Formal Aspects of Computing (1996) 8: 706–715

Formal Aspects of Computing

An Algebraic Semantic Framework for Object Oriented Languages with Concurrency (Extended Abstract)*

Ruth Breu[1] and Elena Zucca[2]

[1] Inst. für Informatik, TU München, München, Germany;
[2] DISI - Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Genova, Italy

Keywords: Object oriented languages; Semantics; Concurrency; Algebraic specifications; Labelled transition systems; Classes; Inheritance; Object identity

Abstract. This paper presents an algebraic semantics schema for object oriented languages including concurrent features. A class, the basic syntactic unit of an object oriented language, in our approach denotes a set of algebras determined by an algebraic specification. This specification describes a system of (possibly active) objects interacting via method calls. Extending other approaches, structured classes are modelled in a fully compositional way. This means that the semantic counterpart of class combinators such as inheritance and clientship are specification combinators. A model of records with sharing allows us to describe typical object oriented features like object sharing, inheritance polymorphism and dynamic binding. For modelling the dynamic behaviour of objects, we rely on an algebraic description of labelled transition systems.

Introduction

The aim of this paper is to supply a guideline for giving the semantics of a variety of concrete object oriented languages with concurrent features. To this end, we propose an algebraic framework which combines notions and results from different approaches.

Correspondence and offprint requests to: Elena Zucca, DISI, Università di Genova, Via Dodecaneso, 35, 16146 Genova, Italy. Email: zucca@disi.unige.it

* This is an extended abstract of [BrZ96], which can be retrieved by downloading the (compressed Postscript) file: FAC_8E.ps.z found in directory pub/fac on the ftp.cs.man.ac.uk. Work partially supported (for E. Zucca) by WG n.6112 COMPASS, Murst 40% - Modelli della computazione e dei linguaggi di programmazione and HCM-MEDICIS.

The first idea on which this paper is based is that the class structuring mechanism of object oriented languages is really close to the structuring mechanism of algebraic specifications. In our paper, a class denotes a set of algebras determined by an algebraic specification. More precisely, a class interface (name and type of the operations which can be performed on objects) can be modelled in a very straightforward way by the formal notion of signature, while a class implementation (internal structure of objects and meaning of the methods) can be modelled in an axiomatic way stating the properties that operations must satisfy. That approach enables us to give a compositional semantics of classes; in this respect it is different from the traditional denotational style like used e.g. in [Wol87] for Smalltalk. Indeed, a class combinator (e.g. inheritance) is semantically interpreted as a function which handles sets of algebras, or, in an equivalent way, as a specification combinator. In particular, we use the algebraic specification language ASL [Wir86].

Interpreting classes as algebraic specifications enables us, on one hand, to use the algebraic approach as a foundation for explaining and formally defining concepts of object orientation. On the other hand, a common framework is provided for expressing different levels of description of a system, from a very high-level specification expressing properties in an abstract way to a lower level one corresponding to a particular class implementation. An analysis and development of these aspects in a sequential context has been done by one of the authors in [Bre91].

A second main aspect addressed in this paper is modelling the execution of an object oriented program as the evolution of a dynamic system in which many objects exist and interact. In that respect, we strongly agree with the classification introduced in [Weg87] and view an object as an entity which has an immutable identity and an internal state which changes during the lifetime of the object. Moreover, we are interested in possibly concurrent systems, i.e. systems in which many objects can evolve in parallel. Object interaction in a concurrent environment includes features like synchronization via message passing, mutual exclusion, priorities and so on. The relationship between the object and the process paradigm is not yet clear (see [Str86] for some discussion about that). Many authors take the point of view of modelling objects as processes tout court. We do not completely agree with that vision and prefer to keep a more modular approach, distinguishing between a passive and an active part of a concurrent object system.

The passive part (current configuration of the existing objects and the relations between them), what we call an object environment in the paper, is modelled as a usual data type. In particular, the framework of algebraic specifications enables us to give a quite abstract description of an object environment. This abstract description is satisfied by different concrete models which can be modularly inserted in our schema. In the full version of this paper [BrZ96] we present also a concrete model which is interesting in itself since it gives a quite unusual view of object sharing (modelled as a congruence over terms denoting objects).

For what concerns the active part (processes evolving concurrently and performing actions which can change the passive part) we rely on the approach to concurrency based on algebraic transition systems (see [AsR93]) for a general presentation). That enables us from one side to insert our concurrency treatment in a modular way in the overall framework. From the other side, a semantics schema for handling a variety of concrete languages is presented, by using parametric specifications to be instantiated by the actual concurrent features

one is dealing with. Hence our aim is more general than giving the semantics for a particular concurrent object oriented language, like e.g. in [AdB86] for POOL.

In this extended abstract, the main aim is to give to the reader the overall flavour of our semantics framework. To this end, we present in Section 1 the features of the object oriented languages we are considering, and the corresponding schema of abstract syntax; in Sections 2 and 3 we define the top-level semantic functions. In particular, Section 2 shows the interpretation of class combinators as specification combinators, and Section 3 gives an outline of the concurrent object system (passive and active part) we associate with a class. We refer to the full version of this paper [BrZ96] for the complete semantics definition, all the technicalities and examples of application.

This paper has been written largely as a result of the experience gained by the authors during the work in the Esprit project DRAGON, in which one of the aims was the development of an object oriented notation particularly suitable for reuse and distribution [BKS88]. A preliminary version has been presented at the 9th FST & TCS Conference [BrZ89].

1. The Language Schema

We consider a schema of object oriented languages which includes the following main concepts.

- A *class* describes a collection of *objects*. Each object consists of an internal state which is determined by the *attributes* of the class, together with a set of *methods* and (possibly) a *thread*. The internal state of an object can be manipulated by the outside through a method call or by the object itself by performing its thread. Classes (and objects) are called *active* if they contain a thread, *passive* otherwise. Objects are dynamic entities which exist only after they have been *created*.
- New classes can be defined using a set of *class combinators*. The most important structuring relations between classes are *inheritance* and *clientship*. A class C_1 which *inherits* from a class C_2 ("C_1 is an *heir* of C_2", "C_2 is an *ancestor* of C_1"), owns all the attributes and methods of C_2 to which it may add its own. A class can have several ancestor classes, i.e. we consider *multiple inheritance*. A class C_1 which *uses* a class C_2, ("C_1 is a *client* of C_2", "C_2 is a *server* of C_1"), can have attributes and method parameters of class C_2. These two class relations are modelled by a unique syntactic combinator in our language schema.

  In addition, we consider two other class combinators, the *export* and the *rename* operators, which enables one to hide and to rename methods, respectively.
- We consider a typed language, thus attributes and method parameters have a *type*. We rely on a mixed paradigm, thus type means here either a basic type or a class. Basic types such as booleans or integers denote sets of values. The mechanism of (*inheritance*) *polymorphism* allows the assignment of an object of any heir class of C to an attribute of a class C. As a consequence, the class of the object an attribute refers to can change dynamically, i.e. during execution (*dynamic binding*).

Formally, the abstract syntax is defined as follows.

Class::=	Flat | Inherits-Uses | Export | Rename
Inherits-Uses::=	Set(Class) Set(Class) Flat
Flat::=	Set(Attr) Set(Method) Body
Export::=	Set(Method) Class
Rename::=	Renaming Class
Attr::=	Attr-Id Type-Id
Type-Id::=	Basic-Type | Class-Id
Method::=	Method-Id List(Param) [Type-Id]
Param::=	Param-Id Type-Id
Body::=	Set(Method Method-Impl) [Thread]
Thread::=	Command
Method-Impl::=	Command
Renaming::=	Set(Method-Renaming)
Method-Renaming::=	Method-Id Method-Id

A hierarchical class is based on two sets of classes which are the inherited and the used classes, respectively. A flat class is the special case in which these two sets are empty. A method whose implementation is defined in the class body has to be declared in the method set part of the class. As a particular case of inheritance, a method which has no implementation in the ancestor class (called *deferred*), can be made *concrete* in an heir class, i.e. an implementation can be introduced. Moreover, methods can also be *redefined* in heir classes.

The syntax of commands is left open since it depends on the particular language being considered. For simplicity, we assume each class expression c to be associated implicitly with a class identifier C, i.e. an environment of classes is not explicitly modelled.

2. The Class Structure

For the semantics definition we use the algebraic specification language ASL [Wir86]. This language supports full first-order logic with partial functions, and in an extension also higher-order logic. For a brief introduction to the syntactic and semantic features of ASL, we refer to the full version of the paper.

First of all, we associate with a class c with name C two signatures, which model the interfaces of a class describing visibility relationships. The signature *Interface*(C, c), called the *(full) interface*, contains all the methods which are visible in a class. These methods may be used inside its body both in the method implementations and in the thread. The signature *User-Interface*(C, c), called the *user interface*, is the restriction of the interface signature to those methods which are visible to client classes of c.

Interface: Class-Id × Class → Method-Sig,
User-Interface: Class-Id × Class → Method-Sig

Here above, Method-Sig denotes the class of the *method signatures*. A method signature is a triple $< BS, CS, M >$ consisting of a set BS of *basic sorts*, a set CS of *class sorts* and a set M of *method operations*, where each method operation consists of a symbol denoting its name, a class sort denoting the class to which the method belongs and a tuple of sorts denoting the types of the parameters and the result type, if any. We use a method signature just as an abbreviated notation for a usual signature (the formal definition of the conversion is given in [BrZ96]).

The full interface of a hierarchical class contains the own methods (the semantic function *Methods-Sig*, whose definition is straighforward, translates a set of methods into a method signature), together with the user interfaces of the server classes and the full interfaces of the ancestors. For the latter ones, the name C replaces the name of the ancestor, reflecting the fact that objects of an heir class are also objects of their ancestor class and thus own all their features. This principle has been applied throughout the semantics definition.

The user interface contains the own methods together with the user interfaces of the ancestor classes. Note that clientship is a non-transitive relation; that means, the user interface of a server class of c is not exported to clients of c.

We give now the semantic clauses defining the full and the user interface of a class.

Let c = Inherits-Uses($\{c_1, \ldots, c_n\}, \{c_1', \ldots, c_m'\}$, attrs, methods, body)
and Name(c_i') = C_i', $i = 1, \ldots, m$.
Interface$(C, c) =_{def}$
Methods-Sig$(C,$ methods$)$ + *Interface*$(C, c_1) + \ldots +$ *Interface*(C, c_n)
+*User-Interface*$(C_1', c_1') + \ldots +$ *User-Interface*(C_m', c_m')
User-Interface$(C, c) =_{def}$
Methods-Sig$(C,$ methods$)$
+*User-Interface*$(C, c_1) + \ldots +$ *User-Interface*(C, c_n)

We assume for simplicity that in a class expression a unique identifier, denoted by Name(c), is associated with each class subexpression c constructed by the operators Inherits-Uses and Flat, and then we define Name(Export(m, d)) $=_{def}$ Name(d) and Name(Rename(r, d)) $=_{def}$ Name(d).

The application of the renaming operator causes a renaming of interfaces. The export operator restricts the user interface; it has no effect on the full interface.

Interface$(C,$ Export(methods, $d)) =_{def}$ *Interface*(C, d)
User-Interface$(C,$ Export(methods, $d)) =_{def}$
< BasicSorts(*User-Interface*(C, d)), ClassSorts(*User-Interface*(C, d)),
Methods(*User-Interface*(C, d))
$\cap$ Methods(*Methods-Sig*$(C,$ methods$)$) >
Interface$(C,$ *Rename*$(r, d)) =_{def}$
rename *Interface*(C, d) **by** *Method-Renaming*(r),
User-Interface$(C,$ Rename$(r, d)) =_{def}$
rename *User-Interface*(C, d) **by** *Method-Renaming*(r)

The semantic function *Method-Renaming* translates a renaming into the corresponding signature morphism which renames (method) signatures. For simplicity, we assume the same syntax for renamings and (method) signature morphisms, and define *Method-Renaming*$(r) =_{def} r$.

We define now the specification associated with a class.

The specification of a hierarchical class is obtained as the sum of the specification of the corresponding concurrent object system *Conc-Obj*(C, c) (outlined in the following section), the specification of basic types BASE, the specifications of ancestors where redefined methods have been forgotten and the specifications of clients, enriched by the operations corresponding to methods and the axioms corresponding to method definitions.

Spec: Class-Id $\times$ Class $\rightarrow$ Spec
Let c = Inherits-Uses($\{c_1, \ldots, c_n\}, \{c_1', \ldots, c_m'\}$, attrs, methods, body)

and Name(c_i') = C_i', $i = 1, \ldots, m$.
Spec(C, c) = $_{def}$
enrich *Conc-Obj*(C, c) + BASE
+ **forget** *Redefined-Methods*(C, body, c_1) **from** *Spec*(C, c_1) + ...
+ **forget** *Redefined-Methods*(C, body, c_n) **from** *Spec*(C, c_n) +
+ *Spec*(C_1', c_1') + ... + *Spec*(C_m', c_m') **by**
opns *Method-Opns*(C, methods)
axioms *Method-Axioms*(C, c)

We assume that *Method-Opns*(C, methods) denotes the set of the operations in the method signature *Method-Sig*(C, methods). The definition of *Method-Axioms* can be found below. The expression *Redefined-Methods*(C, body, c_i) returns the (operations corresponding to) methods of an ancestor class c_i which are redefined in the body of C. The formal definition is straightforward and given in [BrZ96].

The application of the Export operator does not affect a class specification. The Rename operator on classes has its correspondence in the **rename** operator of ASL.

Spec(C, Export(methods, d)) =$_{def}$ *Spec*(C, d)
Spec(C, Rename(r, d)) =$_{def}$ **rename** *Spec*(C, d) **by** *Method-Renaming*(r)

The semantic function *Method-Axioms* translates each method implementation contained in the body of the class into an axiom.

Method-Axioms: Class-Id × Class → Axioms
Method-Axioms(C, Inherits-Uses($\{c_1, \ldots, c_n\}, \{c_1', \ldots, c_m'\}$,
attrs, methods, body)) =$_{def}$
$\bigcup_{(m,\text{com})\in\text{body}} \{$*Method-Axiom*($C, m$, com)$\} \cup \{$*Create-Axiom*($C, c$)$\}$

The translation of a method implementation into an axiom (*Method-Axiom*) and the specification of the default Create method (*Create-Axiom*) depend on the syntax of commands. In [BrZ96], we give some examples of translation of both sequential and concurrent commands.

Deferred methods, i.e. methods without an implementation, have no related axiom in the specification. In other words, the specification of a deferred method is totally *loose*. This means that the interpretation of this method in some model of the class specification is an arbitrary process. That way, the mechanism of deferring methods in ancestors and introducing implementations in heirs has an intuitive semantic counterpart in the mechanism of model set inclusion used in the algebraic framework to model implementation relationship.

3. Objects in a Concurrent Environment

In this section, we outline the definition of the semantic function *Conc-Obj* which gives the specification of the concurrent object system corresponding to a class. As explained in the introduction, the concurrent object system consists, roughly, of two parts: the passive part, i.e. the current states of the existing objects, and the active part, i.e. the processes which execute threads of objects of active classes.

We first give an outline of the definition of the semantic function *Obj* returning the specification of the passive part, which we call *object environment*. In this paper we assume that an object state has a record structure, as it is in most existing object oriented languages. Hence an object environment is determined by the

set of the currently existing objects and by the current interpretation of unary operations modelling attributes. The basic operations which can be performed on an object environment are reading/updating an attribute of an existing object and creating a new object of an existing class. Methods can be seen as operations derived from these primitives, i.e. the effect of a method execution (if terminating) is always a finite sequence of elementary steps each one being a basic operation. Formally:

- for each C class name we have two operations

  $$\text{IsNew}_C : (obj\text{-}env, name_C) \rightarrow boolean$$
  $$\text{Create}_C : (obj\text{-}env, name_C) \rightarrow obj\text{-}env$$

  whose intuitive meaning is testing whether an object name is a new name of class C (i.e. it is not yet used as the name of any existing object) and adding an object of class C to an object environment, respectively;
- for each $X : T$ attribute of a class C we have two operations

  $$\_[\_].X_T : (obj\text{-}env, name_C) \rightarrow \underline{T},$$
  $$\_[\_.X_T \leftarrow \_]_C : (obj\text{-}env, name_C, \underline{T}) \rightarrow obj\text{-}env$$

  whose intuitive meaning is reading and updating the attribute X of an object of class C (here and in what follows $\underline{T}$ denotes T if T is a basic type, $name_T$ otherwise (i.e. if T is a class).

Each class defines a corresponding structure of object environments; for example the following class definitions

```
class C is                  class C' is
attrs                       uses C;
    X : INTEGER             attrs X1 : C; X2 : C;
    Y : BOOLEAN             ...
...                         end class C'
end class  C
```

define object environments in which objects of class `C` and `C'` can be created and there are operations for reading/updating attributes `X`, `Y`, $\texttt{X}_1$, $\texttt{X}_2$.

In order to formalize that correspondence, we introduce, similarly to the notion of method signature mentioned before, the auxiliary notion of *attribute signature* which captures the informal idea of the attribute structure of a class. An *attribute signature* is a 4-tuple $< BS, CS, \leq, A >$ consisting of a set BS of *basic sorts*, a set CS of *class sorts*, a partial ordering $\leq$ on CS, called the *inheritance* relation, and a family A of *attributes* of the form $X\colon\ C \rightarrow T$.

The semantic function *Attr-Sig*, which returns the attribute signature corresponding to a class, is defined in [BrZ96].

In this way, the specification $Obj(C, c)$ of the object environment associated with a class c with name C is defined by

$$Obj(C, c) =_{def} \text{OBJ-ENV}(Attr\text{-}Sig(C, c))$$

where OBJ-ENV is a parametric specification of object environments providing, for a given attribute signature, the operations for reading/updating attributes, creating objects and testing mentioned above. In [BrZ96], we give two different definitions of OBJ-ENV: an abstract version, stating only properties we require an object environment to satisfy, and an implementation (based on modelling object sharing as a congruence over terms denoting objects) which is interesting by itself.

We illustrate now the active part of the concurrent object system associated with a class. We first give a brief introduction to the approach to concurrency based on algebraic transition systems (see [AsR93]), starting from the notion of *labelled transition system.*

A labelled transition system consists of a set of *states* (modelling execution stages), a set of *actions* or *flags* (modelling interaction of the system with the outside) and a set of *transitions* which are triples (s, a, s') where s, s' are states and a is an action, usually written $s \xrightarrow{a} s'$. The intuitive meaning of a transition $s \xrightarrow{a} s'$ is that the system can evolve from the state s into the state s' by performing an interaction with the outside which is labelled by a.

In particular, we use a specialized version of labelled transition systems which we call *concurrent object systems.* A concurrent object system is a labelled transition system with states of the form $< pr_1 \mid \ldots \mid pr_n, \iota, \sigma >$ where $pr_1, \ldots, pr_n$ are states of a lower-level labelled transition system which is called *process transition system*, $pr_1 \mid \ldots \mid pr_n$ is a multiset of process states (the vertical bar is used to suggest parallel composition), ι is a *process information* (an entity keeping trace of information needed for handling interaction of processes, e.g. priorities or locking of variables), σ is the object environment described above. Process information and object environment have no own transitions, but may be changed as a consequence of the transition of processes.

The transitions of the concurrent system are deduced from the transitions of the process transition system by a set of axioms, each one being of the form

$$(*)\quad pr_1 \xrightarrow{a_1} pr_1' \wedge \ldots \wedge pr_n \xrightarrow{a_n} pr_n' \wedge \mathrm{Cond}(a_1, \ldots, a_n, \iota, \sigma) \supset$$
$$< pr_1 \mid \ldots \mid pr_n \mid prset, \iota, \sigma > \Rightarrow < pr_1' \mid \ldots \mid pr_n' \mid prset, \iota', \sigma' >$$

The intuitive meaning of such an axiom is that, if $< pr_1 \mid \ldots \mid pr_n \mid prset, \iota, \sigma >$ is the current state of the concurrent system and each pr_i can evolve into the new process state pr_i' by performing the action a_i, and if moreover the condition $\mathrm{Cond}(a_1, \ldots, a_n, \iota, \sigma)$ holds, then the whole system can evolve in the new state $< pr_1' \mid \ldots \mid pr_n' \mid prset, \iota', \sigma' >$. We assume for simplicity no flags for the transitions of the concurrent system (that means that we consider the concurrent system as a closed system, whose evolution is always internal). Here *prset* denotes a multiset of processes which do not take part in this transition.

Labelled transition systems can be described by algebraic specifications just as usual data types. The algebraic specification corresponding to a particular concurrent object system is obtained instantiating a parametric specification, as shown below.

CONC-OBJ-SYSTEM(PROCESS, PROCESS-INF,
OBJ, TRANSITION-AXIOMS) $=_{def}$
enrich MSET(PROCESS) + OBJ + PROCESS-INF **by**
opns
{ $< \text{-}, \text{-}, \text{-} > \Rightarrow < \text{-}, \text{-}, \text{-} > : (mset(process), process\text{-}inf, obj\text{-}env,$
$mset(process), process\text{-}inf, obj\text{-}env) \rightarrow bool\}$
axioms TRANSITION-AXIOMS

Here we assume that PROCESS is the specification of the process transition system, where *process* is the sort of the states and the transition relation is denoted by $\rightarrow$ (not to be confused with $\Rightarrow$ which denotes the transition relation of the whole system), MSET is a parametric specification of multisets, OBJ and PROCESS-INF are the specifications of the object environment and of the process

information, respectively, and TRANSITION-AXIOMS is a set of axioms each one being of the form (*).

The specification of the concurrent object system associated with a class can now be obtained instantiating the above parametric specification; each actual parameter is the result of a semantic function (defined in [BrZ96]).

$$\begin{aligned}&\mathit{Conc\text{-}Obj}(C,c) =_{def}\\ &\quad \text{CONC-OBJ-SYSTEM}(\mathit{Process}(C,c), \mathit{Process\text{-}Inf}(C,c),\\ &\qquad \mathit{Obj}(C,c), \mathit{Transition\text{-}Axioms}(C,c))\end{aligned}$$

As an example, we show below the definition of the transition axioms in the case in which the language has no concurrent features, hence the only process actions are "standard", i.e. related to reading/updating attributes and creating new objects. In the case of a language with significant concurrent features, the definitions below have to be extended (see [BrZ96]).

$$\mathit{Transition\text{-}Axioms}(C,c) =_{def} \mathit{Standard\text{-}Transition\text{-}Axioms}(\mathit{Attr\text{-}Sig}(C,c))$$

$$\mathit{Standard\text{-}Transition\text{-}Axioms}: \text{Attr-Sig} \to \text{Axioms}$$

$$\mathit{Standard\text{-}Transition\text{-}Axioms}(A\Sigma) =_{def}$$

$$\{pr \xrightarrow{n.\text{ASSIGN-X}_T(x)} pr' \supset$$
$$< pr \mid prset, \iota, \sigma > \Rightarrow < pr' \mid prset, \iota, \sigma[n.X_T \leftarrow x] >,$$

$$pr \xrightarrow{n.\text{GET-X}_T(x)} pr' \wedge \sigma[n].X_T = x \supset < pr \mid prset, \iota, \sigma > \Rightarrow < pr' \mid prset, \iota, \sigma >$$
$$\mid X : C \to T \in \text{Attrs}(A\Sigma)\}$$

$$\cup\{pr \xrightarrow{n.\text{CREATE}_C(thread)} pr' \wedge \text{IsNew}_C(\sigma, n) \wedge \text{Is}_C(n) \supset$$
$$< pr \mid prset, \iota, \sigma > \Rightarrow < pr' \mid prset \mid thread, \iota, \text{Create}_C(\sigma, n) >$$
$$\mid C \in \text{ClassSorts}(A\Sigma)\}$$

4. Concluding Remarks

In the preceding sections, an algebraic semantic framework for object oriented programming languages with concurrency has been presented. A class is modelled by an algebraic specification which describes in a loose approach the objects of the class. The use of the algebraic specification language ASL enables us to model structured classes in a compositional way, based on the structuring operators on specifications. Concerning the model of objects, we consider a general framework of objects in a concurrent environment. The methods of a class are described by processes acting on the passive parts of objects which are determined by the attribute structure of the class. Our object model describes typical object oriented features like object sharing, inheritance polymorphism and dynamic binding of methods.

The work done in this paper has been and will be continued in at least two directions.

First, an integration of abstract specification facilities into an object oriented development method is desirable. By the algebraic semantics, a common framework for classes and abstract data types has been provided. This common framework enables an integration of (implementation-oriented) classes and (property-oriented) specifications via a formal correctness relation. The relation

between a sequential object oriented approach and algebraic specifications has been deeply investigated in [Bre91].

For what concerns the model of object systems, a main result of this paper is to show that it is possible to insert in a modular way a model of the dynamic evolution in the overall algebraic framework, giving a compositional interpretation of classes. That is achieved by what we call the "state-as-term" approach, i.e. modelling the current configuration of a dynamic system as a term of a special sort and the transformations from a configuration to another as operations (predicates). This idea is extremely fruitful and at the root of a variety of formalisms (see [AsR93] for references). Nevertheless, somebody might not like the fact that in this approach there is no strong distinction between static and dynamic aspects of a system.

Indeed some work has been done in extending in a more proper way the algebraic treatment to the dynamic case, introducing a new kind of structures which are a generalization of usual algebras, called d-oids [AsZ95]. In this approach, the "state-as-algebra" view is taken, seeing configurations as usual algebras and transformations as operations at a higher level, i.e. operations handling algebras. An interesting further work in that direction would be to give within this new framework a compositional model of classes as the one proposed in this paper.

Acknowledgment

The authors are grateful to their colleagues in the DRAGON project, especially to Martin Wirsing, Rolf Hennicker (for the Münich side), Egidio Astesiano and Gianna Reggio (for the Genova side), for many helpful discussions and comments.

References

[AdB86] America, P., de Bakker, J., Kok J. and Rutten, J. Denotational semantics of a parallel object-oriented language. Tech. Report CS-R8626, Amsterdam, 1986.

[AsR93] Astesiano, E. and Reggio, G. Algebraic specification of concurrency. In *Recent Trends in Data Type Specification (Dourdan, France, August 1991)*, LNCS 655, 1993.

[AsZ95] Astesiano, E. and Zucca, E. D-oids: a model for dynamic data-types. *Mathematical Structures in Computer Science*, 5(2):257–282, June 1995.

[BKS88] Bayan, R., Kaag, F., Spasojevic, A., Di Maio, A., Cardigno, C., Gatti, S., Crespi Reghizzi, S., Astesiano, E., Giovini, A., Gautier, B., Atkinson, C., Wirsing, M., Hennicker, R. and Zucca, E. An object oriented approach to dragon. Deliverable of the project "DRAGON" (Esprit 1550), 1988.

[Bre91] Breu, R. *Algebraic Specification Techniques in Object Oriented Programming Environment.* LNCS 562, 1991.

[BrZ89] Breu, R. and Zucca, E. An algebraic compositional semantics of an object oriented notation with concurrency. In *Foundations of Software Technology and Theoretical Computer Science 1989*, LNCS 405, 1989.

[BrZ96] Breu, R. and Zucca, E. An algebraic semantic framework for object oriented languages with concurrency. *Formal Aspects of Computing*, 8(E):289, 1996. (For details see page 705).

[Str86] Strom, R. A comparison of the object-oriented and process paradigm. *SIGPLAN Notices*, 21(10):88–97, 1986.

[Weg87] Wegner, P. Dimensions of object based language design. In OOPSLA '87, 1987.

[Wir86] Wirsing, M. Structured algebraic specifications: A kernel language. *Theoretical Computer Science*, 43:123–250, 1986.

[Wol87] Wolczko, M. Semantics of Smalltalk-80. In *ECOOP '87*, LNCS 276, 1987.

Received April 1990

Accepted in revised form July 1996 by C. B. Jones