OOD Frameworks in Component-based Software Development in Computational Logic*

Kung-Kiu Lau
Department of Computer Science
University of Manchester
Manchester M13 9PL
United Kingdom
kung-kiu@cs.man.ac.uk

Mario Ornaghi
Dipartimento di Scienze dell'Informazione
Universita' degli studi di Milano
Via Comelico 39/41, 20135 Milano
Italy
ornaghi@dsi.unimi.it

Extended Abstract

1 Introduction

Currently, component-based software development, more usually known as just Component-based Development (CBD) is synonymous with Object-oriented Design (OOD). Most of the existing (semi-formal) OOD methods such as Fusion [2, 4] and Syntropy [3] 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.

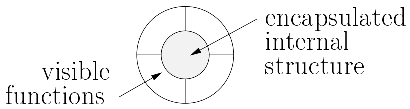

Figure 1: Traditional view of an object.

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. [7, 12]). 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 CBD methodologies, we can expect components to be not just objects, but groups of (interacting) objects. In the CBD methodology *Catalysis* [5, 11], for example, we use open OOD frameworks, in order to allow objects to have multiple roles in different frameworks, and to compose them by composing frameworks, as depicted in Figure 2.

2 CBD in Computational Logic

In computational logic, we have formulated a formal approach to program development [9, 10, 8]. In this paper, we will introduce states to our approach 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.

We will distinguish two kinds of frameworks: ADT-frameworks and OBJ-frameworks.

*Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

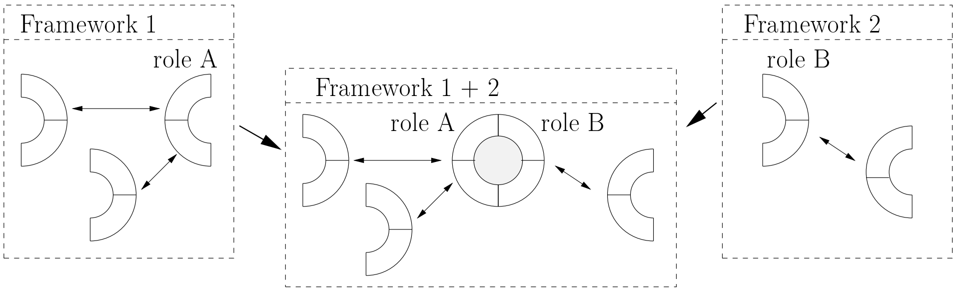

Figure 2: Objects by composing OOD frameworks.

2.1 ADT-frameworks

An *ADT-framework* is a first-order theory that axiomatises an Abstract Data Type, or a problem domain, or both. It is *closed* iff it axiomatises an isoinitial model, as its unique (up to isomorphism) intended model. Otherwise, it is *open*. An open framework axiomatises a class of intended models.

Example 2.1 The following *closed* framework formalises a finite domain containing 3 elements.

Closed-ADT-Framework $\mathcal{FD}3$;
SORTS: $D3$;
FUNCTIONS: $0, 1, 2 \quad : \quad \to D3$;
RELATIONS: $r \quad : \quad (D3)$;
AXIOMS: C-AXS$(0, 1, 2 : D3)$;
DEF$(r = \{0, 2\})$.

Equality $=$ and its standard axioms are assumed. C-AXS$(0, 1, 2 : D3)$ contains the axioms $\neg 0 = 1$, $\neg 0 = 2$, $\neg 1 = 2$ (i.e. Clark's Equality Theory for $\{0, 1, 2\}$) and DEF$(r = \{0, 2\})$ is the (definition) axiom $(\forall x)(r(x) \leftrightarrow x = 0 \vee x = 2)$.

Any model of $\mathcal{FD}3$ with a domain containing three elements (interpreting the constants $0, 1, 2$) is an isoinitial model of $\mathcal{FD}3$. All the isoinitial models are isomorphic. A particular one is the term model with domain $\{1, 2, 3\}$.

C-AXS$(0, 1, 2 : D3)$ and DEF$(r = \{0, 2\})$ are instances of axiom templates. From an abstract point of view, an axiom template can be seen as a recursive set Ax of sets of axioms. Concretely, some computable definition of Ax is needed. Axiom schemas, well-known in logic, are typical examples. However, we may yet have a more general kind of templates. In the following we will use two templates:

- The template C-AXS$(\ldots)$ indicates the *constructor axioms* (i.e. Clark's Equality Axioms) based on the constant and function symbols ... An instance is obtained by fixing the list ... For example, C-AXS$(\mathit{empty} :\to \mathit{Stacks}, \mathit{push} : (\mathit{Elem}, \mathit{Stacks}) \to \mathit{Stacks})$ indicates the following set of axioms (we omit universal quantifiers):

$$\neg \mathit{empty} = \mathit{push}(x, S);$$
$$\mathit{push}(x_1, S_1) = \mathit{push}(x_2, S_2) \to x_1 = x_2 \wedge S_1 = S_2.$$

- The template DEF(R) has instances of the form DEF$(r = \{t_1, \ldots, t_n\})$, which abbreviates the axiom $(\forall x)(R(x) \leftrightarrow x = t_1 \vee \ldots \vee x = t_n)$.

Example 2.2 As an example of an *open* framework, the theory of stacks can be axiomatised as follows.

Open-ADT-Framework $\mathcal{STACK}(\mathit{Elem})$;
SORTS: $\mathit{Elem}, \mathit{Stacks}$;
FUNCTIONS: $\mathit{empty} \quad : \quad \to \mathit{Stacks}$;
$\mathit{push} \quad : \quad (\mathit{Elem}, \mathit{Stacks}) \to \mathit{Stacks}$;
D-AXIOMS: C-AXS$(\mathit{empty}, \mathit{push})$;
P-AXIOMS: $(\exists x, y)(\neg x = y)$.

We have a parameter $Elem$ and two kinds of axioms. The p-axioms, that state some assumptions on the parameter $Elem$ (in this case we require that $Elem$ contains at least two distinct elements). The d-axioms completely characterise the non-parametric symbols in terms of the parameter. This means that via $Elem$ the framework $\mathcal{STACK}$ axiomatises a class of isoinitial models. That is, whenever we choose a particular set of elements (fixed by a closed framework), we get an axiomatisation of stacks with the chosen elements.

An open framework can be seen as an incomplete axiomatisation, to be completed by suitable completion procedures, and on completion it becomes a closed framework.

One such completion procedure is parameter passing. When the parameters Π of a framework $\mathcal{F}(\Pi)$ are instantiated by symbols of a closed framework $\mathcal{G}$, the p-axioms of $\mathcal{F}(\Pi)$ must become theorems. In this case, we say that the instance is *defined* and we indicate it by $\mathcal{F}[\rho, \mathcal{G}]$, where ρ is the (renaming involved in the) parameter passing. We omit here a detailed description of the instantiation operation. More details are given in [8].[1] A more general kind of parameter passing, based on the properties of explicit definitions in first-order theories, will be presented in the full paper.

Important properties of parametric frameworks are adequacy and additivity:

Definition 2.1 An open framework $\mathcal{F}$ is *adequate* if, for every closed framework $\mathcal{G}$ and parameter passing ρ, if $\mathcal{F}[\rho, \mathcal{G}]$ is defined, then it is a closed framework and its isoinitial model $iso(\mathcal{F}[\rho, \mathcal{G}])$ is an expansion of $iso(\mathcal{G})$.

An adequate framework is thus parametric in the sense that parameter passing will work correctly.

Definition 2.2 An adequate open framework is *additive* if, for every (defined) closed instance $\mathcal{F}[\rho, \mathcal{G}]$, $iso(\mathcal{F}[\rho, \mathcal{G}])$ is an expansion of $iso(\mathcal{G})$.

Additivity means that the semantics of $\mathcal{G}$ is preserved in $\mathcal{F}[\rho, \mathcal{G}]$.

The definition of adequacy and related results in our previous work (e.g. [8]) included additivity. Here we separate the two aspects, because for OBJ-frameworks we will require a different notion of additivity.

2.2 OBJ-frameworks

To handle objects, we need a notion of state. Here we introduce states, following an idea sketched in [9]. The approach is similar to evolving algebras [6].

Example 2.3 For example, the following is an OBJ-framework:

OBJ-Framework $OO.\mathcal{OBJ}(r)$;

SORTS:	obj;
FUNCTIONS:	$OO \ : \ \to obj$;
RELATIONS:	$\mathcal{OBJ}, r \ : \ (obj)$;
I-AXIOMS:	$(\forall x : obj)(\mathcal{OBJ}(x) \leftrightarrow x = OO)$; $\neg r(OO)$; $(\forall x : obj)(x = OO \vee r(x))$;
E-AXIOMS:	C-AXS(OO); $(\forall x : obj)(\neg r(x))$.

The constant OO is the name of the object. The set of evolving axioms contains C-AXS(OO), which is empty, and $(\forall x : obj)(\neg r(x))$.

An OBJ-framework may be created at a given time, by suitable creation methods, and it can evolve by means of suitable state-transition methods. For example, the e-axioms in $\mathcal{OBJ}$ may evolve into C-AXS(OO, a), namely $\neg a = OO$, and $(\forall x : obj)(r(x) \leftrightarrow x = a)$. These new axioms contain a new constant a, that is automatically added to the signature of the OBJ-framework. Thus, when axioms evolve, so may the signature of the OBJ-framework.

An OBJ-framework may use some (pre-defined) ADT-framework, i.e. it can be defined in a context of the latter. Of course, an ADT-framework cannot evolve.

An OBJ-framework is an instance of an open framework called an *object template*.

[1] Instantiation could also be defined as a pushout, as in algebraic ADTs [14].

Example 2.4 For example, $\mathcal{OBJ}$ is an instance of the following object-template:

OBJ-Template $.\mathcal{OBJ}(R)$;	
SORTS:	obj;
FUNCTIONS:	$\#id \;:\; \rightarrow obj$;
RELATIONS:	$\mathcal{OBJ}, R \;:\; (obj)$;
I-TEMPLATES:	$(\forall x : obj)(\mathcal{OBJ}(x) \leftrightarrow x = \#id)$; $\neg R(\#id)$ $(\forall x : obj)(x = \#id \vee R(x))$;
E-TEMPLATES:	C-AXS$(\#id, CONST^* : obj)$. DEF(R)

R is a parameter, like in parametric ADT-frameworks. $\#id$ indicates an open name. For example, it is instantiated to OO in the instance $OO.\mathcal{OBJ}(r)$. In the e-templates, $CONST^* : obj$ denotes a possibly empty sequence of constants of sort obj. e-templates state the form of the evolving axioms.

Having shown these examples, we can now define objects and object templates as follows.

Definition 2.3 (Objects)
An *object* is a closed framework $\mathcal{O}$ where we distinguish two kinds of symbols and axioms:

- ADT-symbols and ADT-axioms. They axiomatise abstract data types and specify relations over these data types. They are the ADT-frameworks used in $\mathcal{O}$. The meaning of an ADT-symbol is static, i.e. its axioms and their meaning in the current isoinitial model cannot evolve.
- OBJ-symbols and OBJ-axioms. The meaning of an OBJ-symbol is dynamic, i.e. its axioms and their meaning may evolve. We distinguish two kinds of axioms, for specifying OBJ-symbols:
  - i-axioms. These axioms cannot be updated. They represent the object's class invariant.
  - e-axioms. They belong to a class $\mathcal{A}$ of possible instances. The current instance of the evolving axioms axiomatises (together with the other axioms of $\mathcal{O}$) an isoinitial model. The meaning of the OBJ-symbols in this model represents the current state of the object described by the current evolving axioms. Evolving axioms can be updated; an update gives rise to a new instance of $\mathcal{A}$, axiomatising a new isoinitial model.

Definition 2.4 (Object Templates)
An *object template* is an open framework $.\mathcal{O}$ where we distinguish the following kinds of symbols and axioms:

- ADT-symbols and ADT-axioms. They axiomatise abstract data types.
- OBJ-symbols. A template has two kinds of OBJ-symbols: dotted and non-dotted symbols. Dotted symbols are renamed in every instance, by prefixing the identifier of the instance. Non-dotted symbols, like the ADT-symbols, are the same in every instance. The non-dotted sort symbol obj is always present.
- OBJ-templates. They are axiom-templates. We distinguish two kinds of templates:
  - i-templates. These are templates for the invariants.
  - e-templates. These are templates for the evolving axioms.

Definition 2.5 (Object Template Instances)
An *instance* $n.\mathcal{O}$ of an object template is any OBJ-framework such that:

(i) The signature of $n.\mathcal{O}$ contains the constant $n : \to obj$ (the object identifier) and all the symbols of the template. Dotted template symbols are prefixed by n. In contrast, the sort symbol obj and ADT-symbols are never dotted, hence they will be the same in all the instances.

(ii) The invariants of $n.\mathcal{O}$ are an instance Ax_i of the i-templates, and the evolving axioms Ax_e are an instance of the e-templates. If Ax_i or Ax_e require extra symbols, not already present in the signature defined in (i), these symbols are added to the signature.

Example 2.5 In this example, we explain the dot-notation. The following OBJ-template uses the ADT-framework $\mathcal{INT}$ for integer, which we assume to be pre-defined.

$$
\begin{array}{lrcl}
\textbf{OBJ-Template } .\mathcal{CAR}; & & & \\
\text{USES} & \mathcal{INT}; & & \\
\text{FUNCTIONS:} & \#id & : & \to obj; \\
& .weight & : & \to int \\
& .km & : & \to int \\
\text{I-TEMPLATES:} & \multicolumn{3}{l}{.weight = CONST : int} \\
\text{E-TEMPLATES:} & \multicolumn{3}{l}{.km = CONST : int.}
\end{array}
$$

Let us consider, for example, the instances $c1.\mathcal{CAR} = (\#id = c1, .weight = 50, .km = 100)$ and $c2.\mathcal{CAR} = (\#id = c2, .weight = 40, .km = 100000)$. The signature of $c1$ contains the renamed symbols $c1.weight$ and $c1.km$, and that of $c2$ contains $c2.weight$ and $c2.km$.

Like open parametric frameworks, object templates must be adequate:

Definition 2.6 (Adequate Object Templates)
An object template is *adequate* iff every consistent instance of it is a closed framework.

Note that an instance may be inconsistent, i.e. Ax_e may be inconsistent with respect to the other axioms. Inconsistency has to be avoided: when a creation method is invoked, it must create a consistent instance; when a state-transition method is invoked, it must preserve consistency. Beside consistency and adequacy, another important property is additivity.

Definition 2.7 (Multiple Instances)
Let $a_1, \ldots, a_n$ be distinct names. A *multiple instance* $[a_1, \ldots, a_n].\mathcal{O}$ is defined as the union (of the signatures and of the axioms) of all the single instances $a_i.\mathcal{O}$, together with the axioms C-AXS$(a_1, \ldots, a_n)$.

Definition 2.8 (Additive Object Templates)
An adequate object template $.\mathcal{O}$ is *additive* if its multiple instances $[a_1, \ldots, a_k].\mathcal{O}$ satisfy the following properties:

(i) If, for $1 \leq j \leq k$, $a_j.\mathcal{O}$ is consistent, then so is $[a_1, \ldots, a_k].\mathcal{O}$.

(ii) For every j, $1 \leq j \leq k$, there is a unique isomorphic embedding $I_j : iso(a_j.\mathcal{O}) \to iso([a_1, \ldots, a_k].\mathcal{O})|\Sigma_j$, where $|\Sigma_j$ indicates the reduction to the signature Σ_j of $a_j.\mathcal{O}$.

Here additivity is different from that of ADT-frameworks. Indeed, in the isoinitial term model of $[a_1, \ldots, a_k].\mathcal{O}$, obj is interpreted as the set $\{a_1, \ldots, a_k\}$, while in the one of $a_j.\mathcal{O}$ it is interpreted as the set $\{a_j\}$. Therefore the interpretation of obj is not preserved: when a new object is added, its name modifies the current language and the current interpretation of obj, according to the intuitive meaning of obj. However, by (ii) of Definition 2.8, we get:

Theorem 2.1 *(Representativity) Let $[a_1, \ldots, a_k].\mathcal{O}$ be a multiple instance of an adequate, additive $.\mathcal{O}$. A ground atom of the language of $a_j.\mathcal{O}$ is true in its isoinitial model if and only if it is true in the isoinitial model of $[a_1, \ldots, a_k].\mathcal{O}$.*

This means that, in a system of objects, the meaning of the symbols observable in a single object (i.e. belonging to its signature) is not altered when other objects are added or deleted. Of course, the global meaning (observable at the level of the whole system) will change, as expected.

We can prove the following sufficient condition:

Theorem 2.2 *(Sufficient Additivity Condition) Let $.\mathcal{O}$ be an adequate object template. If all the symbols of $.\mathcal{O}$, except obj, #id and the ADT-symbols, are dotted, then $.\mathcal{O}$ is additive.*

3 Systems of Objects

A system contains a family of evolving objects. We can structure a system in many different ways. Here we present a first simple structure.

3.1 A Simple Kind of Systems

We consider the simple case where each object is identified by a unique identifier. This policy is reflected by the methods for the special object template $\mathcal{OBJ}(R_1, \ldots, R_m)$. It has been shown in Section 2.2 in the case $m = 1$. Its extension to $m > 1$ should be obvious.

To explain the methods of $\mathcal{OBJ}$, we first explain how a system is structured.

System-Framework $\mathcal{S}$;
IMPORT $OO.\mathcal{OBJ}(\mathcal{O}_1, \ldots, \mathcal{O}_m)$
IMPORT TDA
IMPORT $.\mathcal{O}_1, \ldots, .\mathcal{O}_m$

In a system of this kind, there is a (possibly empty) evolving set $\{OO, n_1, \ldots, n_k\}$ of constants of sort *obj*, that represent the *object names*, and a fixed set $\{\mathcal{OBJ}, \mathcal{O}_1, \ldots, \mathcal{O}_m\}$ of predicates of arity (*obj*), that we will call *object predicates*.

The i- and e-templates of $OO.\mathcal{OBJ}(R_1, \ldots, R_m)$ are given in such a way that, for every parameter passing, every consistent instance has an isoinitial model where the sort *obj* is interpreted as the set containing OO and other (possible) object names $n_1, \ldots, n_k$, the predicate $\mathcal{OBJ}(X)$ is true only for $X = OO$, and the union of the object predicates is the domain *obj*.

These properties are guaranteed by our definition of $\mathcal{OBJ}$. However, nothing is said about the relationship between the isoinitial model of $\mathcal{OBJ}$ and the state of the system of objects. We cannot define this link within the OBJ-framework representing the system. We need the following meta-theoretical system invariant:

> For every object name n and object predicate $\mathcal{O}_i$ of $\mathcal{OBJ}$, the system contains one instance $n.\mathcal{O}_i$ of $.\mathcal{O}_i$ if and only if $\mathcal{O}_i(n)$ is true in the isoinitial model of $\mathcal{OBJ}$.

Therefore, a system-framework is an OBJ-framework containing $\mathcal{OBJ}$ and satisfying the above system invariant.

Methods for creating and destroying objects have to work in such a way that, at any one time, we have a consistent instance of $\mathcal{OBJ}$ and the system invariant is preserved. In the starting state the system will contain only the ADT-framework and the instance of $\mathcal{OBJ}$ contains only OO (i.e. only one object is active, namely $\mathcal{OBJ}$ itself). We can easily imagine procedures for creating and destroying objects while maintaining the system invariant.

3.2 Composing OOD Frameworks

Since an object is a closed framework, it can be used to complete another object. This does not require any additional formalism. We can have composite objects and composite object templates.

Example 3.1 $.\mathcal{DRIVER}$ allows us to define composite objects:

$$
\begin{array}{ll}
\textbf{OBJ-Template} & .\mathcal{DRIVER}; \\
\textsc{uses} & \mathcal{PERSON}, \mathcal{CAR}; \\
\textsc{functions:} & \#id \;:\; \to obj; \\
\textsc{relations:} & .drives \;:\; (obj); \\
\textsc{i-templates:} & \mathcal{PERSON}(\#id); \\
 & (\forall c : obj)(.drives(c) \to \mathcal{CAR}(c)); \\
 & (\exists c : obj)(.drives(c)); \\
\textsc{e-templates:} & \mathrm{DEF}(.drives : (obj)); \\
\textsc{specification:} & \forall x : obj(.component(x) \leftrightarrow x = \#id \vee .drives(x))
\end{array}
$$

where we assume some pre-defined OBJ-template $.\mathcal{PERSON}$, and the OBJ-template $.\mathcal{CAR}$ is as defined in Example 2.5.

The composite template $\mathcal{DRIVER}$ states links between objects. A driver must be also a person, and he has to drive at least one car. This means that a method for creating a driver can be applied only if the system contains at least one driver and one car. Similarly, if $n.component(m)$ holds, we cannot destroy m, unless we destroy also n.

This example shows that frameworks can express relationships between objects, as discussed in the Introduction. A composite object $n.\mathcal{O}$ may exist only if its components, specified by the predicate $n.component(X)$, exist. Multiple instances and additivity of composite object templates must be defined accordingly.

The general model explained in this abstract is essentially static. At any one time, a system is just a closed framework and it has an intended model, but we do not have a theory for the dynamics of a system. At the static level, we can introduce program specifications and correctness in the way we have already developed for logic programs in open and closed frameworks.

The next step is to add a dynamic model to the static one and formalise specification and correctness of dynamic methods.

4 Discussion

In [11] we started to formalise a notation for OOD frameworks in *Catalysis*. In this paper, we have proposed a preliminary formalisation of the frameworks themselves, by introducing states into specification frameworks that we have used in our approach to formal development of logic programs.

We have used isoinitial semantics rather than initial semantics, popularly used in algebraic ADTs. This is because under initial semantics Theorem 2.1 would become: 'If a ground atom of the language of $a_j.\mathcal{O}$ is true in its initial model, then it is true in the initial model of $[a_1, \ldots, a_k].\mathcal{O}$.'. Indeed the existence of an homomorphism guarantees that true formulas are preserved, but it may happen that an atom is false in the single object, but true in the system. This cannot happen with isoinitial semantics, since isomorphic embeddings (which are the counterparts of homomorphisms used in initial semantics) also preserve falsehood.

We have described a static model, i.e. we can describe the actual state of an evolving set of objects, where each object is a framework and the system is the union of the frameworks. Dynamic aspects, e.g. creation and state-transition methods, will be studied in another paper.

The work that is most closely related to our work here is the formalisation of OMT [13] object model diagrams by Bourdeau and Cheng [1]. However, their formalisation endows such diagrams with algebraic semantics, which is initial, in contrast to our isoinitial semantics. They model states as sorts, in contrast to our notion of abstract states as evolving axioms.

References

[1] R.H. Bourdeau and B.H.C. Cheng. A formal semantics for object model diagrams. *IEEE Trans. Soft. Eng.*, 21(10):799-821, 1995.

[2] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.

[3] S. Cook and J. Daniels. *Designing Object Systems*. Prentice-Hall, 1994.

[4] 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.

[5] D.F. D'Souza and A.C. Wills. *Component-Based Development using Catalysis*. Draft book.

[6] Y. Gurevich, Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36, Oxford University Press, 1995.

[7] R. Helm, I.M. Holland, and D. Gangopadhay. Contracts — Specifying behavioural compositions in OO systems. *Sigplan Notices* 25(10) (*Proc. ECOOP/OOPSLA 90*).

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] R. Mauth. A better foundation: development frameworks let you build an application with reusable objects. *BYTE* 21(9):40IS 10-13, September 1996.

[13] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Sorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[14] M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.