OOD Frameworks in Component-based Software Development in Computational Logic

Kung-Kiu Lau[1] and Mario Ornaghi[2]

[1] Department of Computer Science, University of Manchester
Manchester M13 9PL, United Kingdom
kung-kiu@cs.man.ac.uk
[2] Dipartimento di Scienze dell'Informazione, Universita' degli studi di Milano
Via Comelico 39/41, 20135 Milano, Italy
ornaghi@dsi.unimi.it

Abstract. Current Object-oriented Design (OOD) methodologies tend to focus on objects as the unit of reuse, but it is increasingly recognised that frameworks, or groups of interacting objects, are a better unit of reuse. Thus, in next-generation Component-based Development (CBD) methodologies, we can expect components to be frameworks rather than objects. In this paper, we describe a preliminary attempt at a formal semantics for OOD frameworks in CBD in computational logic.

1 Introduction

Most of the existing (semi-formal) Object-oriented Design (OOD) methods such as Fusion [4, 6] and Syntropy [5] use classes or objects as the basic unit of design or reuse. These methods are based on the traditional view of an object, as shown in Figure 1, which regards an object as a closed entity with one fixed role.

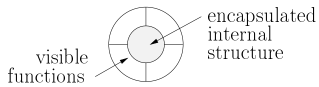

Fig. 1. Traditional view of an object.

This, however, does not reflect the nature of objects (and classes that describe them) in practical systems. In particular, objects tend to have more than one role in more than one context or framework. Consequently, it is increasingly recognised that classes are not the best focus for design (see e.g. [13, 25]). Typical design artefacts are rarely just about one object, but about groups of objects and the way they interact.

Therefore in the next generation of component-based development (CBD) methodologies, we can expect components to be not just objects, but groups of (interacting) objects. In the CBD methodology *Catalysis* [7, 23, 16], for example, objects can have multiple roles in different OOD frameworks, and are formed by composing frameworks, as depicted in Figure 2.

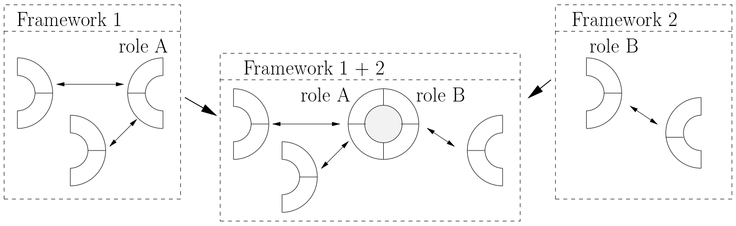

Fig. 2. Objects by composing OOD frameworks.

In computational logic, we have formulated a formal approach to program development [18, 19, 14], in which we use what we call *specification frameworks* to formalise problem domains. These frameworks have purely declarative (model-theoretic) semantics. In this paper we will show (how) we can extend such frameworks to make them correspond to OOD frameworks. The latter are groups of (interacting) objects with local states. Our extension thus has to introduce states and hence objects. However, in the paper we will not address temporal aspects such as state transitions.

2 Overview of Our Approach to Program Development

Our approach to program development in computational logic is based on a three-tier formalism (with model-theoretic semantics) illustrated in Figure 3.

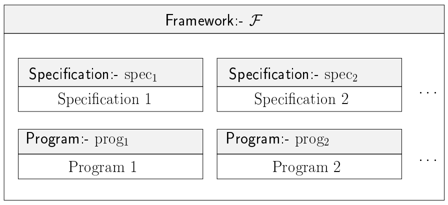

Fig. 3. A three-tier formalism.

At the top level, we have *specification frameworks* (or *frameworks* for short).[1] A framework is a first-order theory that axiomatises a problem domain. Model-theoretic semantics for frameworks is based on *isoinitial models*.

In the middle, we have *specifications*. Specifications are always given in the context of a framework. A specification is a set of formulas that define a new relation or function symbol in terms of the symbols of the framework.

At the bottom level are *(correct) programs*. For every specified relation r, a framework may contain a correct program for computing r. We have a model-theoretic definition of correctness, based on *steadfastness* [22].

Frameworks may be parametric, thus supporting classes, subclasses, inheritance, and framework composition. In particular, a composite framework inherits all the properties, specifications and correct programs of its components; correctness is preserved by framework composition, and the inherited programs can be composed to yield new correct (composed) programs. Thus, frameworks capture the declarative semantics of reusable components needed by CBD methodologies, and of their correctness and their composition [21].

However, so far our frameworks are *static*, i.e. they are immutable theories. In this paper, we will introduce states into our frameworks, and we will show that we thus obtain a formalism suitable for next-generation CBD: in particular, our approach provides a formalisation of OOD frameworks and their composition.

3 Specification Frameworks

Specification frameworks are defined in the context of many sorted first-order logic with identity. In this section we introduce and formalise such frameworks. We will attempt to give a self-contained treatment, because our existing formalisation of frameworks does not contain or emphasise the key aspects that are required for introducing states and objects. So we wish to hereby warn the reader about the somewhat heavy, formal material that lies ahead.

We begin by recalling some relevant terminology and notation.

A many-sorted first-order *signature* Σ consists of a set S of *sort symbols*, a set F of *function declarations* and a set R of *relation declarations*. A function declaration is of the form $f : [a] \to s$, where f is a function symbol, $[a]$ is its arity[2] and s is its sort. Function symbols of empty arity are called constant symbols. A relation declaration is of the form $r : [a]$, where r is a relation symbol and $[a]$ is its arity.

The meaning of the symbols of a signature Σ is given by *Σ-structures* (or *Σ-interpretations*). As usual, the meaning of a sort symbol s in a structure i is a set, called the *domain* of s, while function and relation declarations[3] are interpreted in i as functions and relations. The meaning of a (declaration of a) symbol σ in a structure i will be denoted by σ^{i}.

[1] To avoid confusion, we will always use frameworks to refer to specification frameworks only. OOD frameworks will always be called OOD frameworks.

[2] $[a]$ is a list of sort symbols.

[3] We interpret declarations, in order to allow overloading.

A Σ-structure i is *reachable* if, for every sort symbol s of Σ and every element α of the domain s^{i}, there is a ground term t such that $val_{\mathsf{i}}(t) = \alpha$, where $val_{\mathsf{i}}(t)$ is the value of t in the interpretation i, and is recursively defined in the usual way.

For a set Ax of Σ-sentences, $\mathsf{i} \models Ax$ means that every sentence of Ax is true in i. If $\mathsf{i} \models Ax$, then i is called a *model* of Ax.

In first-order logic with identity, overloaded identity $=: (s, s)$ (for every sort s) is always understood, and $=: (s, s)$ is interpreted, in *every* interpretation i, as the identity over s^{i}. Moreover, the standard identity axioms are always understood.

In this context, a *framework* $\mathcal{F} = \langle \Sigma, Ax \rangle$ is composed of a signature Σ and a finite or recursive set Ax of Σ-axioms.

The purpose of a framework is to axiomatise a problem domain and to reason about it, with the help of an interactive theorem prover. In our approach, a *problem domain* contains the ADT's and classes needed to define the objects of the application at hand.

We distinguish between *closed* and *open* frameworks. The relationship between open and closed frameworks plays a crucial role in our interpretation of objects. Roughly speaking, in object-oriented programming terminology, open frameworks represent classes, and closed frameworks represent their instances, i.e. objects.

3.1 Closed Frameworks

In a closed framework $\mathcal{F} = \langle \Sigma, Ax \rangle$ the purpose of the axioms Ax is to 'completely' characterise the meaning of the symbols in the signature Σ. This can be achieved by singling out an *intended model* i of Ax. We will use reachable isoinitial models as intended models.

Definition 1 (Isoinitial Models). Let Ax be a set of Σ-axioms. A Σ-structure i is an *isoinitial model* of Ax iff, for every other model m of Ax, there is one isomorphic embedding $i : \mathsf{i} \to \mathsf{m}$.

Definition 2 (Closed Frameworks). A framework $\mathcal{F} = \langle \Sigma, Ax \rangle$ is *closed* iff there is a reachable isoinitial model i of Ax, called the *intended model* of Ax.

Thus, our closed frameworks are *isoinitial theories*, namely theories with reachable isoinitial models. They are similar to *initial theories*, which axiomatise reachable initial models. The latter are popular in algebraic abstract data types and specifications [30]. The difference is that initial models use homomorphisms, instead of isomorphic embeddings. We briefly discuss some relevant consequences here (for a more detailed comparison, see [2]).

Let Ax be a set of Σ-axioms with a reachable model i, and consider the following properties:

(i) For every ground atom A, $\mathsf{i} \models A$ iff $Ax \vdash A$.
(ii) For every ground literal L, $\mathsf{i} \models L$ iff $Ax \vdash L$.

(iii) For every ground atom A, $Ax \vdash A$ or $Ax \vdash \neg A$.

$\mathfrak{i}$ is an initial model of Ax if and only if (i) holds, while $\mathfrak{i}$ is an isoinitial model of Ax if and only if (ii) holds. Thus, in an initial theory the falsity of a ground atom A corresponds to the non-provability of A, while in isoinitial theories it corresponds to the provability of $\neg A$. That is, isoinitial theories behave more properly, with respect to negation.

Since negation is important for reasoning about specification and correctness, we have chosen isoinitial theories. A second reason for this choice is that condition (ii) can be replaced by condition (iii), called *atomic completeness*. This is a purely proof-theoretical condition, i.e., $\mathfrak{i} \models \ldots$ disappears. This makes it more manageable. For example, we have used it to link isoinitiality to termination of logic programs [20], and in [26] a constructive proof-theory for isoinitiality has been based on atomic completeness. A third reason lies in our distinction between closed and open frameworks. Closed frameworks are required to be rich enough, to have an intended model. If such a model does not exist, then they are open. If we use initial theories, this distinction becomes very weak, as shown by the following example.

Example 1. Consider a framework for natural numbers, of the form:

Framework $\mathcal{NAT}$;

SORTS: Nat;

DECLS: $0 : [\,] \to Nat$;
$s : [Nat] \to Nat$;

AXIOMS: Ax;

where, as we pointed out before, $=$ and its standard axioms are understood.

$\mathcal{NAT}$ is a closed initial framework, even for an empty set of axioms (i.e., with $Ax = \emptyset$). Indeed, the Herbrand Universe $\mathcal{H}(0, s)$ generated by $\{0, s\}$ is a reachable initial model of $\emptyset$. This shows that very weak frameworks are accepted as being closed. In particular, in such frameworks we cannot prove negated facts, like, in this example, $\neg s(0) = s(s(0))$.

By contrast, to get an isoinitial theory axiomatising $\mathcal{H}(0, s)$, we need the following *freeness axioms* for $\{0, s\}$:

$$Ax = \{\forall x \,.\, \neg 0 = s(x),\ \forall x, y \,.\, s(x) = s(y) \to x = y\}$$

These state that different ground terms represent different numbers, i.e., the framework works properly with respect to negation.

A framework with an isoinitial model can be constructed incrementally, by *adequate expansions*, namely expansions by new symbols δ, that preserve the meaning of the old signature:

Definition 3 (Adequate Expansions of Closed Frameworks). $\mathcal{F}' = \langle \Sigma + \delta, Ax \cup D_\delta \rangle$ is an *adequate expansion* of $\mathcal{F} = \langle \Sigma, Ax \rangle$ if $\mathcal{F}'$ is a closed framework and its isoinitial model is a $(\Sigma + \delta)$-expansion[4] of that of $\mathcal{F}$.

[4] As usual, a $(\Sigma + \delta)$-expansion of a Σ-interpretation $\mathfrak{i}$ is a $(\Sigma + \delta)$-interpretation $\mathfrak{i}'$ that coincides with $\mathfrak{i}$ over Σ.

In an expansion, the axioms D_δ are called a *definition* of δ. A definition is adequate iff the corresponding expansion is. Adequate definitions are a very useful tool for incremental construction of frameworks, starting from a small kernel. There are various kinds of adequate definitions.

Example 2. We can expand the kernel $\mathcal{NAT}$ of natural numbers by useful functions and/or predicates. For example, the following is an adequate expansion of $\mathcal{NAT}$.

Framework $\mathcal{PA}$;

IMPORT: $\mathcal{NAT}$;

$$
\begin{array}{lll}
\text{NEW:} & + : [nats, Nat] \to Nat; & \text{WHERE } x + 0 = x \\
& & \phantom{\text{WHERE }} x + s(y) = s(x + y) \\
& * : [nats, Nat] \to Nat; & \text{WHERE } x * 0 = 0 \\
& & \phantom{\text{WHERE }} x * s(y) = x * y + x \\
& 1 : [\,] \to Nat; & \text{WHERE } 1 = s(0) \\
& \leq : [Nat, Nat]; & \text{WHERE } x \leq y \leftrightarrow \exists z \, . \, x + z = y
\end{array}
$$

All the new symbols are introduced by adequate definitions: $+$ and $*$ by (adequate) *recursive* definitions, and 1 and $\leq$ by (adequate) *explicit* definitions.

Of course, we need methods for proving the adequacy of a definition. Let us briefly consider this issue for *explicit* definitions, which suffice for our purposes. In a framework $\mathcal{F} = \langle \Sigma, Ax \rangle$, explicit definitions have the following general forms:

- An *explicit definition* of a new relation r is a $(\Sigma + r)$-formula of the form $\forall x \, . \, r(x) \leftrightarrow R(x)$ where the *defining formula* $R(x)$ is a Σ-formula.
- An *explicit definition* of a new function f is a $(\Sigma + f)$-formula of the form $\forall x \, . \, F(x, f(x))$ where the *defining formula* $F(x, y)$ is a Σ-formula such that $Ax \vdash \forall x \exists ! y \, . \, F(x, y)$.

In [17] we have shown how adequate explicit definitions can be established by means of programs synthesis. In [26] it is shown that constructive proof-systems can be used to prove adequacy. For this paper, it suffices to know that if the defining formula is quantifier-free, then an explicit definition is *always* adequate.

3.2 Open Frameworks

An open framework does not have an isoinitial model, since its axioms leave open the meaning of some symbols of the signature, that we call *open* symbols. Non-open symbols will be called *defined* symbols.

We denote an open framework by $\mathcal{F}(\Omega) = \langle \Sigma, Ax \rangle$ where Ω are the open symbols. Since Ω may contain open sorts, we consider Ω-reachable models, where Ω-reachability is reachability in an expansion containing a new constant for each element of each open sort.[5] Moreover, we require that the axioms be sufficient

[5] We do not worry about the cardinality of the new constants.

to characterise the meaning of the defined symbols, in terms of the open ones. This corresponds to adequacy, which, for open frameworks, can be introduced as follows.

Definition 4 (Ω-equivalence). Let $\mathcal{F}(\Omega) = \langle \Sigma, Ax \rangle$ be an open framework, and let i and i' be two models of Ax. We say that i and i' are Ω-equivalent if, for every open symbol $\sigma \in \Omega$, $\sigma^{\mathsf{i}} = \sigma^{\mathsf{i}'}$.

Ω-equivalence is an equivalence relation. Therefore it partitions the class of models of $\mathcal{F}(\Omega) = \langle \Sigma, Ax \rangle$ into *Ω-equivalence classes*, or *Ω-classes* for short. In an Ω-class, we will consider Ω-isomorphic embeddings, i.e., isomorphic embeddings that behave as the identity function over the (possible) open sorts. Ω-isoinitial models are defined like isoinitial models, but they use Ω-isomorphic embeddings instead of isomorphic embeddings.

Definition 5 (Adequate Open Frameworks). $\mathcal{F}(\Omega) = \langle \Sigma, Ax \rangle$ is an *adequate open framework* if and only if every Ω-class contains an Ω-reachable Ω-isoinitial model.

Thus an adequate open framework axiomatises a class of intended models, namely the Ω-isoinitial models contained in the Ω-classes. An Ω-isoinitial model j of an Ω-class represents the intended meaning of the defined symbols, when the open symbols are interpreted as in j itself. We say that j is an *intended model* of the open framework. Since the framework is open, we have many non-isomorphic intended models.

Example 3. The following example is trivial, but it serves to explain ideas and problems simply and clearly.

Framework $\mathcal{TRIV}(X, q)$;

SORTS: X, s;

DECLS: $j : [X] \to s$;
$p : [X]$;
$q : [X, X]$;

AXIOMS: $j(x) = j(y) \to x = y$
$p(x) \leftrightarrow \exists y \,.\, q(x, y)$;

Let us interpret, for example, X as the term structure $\{0, s(0), s(s(0)), \ldots\}$ of the natural numbers, and $q(x, y)$ as the predicate $x > y$. We get the interpretation $\{j(0), j(s(0)), j(s(s(0))), \ldots\}$ of s and $x \neq 0$ of $p(x)$. Indeed, this interpretation is an X-reachable isoinitial model in the (X, q)-class interpreting X and q as before, i.e., it is an intended model.

We can prove that, if a framework $\mathcal{F}(\Omega)$ is adequate, then, for every intended model j, there is a suitable closed expansion of $\mathcal{F}(\Omega)$ that axiomatises it. The new axioms of such an expansion essentially coincide with all the Ω-formulas that are true in the interpretation of the parameters. However, since we need axiomatisations that can be managed by a theorem prover, the problem is to state which axioms are needed to get the desired closed instances.

Example 4. Consider the framework $\mathcal{TRIV}(X, q)$ of Example 3. To fix an interpretation of the open symbols X and q, a completion procedure has to add a closed set Ax_{new} of new axioms, whose isoinitial model corresponds to that interpretation. Furthermore, the new axioms must satisfy the following requirement:

$$\text{for every ground } t,\ Ax_{new} \vdash \exists y \,.\, q(t, y) \text{ or } Ax_{new} \vdash \neg\exists y \,.\, q(t, y) \qquad (1)$$

which ensures atomic completeness for the defined symbol p.

In this example, condition (1) guarantees that the new axioms complete the open framework into a closed one. Conditions of this kind will be called *constraints*. We have the following situation:

> An open framework explicitly contains its *constraints* (if any), and a *completion procedure* is a procedure to add axioms that satisfy the constraints. By a completion procedure, we can build many closed expansions, that we call *instances*.

Also, adequate open frameworks can be built incrementally, starting from a small, well understood kernel, and defining new symbols by *adequate definitions*, namely definitions that are inherited in a sound way (for more details, see [17, 14]).

The existence of many instances implies *reusability*: every theorem, expansion, specification or correct program developed in an open framework are inherited, and can be *reused* in all its instances. Reusability was an important motivation behind open frameworks and has been used in [22] to treat parametric frameworks. In this paper, we will also use them to define objects with states. To this end, we introduce a completion procedure based on explicit definitions.

We shall assume that we start with an open framework that already contains a closed subframework $\mathcal{G}$, i.e., is of the form $\mathcal{F}(\Omega) + \mathcal{G}$. This assumption is not restrictive, since it suffices to import $\mathcal{G}$ in $\mathcal{F}(\Omega)$. We can use $\mathcal{G}$ to close the meaning of the open relations and functions, as follows:

- *Sort closure.* The operation:

  $$\text{CLOSE} \quad X \quad \text{BY} \quad s$$

  renames the open sort X by a sort s of the closed subframework $\mathcal{G}$. This operation generates the framework $\mathcal{F}'(\Omega') + \mathcal{G}$, where $\mathcal{F}'(\Omega')$ is obtained by renaming X by s in $\mathcal{F}(\Omega)$.
- *Relation closure.* The operation:

  $$\text{CLOSE} \quad r \quad \text{BY} \quad \forall x \,.\, r(x) \leftrightarrow R(x)$$

  defines the meaning of an open relation $r(x)$ by a formula $R(x)$ of the closed subframework $\mathcal{G}$. We require that the declaration of r contains only sorts of $\mathcal{G}$, and $\forall x \,.\, r(x) \leftrightarrow R(x)$ is an *adequate* explicit definition of r in the closed framework $\mathcal{G}$.

– *Function closure.* The operation:

$$\text{CLOSE} \quad f \quad \text{BY} \quad \forall x \,.\, F(x, f(x))$$

defines the meaning of an open function $f(x)$ by a formula $F(x, y)$ of the closed subframework $\mathcal{G}$. We require that the declaration of f contains only sorts of $\mathcal{G}$, and $\forall x \,.\, F(x, f(x))$ is an *adequate* explicit definition of f in $\mathcal{G}$.

To close all the open symbols of an open framework $\mathcal{F}(\Omega)$ by a closed framework $\mathcal{G}$, we can start from $\mathcal{F}(\Omega) + \mathcal{G}$ and proceed as follows:

(i) Close the open sorts, and obtain a framework $\mathcal{F}'(\Omega') + \mathcal{G}$, where, in the open part $\mathcal{F}'(\Omega')$, all the open sorts of $\mathcal{F}(\Omega)$ have been renamed by closed sorts of $\mathcal{G}$.
(ii) Then close the relation and function symbols, and obtain a framework $\mathcal{F}'(\Omega') + \mathcal{G}'$, where $\mathcal{G}'$ is the (adequate) expansion of $\mathcal{G}$ by the explicit definitions used to close the open relations and functions.

We can prove the following theorem (we omit the proof, for conciseness).

Theorem 1. *Let $\mathcal{F}(\Omega)$ be an adequate open framework and $\mathcal{G}$ be a closed framework. Let $\mathcal{F}'(\Omega') + \mathcal{G}'$ be the final result of steps (i) and (ii) above. If $\mathcal{G}'$ satisfies the (possible) constraints of $\mathcal{F}'(\Omega')$, then $\mathcal{F}'(\Omega') + \mathcal{G}'$ is an adequate expansion of $\mathcal{G}'$ and its isoinitial model is an isoinitial model of the Ω'-class[6] that interprets Ω' as the isoinitial model of $\mathcal{G}'$.*

That is, the closed instance of $\mathcal{F}'(\Omega') + \mathcal{G}'$ is an adequate expansion of $\mathcal{G}$, and its intended model is defined according to $\mathcal{F}'(\Omega')$. Indeed, it coincides with the intended model of $\mathcal{F}'(\Omega')$ that interprets the sorts of Ω' according to $\mathcal{G}$, and the functions and relations of Ω' according to the explicit definitions used to close them.

Example 5. We can close $\mathcal{TRIV}(X, q)$ as follows:

$$\begin{array}{lll} \multicolumn{3}{l}{\text{IN } \mathcal{TRIV}(X, q):} \\ \multicolumn{3}{l}{\text{IMPORT } \mathcal{PA};} \\ \text{CLOSE:} & X & \text{BY} \quad Nat; \\ & q & \text{BY} \quad \forall y \,.\, q(x, y) \leftrightarrow y < x. \end{array}$$

The result is the following closed framework:

Framework $\mathcal{TRIV}(Nat, q) + \mathcal{PA}'$

$$\begin{array}{ll} \text{IMPORT} & \mathcal{PA}; \\ \text{SORTS:} & s; \\ \text{DECLS:} & j : [Nat] \to s; \\ & p : [Nat]; \\ & q : [Nat, Nat]; \\ \text{AXIOMS:} & q(x, y) \leftrightarrow y < x \\ & j(x) = j(y) \to x = y \\ & p(x) \leftrightarrow \exists y \,.\, q(x, y); \end{array}$$

[6] $\mathcal{F}'(\Omega')$ is an adequate open framework, since it is a renaming of $\mathcal{F}(\Omega)$.

$\mathcal{PA}'$ is the expansion of $\mathcal{PA}$ by the explicit definition $q(x,y) \leftrightarrow y < x$, used to close q. According to $\mathcal{PA}'$, Nat is interpreted as the set $\{0, s(0), s(s(0)), \ldots\}$ of natural numbers, and $q(x,y)$ as $y < x$. The corresponding intended model of $\mathcal{TRIV}(Nat,q)$ interprets s as $\{j(0), j(s(0)), j(s(s(0))), \ldots\}$ and $p(x)$ as $x \neq 0$ (see Example 3). The same interpretation is given by the isoinitial model of $\mathcal{TRIV}(Nat,q) + \mathcal{PA}'$, i.e., $\mathcal{TRIV}(Nat,q) + \mathcal{PA}'$ is an adequate expansion of $\mathcal{PA}'$, where the interpretation of p and s is that defined by $\mathcal{TRIV}(Nat,q)$, when the parameters are interpreted according to $\mathcal{PA}'$.

To conclude this section, we remark that Example 5 corresponds to parameter passing which replaces X by Nat and q by $<$. The only difference is that both q and $<$ are in the final signature.[7] That is, our closure operations can be used to implement parameter passing. On the other hand, our mechanism is more general, since we could close open relations and functions by any kind of adequate, possibly recursive, definitions.

4 Classes and Objects

In the previous section we have presented the foundations for (specification) frameworks. In this section, we will show that these frameworks can be used to model both classes and objects in the terminology of object-oriented programming.

We shall use only frameworks that are adequate: frameworks that represent ADTs are called *ADT-frameworks*, and those that represent classes and objects are called *OBJ-frameworks*, respectively.

4.1 ADT-frameworks

A *closed ADT-framework* is a closed framework $\mathcal{F}$ that axiomatises an abstract data type (ADT). For example, the closed framework $\mathcal{PA}$ is a closed ADT-framework. It axiomatises the data type of natural numbers.

An *open ADT-framework* is an open framework $\mathcal{F}(\Omega)$, that axiomatises a class of intended ADT's. The intended ADT's are the intended models of $\mathcal{F}(\Omega)$, as defined in the previous section.

The *instances of* an open ADT-framework can be built by closing the open symbols in the way explained in the previous section. Once an instance is obtained, it is and remains static, i.e., ADT's cannot evolve.

Example 6. The following is the kernel of an adequate (open) framework of lists:

[7] However, by eliminability of explicit definitions, we could eliminate q if we wanted.

ADT-Framework $\mathcal{LIST}(X, \lhd)$

IMPORT: $\mathcal{PA}$;

SORTS: $X, List$;

DECLS:
$$\begin{array}{rl} nil & : [\,] \rightarrow List; \\ \cdot & : [X, List] \rightarrow List; \\ nocc & : [X, List] \rightarrow Nat; \\ elemi & : [List, Nat, X]; \\ \lhd & : (X, X); \end{array}$$

AXIOMS:
$$\begin{array}{l} free(nil, .); \\ nocc(x, nil) = 0; \\ a = b \rightarrow nocc(a, b.L) = s(nocc(a, L)); \\ \neg a = b \rightarrow nocc(a, b.L) = nocc(a, L); \\ elemi(L, i, a) \leftrightarrow \exists x : X \,.\, \exists N : List \,.\, \exists j : Nat \,. \\ \quad (i = 0 \wedge L = a.N) \vee (i = s(j) \wedge L = x.N \wedge elemi(N, j, a)); \end{array}$$

CONSTRS: $TotOrd\lhd$.

where the abbreviations $free(nil, \cdot)$ and $TotOrd\lhd$ stand, respectively, for the freeness axioms of nil and $\cdot$, and the total ordering axioms for the *open* symbol $\lhd$.

The axioms characterise the *defined* symbols $List$, nil, $\cdot$, $nocc$ and $elemi$ in terms of the open ones, while the constraints are used to constrain the open symbol $\lhd$ to be a total ordering.

Once we have fixed an interpretation of the open symbol X, we get the structure of finite lists with elements in X. If, in this structure, we interpret $\lhd$ as a total ordering, then we get an X-reachable isoinitial model in the class of models of $\mathcal{LIST}(X, \lhd)$ that have the same interpretation of X and $\lhd$.

4.2 OBJ-frameworks

A OBJ-framework is also an adequate open framework $\mathcal{F}(\Omega)$. Its *closed instances* can be built in the way explained in the previous section. An OBJ-framework $\mathcal{F}(\Omega)$ will also be called a class, and its instances will be called *objects* of that class. The axioms used to close $\mathcal{F}(\Omega)$ into an object represent the *state* of the object, and are called *state axioms*. State axioms can be updated, i.e. an object is a dynamic entity.

When an object is created, the symbols of the signature of the class may be renamed, so that different objects have different signatures. We will use the following convention:

- The signature of an OBJ-framework may contain dotted symbols, namely symbols whose names begin with a dot.
- Each object has a *name*, and different names represent different objects. When creating a new object with name n, dotted symbols are renamed by prefixing their names with n. Therefore, the signatures of two different objects will contain different dotted symbols. Dotted symbols are used for

dynamic symbols, whose axioms and meaning may evolve in different ways for different objects.
– Non-dotted symbols are left unchanged. They are used for the static ADT-parts of the OBJ-framework (typically imported from pre-defined ADT-frameworks) that will be the same in all objects. For example, the ADT of natural numbers is always the same, in any object that uses it.

In an OBJ-framework $\mathcal{F}(\Omega)$, dotted symbols are dynamic. Therefore they must be open, and as such we do not need to list them as open symbols, i.e., we can write $\mathcal{F}$ instead of $\mathcal{F}(\Omega)$. This reflects the different uses ADT and OBJ-frameworks are intended for. In the former, the open symbols act as *parameters*, that can be closed by *external* closed frameworks, giving rise to immutable closed ADT-frameworks. In the latter, dynamic symbols act as a *state*. They can be dynamically closed *within* the framework itself, giving rise to dynamic objects.

Example 7. An *OBJ*-framework for car objects could be:

$$\begin{array}{ll}
\multicolumn{2}{l}{\textbf{OBJ-Framework } \mathcal{CAR}(.km, .option);} \\
\textsc{import:} & \mathcal{INT}, year96; \\
\textsc{decls:} & \quad .km : [\,] \rightarrow Int \\
 & .option : [year96.opts] \\
\textsc{constrs:} & .km \geq 0
\end{array}$$

where $\mathcal{INT}$ is a predefined ADT of integers, and $year96$ is an object that contains the sort $year96.opts$ of the possible options for a car in the year 96.

An object of class $\mathcal{CAR}$ is created, for example, by:

$$\begin{array}{llll}
\multicolumn{4}{l}{\textsc{new } spider : \mathcal{CAR};} \\
\textsc{close:} & spider.km & \textsc{by} & spider.km = 25000; \\
 & spider.option & \textsc{by} & spider.option(x) \leftrightarrow x = Airbag \lor x = AirCond.
\end{array}$$

where $spider.km = 25000$ is the explicit definition that closes the constant $spider.km$, while $spider.option(x) \leftrightarrow x = Airbag \lor x = AirCond$ is the explicit definition that closes the predicate $spider.option(x)$.

The state of a spider object can be updated, by redefining its state axioms:

$$\begin{array}{l}
\textsc{update } spider : \mathcal{CAR}; \\
spider.km = 27000 \\
spider.option(x) \leftrightarrow x = Airbag \lor x = AirCond
\end{array}$$

As we can see, the constant $spider.km$ has been changed.

When creating and updating objects, we have to guarantee that the explicit definitions used to close the open symbols are adequate, and that the constraints are satisfied. The former condition is automatically satisfied if we use quantifier-free defining formulas. The latter condition, in general, requires a proof. Since proofs are typically time consuming, it is better to design methods for the whole class that guarantee constraint satisfaction.

Example 8. In the previous example, we have used quantifier-free defining formulas. Therefore, adequacy is guaranteed. Methods for the class $\mathcal{CAR}$, that are guaranteed to satisfy the constraints are:

$$create(k, y_1, \ldots, y_k) :- k \geq 0, assert(.km = k),$$
$$assert(\forall x(.optio(x) \leftrightarrow x = y_1 \vee \ldots \vee x = y_k).$$

$$kmupdate(x) :- x > .km, replace((.km = K), (.km = x)).$$

where $replace(A, B)$ is defined by $retract(A).assert(B)$. In the creation method *create*, the constraint $.km \geq 0$ follows from $k \geq 0$. In the updating method *kmupdate*, the new values for $.km$ are required to be greater that the old ones. Since the latter already satisfy the constraint, we are all right.

Note that here the syntax is not relevant. We have not yet fully formalised the dynamic aspects such as state transition (see the discussion in the conclusion). The above examples give only an idea.

5 OOD Frameworks

Having introduced classes and objects in the previous section, in this section we show how we can define OOD frameworks as frameworks containing composite objects, i.e. as composite OBJ-frameworks. We shall do so by formalising composite objects as systems of objects. We then show that our composite OBJ-frameworks indeed correspond to OOD frameworks as used in CBD methodologies like *Catalysis*.

5.1 Systems of Objects

To introduce systems of objects, we need the notion of multiple instance of a set $\mathcal{O}_1, \ldots, \mathcal{O}_n$ of OBJ-frameworks.

Definition 6 (Multiple Instances). Let $\mathcal{O}_1, \ldots, \mathcal{O}_n$ be a set of OBJ-frameworks, and $a_1, \ldots, a_k$ be instances of them.[8] A *multiple instance* $\{a_1, \ldots, a_k\}$ is defined as the union (of the signatures and of the axioms) of all the single instances $a_i, 1 \leq i \leq k$.

To give a semantics of multiple instances, we need a link between the isoinitial models of the individual instances $a_j, 1 \leq j \leq n$, and the isoinitial model of the multiple instance.

Definition 7 (Additivity and Strong Additivity). A set of adequate OBJ-frameworks is *additive* if its multiple instances $\{a_1, \ldots, a_k\}$ satisfy the following properties:

(i) If each $a_j, 1 \leq j \leq n$ satisfies the constraints, then $\{a_1, \ldots, a_k\}$ is a closed framework.

[8] Each OBJ-framework may have 0, 1 or more instances.

(ii) Let i be the isoinitial model[9] of $\{a_1, \ldots, a_k\}$. For every $j, 1 \leq j \leq k$, the isoinitial model i_j of a_j is isomorphically embedded into the reduct of i to the signature of a_j.

It is a *strongly additive* set of OBJ-frameworks if, in (ii), the isoinitial model i_j of a_j is isomorphic to the reduct of i to the signature of a_j.

We have the following theorem:

Theorem 2 (Representativity). *Let $\{a_1, \ldots, a_k\}$ be a multiple instance of a set $\mathcal{O}_1, \ldots, \mathcal{O}_n$ of adequate OBJ-frameworks.*

If $\mathcal{O}_1, \ldots, \mathcal{O}_n$ is strongly additive, then every sentence of the language of $a_j, 1 \leq j \leq k$, is true in its isoinitial model if and only if it is true in the isoinitial model of the multiple instance.

If $\mathcal{O}_1, \ldots, \mathcal{O}_n$ is additive, then every quantifier-free sentence of the language of $a_j, 1 \leq j \leq k$, is true in its isoinitial model if and only if it is true in the isoinitial model of the multiple instance.

In a multiple instance $\{a_1, \ldots, a_k\}$, the formulas of the language of an individual instance a_j correspond to the properties that are 'locally observable' in a_j, while the formulas of the (larger) language of the multiple instance represent properties that are 'globally observable'.

Strong additivity entails that the truth of any locally observable sentence is preserved in the global multiple instance. Additivity guarantees to preserve only those local properties that can be expressed by quantifier-free sentences.

Additivity is sufficient to prove the following proposition:

Proposition 1. *If a set of OBJ-frameworks is additive, then we can update a multiple instance, by adding, deleting or updating groups of individual instances, without altering the isoinitial models of the other individual instances.*

This guarantees that we can confine our updates to a part of a set of cooperating objects, without undesired side-effects on the other parts. Additivity suffices for our purposes. As we will see, there are cases where strong additivity is too restrictive.

Now, we can introduce the notion of a system of objects. Composite OBJ-frameworks will be based on such systems.

A system of cooperating objects $\mathcal{S}[\mathcal{O}_1, \ldots, \mathcal{O}_n]$, based on an additive set $\mathcal{O}_1, \ldots, \mathcal{O}_n$ of adequate OBJ-frameworks, is an object whose state, at any time t, is characterised by a multiple instance of $\mathcal{O}_1, \ldots, \mathcal{O}_n$, and (possibly) by other dynamic symbols, defined at the system level.

The methods for changing the state of $\mathcal{S}[\mathcal{O}_1, \ldots, \mathcal{O}_n]$ are the methods for creating new instances of a class $\mathcal{O}_j \in \{\mathcal{O}_1, \ldots, \mathcal{O}_n\}$ and deleting or updating an already existing individual object. Since $\mathcal{O}_1, \ldots, \mathcal{O}_n$ is additive, these operations update the global state of the system, without altering the isoinitial models of the untouched individual objects.

[9] i is unique up to isomorphism.

There are various ways of defining system methods. For example, they may be based on messages, as in many object-oriented programming languages. A possibility that we are investigating is introducing a meta-level where time is made explicit, and where system methods become logic programs. This formalisation is in its infancy and will not be presented here. Nevertheless, regardless of how system methods are formalised, it is clear that, at every time t, a system has the following properties:

- $\mathcal{S}[\mathcal{O}_1, \ldots, \mathcal{O}_n]$ has dynamic signatures and axioms. At each time t, it 'contains' a multiple instance of $\mathcal{O}_1, \ldots, \mathcal{O}_n$. It has a *global* signature $\Sigma_{sys}(t)$ and a *global* set of axioms $Ax_{sys}(t)$, and each individual instance a_j has its own signature $\Sigma_j(t)$ and axioms $Ax_j(t)$.
- Since the system is a multiple instance, for each individual instance a_j in it, $\Sigma_j(t)$ is a subsignature of $\Sigma_{sys}(t)$, and $Ax_j(t)$ is the restriction of $Ax_{sys}(t)$ to the signature $\Sigma_j(t)$. Moreover, in $\Sigma_{sys}(t)$, we can distinguish a static and a dynamic part.
- The static part consists of the signature and the axioms of the (closed) ADT's included in the OBJ-frameworks $\mathcal{O}_1, \ldots, \mathcal{O}_n$. This part is, globally, a closed ADT. We will denote it by ADT_{sys}.
- The dynamic part contains the dotted symbols and the axioms that have been used to close them in the various individual objects contained in the system at time t. Beside the dotted symbols, a system may contain non-dotted dynamic symbols, together with the related closing axioms. Non-dotted dynamic symbols are useful for expressing dynamic properties that are common to groups of objects and, hence, are not to be renamed (in contrast to dotted symbols which always are). In particular, there may be non-dotted system symbols, whose axioms are managed at the system level, i.e., outside the constituent OBJ-frameworks.

The presence of non-dotted dynamic symbols may affect additivity, so they must be treated with some care. The following proposition states a sufficient condition for strong additivity:

Proposition 2. *Let $\mathcal{S}[\mathcal{O}_1, \ldots, \mathcal{O}_n]$ be a system of objects. If all the dynamic symbols are dotted, then the system is strongly additive.*

If there are dynamic non-dotted symbols, then strong additivity and additivity may not hold. However, there are interesting cases where at least additivity is guaranteed. For example, if we have a non-dotted dynamic sort s, and s is closed, in each individual instance, by the freeness axioms of a set of constant symbols, then additivity is guaranteed.

We may also organise the objects of a system into sub-groups, so that we isolate properties that are common to a subgroup, but are not visible by the objects of another subgroup. This, of course, requires us to introduce a suitable renaming policy, e.g., by allowing multiple dottings, like path-names of a system with a tree-structure. We do not discuss this issue here, although our theoretical foundations are strong enough to embrace it.

Considering the possibility of non-dotted system symbols, a very useful tool is introducing the names of the individual objects (or of groups of objects, in a more complex renaming structure) in the signature. To this end, we assume that every system $\mathcal{S}[\mathcal{O}_1, \ldots, \mathcal{O}_n]$ has itself a reserved name *sys* and its global signature always contains a system reserved sort symbol *obj* and system reserved predicates $O_1, \ldots, O_n$. We require that the following *system constraint* is preserved by the methods for creating and destroying objects.

At each time t, the isoinitial model $\mathsf{i}(t)$ is such that:

(i) $obj^{\mathsf{i}(t)}$ is the set of names of the objects contained in the system at time t.

(ii) $\mathsf{i}(t) \models O_j(n)$ if and only if n is the name of an individual object of class $\mathcal{O}_j$, contained in the system at time t.

Once we have the names of the objects, we can also introduce expressive axiom schemas,[10] as shown by the following example:

Example 9. Suppose we want to define a predicate $.id : [obj]$, such that, in every object with name n, $n.id(x)$ holds if and only if x is the name of the object itself, i.e., $x = n$. We write down the axiom schema:

$$\#N.id(n) \leftrightarrow n = \#N$$

where the meta-symbol $\#N$ stands for any object name. So, if *ted* is an object-name, we get the instance $ted.id(n) \leftrightarrow n = ted$.

Now, we can introduce composite objects.

5.2 Composite Objects

A composite object is like a system $\mathcal{S}[\mathcal{O}_1, \ldots, \mathcal{O}_n]$, but now the additional global signature does not contain the symbols $sys, obj, \ldots$, that we have introduced in a system framework. It contains other (in general dotted) symbols, that are open symbols of the composite object not already contained in the components. Apart from this, we can reason exactly as we have done for systems of objects.

Example 10. Let us assume that we are working in a system containing at least the following OBJ-frameworks:

OBJ-Framework $\mathcal{CAR}$;

IMPORT: ...;

DECLS: $.weight : \to int$

$.km : \to int$

[10] As usual, an axiom schema is a template for a given general form of axioms.

OBJ-Framework $\mathcal{PERSON}$;

IMPORT ...;

DECLS: $.name : \to string;$
$.age : \to int; \ldots$

CONSTRS: $.age \geq 0$

We want to compose them into a driver framework, where a driver is a person allowed to drive a non-empty set of cars. We can set up a composite framework like

OBJ-Framework $\mathcal{DRIVER}[\mathcal{PERSON}, \mathcal{CAR}]$;

DECLS: $.drives : [obj];$

CONSTRS: $\#X.drives(c) \to PERSON(\#X) \wedge \#X.age \geq 18 \wedge CAR(c);$
$(\exists c : obj)(.drives(c));$

SPEC: $\#X.DriverComponent(n) \leftrightarrow n = \#X \vee \#X.drives(n);$

PROG: $\#X.Component(N) : - \ N = \#X; \#X.drives(N).$

Note that we are in the context of a system framework, and the system language, i.e., the sort *obj* and the predicates *PERSON*, *CAR*, *DRIVER*,..., can be used in the OBJ-frameworks of the system framework.

In the framework $\mathcal{DRIVER}$, we have also shown an example of a specification and a corresponding correct program[11] (or method). As we said, specifications and correct programs may be part of an open or closed framework, and hence of OBJ-frameworks.

Here the composite object states links on its components, constraining creation and deletion methods. We cannot create an object $n.\mathcal{DRIVER}$, if n is not a person. Furthermore, we need at least one car c. Thus, a method for creating a $\mathcal{DRIVER}$ object will be of the form:

$$\begin{array}{l} \#X.create(\#C_1, \ldots, \#C_k) : - \ \ not\#X.error(\#C_1, \ldots, \#C_k), \\ \quad add(\#X, state(\#X.drives(c) \leftrightarrow c = \#C_1 \vee \ldots \vee c = \#C_k)). \\ \#X.error(\#C_1, \ldots, \#C_k) : - \\ \qquad\qquad \neg(PERSON(\#X), CAR(\#C_1), \ldots, CAR(\#C_k)) \end{array}$$

where *error* displays appropriate error messages, and programs for *add*, *PERSON*, *CAR* are defined at the system level. $PERSON(X)$ fails if X is not a constant of sort *obj*, or if it is not the name of a $\mathcal{PERSON}$-object contained in the system. *CAR* is similarly defined. *add* updates the state-axiom closing *obj*, includes the signature of the $\#X$-instance of $\mathcal{DRIVER}$, and adds the state-axioms $state(\ldots)$.

Complementary methods for deleting cars and persons will be of the form:

$$\begin{array}{l} \#X.delete : - \ not\#X.error, cancel(\#X). \\ \#X.error : - \ DRIVER(\#N), \#N.Component(\#X). \end{array}$$

[11] In this example we will use a PROLOG-like formalism, but our discussion is independent of the chosen language.

where, similar to *add* and *cancel* can be defined at the system level. Note that the predicate $.Component$ is needed in deletion methods of components of composite objects. We can assume that each composite OBJ-frameworks has to specify, at least, its own predicate $.Component$, as well as a correct method for computing it.

As we can see from the example, we can specify methods like $.Component(X)$, and we can reason about their synthesis and correctness in the usual way in our approach to program development. In contrast, in our previous work we have never specified methods for creating, deleting or updating objects. This is because we do not have a language to formally specify them; as a consequence, we can speak about their correctness only informally. A formal system specification language should contain appropriate system symbols, and making time explicit seems to us very useful. We are studying these aspects. A short discussion will be given in the conclusions.

Finally, we briefly show that composite OBJ-frameworks are OOD frameworks as used in CBD methodologies like *Catalysis*. First, the composite OBJ-framework $\mathcal{DRIVER}$ can be represented in *Catalysis* by the simple (closed) OOD framework shown in Figure 4.[12] Here the person object is fixed (and so is

Fig. 4. The Driver OOD framework.

the composite driver object), like the traditional object in Figure 1.

A more interesting example is one where an object is formed by composing OBJ-frameworks in each of which the object plays a different role.

Example 11. In the OBJ-framework $\mathcal{DRIVER}[\mathcal{PERSON},\mathcal{CAR}]$, the role of a person is to drive a car. So we can rename the Driver OOD framework in Figure 4 as the PersonAsDriver OOD framework.

Equally a person could also be a guest at a motel, i.e., we can build an OBJ-framework $\mathcal{GUEST}[\mathcal{PERSON},\mathcal{MOTEL}]$. Thus a person can play the roles of driver and guest respectively in the PersonAsDriver and PersonAsGuest OOD frameworks, as shown in Figure 5. Then we can compose the two frameworks into $\mathcal{DRIVER}+\mathcal{GUEST}[\mathcal{PERSON},\mathcal{CAR},\mathcal{MOTEL}]$, where a person has two roles. This is depicted by Figure 6. In PersonAsDriverGuest, a person object is a composite object of the kind depicted in Figure 2.

Finally, we briefly discuss *open* OBJ-frameworks. In the previous example, both $\mathcal{DRIVER}$ and $\mathcal{GUEST}$ have the general structure of an user, i.e., they have

[12] We will present a simplified version of the (UML-like) notation used in *Catalysis*. More details can be found in [23, 16].

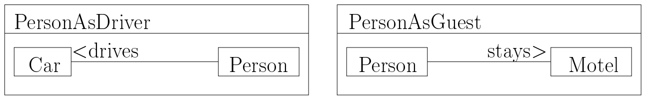

Fig. 5. PersonAsDriver and PersonAsGuest OOD frameworks.

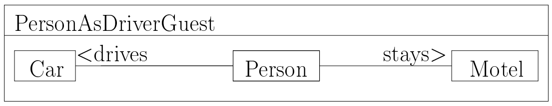

Fig. 6. PersonAsDriverGuest OOD framework.

the form $< \mathcal{USER} > [< \mathcal{USING} >, < \mathcal{USED} >]$. Here we have *open* OBJ-frameworks (indicated by the presence of the parameters $< \mathcal{USING} >$ and $< \mathcal{USED} >$), which cannot close their (possible) dynamic symbols by themselves.

In general, open OBJ-frameworks need to incorporate more knowledge, to become OBJ-frameworks of the kind we have considered here, where dynamic symbols can be closed without any additional knowledge. We have not treated open OBJ-frameworks here for lack of space. They would allow us to model open OOD frameworks as used in *Catalysis*.

6 Conclusion

We have presented semantic foundations for OOD frameworks as used in current CBD methodologies, in particular *Catalysis*. Our three-tier formalism makes explicit the existence of two levels of reuse. The first level of reuse occurs at the (top) level of specification frameworks. Here reuse is based on the classical inheritance mechanisms and is governed by constraints. The second level of reuse occurs at the levels of specifications and programs. Here reuse is governed by specifications, which indicate how to correctly compose open correct programs, to get new correct (composite) programs. The second level of reuse thus pertains to *correct program reuse*, which, we believe, contributes to a better understanding of and support for reusability in OOD and CBD. We have already studied correct program reuse within ADT's, through the notion of steadfastness. The next step is to extend our study to state transitions, by formalising the dynamics of object systems.

We have also a model-theoretic, i.e. declarative, semantics. Our semantics is model-theoretic and hence declarative. Such a semantics is important we believe for defining, and reasoning about, the key elements of next-generation CDB methodologies, and should provide a sound theoretical basis for their implementations. As an example, our use of isoinitial theories (and models) is motivated by our desire for a good declarative semantics; but we do not expect ordinary users to handle such theories, which should be well-hidden in CBD implementations. Rather, users will reason in terms of the intended interpretations of the symbols of the ADT's, classes and objects that they use. We believe that the semantics classical first-order logic, based on classes of interpretations that satisfy common properties, models in a natural and intuitive way the variety of uses and meanings that open ADT's and classes can assume in different contexts.

Our approach is similar to that proposed by Maibaum [24] in the use of theories as the units of construction and the relevant role of conservative (and, in our approach, adequate) extensions. However, we have a different emphasis on intended models. The introduction of errors is most likely during the phase of specification design, and we believe that a clear declarative semantics is very important and helpful to reduce the risks of mistakes at this level. Another difference is that our modularity is ruled by framework constraints, rather than guaranteed by general conditions, like interpolability. Moreover, we want to model correct reusability of open programs, rather than stepwise implementation. Nevertheless, refinement based on mediating theories is interesting, as also shown by SPECWARE [29], and it seems that it could be imported in our approach, as an orthogonal technique.

Our model-theoretic characterisation of states and objects stands in contrast to the type-theoretic approach by Abadi & Cardelli [1], however.

The work that is most closely related to ours is that of Bourdeau & Cheng [3], which formalises a semantics for object model diagrams in OMT [27]. Basically they translate object model diagrams into algebraic specifications (and semantics) expressed in Larch [12]. Thus they use initial theories, as opposed to isoinitial theories that we use. Our preference for isoinitial semantics has already been discussed in Section 3. Another motivation is that, using initial semantics, additivity entails only that *positive* locally observable quantifier-free sentences are preserved at the system-level, i.e., any problems with negation will affect additivity.

Moreover, the formalisation proposed by [3] is not oriented to (open) OOD frameworks, i.e. groups of interacting objects. In other words, they consider objects as the unit of reuse, whereas our formalisation supports OOD frameworks as the unit of reuse. As we pointed out before, industry is increasingly recognising that objects are not the best focus for design, and OOD frameworks are becoming widely used as the basic unit of reuse.

Other approaches to formal semantics for CBD methodologies are mainly so-called integrated methods. For example France *et al* [9] derives Z [28] specifications from Fusion [4, 6] diagrams. At present, such methods do not have completely formal semantics.

So far, our semantics provides a sound formalisation of OOD frameworks used in the CBD methodology *Catalysis*. However, in order to implement these frameworks, we need a formalism to describe and specify state transitions. In our semantics, we have only states at particular instants, i.e., each object has an intended model only at a fixed time, and we can treat state transition in a limited way. We can deal with the problem of constraint satisfaction, but not with the global system behaviour. More precisely, we cannot specify temporal properties, like liveness or deadlock freeness.

In order to achieve this capability, we are studying a temporal model for systems of objects. After a first attempt, inspired by evolving algebras [11], we decided to model states by evolving axioms. At each time, an evolving axiom A represents an observable instantaneous property. Evolving axioms are not so far from dynamic attributes (as used in, e.g., [8] and TROLL [10]). For example, a dynamic attribute $a : int$ with (current) value 5 corresponds to the (evolving) axiom $a = 5$, but evolving axioms allow us to describe larger classes of attributes, including data bases. To express dynamic behaviour, we need to model actions and time, taking into account the modularity induced by the presence of composite objects (similar ideas can be found, e.g., in [15]). Our aim is to introduce time in the specification language, and to deal with correctness of state transition methods in this extended context.

Acknowledgements

We wish to thank the referees for their helpful comments and suggestions. We also thank Juliana Küster-Filipe, Braunschweig University, Germany for her detailed feedback on an earlier draft.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.
2. A. Bertoni, G. Mauri, and P. Miglioli. On the power of model theory in specifying abstract data types and in capturing their recursiveness. *Fundamenta Informaticae* **VI**(2):127–170, 1983.
3. R.H. Bourdeau and B.H.C. Cheng. A formal semantics for object model diagrams. *IEEE Trans. Soft. Eng.*, 21(10):799-821, 1995.
4. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.
5. S. Cook and J. Daniels. *Designing Object Systems*. Prentice-Hall, 1994.
6. D.F. D'Souza and A.C. Wills. Extending Fusion: practical rigor and refinement. In R. Malan *et al*, editors, *Object-Oriented Development at Work*. Prentice-Hall 1996.
7. D.F. D'Souza and A.C. Wills. *Catalysis: Components, Objects and Frameworks in UML.* Addison-Wesley, October 1998. Draft available at http://www.trireme.com/catalysis/book.
8. J.L. Fiadeiro and T. Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming* 28:111-138, 1997.
9. R.B. France, J.-M. Bruel, M. Larrondo-Petrie, E. Grant, and M. Saksena. Towards a rigorous object-oriented analysis and design method. *Proc. 1st IEEE Int. Conf. on Formal Engineering Methods*, pages 7-16, IEEE Computer Society, 1997.
10. A. Grau, J. Küster Filipe, M. Kowsari, S. Eckstein, R. Pinger and H.-D. Ehrich. The TROLL approach to conceptual modelling: syntax, semantics and tools. In T.W. Ling, S. Ram and M.L. Leebook, editors, *Proc. 17th Int. Conference on Conceptual Modeling*, *LNCS* 1507:277-290, Springer, 1998.

11. Y. Gurevich, Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36, Oxford University Press, 1995.
12. J.V. Guttag and J.J. Horning. Larch: Languages and Tools for Formal Specification. Springer-Verlag, 1993.
13. R. Helm, I.M. Holland, and D. Gangopadhay. Contracts — Specifying behavioural compositions in OO systems. *Sigplan Notices* 25(10) (*Proc. ECOOP/OOPSLA 90*).
14. C. Kreitz, K.-K. Lau, and M. Ornaghi. Formal reasoning about modules, reuse and their correctness. In D.M. Gabbay and H.J. Ohlbach, editors, *Proc. Int. Conf. on Formal and Applied Practical Reasoning*, *LNAI* 1085:384–399, Springer-Verlag 1996.
15. J. Küster Filipe. Using a modular distributed temporal logic for in-the-large object specification. In A. Brogi and P. Hill, editors, *Proc. 1st Int. Workshop on Component-based Software Development in Computational Logic*, pages 43-57, September 1998, Pisa, Italy.
16. K.-K. Lau, S. Liu, M. Ornaghi, and A. Wills. Interacting Frameworks in *Catalysis*. To appear in *Proc. 2nd IEEE Int. Conf. on Formal Engineering Methods*, 9-11 December 1998, Brisbane, Australia.
17. K.-K. Lau and M. Ornaghi. On specification frameworks and deductive synthesis of logic programs. In L. Fribourg and F. Turini, editors, *Proc. LOPSTR 94 and META 94*, *Lecture Notes in Computer Science* 883, pages 104–121, Springer-Verlag, 1994.
18. K.-K. Lau and M. Ornaghi. Towards an object-oriented methodology for deductive synthesis of logic programs. In M. Proietti, editor, *Proc. LOPSTR 95*, *LNCS* 1048:152–169, Springer-Verlag, 1996.
19. K.-K. Lau and M. Ornaghi. A formal approach to deductive synthesis of constraint logic programs. In J.W. Lloyd, editor, *Proc. 1995 Int. Logic Programming Symp.*, pages 543–557, MIT Press, 1995.
20. K.-K. Lau and M. Ornaghi, Isoinitial Models for Logic Programs: A Preliminary Study, in J.L. Freire-Nistal, M. Falaschi, and M. Vilares-Ferro, editors, *Proceedings of the 1998 Joint Conference on Declarative Programming*, pages 443-455, A Coruña, Spain, July 1998.
21. K.-K. Lau and M. Ornaghi. On specification and correctness of OOD frameworks in computational logic. In A. Brogi and P. Hill, editors, *Proc. 1st Int. Workshop on Component-based Software Development in Computational Logic*, pages 59-75, September 1998, Pisa, Italy.
22. K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. To appear in *J. Logic Programming*.
23. K.-K. Lau, M. Ornaghi, and A. Wills. Frameworks in *Catalysis*: Pictorial Notation and Formal Semantics. *Proc. 1st IEEE Int. Conf. on Formal Engineering Methods*, pages 213-220, IEEE Computer Society, 1997.
24. T. Maibaum. Conservative extensions, interpretations between theories and all that! In *Proc. TAPSOFT 97*, *LNCS* 1214:40-66, Springer Verlag, 1997.
25. R. Mauth. A better foundation: development frameworks let you build an application with reusable objects. *BYTE* 21(9):40IS 10-13, September 1996.
26. P. Miglioli, U. Moscato and M. Ornaghi. Abstract parametric classes and abstract data types defined by classical and constructive logical methods. *J. Symb. Comp.* 18:41-81, 1994.
27. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Sorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
28. J.M. Spivey. *The Z Notation*. Prentice Hall, 2nd edition, 1992.

29. Y.V. Srinvas and R. Jüllig Specware: Formal support for composing software. In *Proc. MCP 95*, *LNCS* 947:399-422, Springer Verlag, 1995.
30. M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.