpretation rule 1). Since $ELEM$ is imported into $AGGR$, $elem$ is also contained in $sorts(AGGR)$ and is interpreted arbitrarily due to the same interpretation rule.
2. With interpretation rule 4, part 1, each operation in $opns(ELEM)$ is interpreted in $\mathcal{A}$ as required for $\mathcal{A}|_{\Sigma_{ELEM}}$. Part 2 provides the necessary values.
3. By assumption, $\mathcal{A}$ is a model for $AGGR$, and $\mathcal{E}$ is a model for $ELEM$.

Thus, the Σ_{ELEM}-reduct of $\mathcal{A}$ is a model of $ELEM$. This is the observation. ■

Concluding this subsection, we refer to Example 1 and show how algebras are obtained by the above set of interpretation rules.

EXAMPLE 2 For better readability, we expand the **import**-clause of specification $SPONGER$ (see Spec. 2 on page 12), and obtain the following specification:

$SPONGER\_EXPANDED =$

sorts	$sponger, grower$
opns	$num : grower \rightarrow nat,$
	$get : grower \rightarrow nat,$
	$set : grower \times nat \rightarrow grower,$
	$grow : grower \rightarrow grower,$
	$grw : sponger \rightarrow grower,$
	$sponge : sponger \times nat \rightarrow nat,$
	$main : sponger \rightarrow nat$
vars	$g : grower, s : sponger, v : nat$
axms	$num(g) = get(g),$
	$get(set(g, v)) = v,$
	$grow(g) = set(g, num(g) + 1),$
	$sponge(s, v) = get(grow(set(grw(s), v))),$
	$main(s) = sponge(s, 7)$

Let $\mathcal{G}$ be a Σ_{GROWER}-algebra. Due to interpretation rule 1, sort $grower$ is interpreted arbitrarily. Due to interpretation rule 3, the operations num, $grow$, get and set are interpreted as follows for all $g, g' \in G_{grower}$ and for all $n \in G_{nat}$:

$$num^{\mathcal{G}}(g) = get^{\mathcal{G}}(g) \tag{1}$$

$$set^{\mathcal{G}}(g, n) = set(g, n) \tag{2}$$

$$get^{\mathcal{G}}(set(g, n)) = n, \tag{3}$$

$$grow^{\mathcal{G}}(g) = set^{\mathcal{G}}(g, num^{\mathcal{G}}(g) + 1) \tag{4}$$

It is easy to prove that $\mathcal{G}$ is a model for $GROWER$.

Let $\mathcal{S}$ be a $\Sigma_{SPONGER}$-algebra. Interpretation rule 1 is responsible for arbitrarily interpreting sort $sponger$. With interpretation rule 3, grw can also be interpreted freely. Operations $sponge$ and $main$ are interpreted as follows for all $s \in S_{sponger}$ and for all $n \in S_{nat}$:

$$sponge^{\mathcal{S}}(s, n) = get^{\mathcal{S}}(grow^{\mathcal{S}}(set^{\mathcal{S}}(grw^{\mathcal{S}}(s), n))) \tag{5}$$

$$main^{\mathcal{S}}(s) = sponge^{\mathcal{S}}(s, 7) \tag{6}$$

Following interpretation rule 4, operations imported from $GROWER$ are interpreted as in $\mathcal{G}$. Further, carrier-sets of sorts in $sorts(SPONGER) \cap sorts(GROWER)$ are joined. For all $s \in S_{sponger}$, $g \in S_{grower}$, and $n \in S_{nat}$ holds:

$$G_{grower} \subseteq S_{grower} \tag{7}$$

$$num^{\mathcal{S}}(g) = num^{\mathcal{G}}(g) \tag{8}$$

$$get^{\mathcal{S}}(g) = get^{\mathcal{G}}(g) \tag{9}$$

$$set^{\mathcal{S}}(g, n) = set^{\mathcal{G}}(g, n) \tag{10}$$

$$grow^{\mathcal{S}}(g) = grow^{\mathcal{G}}(g) \tag{11}$$

Again, it is easy to see that $\mathcal{S}$ is a model for $SPONGER$. By Observation 1 it holds that $\mathcal{S}|_{\Sigma_{GROWER}}$ is a model for $GROWER$.
■

4.3 Semantics of Local Classes

Local classes allow to restrict the visibility of classes by introducing a new scope. To model this property adequately, we use algebraic specifications with hidden symbols.

4.3.1 Transformation of Local Classes

DEFINITION 4 Let $\Sigma = \langle S, \Gamma \rangle$ be a signature. A **specification with hidden symbols** is a triplet $SP_h = \langle \Sigma, H\Sigma, E \rangle$ with

- a **visible signature** Σ,
- an **extended signature** $H\Sigma = \langle HS, H\Gamma \rangle$ with $\Sigma \subseteq H\Sigma$, and
- a **set of axioms** $E \subseteq WFF(H\Sigma)$ over the set of all well-formed formulas over Σ.

Let $sig(SP_h) = \Sigma$. The elements in $HS \backslash S$ are called **hidden sorts**, the elements in $H\Gamma \backslash \Gamma$ are called **hidden functions**. We write **export** Σ **from** $\langle H\Sigma, E \rangle$ for $\langle \Sigma, H\Sigma, E \rangle$. The **loose semantics** $Mod(SP_h)$ of SP_h is defined as the set of all Σ-reducts of models of $\langle H\Sigma, E \rangle$:

$$Mod(SP_h) := \{\mathcal{A}|_{\Sigma} \in Alg(\Sigma) \mid \mathcal{A} \in Mod(\langle H\Sigma, E \rangle)\}$$

Local classes cannot refer to features, i.e. attributes and methods of their embedding class, and thus they can be transformed prior to this class. Again we obtain a tree-structure over classes modeling the locality relationship. The root of this tree is an embedding class, the inner nodes are local classes which may themselves contain local classes. The embedding class, however, can make use of their local classes, but those are not visible outside this class. Transformation rules have to ensure that the features of a local class are denoted by hidden operations. Operations of the embedding class using locally defined classes or methods have to be hidden as well. This is captured by the visible signature. Local classes can be complex as well. This is discussed in Sect. 4.4. Whatever local classes look like, we can assume them to be already properly transformed.

TRANSFORMATION RULE 5 1. The tree-structure of the locality relationship is transformed in post-order.

2. A *DoDL*-class containing local classes is transformed into an algebraic specification with hidden symbols.

In the sequel, let `class` be a *DoDL*-class containing local classes, and let $CLASS = \langle \Sigma, H\Sigma, E \rangle$ be its transformation.

TRANSFORMATION RULE 6 Transformation rules 2, 3, and 4 hold analogously respecting $H\Sigma$ instead of $CLASS$. That is, this rule replaces $sorts(CLASS)$ by $sorts(H\Sigma)$, and $opns(CLASS)$ by $opns(H\Sigma)$.

Remarks: Note that now $attrType$ can be a local class designator, and that $methType$ as well as $parType$ can be identifiers used in local classes. Only those operations in $H\Sigma$ are in Σ which are accessible from outside the embedding class. Accessible are all those operations the signatures of which do not contain identifiers defined in a local class. The treatment of aggregation carries over to classes with local classes.

Each flat algebraic specification $\langle \Sigma, E \rangle$ can be denoted by a specification with hidden symbols where $H\Sigma = \Sigma$. Hence, we have to consider which sorts and symbols of an embedding class have to be hidden.

TRANSFORMATION RULE 7 In $CLASS = \langle H\Sigma, \Sigma, E \rangle$, Σ is the smallest signature with:

1. Each sort $s \in sorts(H\Sigma)$ is in $sorts(\Sigma)$, if and only if s is not a local class designator in $CLASS$.
2. Each operation $f : s_1 \times \ldots \times s_n \to s \in opns(H\Sigma)$ is in $opns(\Sigma)$, if and only if each s_i, $i = 1, \ldots, n$, and s are in $sorts(\Sigma)$.

Remarks: Note that local class designators are not contained in $sorts(\Sigma)$ and are thus hidden. Instances of local class as well as operations using at least one local class designator are hidden. Other operations are visible.

Introducing **`public`** or **`private`** qualifiers in *DoDL* can easily be respected by changing the above transformation rule: The **`private`** qualifier enforces an operation to occur in the hidden signature.

We explain the transformation of classes with local classes.

EXAMPLE 3 With reference to Example 1, Listing 9 on page 17 shows classes `grower` and `sponger` again, now with `grower` declared locally in `sponger`.

SPECIFICATION 3 An explanatory local class specification

$SPONGER =$ **export** $sponger, sponge, main$ **from**
import $GROWER$ **into**
sorts $sponger$
opns $grw : sponger \to grower,$
$sponge : sponger \times nat \to nat,$
$main : sponger \to nat$
vars $s : sponger, v : nat$
axms $sponge(s, v) = get(grow(set(grw(s), v))),$
$main(s) = sponge(s, 7)$

LISTING 9 A local class definition in *DoDL*

```
class sponger is
   declare
      class grower is                // a local class definition
         attributes num: nat;
         construct
            nat get(void){ return num; }
            grower set(nat val){ num = val; }
            grower grow(void){ num = num + 1; }
      end grower;

   attributes grw: grower;           // a composition
   construct
      nat sponge(nat val){
         grw.set(val);
         grw.grow();
         return grw.get();
      }
      nat main(void){ print(this.sponge(7)); }
end sponger;
```

Following transformation rule 5, part 1, the local class has to be transformed first. Since `grower` is a simple class, we follow transformation rule 1, and take over its formalization from Example 1. Then, we follow transformation rule 5, part 2, and import $GROWER$ into $SPONGER$ (see Spec. 3 on page 16). Transforming the attributes and methods of class `sponger` is accomplished by rule 6.

Now we have to consider which parts of this specification are visible using transformation rule 7. Sort *sponger* is obviously not a local class designator and hence visible. Sort *grower*, which is in $sorts(SPONGER)$ due to the **import**-clause, is a local class designator and hence not visible. The operation symbols in $GROWER$ (see Spec. 1 on page 12) use the hidden sort *grower* and are thus hidden. The transformed attribute declaration `grw: grower` is hidden for the same reason; the operation symbols *sponge* and *main* are both visible for the opposite reason. The result is denoted by the **export**-clause abbreviating the visible signature Σ. It formalizes an interface to class *sponger* and allows to declare instances of class `sponger` as well as to invoke the methods `sponge` and `main`.
■

4.3.2 Interpretation of Local Classes

The interpretation rules given in the previous section carry over to specifications with hidden symbols, thereby respecting $H\Sigma$:

INTERPRETATION RULE 5

1. Interpretation rules 1 to 4 hold analogously respecting $H\Sigma$ instead of $CLASS$. That is, this rule replaces $sorts(CLASS)$ by $sorts(H\Sigma)$, and $opns(CLASS)$ by $opns(H\Sigma)$.
2. Aggregation and composition are treated equally. That is, interpretation rule 4 holds for using local classes in attribute declaration, thereby respecting $H\Sigma$ instead of $AGGR$.

Remarks: The case of local classes contained in local or element classes is covered by this rules. By Definition 4, the signature of a specification with hidden symbols is equal to its visible signature. Hence, only sorts and symbols visible in local or element classes are imported and thus need interpretation.

Corresponding to Observation 1, we have to show that the above interpretation rules respect the semantics given for specifications with hidden symbols (see Def. 4).

OBSERVATION 2 Let $EMBED = \langle \Sigma, H\Sigma, E \rangle$ be a transformed *DoDL*-class containing local classes, and let $LOCAL$ be the transformation of such a local class. Let both specifications be obtained by the above transformation rules. Let $\mathcal{A}$ be a Σ_{EMBED}-algebra obtained by the above interpretation rules. Then, $\mathcal{A} \mid_{\Sigma_{LOCAL}}$ is a model for $LOCAL$.

Proof. The proof follows immediately from observation 1, since local classes are imported into their embedding class. ■

4.4 Semantics of Inheritance

In the literature, two algebraic approaches to modelling inheritance can be found:

1. Inheritance is modelled by a relation between classes. It is understood as replicating all superclass operations within a subclass. Sorts are restricted by subsorts. This approach is studied, for example, in [11] and [45], and is called *inheritance by restriction* and *reusing inheritance*, respectively.
2. Inheritance is modelled by model-inclusion on algebras. This approach is studied in [11] and [45] as well, and is called *inheritance by specialization* and *specialization inheritance*, respectively.

We use hierarchical specifications to express inheritance in *DoDL*. Inheritance by specialization is proposed for hierarchical specifications [58].

As we discussed in Section 3.3, the type of a caller `c` of a method `m` determines the class `m` is chosen from. In case the compiler cannot resolve the correct method in a method invocation of the form `c.m()`, this is called *late binding* and requires a suitable lookup mechanism. Looking up the correct method in a class hierarchy starts at the class `C` the caller is declared an instance of, and follows the hierarchy via superclass definition. Lookup stops at a class `D` that defines `m`, and for which no other class `D'` exists in the hierarchy that redefines `m` and is defined between `D` and `C`. A very similar algorithm can be found, for example, in [35], covering local class hierarchies as well.

Since the semantics of **`this`** depends on the caller's type (see Sect. 3.3), **`this`** and **`super`** are not directly transformed. Moreover, we define suitable axioms and simulate a lookup mechanism on algebras.

4.4.1 Transformation of Subclasses

DEFINITION 5 Let $SP_A = \langle \Sigma, E \rangle$ be a specification, and let SP_B be a subspecification of SP_A. The pair $HS = \langle SP_A, SP_B \rangle$ is called **hierarchical specification**. SP_A is called **simple specification**, SP_B is called **primitive part** of HS. Let $sig(HS) = \Sigma$. The **loose semantics** $Mod(HS)$ of HS is defined as the set of all models of SP_A the Σ_{SP_B}-reduct of which is a model of SP_B:

$$Mod(HS) := \{\mathcal{A} \in Mod(SP_A) \mid \mathcal{A}|_{\Sigma_{SP_B}} \in Mod(SP_B)\}$$

Remarks: The notions of simple specification and primitive part are somewhat awkward, since what is called simple or primitive will turn out to be structured in our approach. Hence, we prefer to call these parts *subclass specification* and *superclass specification*, respectively.

Similar to local classes, we provide a law of transformation for a class and its superclass. The tree-structure over classes modeling the subclass relationship has a superclass as root and subclasses as inner nodes which may themselves be superclasses. Since any class can be used as superclass without being effected thereby, and due to the structure of hierarchical specifications, we define the following rule:

Transformation Rule 8 1. The tree-structure of the subclass relationship is transformed in pre-order.

2. A *DoDL*-class that is subclass to any other class is transformed into a hierarchical specification, where the superclass specification is given by the transformation of its superclass.

It remains to consider how to obtain the subclass specification. A superclass may be complex and therefore contain local classes. Then, the superclass is transformed into a specification with hidden symbols. Local classes are inherited by a subclass. Thus, the subclass must be represented by a specification with hidden symbols as well. Even if a superclass does not declare local classes, its subclasses may define them in their **`declare`**-sections. Then again, a subclass has to be transformed into a specification with hidden symbols. Only in case both a subclass and its superclass are simple (apart from inheritance), the transformation into a flat specification is sufficient. Nonetheless, each flat specification $\langle\Sigma, E\rangle$ can be denoted as a specification with hidden symbols of the form $\langle\Sigma, \Sigma, E\rangle$. We fix the following rule:

Transformation Rule 9 The subclass specification of a hierarchical specification is denoted as a specification with hidden symbols. Symbols and axioms of the superclass specification are replicated and supplemented by the subclass under consideration.

Remarks: The supplement is formed by those local classes, attributes and methods that are defined in the subclass, called the *increment of the superclass*.

Replication is defined next. Moreover, we take care of the **`super`**-construct. Up to now, transformed *DoDL*-classes occur in three different flavors: as flat specifications, as specifications with hidden symbols, and as hierarchical specifications. The replication transforms any of these structured specifications into a specification with hidden symbols. In the sequel, let `class` be *DoDL*-class and $CLASS$ its transformation.

Definition 6 The **replication** of $CLASS$, $\rho(CLASS)$ for short, is defined as follows:

$$\rho(CLASS) := \begin{cases} \langle\Sigma_{CLASS}, \Sigma_{CLASS}, E\rangle, & \text{if } CLASS \text{ is a flat specification of the form } \langle\Sigma_{CLASS}, E\rangle \\ CLASS, & \text{if } CLASS \text{ is a specification with hidden symols} \\ CLASS', & \text{if } CLASS \text{ is a hierarchical specification of the form } (CLASS', C). \end{cases}$$

Remarks: The replication of a specification with hidden symbols is the identity, whereas the replication of a hierarchical specification considers the subclass specification only: the replication of the superclass specification is already contained in this part.

Henceforth, it is sufficient to consider the increment of a class when a transformation is carried out. This leads to transformation rules similar to those defined in the previous sections. For clarity, however, we define some of them explicitly. In the sequel, let `subclass` be a subclass of `class`, `subclass` $\sqsubseteq$ `class` for short, and let $SUBCLASS$ be its transformation. The replication of $CLASS$ is denoted by $\rho(CLASS) = \langle\Sigma, H\Sigma, E\rangle$.

The increment of `class`, i.e. attribute declarations and methods defined in `subclass`, is treated as usual:

Transformation Rule 10 Transformation rules 2 and 3 hold analogously respecting $H\Sigma$ in $\rho(CLASS)$ instead of $CLASS$. That is, this rule replaces $sorts(CLASS)$ by $sorts(H\Sigma)$, and $opns(CLASS)$ by $opns(H\Sigma)$.

The signatures of inherited methods have to be adapted to the subclass in order to make these methods applicable to subclass instances. In particular, the method type is changed from `class` to `subclass`.

TRANSFORMATION RULE 11

1. A n-ary method declaration in `class` of the form `class methID(parType`$_1$ `parID`$_1$`, ..., parType`$_n$ `parID`$_n$`)` is transformed into a $(n+1)$-ary operation in $opns(H\Sigma)$ of the form $methID : subClass \times parType_1 \times \ldots \times parType_n \to subclass$.
2. Any other n-ary method declaration in `class` of the form `methType methID(parType`$_1$ `parID`$_1$`, ..., parType`$_n$ `parID`$_n$`)` with `methType` $\neq$ `class` is transformed into a $(n+1)$-ary operation in $opns(H\Sigma)$ of the form $methID : subClass \times parType_1 \times \ldots \times parType_n \to methType$.
3. A unary method declaration in `class` of the form `class methID(void)`or `methType methID(void)` is transformed into a unary operation in $opns(H\Sigma)$ of the form $methID : subClass \to subclass$ or $methID : subClass \to methType$, respectively.
4. The identifiers $subClass$, $methType$, $parType_i$, $i = 1, \ldots, n$ and $void$ are added to $sorts(H\Sigma)$.

We characterize polymorphism through substitution, and therefore each instance of a class must be an instance of its superclass. This leads to two operations which are vital for algebraic transformation (c.f. [10], Def. 3.83 on pages 94, 95): the usual injection $in : subClass \to class$, defining substitution, and the usual partial projection $cast : class \to subClass$ together with the projection axiom $cast(in(s)) = s$ for each $s \in subClass$, defining the semantics of **`super`**:

TRANSFORMATION RULE 12 The operations

$$in : subClass \to class \qquad \text{and} \qquad cast : class \to subClass$$

are added to $opns(H\Sigma)$. In E, an axiom of the form $\forall s : SubClass\ .\ cast(in(s)) = s$ is required.

If a method is redefined, suitable axioms have to be given (see Sect. 4.7). If a non-redefined method `m` in `subclass` uses a method redefined in `subclass`, we call `m` **indirectly redefined**, and hence suitable axioms have to be given here as well. If a method is inherited and not (indirectly) redefined, its semantics carries over to the subclass. We distinguish between different method types as in rule 11:

TRANSFORMATION RULE 13

1. For each non-redefined operation in $opns(H\Sigma)$ of the form $methID : subClass \times parType_1 \times \ldots \times parType_n \to subClass$, which does not use a method redefined in $subClass$, there is defined an axiom in E of the form

$$\begin{aligned}\forall s :& subClass, p_1 : parType_1, \ldots, p_n : parType_n\ .\\ & methID(s, p_1, \ldots, p_n) = cast(methID(in(s), p_1, \ldots, p_n))\end{aligned}$$

2. For each non-redefined operation in $opns(H\Sigma)$ of the form $methID : subClass \times parType_1 \times \ldots \times parType_n \to methType$, which does not use a method redefined in $subClass$, there is defined an axiom in E of the form

$$\begin{aligned}\forall s :& subClass, p_1 : parType_1, \ldots, p_n : parType_n\ .\\ & methID(s, p_1, \ldots, p_n) = methID(in(s), p_1, \ldots, p_n)\end{aligned}$$

Remarks: The term $methID(s, p_1, \ldots, p_n)$ refers to the operation $methID : subClass \times \ldots$, whereas $methID(in(s), p_1, \ldots, p_n)$ refers to $methID : class \times \ldots$.

The visible signature Σ can be defined in analogy to local class transformation:

TRANSFORMATION RULE 14 Transformation rule 7 holds for $\rho(CLASS)$ and $subClass$ analogously.

In particular, this rule adds the operations in and $cast$ to the visible signature. It is furthermore easy to see that these rules establish a well-defined hierarchical specification, where the superclass specification is a subspecification of the subclass specification. Hence, the transformation of an inheriting class is anti-monotonous w.r.t. inclusion.

EXAMPLE 4 We refer to the previous example and introduce class `sponger` as a subclass of `grower`, shown in Listing 10. The transformation of a class inheriting from a class with local classes works analogously and is omitted here. Such an example can be found in [22].

LISTING 10 A subclass definition

```
class grower is
   attributes num: nat;
   construct
      nat get(void){ return num; }
      grower set(nat val){ num = val; }
      grower grow(void){ num = num + 1; }
end grower;

class sponger is grower with      // a subclass definition
   attributes value: nat;         // an additional attribute
   construct
      nat getV(void){             // a new method
         return value;
      }
      sponger set(nat val_1, nat val_2){ //an overloaded method
         super.set(val_1);        // call super.set()
         value = val_2;
      }
      sponger grow(void){         // a redefined method
         num = num + value;
      }
      nat sponge(nat val_1, nat val_2){
         this.set(val_1, val_2);
         super.grow();            // add 1 to num
         this.grow();             // use redefinition
         return this.get();
      }
      nat main(void){ print(this.sponge(7, 8)); }
end sponger;
```

Class `sponger` introduces an attribute `value` and a method `getV`. Method `set` is overloaded since its signature has changed, and uses method `set` from class `grower`. Method `grow` is redefined. Methods `sponge` and **`main`** are adapted to the new situation.

The transformation of `sponger` yields a hierarchical specification of the form $SPONGER = (SPONGER', GROWER)$. With transformation rule 8, part 2, the superclass specification is as given in specification 1 on page 12. Transformation rule 9 prepares the subclass specification for taking up the superclass replication. By Definition 6 and since $GROWER$ is a flat specification, this replication is a specification of the form $\langle \Sigma_{GROWER}, \Sigma_{GROWER}, axms(GROWER)\rangle$. The subclass specification $SPONGER'$ is shown in specification 4 on page 22 and explained in the remainder of this example.

Transformation rule 11 rewrites the operations in *GROWER* for usage with sort *sponger*. Rule 10 takes care for the increment of class `grower`, i.e. the features defined in `sponger`, as usual. The injection and projection together with the required axiom are added by rule 12. The attribute declaration `num` and method `get` are not redefined, and thus rule 13.2 fixes the required axioms. The method `set` in class `grower` has method type `grower` and was rewritten by rule 11. Therefore, casting applies and is fixed by rule 13.1. The methods introduced in class `sponger` need new axioms. Their forms correspond to those given in Example 1.

SPECIFICATION 4 An explanatory subclass specification

$SPONGER' =$ (by replication with $H\Sigma = \Sigma$)

export $grower, num, get, set, grow, sponger, value, getV, sponge, main, in, cast$ **from**

sorts $grower, sponger$ (by rule 10)

opns
$num : grower \to nat,$ (by replication)
$get : grower \to nat,$ (by replication)
$set : grower \times nat \to grower,$ (by replication)
$grow : grower \to grower,$ (by replication)

$num : sponger \to nat,$ (by rule 11)
$get : sponger \to nat,$ (by rule 11)
$set : sponger \times nat \to sponger,$ (by rule 11)
$grow : sponger \to sponger,$ (by rule 11)

$value : sponger \to nat,$ (by rule 10)
$getV : sponger \to nat,$ (by rule 10)
$set : sponger \times nat \times nat \to sponger,$ (by rule 10)
$sponge : sponger \times nat \times nat \to nat,$ (by rule 10)
$main : sponger \to nat,$ (by rule 10)

$in : sponger \to grower,$ (by rule 12)
$cast : grower \to sponger$ (by rule 12)

vars $s : sponger, g : grower, n, n_1, n_2 : nat$

axms
$get(g) = num(g),$ (by replication)
$get(set(g, v)) = v,$ (by replication)
$grow(g) = set(g, num(g) + 1),$ (by replication)

$num(s) = num(in(s)),$ (by rule 13.2)
$get(s) = get(in(s)),$ (by rule 13.2)
$set(s, n) = cast(set(in(s), n)),$ (by rule 13.1)

$value(s) = getV(s),$
$getV(set(s, n_1, n_2)) = n_2,$
$grow(s) = set(s, num(s) + value(s), value(s)),$
$sponge(s, n_1, n_2) = get(grow(cast(grow(in(set(s, n_1, n_2)))))),$
$main(s) = sponge(s, 7, 8),$

$cast(in(s)) = s$ (by rule 12)

Method `sponge` needs explanation. The sequence of method invocations can be rewritten as `(((this.set(val_1, val_2)).super.grow()).grow()).get()`. This expression is rewritten as a term by writing it down vice versa: $get(grow(grow(set(n_1, n_2))))$. To achieve the correct sorts and to reach the correct super-methods, we use injection and projection and yield the given axiom.

■

Local classes may also use inheritance. Local class hierarchies are transformed as any other hierarchies and hence need no further rules. Nonetheless, we discuss them briefly for completeness. Scoping rules allow for three different kinds of using local classes:

1. Local classes may use other local classes regarding *delcaration before use*.
2. Local classes may inherit from classes defined in the scope of their embedding class.
3. Local classes may inherit from local classes defined in the superclass of their embedding class.

With transformation rules 5 and 8, parts 1, it holds that local classes are to be transformed prior to their embedding class, and that superclasses are to be transformed prior to a subclass. The transformation of a class declaring a local class that inherits in one of the above mentioned ways embraces the transformation of this "local" superclass which is done prior to the local class transformation. This yields a hierarchical specification replicated as shown above. The embedding class is transformed by the rules for local classes and imports the subclass specification of its transformed local classes.

Moreover, the embedding class may itself inherit from another class. Following rules 5 and 8, parts 2, this class is transformed into a hierarchical specification and into a specification with hidden symbols simultaneously. This conflict is solved regarding transformation rule 9 as follows. A subclass is transformed into a hierarchical specification the superclass specification of which is the transformed superclass. The subclass specification, however, is a replication of this specification, and in any case a specification with hidden symbols. The features of local classes can be added to the replication without any problem. The transformation of a subclass is thus prepared for local classes *per definitionem*, and can therefore be understood as the structurally stronger transformation.

4.4.2 Interpretation of Subclasses

In the sequel, let `inh` be a *DoDL*-class inheriting from a class `root`. The transformations are $INH = (INH', ROOT)$ and $ROOT$, respectively. Further, let INH' be the replication $(\Sigma, H\Sigma, E)$ of $ROOT$ supplemented with `inh`, and let $\mathcal{A}$ be a $H\Sigma$-algebra, $\mathcal{R}$ a Σ_{ROOT}-algebra.

INTERPRETATION RULE 6 Interpretation rules 1 to 4 hold analogously respecting $H\Sigma$ instead of $CLASS$. That is, this rule replaces $sorts(CLASS)$ by $sorts(H\Sigma)$, and $opns(CLASS)$ by $opns(H\Sigma)$.

For clarity, we rewrite the interpretation of aggregated class methods:

INTERPRETATION RULE 7 Let `elem` be an element class in `inh`, $ELEM$ its transformation, and $\mathcal{E}$ a Σ_{ELEM}-algebra. Further, let M denote the set of all operations defined in an element class of `inh`, i.e. $M = (opns(H\Sigma) \setminus opns(ROOT)) \cap opns(ELEM)$.

1. Each operation f in $opns(ELEM)$ is interpreted in $\mathcal{A}$ as in $\mathcal{E}$:

$$\begin{aligned}&\forall f : s_1 \times \ldots \times s_n \to s \in M, a \in E_{s1} \times \ldots \times E_{sn} :\\ &\quad f^{\mathcal{A}}(a) = f^{\mathcal{E}}(a)\end{aligned}$$

2. It is required that $E_s \subseteq A_s$ holds for each $s \in sorts(ELEM)$.

Additionally, we need interpretation rules for injection and projection.

INTERPRETATION RULE 8 For the carrier-sets A_{inh} and A_{root}, $A_{inh} \subseteq A_{root}$ holds.

Remarks: inh is interpreted as a subsort of $root$ (see Sect. 4.4.3).

INTERPRETATION RULE 9 For the injection and projection operations we define:

1. $\forall i \in A_{inh} : in^{\mathcal{A}}(i) = i$
2. $cast^{\mathcal{A}}(i) = \begin{cases} i, & \text{if } i \in A_{inh} \\ \perp, & \text{else} \end{cases}$

Inherited methods have to be interpreted depending on whether they are (indirectly) redefined or non-redefined:

INTERPRETATION RULE 10

1. Each operation in $opns(H\Sigma)$ of the form $f : inh \times s_1 \times \ldots \times s_n \rightarrow inh$ that is (indirectly) redefined in INH is interpreted in $\mathcal{A}$ as follows:

$$\forall r \in A_{root} \setminus A_{inh}, a \in A_{s1} \times \ldots \times A_{sn} :$$
$$f^{\mathcal{A}}(r, a) = cast^{\mathcal{A}}(f^{\mathcal{R}}(r, a))$$

2. Each operation in $opns(H\Sigma)$ of the form $f : inh \times s_1 \times \ldots \times s_n \rightarrow s$ that is (indirectly) redefined in INH is interpreted in $\mathcal{A}$ as follows:

$$\forall r \in A_{root} \setminus A_{inh}, a \in A_{s1} \times \ldots \times A_{sn} :$$
$$f^{\mathcal{A}}(r, a) = f^{\mathcal{R}}(r, a)$$

3. For values in A_{inh}, f is interpreted following the respective axioms given.
4. Any non-redefined operation in $opns(H\Sigma)$ of the same form, for which there exists an axiom in E of the form $f(i, a) = cast(f(in(i), a))$, with $i \in \mathcal{A}_{inh}, a \in A_{s1} \times \ldots \times A_{sn}$, is interpreted in $\mathcal{A}$ as follows:

$$\forall i \in A_{inh}, a \in A_{s1} \times \ldots \times A_{sn} :$$
$$f^{\mathcal{A}}(i, a) = cast^{\mathcal{A}}(f^{\mathcal{R}}(in^{\mathcal{A}}(i), a))$$

5. Any non-redefined operation in $opns(H\Sigma)$ of the same form, for which there exists an axiom in E of the form $f(i, a) = f(in(i), a)$, with $i \in \mathcal{A}_{inh}, a \in A_{s1} \times \ldots \times A_{sn}$, is interpreted in $\mathcal{A}$ as follows:

$$\forall i \in A_{inh}, a \in A_{s1} \times \ldots \times A_{sn} :$$
$$f^{\mathcal{A}}(i, a) = f^{\mathcal{R}}(in^{\mathcal{A}}(i), a)$$

Remarks:

1. This rule establishes a lookup method as follows: Inherited methods are overloaded in $opns(H\Sigma)$. Thus, two versions of f exist with different characteristics: $f : inh \times \ldots$ and $f : root \times \ldots$. In case, f is not redefined, the interpretation of $f : inh \times \ldots$ cascades, i.e. it is "forwarded" to the interpretation of $f : root \times \ldots$ by the second part of the above rule. If it is redefined, new axioms are given in INH.
2. For multiple inheritance, these properties are formulated, for example, in [42], page 75.

In analogy to the previous sections, these interpretation rules have to maintain the semantics for hierarchical specifications. The following observation is helpful to prove this proposition.

OBSERVATION 3 The semantics of a hierarchical specification, (SP_A, SP_B), is identical to the semantics of a specification established by means of the specification-building function $\mathit{import}\ SP_B\ \mathit{into}\ SP_A$.

Proof. Since $SP_B \subseteq SP_A$, and with the definition of *import* (see Sect. 4.1), it follows:

$$\begin{aligned} sig(import\ SP_B\ into\ SP_A) &= sig(SP_B) \cup sig(SP_A) \\ &= sig(SP_A) \end{aligned}$$

The equivalence of the model sets follows immediately with definition 5 and *import*, such that $SP = SP_B$ and $SP' = SP_A$. ■

Remarks: A similar result is given in [56], page 746.

OBSERVATION 4 Let $INH = (\langle \Sigma, H\Sigma, E\rangle, ROOT)$ be the transformation of `inh`, and let $\mathcal{A}$ be a model for $\langle \Sigma, H\Sigma, E\rangle$. Then, $\mathcal{A}|_{\Sigma_{ROOT}}$ is a model for $ROOT$.

Proof. With the assumption that $\mathcal{A}$ is a model for $\langle \Sigma, H\Sigma, E\rangle$, the model set of $\langle \Sigma, H\Sigma, E\rangle$ is the set of all Σ-reducts of $\mathcal{A}$ for which $\mathcal{A}$ is a model of $\langle H\Sigma, E\rangle$ (by Def. 4). Let $\mathcal{A}|_{\Sigma}$ be such a model.

The model set of *inh* is defined (by Def. 5) as the set of all models of $\langle \Sigma, H\Sigma, E\rangle$ the Σ_{ROOT}-reduct of which is a model of $ROOT$. To prove the assumption, it is sufficient to show:

1. $(\mathcal{A}|_{\Sigma})_{|\Sigma_{ROOT}}$ is a model for $ROOT$
2. $\mathcal{A}|_{\Sigma_{ROOT}} = (\mathcal{A}|_{\Sigma})_{|\Sigma_{ROOT}}$

Ad 1: The transformation rules yield $ROOT \subseteq \langle H\Sigma, E\rangle$. By Observation 3 and the *import*-operation, INH can be formulated equivalently as

$$INH := \textbf{import}\ ROOT\ \textbf{into}\ \langle \Sigma, H\Sigma, E\rangle$$

With Observation 1 on page 14 it follows immediately that $(\mathcal{A}|_{\Sigma})_{|\Sigma_{ROOT}}$ is a model for $ROOT$.

Ad 2: The transformation rules yield $\Sigma_{ROOT} \subseteq \Sigma_{INH}$. Since $\Sigma_{INH} = \Sigma$, is follows immediately that $\mathcal{A}|_{\Sigma_{ROOT}} = (\mathcal{A}|_{\Sigma_{INH}})_{|\Sigma_{ROOT}} = (\mathcal{A}|_{\Sigma})_{|\Sigma_{ROOT}}$. ■

We conclude this section by an example illustrating the interpretation rules.

EXAMPLE 5 Let $\mathcal{S}$ be a $\Sigma_{SPONGER}$-algebra, and $\mathcal{G}$ a Σ_{GROWER}-algebra. We interpret specification $SPONGER$ as given in Example 4. Specification $GROWER$ follows Example 2. There, we had for each $g \in G_{grower}, n \in G_{nat}$:

$$num^{\mathcal{G}}(g) = get^{\mathcal{G}}(g) \tag{1}$$
$$set^{\mathcal{G}}(g, n) = set(g, n) \tag{2}$$
$$get^{\mathcal{G}}(set(g, n)) = n, \tag{3}$$
$$grow^{\mathcal{G}}(g) = set^{\mathcal{G}}(g, num^{\mathcal{G}}(g) + 1) \tag{4}$$

For non-redefined methods, we follow rule 10, part 4 and part 5, yielding for each $s \in S_{sponger}, n \in S_{nat}$:

$$num^{\mathcal{S}}(s) = num^{\mathcal{G}}(in^{\mathcal{S}}(s)) \tag{12}$$
$$get^{\mathcal{S}}(s) = get^{\mathcal{G}}(in^{\mathcal{S}}(s)) \tag{13}$$
$$set^{\mathcal{S}}(s, n) = cast^{\mathcal{S}}(set^{\mathcal{G}}(in^{\mathcal{S}}(s), n)) \tag{14}$$

The redefined method *grow* is captured by rule 10, part 1, if $s \in S_{grower} \setminus S_{sponger}$, and by part 3, else:

$$grow^{\mathcal{S}}(s) = \begin{cases} cast^{\mathcal{S}}(grow^{\mathcal{G}}(s)), & \text{if } s \in S_{grower} \setminus S_{sponger} \\ set^{\mathcal{S}}(s, num^{\mathcal{S}}(s) + value^{\mathcal{S}}(s), value^{\mathcal{S}}(s)), & \text{else} \end{cases} \tag{15}$$

The increment of `grower`, i.e. the features of class `sponger`, is interpreted following the axioms in *SPONGER*. For each $s \in S_{sponger}$ and $n, n_1, n_2 \in S_{nat}$ we have:

$$value^{\mathcal{S}}(s) = getV^{\mathcal{S}}(s) \tag{16}$$
$$getV^{\mathcal{S}}(set(s, n_1, n_2)) = n_2 \tag{17}$$
$$set^{\mathcal{S}}(s, n_1, n_2) = set(s, n_1, n_2) \tag{18}$$
$$sponge^{\mathcal{S}}(s, n_1, n_2) = get^{\mathcal{S}}(grow^{\mathcal{S}}(grow^{\mathcal{S}}(in^{\mathcal{S}}(set^{\mathcal{S}}(s, n_1, n_2))))) \tag{19}$$
$$main^{\mathcal{S}}(s) = sponge^{\mathcal{S}}(s, 7, 8) \tag{20}$$

The interpretations of the *set* operations determine the structure of the carrier-sets for S_{grower} and $S_{sponger}$. The former contains terms of the form $set(\ldots(set(g, n_1), \ldots), n_l)$, the latter terms of the form $set(\ldots(set(s, n_1, n'_1), \ldots), n_l, n'_l)$, i.e. they are term-generated. Note that each value in $S_{sponger}$ is also in S_{grower} due to interpretation rule 8:

$$S_{sponger} \subseteq S_{grower} \tag{21}$$

Interpretation rule 9 fixes injection and projection. With the above structure of $S_{sponger}$ and S_{grower}, we can resolve the else-case in *cast* and obtain:

$$\forall s \in S_{sponger}, n_1, n_2 \in S_{nat} : \quad in^{\mathcal{S}}(set(s, n_1, n_2)) = set(s, n_1) \tag{22}$$
$$\forall s \in S_{sponger} : \quad cast^{\mathcal{S}}(s) = s \tag{23}$$
$$\forall s \in S_{sponger}, n_1, n_2, m \in S_{nat} : \quad cast^{\mathcal{S}}(set(in(set(s, n_1, n_2)), k)) = set(s, k, n_2) \tag{24}$$

It is again easy to prove that $\mathcal{S}$ is a model for *SPONGER*. Since $\mathcal{G}$ is a model for *GROWER* and with Observation 4, $\mathcal{A}|_{\Sigma_{GROWER}}$ is model for *GROWER*.
■

4.4.3 A Remark on Subsorts

We conclude Section 4.4 with a remark on subsorts. The transformation of a subclass, `inh`, yields a specification of the form $(INH', ROOT)$, where `root` is the superclass of `inh`, and INH' is as given by transformation rules. In particular, *inh* and *root* are sorts in $sorts(INH')$, and the interpretation of those sorts results in the subset relation given in rule 8. This expresses a relation on sorts alternatively used to model inheritance (c.f. [47, 33]). The language CASL [39], for example, is equipped with a formal semantics using subsorts. If algebraic specifications allow to express relations of the form $S_1 \leq S_2$ over sorts S_1 and S_2, (c.f. [18], Sect. 11.2, or [38]), and if they are respected in order-sorted algebras (c.f. [56], Sect. 3.3.4), unambiguous interpretations can be given without the need to explicitly fix interpretation rules. Nonetheless, this approach only works under certain restrictions (c.f. [33]), but with a subsort-property given by definition: for each Σ_{S_1}-algebra $\mathcal{A}$, $S_1 \leq S_2$ implies $A_{S_1} \subseteq A_{S_2}$ (c.f. [49], p. 137, or [51], Sect. 2.10.4). We simulate this approach with our interpretation rules, and thereby overcome the drawback of technical complexity (c.f. [10], Sect. 3.3.5) inevitably introduced into the structured specifications used here.

4.5 Semantics of Generic Classes

In [9], parametrization of classes is understood as *parametric polymorphism*. With polymorphism characterized by substitution (see Sect. 3.3), generic classes may be used in different situations uniformly by actualizing their formal parameter. In correspondence with *horizontal* and *vertical implementation* [19], substitution may be called vertical in the context of inheritance, and horizontal in the context of parametrization. Parametrical polymorphism allows for parallel development of generic classes and their parameters, and is thus orthogonal to inheritance. A detailed discussion can be found, for example, in [43].

For algebraic specifications, different approaches to parametrization exist. We refer to parameterized specifications as an extension of their parameter (c.f. [20], Sect. 7.1). An actual parameter is provided by *actualization* (c.f. [40], p. 172), thereby replacing the formal one. This approach sufficiently models the simple renaming mechanism required for generic *DoDL*-classes.

DEFINITION 7 Let SP and PAR be two algebraic specifications with $PAR \subseteq SP$.

A **parameterized specification** is a λ-expression of the form

$$PSP = \lambda X \ : \ PAR.SP$$

The specification PAR is called **formal parameter**, or **parameter specification**. Let $sig(PSP) = sig(SP)$.

Let ACT be a specification. An **actualization** of PSP through ACT, $PSP(ACT)$ for short, is defined as a specification of the form

$$PSP(ACT) := SP_{[PAR/ACT]}$$

The specification ACT is called **actual parameter** of PSP.

The **loose semantics** of $PSP(ACT)$, $Mod(PSP(ACT))$ for short, is defined as the set of all models of SP with PAR renamed by ACT:

$$Mod(PSP(ACT)) := Mod(SP_{[PAR/ACT]})$$

For technical simplicity, we do not allow parameter-passing morphisms and hence omit arbitrary actual parameters. Transforming a generic *DoDL*-class then requires the transformation of a parameter specification first, since sorts and symbols introduced there are applied in a generic class.

TRANSFORMATION RULE 15

1. The transformation of a generic *DoDL*-class presumes the transformation of a parameter specification.
2. A generic *DoDL*-class, `G`, is transformed into a parameterized specification of the form $\lambda X \ : \ PAR.SP$. Thereby, SP is a specification obtained by transforming of the features of `G`.
3. The transformation PAR of its parameter specification is imported into SP.
4. Sort par is added to $sorts(SP)$.

Remarks:

1. The transformation of `G` refers to the transformation rules defined previously. That is, if `G` is a simple class (apart from genericity), it is transformed by simple class transformation. Local classes and inheritance are respected analogously.
2. The import of PAR in SP ensures that PAR is a subspecification of SP, as required by Definition 7.

In addition to being parameterized and thus allowing for exchangeable types used in attribute declaration or method definition, a generic class is either simple, may contain local classes or is defined as a subclass. These cases are respected when a generic class is transformed, thereby being reduced to one of the cases already discussed. Only actualizations of parameterized classes are used in other classes. Those classes hence aggregate a parameterized class in attribute declaration, or are a subclass of a parameterized class, or are themselves generic. In any case, actualizations are transformed following those transformation rules that respect the actualizing class structure.

TRANSFORMATION RULE 16

1. The transformation of an actualization, $PSP(ACT)$, presumes the transformation of ACT. In SP, the sort par is renamed with act.
2. An attribute declaration in a class `class` directly actualizing PSP of the form `attrID: attrType[actPar]` is transformed into an operation in $opns(CLASS)$ of the form $attrID : class \rightarrow attrType$.

Remarks: Hierarchical actualization is covered by subclass transformation. Generic actualization establishes a parameterized specification with SP being established by subclass transformation.

Due to this rule, the interpretation of generic classes totally reduces to the respective interpretation rules given in the previous sections and need no further investigation.

EXAMPLE 6 We refer to Example 1 and show class `sponger` as a generic class in Listing 11. To keep things simple, we use direct actualization in class `useSponger` which actualizes the formal parameter `beingSponged` by class `grower`.

LISTING 11 A generic class in *DoDL*

```
class sponger [beingSponged] is
   attributes spng: beingSponged; //using the parameter
   construct
      nat sponge(nat val){
         spng.set(val);
         spng.grow();
         return spng.get();
      }
end sponger;

class useSponger is
   attributes sp: sponger[grower]; //a direct actualization
   construct
      nat main(void){
         print(sp.sponge(7));
      }
end useSponger;
```

Without an actualization, class `sponger` can only be properly transformed if a parameter specification is given. Class `beingSponged` (see Listing 12) offers attributes and methods an actual parameter may implement. That is, method bodies are empty here. For simplicity, this class resumes class `grower` as given in Example 1 on page 11.

LISTING 12 An interface

```
class beingSponged is
   attributes num: nat;
   construct
      nat get(void){}
      grower set(nat val){}
      grower grow(void){}
end beingSponged;
```

Transforming class `beingSponged` by rule 15, part 1, results in a specification with an empty set of axioms. Since this class is simple, we use simple class transformation and yield specification 5. Such a parameter specification serves as a "template" for actual parameters which define axioms and thus fix a semantics for the operations offered in the template. It is clear that each

$\Sigma_{BEINGSPONGED}$-algebra is a model of this parameter specification. An actual parameter, however, restricts this set of models.

SPECIFICATION 5 A parameter specification

$BEINGSPONGED =$

sorts	$beingsponged$
opns	$num : beingsponged \to nat$,
	$get : beingsponged \to nat$,
	$set : beingsponged \times nat \to beingsponged$,
	$grow : beingsponged \to beingsponged$

Together with class `beingSponged`, class `Sponger` can then be transformed using transformation rule 15, part 2. Again, we use the rules defined for simple classes and import specification *BEINGSPONGED* into *SPONGER* (see Spec. 6). Following Definition 7, the resulting parameterized specification has the form $\lambda X : BEINGSPONGED \;.\; SPONGER$. Interpreting specification *SPONGER* follows the interpretation rules defined for simple classes with aggregation.

SPECIFICATION 6 A generic class transformation

$SPONGER =$ **import** $BEINGSPONGED$ **into**

sorts	$sponger$
opns	$spng : sponger \to beingsponged$,
	$sponge : sponger \times nat \to nat$
vars	$s : sponger, v : nat$
axms	$sponge(s, v) = get(grow(set(spng(s), v)))$

To transform class `useSponger`, we follow transformation rule 16 and presume the transformation of class `grower` as given in Example 1. Notice that class `sponger` is an element class in `useSponger`, and its transformation has thus to be imported into *USESPONGER*. Thereby, we rename sort *beingsponged* with *grower* and yield specification 7. The attribute declaration for `sp` follows rule 16.2. Due to the import of *GROWER* via *SPONGER*, the axioms given there have to be respected when *USESPONGER* is interpreted. This interpretation follows the rules given for simple classes respecting aggregation.

SPECIFICATION 7 An actualization

$USESPONGER =$ **import** $SPONGER_{[BEINGPSPONGED/GROWER]}$ **into**

sorts	$useSponger$
opns	$sp : useSponger \to sponger$,
	$main : useSponger \to nat$
vars	$u : useSponger, v : nat$
axms	$main(u) = sponge(sp(u), 7)$

In case of hierarchical actualization, we transform `useSponger` by the rules given for inheritance. Additionally in this case, `useSponger` may also be generic. Then, the result is a parameterized specification the body of which respects the transformation rules given for inheritance as well. Generic classes containing local classes are a variation of the same theme and respect local class transformation rules.

■

4.6 Semantics of Bindings

Bindings assign values to attributes. This allows to create instances (or objects) during compilation. At run-time, further instances can be created using a **new**-operator. Both concepts are given meaning in this section. The transformation of classes is not affected, but interpretations of transformed attribute declarations have to be adapted such that the specific values given in a binding are respected. This allows to precisely define the notions of *type* and *object* in the algebraic context.

4.6.1 Transformation of Bindings

Transforming a binding means to define an assignment function. In the sequel, let $BINDING$ be the set of all syntactically correct bindings as given by the syntax rules, and let $IDENT$ and $VALUE$ be the respective sets for identifiers and values.

DEFINITION 8 An **assignment** is defined as a partial function from the set of all identifiers to the set of all values:

$$assign := IDENT \nrightarrow VALUE$$

The **set of all assignments** is denoted by $ASSIGN$.

TRANSFORMATION RULE 17 The transformation of a binding, B, yields a function $[\![\_]\!]$ which maps bindings onto assignments as follows:

$$[\![\_]\!] : BINDING \to ASSIGN,$$
$$\forall B \in BINDING, var \in IDENT :$$
$$[\![B]\!](var) := \begin{cases} value, & \text{if } \texttt{var: type = value} \text{ is a simple assignment in } B \\ \bot, & \text{else} \end{cases}$$

This transformation does not yet respect in-assignments. Therefore, we define a binary infix operation _|_ on bindings extracting the simple assignments within an in-assignment. An in-assignment refers to an identifier used in attribute declaration. This identifier is called *qualifying identifier* the type of which is a self-defined class. The attributes in this class are also assigned values. This happens recursively.

TRANSFORMATION RULE 18 Let B be a binding and $qual$ a qualifying identifier in B. The transformation of B is extended for in-assignments in the following way:

$$\_|\_ : BINDING \times IDENT \to BINDING,$$
$$\forall B \in BINDING, qual \in IDENT, var \in IDENT :$$
$$[\![B|_{qual}]\!](var) := \begin{cases} value, & \text{if } qual \text{ is a qualifying identifier of an in-assignment, } I \text{, in } B \text{ and } \texttt{var: type = value} \text{ is a simple assignment in } I \\ \bot, & \text{else} \end{cases}$$

A binding must assign values to precisely those attributes given in the main class of a *DoDL*-program (see Sect. 3.1), and in all its super- and element classes, recursively. Hence, not each arbitrary binding is suitable for a given *DoDL*-class. Moreover, the values fixed in a binding have to be typed correctly. Such bindings are called *fitting*. In the sequel, `attrType` refers to the predefined types `nat`, `bool`, and `string`. Self-defined classes posses their own attributes which are bound in in-assignments.

DEFINITION 9 Let `D` be a *DoDL*-program, and let `C` be the main class in `D`. A binding B **fits** to `C`, if and only if the following holds:

1. For each attribute declaration of the form `attrID: attrType`, either in `C` or in a subclass `C'` with `C` $\sqsubseteq$ `C'`, $[\![B]\!]($`attrID`$)$ has to be defined.
2. For each attribute declaration of the form `attrID: class`, either in `C` or in a subclass `C'` with `C` $\sqsubseteq$ `C'`, where `class` is self-defined, there exists an in-assignment in B with `attrID` the qualifying identifier of type `class'` and `class'` $\sqsubseteq$ `class`, such that $B|_{\texttt{attrID}}$ is a binding fitting to `class'`.
3. No other assignments exist in B.

Remarks: A binding does not fit to `class'`, if `class'` is not a subclass of `class`. The definition is recursive. Value assignment to element class attributes has to be done in a binding fitting to this class. Assignments of attributes not belonging to any class in `D` are of no use in a binding. Hence, they can be omitted.

EXAMPLE 7 Listing 13 shows a *DoDL*-program the classes of which are already known from Examples 4 and 6. We omit the **`construct`**-sections here, since they are not relevant for discussing bindings. The **`main`** method in class `useSponger`, however, is shown to demonstrate the introduction of an attribute `num`.

LISTING 13 A *DoDL*-program

```
class grower is
   attributes num: nat;
   construct ...
end grower;

class sponger is grower with
   construct ...
end sponger;

class useSponger is
   attributes  sp: sponger;
               num: nat;
   construct
      nat main(void){
         print(sp.sponge(num));
      }
end useSponger;
```

A binding B fitting to class `useSponger` has to assign values to both `sp` and `num`. Notice that `sp` is an instance of class `sponger` which does not introduce attributes, but is a subclass of `grower`. There, an attribute `num` is declared which has also to be bound. Since `sponger` is an element class in `useSponger`, this assignment is done within an in-assignment expression. Listing 14 shows the situation.

LISTING 14 Binding attributes

```
binding useSponger is
   in sp: sponger assign
      num: nat = 0;
   end;
   num: nat = 7;
end;
```

The transformation of B results in an assignment function given by the following mappings. The simple assignment for `num` yields $([\![B]\!](num) \longmapsto 7)$, whereas the in-assignment for `sp` yields

($[\![B|_{\mathtt{sp}}]\!](num) \longmapsto 0$). The attribute `num` does not cause a name clash, since it occurs in different scopes. This is respected by introducing $B|_{\mathtt{sp}}$.
■

4.6.2 Interpretation of Bindings

In the previous sections, the interpretation of transformed attribute declarations was left open. Now we can fix these interpretations to the respective values given in a binding. Again, we distinguish between simple assignments and in-assignments.

In the sequel, let `class` be a *DoDL*-class, *CLASS* its transformation, and $\mathcal{C}$ a Σ_{CLASS}-algebra. Further, let `elem` be an element class in *class*, *ELEM* its transformation, and $\mathcal{E}$ a Σ_{ELEM}-algebra, let `super` be a *DoDL*-class with `class` $\sqsubseteq$ `super`, *SUPER* its transformation, and $\mathcal{S}$ a Σ_{SUPER}-algebra. Let B be a binding fitting to `class`. Again, `attrType` refers to the pre-defined types `nat`, `bool`, and `string`.

INTERPRETATION RULE 11 For each attribute declaration in `class` of the form `attrID: attrType` we define:

$$\forall attrID : class \rightarrow attrType \in opns(CLASS), c \in A_{class} :$$
$$attrID^{\mathcal{A}}(c) = [\![B]\!](\mathtt{attrID})$$

INTERPRETATION RULE 12 For each attribute declaration in `elem` of the form `attrID: attrType` we define:

$$\forall attrID : elem \rightarrow attrType \in opns(ELEM), e \in E_{elem} :$$
$$attrID^{\mathcal{E}}(e) = [\![B|_{\mathtt{e}}]\!](\mathtt{attrID})$$

INTERPRETATION RULE 13 For each attribute declaration in `super` of the form `attrID: attrType` we define:

$$\forall attrID : super \rightarrow attrType \in opns(SUPER), s \in S_{super} :$$
$$attrID^{\mathcal{S}}(s) = [\![B]\!](\mathtt{attrID})$$

The interpretation of attribute declarations in element classes as well as in superclasses is forwarded to aggregates and subclasses by the respective interpretation rules given in the previous sections. Therefore, a binding has to be respected in Σ_{ELEM} and Σ_{SUPER}-algebras.

These interpretation rules lead to models which we call *bound*:

DEFINITION 10 Each Σ_{CLASS}-algebra following the above interpretation rules is called **bound model** of *CLASS*. The **set of all bound models** of *CLASS* is denoted by $Bound(CLASS)$.

It is easy to prove that bound models are models:

OBSERVATION 5 Each Σ_{CLASS}-algebra in $Bound(CLASS)$ is in $Mod(CLASS)$.

Proof. Interpretation rules 11, 12 and 13 fix the interpretation of attribute declarations to certain values of the respective carrier-sets. This interpretation has thus no effect on observations 1 and 2. Hence, the rules maintain model properties. ■

EXAMPLE 8 Referring to Example 4, let $\mathcal{U}$ be a $\Sigma_{USESPONGER}$-algebra, $\mathcal{S}$ a $\Sigma_{SPONGER}$-algebra, and $\mathcal{G}$ a Σ_{GROWER}-algebra. We apply interpretation rules 11, 12, and 13 on binding B as given in Example 7 and obtain the following interpretations:

1. With $[\![B]\!](num) = 7$, we have $num^{\mathcal{U}}(u) = 7$ for each $u \in U_{useSponger}$ by rule 11.

2. With $[\![B|_{sp}]\!](num) = 0$ and rule 12, together with $B|_{sp}$ is a binding fitting to class `sponger`, we have $num^{\mathcal{G}}(g) = 0$ for each $g \in G_{grower}$ by rule 13.

3. The interpretation $num^{\mathcal{S}}(s) = num^{\mathcal{G}}(s)$ is obtained by rule 6.

■

With the notion of bound models we can formally define types and objects. For pre-defined classes, their type is their respective model set given in the usual way. For self-defined classes, we define:

DEFINITION 11 Let `attrID: attrType` be an attribute declaration in `class` such that `attrType` is a self-defined class. The **type** of `attrID`, *type*(`attrID`) for short, is defined as the loose semantics $[\![ATTRTYPE]\!] = \mathit{Mod}(ATTRTYPE)$ of the transformation $ATTRTYPE$.

In correspondence with the usual characterization of objects found in the literature (c.f. [37], Sect. 8.3), we define an **object** as an *individually identifiable class instance*. Moreover, we use the designators given in attribute declarations to refer to an object. Each class hence describes a set of individuals through attributes and methods. The **state of an object** is given by the set of attribute/value pairs fixed in a binding (c.f. [50], Sect. 5.1). Methods are responsible to controllably change this state, that is, methods allow to change values in attribute/value pairs. The **behavior of an object** paraphrases the conditions under which an objects is allowed to change its state (c.f. [17], Sect. 12.1.1). **Initialization** corresponds to the initial assignment of values to attributes (c.f. [37], Sect. 8.3), in *DoDL* done by bindings or the **new**-operator.

Types are given through model sets. An instance of a *DoDL*-class `class` thus corresponds to a model in $\mathit{Mod}(CLASS)$. Without bindings, those models are of no pragmatic use, since we cannot refer to the object's state and hence cannot manipulate it. Therefore, we respect bindings:

DEFINITION 12 Each model in $\mathit{Bound}(CLASS)$ is an **object** of class `class`.

To create objects at runtime, we use a simple **new**-operator. This operator is parameterized with a binding fitting the class an instance of which is created. By this binding, we can establish a bound model in the way discussed above. That is, the semantics of the **new**-operator is a bound model. A concluding example may illustrate how instantiation works at runtime.

EXAMPLE 9 We refer to class `useSponger` as shown in Listing 13 in Example 7 on page 31.

LISTING 15 The **new**-operation

```
class useSponger is
   attributes  sp: sponger;
               num: nat;
   construct
      nat main(void){ print(sp.sponge(num)); }
end useSponger;

class surround is
   construct
      void showCreate(){
         useSP: useSponger;
         useSP = new(sp(num = 0), num = 7);
      }
end surround;
```

Let us assume, a class `surround` contains a method `showCreate` that declares a variable `useSP` of type `useSponger` and thereby an instance of class `useSponger` (see Listing 15 on page 33). Since this variable is not a class attribute, it is not respected by a binding as discussed above and needs another form of initialization.

The **`new`**-operator expects a value for `sp` and a value for `num`. With references, we could refer to an instance of `sponger` already defined in a binding. But references do not exist in *DoDL*, and we have to create a new instance of `sponger`. Moreover, it is easy to convert the parameters of the **`new`**-operator into a binding fitting to `useSponger`. The simple assignment `num = 7` corresponds to a simple assignment in such a binding. The first parameter, the structured assignment `sp(num = 0)`, corresponds to an in-assignment expression: the qualifying identifier `sp` here prefixes a list of assignments written in brackets.

The transformation of **`new`** results in a term of the form $useSP(s) = set(set(sp, 0), 7)$, where $s \in S_{Sourround}$ and $sp \in S_{sponger}$ for a $\Sigma_{SOURROUND}$-algebra $\mathcal{S}$. Its interpretation affects the interpretation of $num : useSponger \rightarrow nat$ and $num : sponger \rightarrow nat$ only.
■

Finally, we can define the semantics of a *DoDL*-program as the model set of its main class, and show that class inheritance implies type inheritance:

DEFINITION 13 Let `D` be a *DoDL*-program, let `class` be the main class in `D`, and $CLASS$ its transformation. Further, let B be a binding fitting to `class`, and let $\mathcal{M}_B$ be a bound model for $CLASS$. Let $[\![\_]\!]$ be a mapping from a *DoDL*-program and a binding onto a model set. Then, the **semantics of `D` under B**, $[\![\texttt{D}]\!]_B$ for short, is defined as the smallest model $\mathcal{M} \in Mod(CLASS)$ with $\mathcal{M} \subseteq \mathcal{M}_B$.

Remarks: Since fitting bindings do not affect the model property, it is easy to see that such a model $\mathcal{M}_B$ exists.

With the assumption of Observation 4, each model of a hierarchical specification reduced to the signature of its superclass specification is a model of the superclass specification. For types, we introduce a relation $\preccurlyeq$ similar to class inheritance, $\sqsubseteq$, which respects Def. 5:

DEFINITION 14 **Type inheritance** is defined as a relation $\preccurlyeq \subseteq MOD \times MOD$ on model sets. It is defined for all transformed *DoDL*-classes, $CLASS$ and $CLASS'$, as follows:

$$Mod(CLASS) \preccurlyeq Mod(CLASS') :\Longleftrightarrow \forall \mathcal{A} \in Mod(CLASS) : \mathcal{A}|_{\Sigma_{CLASS'}} \in Mod(CLASS')$$

OBSERVATION 6 Let `class` and `class'` be two *DoDL*-classes, $CLASS$ and $CLASS'$ their transformations. Then,

$$\texttt{class} \sqsubseteq \texttt{class'} \Longrightarrow Mod(CLASS) \preccurlyeq Mod(CLASS')$$

holds.

Proof. The assumption immediately follows from the transformation rules and Definition 14. ■

Remarks: A similar observation between classes and instances is made in [1], page 18, and is there called *subtype relation*.

4.7 Embedding Imperative Parts

Primitive statements and control structures are used within method bodies. We allow declaration of variables, value assignment, sequences, alternatives, as well as `for`- and `while`-loops. Additionally, method invocation and lookup together with the concepts of `this` and `super` are crucial and have already been discussed in Section 4.4. The method bodies are left to be transformed into suitable axioms. Exactly there, the semantics discussed so far and the semantics of the imperative parts are integrated. An approach to combine formal semantics of these different aspects is, for example, proposed in [3]. We proceed as follows, thereby utilizing term evaluation and denotational semantics of imperative languages.

The execution of a *DoDL*-program utilizes the mechanisms usually applied for imperative languages (c.f. [52]): assignments and method invocation form the primitive statements, control structures are responsible for the order of their execution. As soon as a *DoDL*-program is transformed and an appropriate interpretation is given, program execution reduces to evaluation of these interpretations. We show this by an example, after defining term evaluation, in the literature usually called term interpretation. In order not to collide with the notion of interpretation used in the present paper, we refer to term interpretation as term evaluation.

DEFINITION 15 A mapping $v : X \to \mathcal{A}$ is called **valuation**. It is defined through an S-indexed family $\{v_s\}_{s\in S}$ of mappings $v_s : X_s \to A_s$ for each $s \in S$.

Let $\Sigma = \langle S, \Gamma\rangle$ be a signature, $\mathcal{A}$ a Σ-algebra, and $v : X \to \mathcal{A}$ a valuation. For each $s \in S$, a S-sorted mapping $v^*_s : \mathcal{T}(\Sigma, \mathrm{X})_s \to A_s$ is defined as follows:

1. for all variables $x \in X_s$ let $v^*_s(x) := v(x)$
2. for all constants $f :\to s \in \Gamma$ let $v^*_s(f) := f^{\mathcal{A}}$
3. for all operations $f : s_1 \times \ldots \times s_n \to s \in \Gamma$ and all terms $t_1 \in \mathcal{T}(\Sigma, \mathrm{X})_{s_1}, \ldots, t_n \in \mathcal{T}(\Sigma, \mathrm{X})_{s_n}$ let $v^*_s(f(t_1, \ldots, t_n)) := f^{\mathcal{A}}(v^*_s(t_1), \ldots, v^*_s(t_n))$

A mapping $v^* : \mathcal{T}(\Sigma, \mathrm{X}) \to \mathcal{A}$ is called **term evaluation** of terms in $\mathcal{A}$ with respect to v, and is defined through an S-indexed family $\{v^*_s\}_{s\in S}$ of mappings v^*_s.

EXAMPLE 10 In Example 2, we gave interpretations for classes `grower` and `sponger`. Executing the **`main`** method in `sponger` means to evaluate the term $main(s)$ following the respective interpretations:

$$\begin{aligned}
v^*((main)(s)) &= main^{\mathcal{S}}(v^*(s)) && \text{(Def. } v^*)\\
&= sponge^{\mathcal{S}}(v^*(s), 7) && \text{(eq. (6))}\\
&= get^{\mathcal{S}}(grow^{\mathcal{S}}(set^{\mathcal{S}}(grw^{\mathcal{S}}(v^*(s)), 7))) && \text{(eq. (5))}\\
&= get^{\mathcal{G}}(grow^{\mathcal{G}}(set^{\mathcal{G}}(v^*(grw(s)), 7))) && \text{(eqs. (9) – (11))}\\
&= get^{\mathcal{G}}(set^{\mathcal{G}}(v^*(set(grw(s), 7)), num^{\mathcal{G}}(v^*(set(grw(s), 7))) + 1)) && \text{(eq. (4))}\\
&= get^{\mathcal{G}}(v^*(set(set(grw(s), 7), 7 + 1))) && \text{(eqs. (2), (3))}\\
&= num^{\mathcal{G}}(v^*(set(set(grw(s), 7), 8))) && \text{(eq. (1))}\\
&= 8 && \text{(Def. } v^*)
\end{aligned}$$

A Σ_{GROWER}-algebra $\mathcal{G}$ with $num^{\mathcal{G}}(g) = 8$ is a model of *GROWER* and hence an instance of class `grower`; analogously, a $\Sigma_{SPONGER}$-algebra $\mathcal{S}$ with the interpretations as given is a model of *SPONGER* and hence an instance of class `sponger`. Intuitively spoken, the semantics of the main method corresponds to the evaluation of its transformation.

■

Following Definition 13 on page 34, the transformation of a *DoDL*-program starts at its main class. Thereby, we follow the transformation rules given in the previous sections. We can thereupon define semantic functions, $[\![\_]\!]$, which denote a mathematical object to each syntactic construct. For example, the **`declare`**-section is given meaning through a function $[\![\_]\!]$ that maps a **`declare`**-section, $declsec$, onto a mapping from local classes, loc_i, to their transformations, LOC_i, i.e. $[\![declsec]\!](loc_i) = LOC_i$. Analogously, the **`attribute`**- and **`construct`**-section can be given a meaning.

Method bodies represent the lowest level of such recursive class transformation, and hence semantic functions for imperative constructs have to be formalized algebraically. We can respect denotational semantics for imperative languages. For example, the denotational semantics of an **`if`**-statement of the form `if` b `then` s_1 `else` s_2 is given as follows (c.f. [2], chap. 7):

$$[\![\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2]\!] = \begin{cases} [\![s_1]\!], & \text{if } [\![b]\!] = true \\ [\![s_2]\!], & \text{else} \end{cases}$$

To interpret this statement algebraically, we have to evaluate the semantics of s_1, if the interpretation of b in a Σ_{BOOL}-algebra $\mathcal{B}$ yields $true$. Otherwise, we evaluate s_2:

$$[\![\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2]\!] := v^*(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2) = \begin{cases} [\![s_1]\!], & \text{if } b^{\mathcal{B}} = true \\ [\![s_2]\!], & \text{else} \end{cases}$$

This shows exactly how an **`if`**-statement is transformed into an axiom. Axioms for s_1 and s_2 are established through transformation, and the semantic functions $[\![s_1]\!]$ and $[\![s_2]\!]$ are given by evaluating suitable axioms, thereby reducing the mapping $[\![\_]\!]$ stepwise to the evaluation v^*. For example, a method invocation of the form `expresseion.methodID(`$\texttt{p}_1, \ldots, \texttt{p}_n$`)` is transformed into a term $m(expression, p_1, \ldots, p_n)$. With slight syntactic variations, a sequence of the form shown in method `sponge` of class `sponger` is transformed into a term of the form $m(t, p_1, \ldots, p_n)$ where t is such a transformed expression. The semantics of method invocation corresponds to the evaluation of its transformation. Since interpretations of operations satisfy the axioms given, we simply have

$$\begin{aligned}[\![\texttt{expr.m(}\texttt{p}_1, \ldots, \texttt{p}_n\texttt{)}]\!] &:= v^*(m([\![\texttt{expr}]\!], [\![\texttt{p}_1]\!], \ldots [\![\texttt{p}_n]\!])) \\ &= m^{\mathcal{C}}(v^*(expr), v^*(p_1), \ldots, v^*(p_n))),\end{aligned}$$

where `expr` is an expression denoting an instance of class `class`, and $\mathcal{C}$ is a Σ_{CLASS}-algebra.

For variable declaration and assignment, sequences, and loops we can proceed analogously. Together with the evaluations given above, the semantics of a program is deduced by recursive transformation and evaluation. Hence, our algebraic semantics is denotational.

EXAMPLE 11 As a last example, we refer to Example 5 and show the semantics of the `main` method. It is given by the following evaluation. Let $\mathcal{S}$ be a Σ_{SPONGE}-algebra as in Example 5. Numbers over the equivalence sign denote interpretations in Example 5.

$$\begin{aligned}
[\![\texttt{main()}]\!] &\overset{def}{=} v^*(main(s)) \\
&\overset{v^*}{=} main^{\mathcal{S}}(s) \\
&\overset{(20)}{=} sponge^{\mathcal{S}}(v^*(s), 7, 8) \\
&\overset{(19)}{=} get^{\mathcal{S}}(grow^{\mathcal{S}}(grow^{\mathcal{S}}(in^{\mathcal{S}}(set^{\mathcal{S}}(v^*(s), 7, 8))))) \\
&\overset{(18)}{=} get^{\mathcal{S}}(grow^{\mathcal{S}}(grow^{\mathcal{S}}(in^{\mathcal{S}}(set(v^*(s), 7, 8))))) \\
&\overset{(15).if}{=} get^{\mathcal{S}}(grow^{\mathcal{S}}(cast^{\mathcal{S}}\left[grow^{\mathcal{G}}(in^{\mathcal{S}}(set(v^*(s), 7, 8)))\right])) \\
&\overset{(4)}{=} get^{\mathcal{S}}(grow^{\mathcal{S}}(cast^{\mathcal{S}}\left[set^{\mathcal{G}}\left[in^{\mathcal{S}}(set(v^*(s), 7, 8)), num^{\mathcal{G}}\left(in^{\mathcal{S}}(set(v^*(s), 7, 8))\right) + 1\right]\right])) \\
&\overset{(22)}{=} get^{\mathcal{S}}(grow^{\mathcal{S}}(cast^{\mathcal{S}}\left[set^{\mathcal{G}}\left[in^{\mathcal{S}}(set(v^*(s), 7, 8)), num^{\mathcal{G}}\left(set(v^*(s), 7)\right) + 1\right]\right])) \\
&\overset{(1),(3)}{=} get^{\mathcal{S}}(grow^{\mathcal{S}}(cast^{\mathcal{S}}\left[set^{\mathcal{G}}\left[in^{\mathcal{S}}(set(v^*(s), 7, 8)), 7 + 1\right]\right])) \\
&\overset{(24)}{=} get^{\mathcal{S}}(grow^{\mathcal{S}}(set(v^*(s), 7 + 1, 8))) \\
&\overset{(15).else}{=} get^{\mathcal{S}}(set^{\mathcal{S}}\begin{bmatrix} set(v^*(s), 8, 8), \\ \quad num^{\mathcal{S}}(set(v^*(s), 8, 8)) + value^{\mathcal{S}}(set(v^*(s), 8, 8)), \\ \qquad value^{\mathcal{S}}(set(v^*(s), 8, 8)) \end{bmatrix})
\end{aligned}$$

$$
\begin{aligned}
&\overset{(12),(16)}{=} get^{\mathcal{S}}(set^{\mathcal{S}} \begin{bmatrix} set(v^*(s),8,8), \\ \quad num^{\mathcal{G}}(in^{\mathcal{S}}(set(v^*(s),8,8))) + getV^{\mathcal{S}}(set(v^*(s),8,8)), \\ \qquad getV^{\mathcal{S}}(set(v^*(s),8,8)) \end{bmatrix}) \\
&\overset{(1),(22),(17)}{=} get^{\mathcal{S}}(set^{\mathcal{S}}(set(v^*(s),8,8), get^{\mathcal{G}}(set(v^*(s),8)) + 8, 8)) \\
&\overset{(13),(3)}{=} get^{\mathcal{G}}(in^{\mathcal{S}}(set^{\mathcal{S}}(set(v^*(s),8,8), 8+8, 8))) \\
&\overset{(22)}{=} get^{\mathcal{G}}(set(set(v^*(s),8,8), 8+8)) \\
&= 16
\end{aligned}
$$

This value corresponds to an intuitive expectation of calling `sponge(7, 8)`. Moreover, with the interpretation of the set operations, we establish terms that reflect the state changes of an object of class $sponger$. Each carrier-set of a sort of interest (see Sect. 2), e.g. $grower$ and $sponger$, represents a set of all states of an object of class `grower` and `sponger`, respectively.
■

5 Conclusion and Further Work

5.1 Conclusion

In this paper, we studied the algebraic semantics of a simple object-oriented language, *DoDL*. The approach is based on structured algebraic specifications used to denote the object-oriented concepts of *DoDL*.

First, we defined transformation rules to yield an adequate algebraic specification for each class definable in *DoDL*. Using interpretation rules, we established a mathematical model reflecting the classes semantics. In a second step, algorithmic parts were embedded into the semantics by using the usual denotational approach to imperative languages. Our approach thus exploits the fact that an object-oriented language introduces code structuring mechanisms not found in imperative languages but extending them. Semantical aspects such as polymorphism and late binding not expressible on the specification level were simulated on the respective algebras. Those concepts were also given a precise semantics by interpretation rules.

Depending on the specific concepts the language under consideration offers, transformation and interpretation rules must be adapted to them. For example, multiple inheritance as provided in C++ is not found in *DoDL* and hence not modelled in this paper; a different inheritance mechanism as e.g. Beta [36, 16] is equipped with also requires a different formalization. Nonetheless, the goal of this paper was to develop an approach to formally defining a semantics for object-oriented languages, here elaborated for a sample language, *DoDL*.

Moreover, we have discussed how algebraic specification can be put to use for complex applications such as formalizing a language's semantics. In contrast to the usual task of specifications, we formalize an implementation instead of developing it from a specification.

These rules allow the generation of mathematical models. A compiled program has to fulfill the properties of such a model. Hence, a model can be used as a test-base (c.f. [24]) for the correctness of a compiled program.

The algebraic models are modular, i.e. they are extensible and exchangeable: the interpretation of a class, say, *c*, is changed from $Mod(C)$ to $Bound(C)$ without the need to change anything else than the interpretation of transformed attribute declarations.

5.2 Further Work

The concept of references can be introduced as follows. First, the notion of types is extended to pairs of an infinite set of identifiers and a model set, $(IDENT, MOD)$ for short. Identifiers correspond exactly to object names. Referencing is then defined as a partial function $ref : IDENT \,+\!\!\!\!\rightarrow$

MOD. If a model, m, is changed for example due to assignments at runtime, the values of ref are changed for all instances i with $ref(i) = m$. This behavior corresponds to object references discussed in [37], Sect. 8.6. Deletion of an object, i, yields undefined references simulated by an undefined value for $ref(i)$. Changing a reference corresponds to changing a value in ref. A **new**-operation creating an object obj of type t corresponds to adding a pair $(obj \mapsto \mathcal{M})$ to ref where $\mathcal{M}$ has to be a model in $Mod(T)$ (where T is the transformation of t), and obj is in $IDENT$.

A technically more complicated yet more detailed approach is discussed in [48] and [59]. So-called *entity algebras* are used to represent dynamic structures and thereby allow for references. Another approach is captured by *evolving algebras* [31].

What follows from references are mutually associated classes. The use-relation then defines a (cyclic) collection of classes. This collection has to be transformed into one single algebraic specification to overcome the problem of sorts used recursively (c.f. [41]).

Some concepts need further investigation: polymorphic object constructors, explicit destructors, abstract classes, virtual methods, interfaces and multiple inheritance are worth a formalization. We are confident that our approach is strong enough to capture these aspects to widen the range of the semantic issues discussed here. Our approach shows that algebraic specification and its model sets can be used to give an algebraic semantics to object-orientation in a uniform manner.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[2] L. Allison. *A practical introduction to denotational semantics*. Cambridge University Press, 1986.

[3] E. Astesiano, M. Cerioli, and G. Reggio. Plugging data constructs into paradigm-specific languages: Towards an application to uml. In T. Rus, editor, *Algebraic Methodology and Software Technology (AMAST 2000)*, volume 1816 of *Lecture Notes in Computer Science (LNCS)*, pages 271 – 292, 2000.

[4] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments*, volume 562 of *Lecture Notes in Computer Science (LNCS)*. Springer, 1991.

[5] T. L. Briggs and J. Werth. A specification language for object-oriented analysis and design. In M. Tokoro and R. Pareschi, editors, *ECOOP '94*, volume 821 of *Lecture Notes in Computer Science (LNCS)*, pages 365–385. Springer, 1994.

[6] M. Broy and M. Wirsing. Algebraic definition of a functional programming language. *IEEE Transactions on Information Theory*, 17(2):137–161, 1982.

[7] M. Broy, M. Wirsing, and P. Pepper. On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems*, 9(1):54 – 99, January 1987.

[8] R. M. Burstall and J. A. Goguen. The semantics of CLEAR, a specification language. In *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, number 86 in Lecture Notes in Computer Science (LNCS). Springer, 1980.

[9] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[10] M. Cerioli, T. Mossakowski, and H. Reichel. From total equational to partial first-order logic. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Reports, pages 31 – 104. Springer, 1999.

[11] S. Clerici, R. Jimenez, and F. Orejas. Semantic constructions in the specification language Glider. In H.-D. Ehrich and F. Orejas, editors, *Recent Trends in Data Type Specification*, volume 785 of *Lecture Notes in Computer Science (LNCS)*, pages 144 – 157. Springer, 1994.

[12] S. Clerici and F. Orejas. The specification language GSBL. In H. Ehrig, K. P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification*, volume 534 of *Lecture Notes in Computer Science (LNCS)*, pages 31 – 51. Springer, 1991.

[13] G. Cousineau. An algebraic definition for control structures. *Theoretical Computer Science*, 12(2):175–198, 1980.

[14] E.-E. Doberkat. A language for specifying hyperdocuments. *Software - Concepts and Tools*, 17:163–172, April 1996.

[15] E.-E. Doberkat. Using logic for the specification of hypermedia documents. In J. Balderjahn, R. Mathar, and M. Schader, editors, *Classification, Data Analysis and Data Highways*, pages 205–212. Springer, 1998.

[16] E.-E. Doberkat and S. Dissmann. *Einf"uhrung in die objektorientierte Programmierung in* BETA. Addison-Wesley, 1996.

[17] H.-D. Ehrich. Object specification. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Reports, pages 435 – 465. Springer, 1999.

[18] H.-D. Ehrich, M. Gogolla, and U. W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen*. Teubner, 1989.

[19] H. Ehrig and H.-J. Kreowski. Refinement and implementation. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Reports, pages 201 – 242. Springer, 1999.

[20] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[21] W. Fey. *Pragmatics, Concepts, Syntax, Semantics, and Correctness Notions of ACT TWO: An Algebraic Module Specification and Interconnection Language*. PhD thesis, Technische Universität Berlin, Fachbereich Informatik, 1988.

[22] A. Fronk. *Algebraische Semantik einer objektorientierten Sprache zur Spezifikation von Hyperdokumenten*. PhD thesis, Lehrstuhl Software-Technologie, Fachbereich Informatik, Universität Dortmund, Shaker Verlag, 2002, 2001.

[23] A. Fronk and J. Pleumann. Der *DoDL*-Compiler. Memorandum 100, Universität Dortmund, Fachbereich Informatik, Lehrstuhl für Software-Technologie, June 1999. ISSN 0933-7725.

[24] M.-C. Gaudel and G. Bernot. The role of formal specifications. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Report, pages 1 – 12. Springer, 1999.

[25] M. Gogolla and R. Herzig. An algebraic semantics for the object specification language TROLL *light*. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, volume 906 of *Lecture Notes in Computer Science (LNCS)*, pages 290 – 306. Springer, 1995.

[26] J. A. Goguen and R. Diaconescu. Towards an algebraic semantics for the object paradigm. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification*, volume 785 of *Lecture Notes in Computer Science (LNCS)*, pages 1 – 29. Springer, 1992.

[27] J.A. Goguen, K. Futatsugi, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Principles of Programming Languages*, ACM SIGPLAN Notices, pages 52 – 66, 1985.

[28] A. Goldberg and D. Robson. *Smalltalk-80, The Language*. Addison-Wesley, 1989.

[29] M. Gordon. *The Denotational Description of Programming Languages*. Springer, Berlin, 1979.

[30] M. Grosse-Rhode. Towards object-oriented algebraic specifications. In H. Ehrig, K. P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specifications*, volume 534 of *Lecture Notes in Computer Science (LNCS)*, pages 98 – 116. Springer, 1991.

[31] Y. Gurevich. Evolving algebras: An attempt to discover semantics. *EATCS Bulletin*, 43:264 – 284, February 1991.

[32] Y. Gurevich and J. K. Huggins. The semantics of the C programming language. In E. Börger, G. Jäger, H. Kleine-Büning, S. Martini, and M.M. Richter, editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science (LNCS)*, pages 274 – 308. Springer, 1992.

[33] C. Hintermeier, C. Kirchner, and H. Kirchner. Sort inheritance for order-sorted equational presentations. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, volume 906 of *Lecture Notes in Computer Science (LNCS)*, pages 319 – 335. Springer, 1995.

[34] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, July 1995.

[35] O. L. Madsen. Semantic analysis of virtual classes and nested classes. *ACM SIGPLAN Notices*, 34(10):114 – 131, 1999.

[36] O. L. Madsen, B. Moeller-Redersen, and K. Nygaard. *Object-Oriented Programming in the* BETA *Programming Language*. Addison-Wesley, 1993.

[37] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2. edition, 1997.

[38] P. D. Mosses. The use of sorts in algebraic specifications. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specifications*, volume 655 of *Lecture Notes in Computer Science (LNCS)*, pages 66 – 91. Springer, 1993.

[39] P. D. Mosses. Cofi: The common framework initiative for algebraic specification and development. In M. Bidoit and M. Dauchet, editors, *Recent Trends in Data Type Specification*, volume 1214 of *Lecture Notes in Computer Science (LNCS)*, pages 115 – 137. Springer, 1997.

[40] F. Orejas. Structuring and modularity. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Reports, pages 159 – 200. Springer, 1999.

[41] P. Padawitz. Swinging UML: How to make class diagrams and state machines amenable to constraint solving and proving. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language*, volume 1939 of *Lecture Notes in Computer Science (LNCS)*, pages 162 – 177. Springer, 2000.

[42] P. Padawitz. Sample swinging types. `http://ls5.cs.uni-dortmund.de/~peter`, June 2001. Manuskript.

[43] J. Palsberg and M. I. Schwartzbach. Type substitution for object-oriented programming. In N. Meyrowitz, editor, *ACM SIGPLAN Notices*, volume 25, pages 151 – 160, October 1990.

[44] F. Parisi-Presicce and A. Pierantonio. An algebraic theory of class specification. *ACM Transactions on Software Engineering and Methodology*, 3(2):166–199, April 1994.

[45] F. Parisi-Presicce and A. Pierantonio. Structured inheritance for algebraic class specifications. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specifications*, volume 785 of *Lecture Notes in Computer Science (LNCS)*, pages 295 – 309. Springer, 1994.

[46] F. Parisi-Presicce and A. Pierantonio. Dynamical behavior of object systems. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications*, volume 906 of *Lecture Notes in Computer Science (LNCS)*, pages 406 – 419. Springer, 1995.

[47] A. Poigne. Identity and existence, and types in algebra - a survey of sorts. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification*, volume 785 of *Lecture Notes in Computer Science (LNCS)*, pages 53 – 78. Springer, 1994.

[48] G. Reggio. Entities: An institution for dynamic systems. In H. Ehrig, K. P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specifications*, volume 534 of *Lecture Notes in Computer Science (LNCS)*, pages 246 – 265. Springer, 1991.

[49] H. Reichel. Specification semantics. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Report, pages 131 – 158. Springer, 1999.

[50] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Objektorientiertes Modellieren und Entwerfen*. Prentice Hall, 1993.

[51] D. Sanella and A. Tarlecki. Algebraic preliminaries. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Report, pages 13 – 30. Springer, 1999.

[52] D. A. Schmidt. *Denotational Semantics - A Methodology for Language Development*. Wm. C. Brown Publishers, 1988.

[53] R. W. Sebesta. *Concepts of programming languages*. Benjamin/Cummings, 1989.

[54] E. G. Wagner. Overloading and inheritance. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specifications*, volume 785 of *Lecture Notes in Computer Science (LNCS)*, pages 79 – 97. Springer, 1994.

[55] J. H. Williams. On the development of the algebra of funtcional programs. *ACM Transactions on Programming Languages and Systems*, 4(4):733 – 757, October 1982.

[56] M. Wirsing. Algebraic specifications. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, pages 675 – 788. Elsevier, 1990.

[57] M. Wirsing. Algebraic specification languages: An overview. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications*, volume 906 of *Lecture Notes in Computer Science (LNCS)*, pages 81 – 115. Springer, 1995.

[58] M. Wirsing, P. Pepper, H. Partsch, W. Dosch, and M. Broy. On hierarchies of abstract data types. *Acta Informatica*, 20:1–33, 1983.

[59] E. Zucca. Implementation of data structures in an imperative framework. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specifications*, volume 906 of *Lecture Notes in Computer Science (LNCS)*, pages 483 – 498. Springer, 1995.

/112/ Corina Kopka, Ursula Wellen
Development of a Software Production Process Model for Multimedia CAL Systems by Applying Process Landscaping
April 2001

/113/ Ernst-Erich Doberkat
The Converse of a Probabilistic Relation
Juni 2001

/114/ Ernst-Erich Doberkat, Eugenio G. Omodeo
Algebraic semantics of ER-models in the context of the calculus of relations.
Part II: Dynamic view
Juli 2001

/115/ Volker Gruhn, Lothar Schöpe (Eds.)
Unterstützung von verteilten Softwareentwicklungsprozessen durch integrierte Planungs-, Workflow- und Groupware-Ansätze
September 2001

/116/ Ernst-Erich Doberkat
The Demonic Product of Probabilistic Relations
September 2001

/117/ Klaus Alfert, Alexander Fronk, Frank Engelen
Experiences in 3-Dimensional Visualization of Java Class Relations
September 2001

/118/ Ernst-Erich Doberkat
The Hierarchical Refinement of Probabilistic Relations
November 2001

/119/ Markus Alvermann, Martin Ernst, Tamara Flatt, Urs Helmig, Thorsten Langer, Ingo Röpling,
Clemens Schäfer, Nikolai Schreier, Olga Shtern
Ursula Wellen, Dirk Peters, Volker Gruhn
Project Group Chairware Intermediate Report
November 2001

/120/ Volker Gruhn, Ursula Wellen
Autonomies in a Software Process Landscape
Januar 2002

/121/ Ernst-Erich Doberkat, Gregor Engels (Hrsg.)
Ergebnisbericht des Jahres 2001
des Projektes "MuSofT – Multimedia in der SoftwareTechnik"
Februrar 2002

/122/ Ernst-Erich Doberkat, Gregor Engels, Jan Hendrik Hausmann, Mark Lohmann, Christof Veltmann
Anforderungen an eine eLearning-Plattform — Innovation und Integration —
April 2002

/123/ Ernst-Erich Doberkat
Pipes and Filters: Modelling a Software Architecture Through Relations
Juni 2002

/124/ Volker Gruhn, Lothar Schöpe
Integration von Legacy-Systemen mit Eletronic Commerce Anwendungen
Juni 2002

/125/ Ernst-Erich Doberkat
A Remark on A. Edalat's Paper *Semi-Pullbacks and Bisimulations in Categories of Markov-Processes*
Juli 2002

/126/ Alexander Fronk
Towards the algebraic analysis of hyperlink structures
August 2002

/127/ Markus Alvermann, Martin Ernst, Tamara Flatt, Urs Helmig, Thorsten Langer
Ingo Röpling, Clemens Schäfer, Nikolai Schreier, Olga Shtern
Ursula Wellen, Dirk Peters, Volker Gruhn
Project Group Chairware Final Report
August 2002