Logic for Component-based Software Development

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

Prologue

The title of this paper is styled on that of Kowalski's seminal book *Logic for Problem Solving* [32]. This is because in this paper we want to discuss how logic can play a crucial part in next-generation component-based software development, just as Kowalski showed in [32] that logic can be used for programming.

Our starting point is the following quote from the Preface of [32]:

> "In contrast with conventional computing methodology, which employs different formalisms for expressing programs, specifications, databases, queries and integrity constraints, logic provides a single uniform language for all these tasks."

with which we whole-heartedly agree. Unfortunately, despite this potential advantage, it would be fair to say that hitherto Logic Programming has not made any impact on Software Engineering. In fact, it has missed the boat, as far as the latter is concerned.

We believe that for Software Engineering, logical systems stronger and more expressive than Logic Programming are needed. In this paper we want to show that full first-order logic can be used as a basis for developing a *declarative* (model-theoretic) approach to Software Engineering, in particular *component-based software development* [56].

Currently Software Engineering is moving from object-oriented to component-based development (CBD), but it will not succeed, in our view, unless components have suitable declarative semantics. We believe that the declarative nature of logic specifications and programs will give Logic Programming the chance of a second bite at the cherry, to become a force in CBD, which has been hailed as "the Industrial Revolution for IT"!

1 Logic for Programming

Kowalski proposed using predicate logic as a programming language in [57, 32], and the rest — as they say — is history. The success of Logic Programming languages, principally Prolog, is undoubtedly due to their declarative nature, as observed in another quote from the Preface of [32]:

> "The meaning of programs expressed in conventional languages is defined in terms of the behaviour they invoke within the computer. The meaning of programs expressed in logic, on the other hand, can be defined in machine-independent, human-oriented terms. As a consequence, logic programs are easier to construct, easier to understand, easier to improve, and easier to adapt to other purposes."

In our view, however, the success of Logic Programming languages has been confined to *programming-in-the-small*. We shall not dwell on this, since our main concern here is software development, i.e. *programming-in-the-large*.

2 Logic for Specification

In [32] Kowalski also pointed out the suitability of logic as a specification language. We quote from Chapter 10 (p. 193):

> "The specification of programs, in particular, is an area in which the standard form of logic (or some appropriate extension of Horn clause form) is more suitable than simple clausal form."

Here Kowalski is comparing full first-order logic with clausal form from the point of view of program specification. We agree fully that the former is more suitable than the latter. Indeed, we believe that first-order logic is good for these purposes anyway.

However, somewhat ironically, the use of logic for specification is much more widespread in Formal Methods like Z [55] and B [1] than in Logic Programming itself. In fact within Logic Programming the prevalent view is that logic programs are executable specifications and therefore do not need specifying themselves. For example, here's a quote from the Conclusion of [33]:

> "Logic sufficiently blurs the distinction between program and specification that many logic programs can just as well be regarded as executable specifications."

This implicitly says that "logic programs are obviously correct since they are logical assertions". This is not satisfactory, in our view, since we believe that the meaning of correctness must be defined in terms of something other than logic programs themselves (we are not alone in this, see e.g. [23, p. 410]).

We believe that it is unfortunate that specifications have not received due attention in Logic Programming, and that logic programs have been equated with specifications. Indeed, we take the view that specifications should be strictly distinguished from programs, especially for the purposes of software development.

We have shown in [37, 38] that in Logic Programming, not only can we maintain this distinction, but we can also define various kinds of specifications for different purposes.

Our approach is based on a three-tier formalism with model-theoretic semantics illustrated in Figure 1.

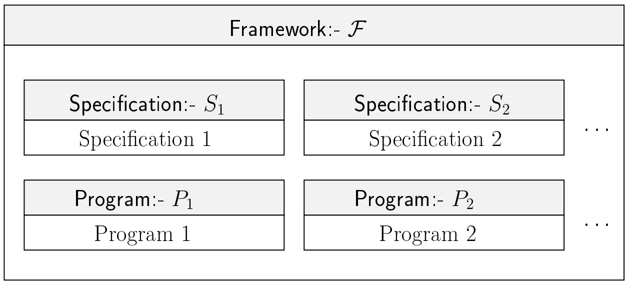

Fig. 1. Three-tier formalism.

- At the top level, we formalise a problem domain as a full first-order logical theory, which we call a *framework* $\mathcal{F}$. $\mathcal{F}$ has an *intended model* $I_\mathcal{F}$,[1] i.e. it has model-theoretic and hence *declarative* semantics.

  For the purpose of this paper the kind of intended model semantics is not relevant. (In our approach, the intended model $I_\mathcal{F}$ of a framework $\mathcal{F}$ is an *isoinitial model* [40].)

  In general, a framework $\mathcal{F}$ may be *open*, i.e., it may have no fixed intended models, but a class of possible ones instead. Open frameworks will be considered in Section 3. Here we consider only closed frameworks, i.e. frameworks with fixed intended models.
- In the middle, inside a framework $\mathcal{F}$, we define *specifications* as certain forms of first-order formulas in $\mathcal{F}$. A specification S_r of a new relation symbol r in a framework $\mathcal{F}$ is a set of axioms that defines the new symbol r in terms of the symbols of the framework.

  The symbols introduced by specifications will be called *specified symbols*, to distinguish them from the *framework signature* $\Sigma_\mathcal{F}$.

  In a closed framework with intended model $I_\mathcal{F}$, the model-theoretic meaning of a specification S_r of a symbol r is the set of $(\Sigma_\mathcal{F} + r)$-expansions[2] of $I_\mathcal{F}$ that satisfy S_r. For conciseness, the interpretations of r in such expansions will be called the *interpretations admitted by* S_r in $I_\mathcal{F}$.

  In an open framework $\mathcal{F}$, S_r associates every intended model I of $\mathcal{F}$ with the set of $(\Sigma_\mathcal{F} + r)$-expansions of I that satisfy S_r.
- At the bottom level, inside a framework $\mathcal{F}$, we have the *programs*, either standard or constraint logic programs for computing (specified) relations. P_r denotes a program that computes the specified relation r. (We could equally well use imperative or functional programs here, as long as they are correct or *steadfast* (see Section 3). However, using (constraint) logic programs here has the advantage of a homogeneous formalism as mentioned in the Prologue, which in turn simplifies the treatment of *steadfastness* (see Section 3).)

---

[1] A canonical representative of an isomorphism class.

[2] $(\Sigma_\mathcal{F}+r)$ extends $\Sigma_\mathcal{F}$ by r, and a $(\Sigma_\mathcal{F}+r)$-expansion of $I_\mathcal{F}$ is a $(\Sigma_\mathcal{F}+r)$-interpretation that coincides with $I_\mathcal{F}$ for the symbols in $\Sigma_\mathcal{F}$.

The framework $\mathcal{F}$ provides an unambiguous semantic underpinning for specifications and programs, and their strict distinction, via the *correctness relationship* between them.

For example, if a specified symbol r has only one interpretation admitted by its specification S_r, then the correctness of a program P_r for computing r is illustrated in Figure 2. P_r is correct wrt S_r iff the interpretation of r in the

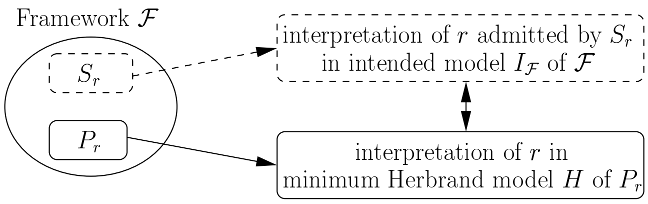

Fig. 2. Strict specifications.

minimum Herbrand model of P_r coincides with the (only) interpretation of r admitted by S_r (in $I_{\mathcal{F}}$).

An example of such a specification is an *if-and-only-if specification* S_r of a new relation r in a framework $\mathcal{F}$:

$$\forall x \,.\, r(x) \leftrightarrow R(x)$$

where $R(x)$ is any $\Sigma_{\mathcal{F}}$-formula.

On the other hand, S_r may be *loose*, i.e., it may admit many interpretations for r. For loose specifications, we have the situation in Figure 3. P_r is correct

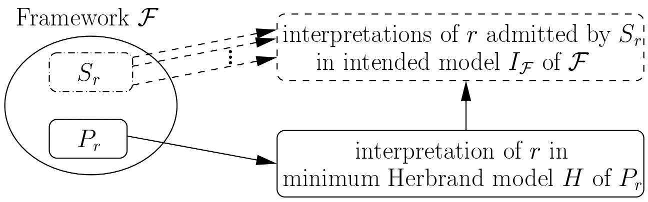

Fig. 3. Loose specifications.

wrt S_r iff the interpretation of r in the minimum Herbrand model of P_r coincides with one of the interpretations of r admitted by S_r (in $I_{\mathcal{F}}$).

An example of such a specification is a *conditional specification* of a new relation r in a framework $\mathcal{F}$:

$$\forall x, y \,.\, IC(x) \rightarrow (r(x, y) \leftrightarrow R(x, y))$$

where the *input condition* $IC(x)$ and the *input-output relation* $R(x, y)$ are $\Sigma_{\mathcal{F}}$-formulas.

A conditional specification is like a *pre-post-condition* style of specification as in Z [55] and B [1].

In a framework, all program predicates have specifications. The knowledge of the problem domain, codified by the framework axioms and theorems, together with the specifications, are the basis for developing *formal correctness proofs* [41].

Moreover, they allow us to enhance program reuse through *specification reduction*: a specification S_r reduces to a specification S'_r if every program P that is correct with respect to S_r is also correct with respect to S'_r. Thus, a non-strict specification S_r reduces to a non-strict specification S'_r if the interpretations of r that are admitted by S_r are admitted also by S'_r (strict specifications work as a limiting case). Specification reduction can be treated at the framework level. For example, in a closed framework $\mathcal{F}$, a conditional specification $I(x) \rightarrow (r(x,y) \leftrightarrow R(x,y))$ reduces to $I'(x) \rightarrow (r(x,y) \leftrightarrow R'(x,y))$ if and only if the formula

$$(I'(x) \wedge R'(x,y) \rightarrow I(x) \wedge R(x,y)) \wedge (\neg I(x) \vee R(x,y) \rightarrow \neg I'(x) \vee R'(x,y)) \quad (1)$$

is true in the intended model $I_{\mathcal{F}}$ of $\mathcal{F}$.

To treat specification reduction in the context of $\mathcal{F}$, the formulas involved, like (1), must not contain program predicates, since the latter do not belong to the framework signature $\Sigma_{\mathcal{F}}$. This excludes, for example, recursive if-and-only-if specifications. We require that our specifications are *completely declarative* in the context of the framework $\mathcal{F}$, that is, reasoning about them can be done in $\mathcal{F}$, independently of the programs. In this way, we can disregard the implementation details. As a concomitant, frameworks should be strong theories, i.e., they should satisfy the following *richness requirements*:

- The framework language should be *expressive* enough to specify programs naturally in a non-recursive, declarative way. To this end, we use full first-order logic, and we consider frameworks $\mathcal{F}$ with a rich signature $\Sigma_{\mathcal{F}}$.
- To reason about program correctness and specification reduction, the framework axioms should give a *powerful deductive system*. Moreover, a repository of useful theorems would facilitate theorem proving.

Thus, in order to meet these richness requirements, our frameworks are *full* first-order theories with powerful axioms, like induction schemas or descending chain principles.

Richness requirements and the emphasis on correctness and declarative specifications are the main features distinguishing our approach from algebraic specifications [58, 4]. A less relevant difference[3] lies in the kind of intended models we choose. We use *isoinitial models* [40], instead of initial ones. This is in line with the richness requirements: for a closed framework, we require at least *reachability* (a model is reachable if every element of its domain can be represented by a ground term) and *atomic completeness* ($\mathcal{F}$ is atomically complete if $\mathcal{F} \vdash A$ or $\mathcal{F} \vdash \neg A$, for every ground atomic formula A). For reachable models atomic completeness is a necessary and sufficient *isoinitiality condition* [40].

[3] As we have said, for the purpose of this paper, what kind of intended model we choose is not a relevant issue.

For the rest, the algebraic approach is close to ours, especially at the framework level. We have in common a model-theoretic semantics, and modularity at the framework level, briefly discussed in the next section, is modelled by theory morphisms, a way to *put theories together* [10, 27] that has been largely studied in the algebraic specification community.

Concerning the expressiveness of specifications, it is worthwhile to briefly comment on logics different from first-order classical logic. For example, temporal logic can be used to model time and change in specifications. Temporal logic has been proposed, e.g., in [19], to model concurrent object systems in an algebraic setting. Aspects like concurrency and resources can be implicitly modelled in non-classical logics, where logical connectives may have a non-classical operational interpretation. A notable example is linear logic, that has been used to specify and model concurrent systems [3, 31, 46]. Finally, compared to first-order logic, higher-order logic has a greater expressive power. It is the basis of various extensions of logic programming, e.g. λProlog [47], and of some logical frameworks, e.g., Isabelle [48], ELF [52], that could be used as a general meta-logic to manage different object logics and theories, as opposed to the categorical approach developed in algebraic specifications.

We do not exclude the use of more expressive logics, in particular temporal logic, in specifications, as long as we can maintain a model-theoretic semantics for frameworks, specifications and program correctness. A model-theoretic semantics is, in our opinion, more declarative than other kinds of semantics, like type-theoretic or proof-theoretic semantics. Moreover, it allows us to introduce *steadfastness*, a model-theoretic notion of *a priori* correctness. Steadfastness and its relevance for correct reusability will be discussed in the next section. So far, we have studied it in the model-theoretic setting of classical logic; the possibility of introducing time and change will be briefly discussed in Section 4.2.

As we will show later, the above features, together with the model-theoretic semantics of frameworks, specifications and programs provide a suitable basis for component-based software development.

3 Logic for Software Engineering

When it comes to Software Engineering, Logic Programming has missed the boat big time. This is primarily due to the view that logic programs are specifications. The quote from the Conclusion of [33] goes on like this:

> "Logic sufficiently blurs the distinction between program and specification that many logic programs can just as well be regarded as executable specifications. On the one hand, this can give the impression that logic programming lacks a programming methodology; on the other, it may imply that many of the software engineering techniques that have been developed for conventional programming languages are inapplicable and unnecessary for logic programs."

Our view is that Logic Programming indeed lacks a programming methodology, notwithstanding [15]. It may well be that many Software Engineering techniques for conventional programming languages are inapplicable to logic programs, but certainly they are not unnecessary.

The lack of emphasis by Logic Programming on Software Engineering is manifested by Logic Programming's bias in the last 20 or so years towards specialised AI and database issues such as non-monotonic logic, which are very distant from Software Engineering problems indeed. This prevented programming language researchers and designers outside Logic Programming from having a stronger positive influence on Logic Programming than they could have done otherwise. Consequently, notions such as modules, parameterisation, polymorphic data types and objects entered Logic Programming relatively recently.

Now we discuss how we might remedy the situation, and make Logic Programming address the issues of Software Engineering.

As we showed in the previous section, we distinguish strictly between specifications and programs within frameworks. This is important for Software Engineering, since it allows us to define *modules* and their *correctness*.

To formalise highly reusable modules, open frameworks are indispensable. Indeed, reusability in a wide context entails that the intended model cannot be fixed in advance, but instead we have to consider a class of possible models.

Intuitively, an open framework contains an as yet incomplete axiomatisation, which can be completed in many ways, by means of different suitable *completion procedures*. An example of a completion procedure is *parameter passing* in parametric frameworks, but we can have other kinds of completion procedures as well, like *internalisation* defined in [39, 34].

The symbols of an open framework $\mathcal{F}$ to be closed by the completion procedures are called *open symbols*. We will denote an open framework with open symbols Π by $\mathcal{F}(\Pi)$. A completion operation gives rise to a more specific framework $\mathcal{F}'$, and can be formalised as a suitable theory morphism $m : \mathcal{F} \to \mathcal{F}'$. The ($m$-reducts of the) intended models of $\mathcal{F}'$ are a subset of those of $\mathcal{F}$. We may have *total* completions, giving rise to closed frameworks, that we call *instances* of $\mathcal{F}(\Pi)$, as well as *partial* completions, yielding open *specialisations* of $\mathcal{F}(\Pi)$. For open frameworks, our three-tier formalism is illustrated in Figure 4.

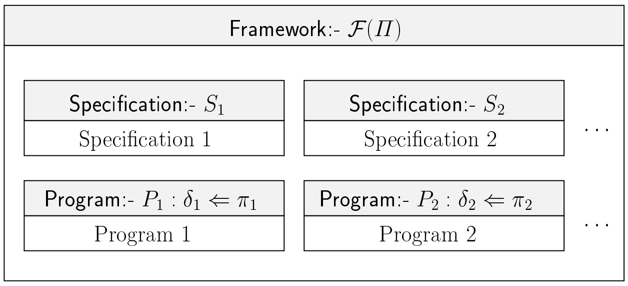

Fig. 4. Three-tier formalism for open frameworks.

Specifications S_1, S_2, $\ldots$, define program relations as in closed frameworks (closed frameworks are limiting cases of open frameworks) and may contain parameters from Π.

In each program P_i, δ_i are the *defined predicates* of P_i (they occur in the head of at least one clause), while π_i are the *open* ones (they occur only in the body of clauses). That is, P_i may be an open program. Its specification in $\mathcal{F}(\Pi)$ is a pair $(S_{\delta_i}, S_{\pi_i})$, where S_{δ_i} are specifications of P_i's defined predicates, and S_{π_i} are specifications of its open predicates.

Correctness can be defined in a model-theoretic way, as in the previous section, but we cannot compare minimum Herbrand models and interpretations admitted by specifications in the intended model, for two reasons: (a) minimum Herbrand models interpret the open predicates as empty relations, whereas the latter are supposed to represent generic, though not yet fixed predicates; and (b) an open framework has no fixed intended models, but instead we have to consider a class of possible models. To define correctness of an open program $P : \delta \Leftarrow \pi$ in a class $\mathcal{I}$ of models, we consider its *minimum j-models* where j is a *pre-interpretation*,[4] i.e. an interpretation of the symbols of P that are distinct from δ, namely its sorts, constants, functions and open predicates. Using *minimum j-models*, we base correctness on *steadfastness*.

Steadfastness embodies at once *modularity*, *reusability*, and *correctness*. A *steadfast* program is one that is correct (wrt to its specification) in each intended model of $\mathcal{F}$. Since the (reducts of the) intended models of its specialisations and instances are intended models of $\mathcal{F}$, a steadfast program is a correctly reusable unit in all specialisations and instances of $\mathcal{F}$. It can thus be used to model correct schemas (see [22]).

A formalisation of steadfastness is given in [41] for definite programs, with both a model-theoretic, hence *declarative*, characterisation and a proof-theoretic treatment of steadfastness. Our treatment readily extends to constraint logic programs. However, the extension to normal programs is not automatic, because the existence of the minimum j-model is no longer guaranteed. As far as semantics is concerned, a possible solution could be the introduction of a notion similar to stable models [26], but this would require a different proof theory for steadfastness. The extension of steadfastness to normal programs is an interesting problem, but we do not deal with it here, since our main interest is in specifications (where negation is treated as in full first-order logic) and correctness.

Here we give a simple example (taken from [41]) to illustrate the intuition behind steadfastness.

Example 1. Suppose we want to iterate n times a binary operation $\circ$ on some domain D. The framework, $\mathcal{ITER}(D, \circ, e)$, would contain (at least) the following:

(i) a (generic) domain D, with a binary operation $\circ$ and a distinguished element e;
(ii) the usual structure of natural numbers, to express the number n;

[4] Our notion of pre-interpretation generalises that in [42].

(iii) the iteration operation $\times(a,n) = e \circ a \underbrace{\circ \cdots \circ}_{(n \text{ times})} a$.

The specification of the relation *iterate* is:

$$S_{iterate}: \qquad iterate(x,n,z) \leftrightarrow z = \times(a,n) \tag{2}$$

Consider the following program $P_{iterate}$:

$$\begin{array}{ll} iterate(a,0,v) & \leftarrow unit(v) \\ iterate(a,s(n),v) & \leftarrow iterate(a,n,w), op(w,a,v) \end{array}$$

where s is the successor function for natural numbers, and the predicates *unit* and *op* are specified in $\mathcal{ITER}$ by the specifications:

$$\begin{array}{lll} S_{unit}: & unit(u) & \leftrightarrow u = e \\ S_{op}\ \ : & op(x,y,z) & \leftrightarrow z = x \circ y \end{array} \tag{3}$$

$P_{iterate}$ is *correct* with respect to the specifications (2) and (3) because it always computes *iterate* correctly in terms of correct computations of *unit* and *op* in *any* interpretation of $\mathcal{ITER}$.

For example, if D is the set of natural numbers, $\circ$ is $+$, e is 0, then $\times(a,n) = 0+a+\cdots+a = na$, i.e., $S_{iterate}$ specialises to $iterate(x,n,z) \leftrightarrow z = na$. Similarly, S_{unit} specialises to $unit(u) \leftrightarrow u = 0$, and S_{op} to $op(x,y,z) \leftrightarrow z = x + y$.

Now, if P^{+}_{unit} computes $u = 0$ and P^{+}_{op} computes $z = x + y$ (i.e., they are correct with respect to their specialised specifications), then $P_{iterate} \cup P^{+}_{unit} \cup P^{+}_{op}$ will compute na, i.e., $P_{iterate}$ is correct (wih respect to its specialised specification) in this first interpretation.

As another example, if D is the set of integers, $\circ$ is $-$, e is 0, then $\times(a,n) = 0 - a - \cdots - a = -na$.

If P^{-}_{unit} computes $u = 0$ and P^{-}_{op} computes $z = x - y$, then $P_{iterate} \cup P^{-}_{unit} \cup P^{-}_{op}$ will compute $-na$ for an integer a, i.e., $P_{iterate}$ is also correct in this second interpretation.

We say $P_{iterate}$ is *steadfast* in $\mathcal{ITER}$.

As an example of a non-steadfast program in $\mathcal{ITER}$, consider the following program $P^{*}_{iterate}$:

$$\begin{array}{l} iterate(a,0,v) \leftarrow unit(v) \\ iterate(a,n,v) \leftarrow m + m = n, iterate(a,m,w), op(w,w,v) \\ iterate(a,n,v) \leftarrow m + s(m) = n, iterate(a,m,w), \\ \qquad\qquad\qquad\qquad op(w,w,z), op(z,a,v) \end{array}$$

$P^{*}_{iterate}$ is more efficient than $P_{iterate}$: the number of recursive calls is linear in $P_{iterate}$, whereas it is logarithmic in $P^{*}_{iterate}$.

$P^{*}_{iterate}$ is correct with respect to (2) and (3) if D is the set of natural numbers with $+$ as $\circ$ and 0 as e. However, it is incorrect if D is the set of integers with $-$ as $\circ$ and 0 as e. For instance, for $iterate(a, s(s(s(s(0)))), v)$, $P^{*}_{iterate}$ computes 0 instead of the correct answer $-4a$. Thus $P^{*}_{iterate}$ is *not* steadfast in $\mathcal{ITER}$.

However, if we require that e and $\circ$ satisfy the additional (unit and associativity) axioms:

$$\forall x \,.\, e \circ x = x$$
$$\forall x, y, z \,.\, x \circ (y \circ z) = (x \circ y) \circ z$$

then we can prove that the following properties hold:

$$\begin{cases} \times(a,n) = \times(a, n \div 2) \circ \times(a, n \div 2) \circ a & \text{if } n \text{ is odd} \\ \times(a,n) = \times(a, n \div 2) \circ \times(a, n \div 2) & \text{if } n \text{ is even} \end{cases}$$

and in the subclass $\mathcal{ITER}^*$ of interpretations of $\mathcal{ITER}$ that satisfy the additional axioms, $P^*_{iterate}$ computes correctly, and thus it is steadfast in $\mathcal{ITER}^*$.

For instance, if D is the set of m-dimensional square matrices, with the m-dimensional identity matrix as e, then since matrix multiplication $\times$ is associative, $P^*_{iterate}$ is correct, where op computes matrix products.

To show how frameworks containing steadfast programs allow correct reuse, we continue from this example.

Example 2. The open framework $\mathcal{ITER}(D, \circ, e)$ can be formalised as follows.

Framework $\mathcal{ITER}(D, \circ, e)$;

IMPORT: $\mathcal{NAT}$;

SORTS: D;

FUNS:
$e \,: [\,] \to D$;
$\circ \,: [D, D] \to D$;
$\times : [D, Nat] \to D$;

RELS:

C-AXS:

D-AXS:
$\forall x : D \,.\, \times(x, 0) = e$;
$\forall x : D \;\; \forall n : Nat \,.\, \times(x, s(n)) = \times(x, n) \circ x$;

SPECS:
$S_{iterate} : iterate(x, n, z) \leftrightarrow z = \times(x, n)$;
$S_{unit} \;\;: unit(u) \leftrightarrow u = e$;
$S_{op} \;\;\;\,: op(x, y, z) \leftrightarrow z = x \circ y$;

PROGS:
$P_{iterate} : iterate(a, 0, v) \leftarrow unit(v)$
$\qquad\qquad iterate(a, s(n), v) \leftarrow iterate(a, n, w), op(w, a, v)$

where $\mathcal{NAT}$ is the closed framework containing first-order Peano Arithmetic.

We distinguish two kinds of axiom: *constraints* (C-AXS) and *definitions* (D-AXS). Constraints are properties to be satisfied by the completion procedures, when the open symbols are closed. Definitions guarantee that, once all the open symbols have been closed, the closed ones ($\times$ in the example) are completely defined, i.e. *total* completions give rise to closed (consistent) frameworks.

In our example, we do not have constraints, i.e. any completion for D, $\circ$ and e will work. This can be proved by considering the following facts: (a) a total completion introduces new Σ-axioms Ax with a reachable isoinitial model I, where Σ contains D, $\circ$, e, and possibly other symbols, but not the defined symbol $\times$; (b) I can be expanded into the reachable $(\Sigma + \times)$-model I' of D-AXS, which interprets $\times$ as the function evaluated according to the recursive equations D-AXS; (c) the atomic completeness of Ax is preserved by D-AXS (due to the evaluability of $\times$).

The intended models[5] of the specialisations and instances of a framework $\mathcal{F}$ are models of $\mathcal{F}$. By the properties of steadfastness, this guarantees that steadfast programs developed in $\mathcal{F}$ are correctly inherited.

For example, we can close the open symbols of $\mathcal{ITER}(D, \circ, e)$ by the following completion by internalisation:

COMPLETION NEG OF $\mathcal{ITER}(Int :: \mathcal{INT}, \circ, e)$:
CLOSE $\circ$ by $x \circ y = x - y$
CLOSE e by $e = 0$

In this completion we have used a parameter passing and two internalisations by explicit definitions. The parameter passing implicitly includes the predefined framework $\mathcal{INT}$ for integers, which expands the already imported $\mathcal{NAT}$. Axioms, specifications and programs are inherited. We can prove the formula $\times(x, n) = -nx$, so we can conclude that $S_{iterate}$ specialises to $iterate(x, n, z) \leftrightarrow z = -nx$, that is, our program correctly computes $-nx$. In general, specialisation is equivalence in the more specific context of a completion, but it may also involve specification reduction. Specialisation by reduction improves reusability and correct overriding, as we will discuss later. Since $P_{iterate}$ is open, we have to provide steadfast (i.e. correct)[6] programs for op and $unit$. The latter is trivial, while the former is likely already present in $\mathcal{INT}$. Such programs correctly compose with $P_{iterate}$, by the properties of steadfastness.

We can also consider:

COMPLETION EXP OF $\mathcal{ITER}(Nat, \circ, e)$:
CLOSE $\circ$ by $x \circ y = x \cdot y$
CLOSE e by $e = 1$

to get a program for computing the exponential function.

We can specialise $\mathcal{ITER}$ to $\mathcal{ITER}^*$ as follows:

Framework $\mathcal{ITER}^*(D, \circ, e)$;
EXTENDS: $\mathcal{ITER}$;
C-AXS: $\forall x : D \,.\, e \circ x = x$;
$\forall x, y, z : D \,.\, (x \circ y) \circ z = x \circ (y \circ z)$;
SPECS: $S_{half} : half(x, y) \leftrightarrow x = y + y$;
PROGS: $P_{iterate} : iterate(a, 0, v) \leftarrow unit(v)$
$iterate(a, n, v) \leftarrow half(n, m), iterate(a, m, w), op(w, w, v)$
$iterate(a, n, v) \leftarrow half(s(n), s(m)), iterate(a, m, w),$
$op(w, w, u), op(u, a, v)$

[5] More precisely, their reducts.
[6] In a closed framework, steadfastness and correctness coincide for closed programs.

We have inherited all the axioms and specifications, while overriding $P_{iterate}$. The new $P_{iterate}$ is more efficient, but it can be used only if its constraints can be proved after the completion operations. Thus, we can replace $\mathcal{ITER}$ by $\mathcal{ITER}^*$ in the closure EXP, but not in NEG.

Open frameworks and steadfastness conjugate reuse and correctness, and yield 3 levels of *correct reusability*:

- Top level.
  Open frameworks are reusable through completion operations, which allow us to implement framework *composition*, *specialisation* and *instantiation*. Axioms and theorems are inherited, because completion operations are theory morphisms.
  We require that open frameworks are *correctly constrained*, i.e. its constraints guarantee that consistency is preserved by completion operations (which must satisfy the constraints), and *totally defined*, namely every total completion yields a closed instance.
  At this level, correctness corresponds to constraint satisfaction. Constraints are therefore the first level of guidance for correctly reusing frameworks.
- Specification level.
  Specifications are inherited. Their meaning can be specialised according to the completion operation, by means of specification reduction (introduced in Section 2 for closed frameworks; in an open framework $\mathcal{F}(\Pi)$, S_r reduces to S'_r if, for every intended model I of $\mathcal{F}(\Pi)$, every interpretation of r admitted by S_r in I is also admitted by S'_r in I). For example, in the closure NEG, the specialised $S_{iterate}$ shows that *iterate* computes $-n \cdot x$. Specifications are the second level of guidance for correctly reusing frameworks, because their specialisation describes in a compact and declarative way how the behaviour of steadfast program specialises.
- Bottom level.
  Steadfast programs are inherited, together with their specifications. If $P : \delta \Leftarrow \pi$ with specification (S_δ, S_π) is steadfast in an open framework $\mathcal{F}(\Pi)$, then it is steadfast in *all* the *specialisations* and *instances* of $\mathcal{F}$ by completion operations, that is, it is correctly inherited with respect to the specifications (S_δ, S_π).
  Moreover, its specifications are a guidance for *correctly composing* $P : \delta \Leftarrow \pi$ with other programs Q_i for its open predicates. To guarantee the correctness of the composite program, it suffices to prove that Q_i's are correct with respect to the specifications S_π.

At the top level, there are no substantial differences with respect to algebraic specifications. However, reusability at the specification and bottom (program) level is peculiar to our approach. Specification reduction plays an important role both at the specification level, for specialising specifications while partially or totally completing frameworks, and at the bottom level, to enhance correct program composition.

In the next section, we will discuss how these features of frameworks can help to provide a basis for building reusable components with a declarative semantics.

4 Logic for Component-based Software Development

CBD [56] has been hailed as the "Industrial Revolution for IT", aimed at delivering Software Engineering from a cottage industry into a "mechanised" manufacturing industry. The goal of CBD is thus to provide the engineering science and tools for constructing software products by plugging components together, like building hardware from kits of component parts. Therefore, the ultimate test for CBD is whether it can allow arbitrary combination, or *third-party assembly*, of software components.

At present the key pre-requisites for CBD to succeed have not been met (see e.g. [9]). The first is a *formal semantics* for components and component reuse. Without formal semantics, it is not possible to achieve a standard, universally understood and accepted definition, which in turn is essential for achieving the ultimate goal of third-party assembly of components.

The second pre-requisite is good component *interfaces*. The interface of a component should be *all we know* about the component. It should therefore provide all the information on what the component does, i.e. its operations, (though not how it does it) and how we can use the component, i.e. its context dependencies. Otherwise, third-party assembly would not be possible. Therefore, an interface should include not just a list of *operations*, but also *context dependencies*. This implies that we need, as a minimum, polymorphism, theory morphism and composition, etc. to describe the semantics of interfaces. Therefore, *pre- and post-conditions* are not enough for specifying interfaces.

Thirdly, we need a good *assembly guide* for selecting the right components. The interface of a reusable component contains a collection of operations. In order to have a good assembly guide, the following pre-requisites should be satisfied:

- We need to know what each component does (correctly). Thus component operations should have *declarative specifications* (we have to know what they do, not how) and composition of components should yield the specification of the operations of the composite.
- We need to know what the reuse of a component within another one (after composition) will yield. This implies that the specifications of component operations should be compositional, namely that the specifications of a composite should indicate when and how we can correctly compose the operations inherited from its components.

This is only possible if we have a notion of correctness of component operations wrt their specifications and require that correctness is preserved by composition (wrt the specification of the composite derived from the specifications of the constituents). This implies that the semantics for components and their interfaces should incorporate a notion of *a priori correctness*, i.e. *pre*-proved correctness of any given component operation wrt its own specification, as well as *pre*-stated conditions that will guarantee that component and operation compositions will preserve correctness. This kind of correctness means *correct reusability* because it preserves *inheritance* and *compositionality*, i.e., programs can be inherited and specifications can be automatically derived from components (in composition) and super-components (when a general super-component is specialised) without destroying correctness.

Thus *a priori* correctness is the key to providing a good assembly guide. It stands in contrast to *a posteriori correctness*. The latter is the usual kind of correctness for verification-based program construction, where correctness of a composite can only be proved *after* the composition has taken place. It therefore cannot provide an assembly guide.

Although there are many strands of existing work in Formal Methods that address correctness, including *a priori* correctness, and composition of modules, collectively they do not meet the above requirements for CBD mainly because their modules are not components in the sense of CBD, such as frameworks [18] or patterns [24] for instance.

4.1 Limitations of Current Software Development Approaches

Now we consider the most relevant approaches to software development, and briefly analyse their potential for CBD, with respect to the prerequisites discussed above.

Object Technology. At present, CBD exists in the form of OO software development, employing current object technology, i.e. tools based on UML [53], together with middleware such as CORBA [28,6], COM [8] and Enterprise Java Beans (EJB) [44].

It lacks formal (declarative) semantics for objects, components, patterns, frameworks, interfaces, component assembly and reuse, component correctness and component assembly correctness. So it cannot provide good interface specifications or a good assembly guide.

Much of current CBD work also suffers from being low-level, consisting in *component-oriented programming* (i.e. a low-level approach akin to programming-in-the-small) because it uses languages like Component Pascal that merely somehow 'upgrade' plain OO programming languages into 'component programming' ones.

Work using CORBA, COM and EJB is rather low-level too, employing as it does what could be called *IDL programming*, i.e. OO programming in C++ or Java, with object integration provided by CORBA/COM/EJB via their respective *interface definition languages* (IDLs).

For CBD to achieve its goal, this low-level, bottom-up approach needs to evolve into a high-level, top-down one, with emphasis on *component assembly*, e.g. architecture description languages [54] and/or 'component assembly' languages.

Another weakness of current CBD is that it relies too much on existing OO (analysis and) design (OOD) methods. These methods use objects (or classes) as the basic unit of design and reuse. However, it is increasingly recognised that classes are not the best focus for design (see e.g. [29,45,18]). Typical design artefacts are rarely just about one object, but about groups of objects and the way they interact.

Frameworks, also known as *OOD frameworks*, are such groups of interacting objects. For example, in *Catalysis* [18], a driver may be represented (in UML) as the framework shown in Figure 5. A driver is a person who drives a car, or

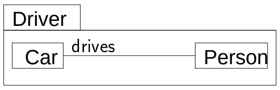

Fig. 5. The Driver framework.

in framework terminology, a driver is a framework composed of a car object and a person object, linked by a 'drives' association (or attribute).

Frameworks are increasingly recognised as better units of reuse in software development than objects (see e.g. [29, 45]). The reason for this is that in practical systems, objects tend to have more than one role in more than one context, and frameworks can capture this, whereas existing OOD methods (e.g. Fusion [11] and Syntropy [12]) cannot. The latter use classes or objects as the basic unit of design or reuse, and are based on the traditional view of an object, as shown in Figure 6, which regards an object as a closed entity with one fixed role. Such objects are very hard to reuse in practice. On the other hand, frameworks

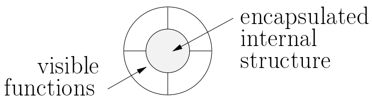

Fig. 6. Objects with one fixed role.

allow objects that play different roles in different frameworks to be composed by composing frameworks. In *Catalysis*, for instance, this is depicted in Figure 7.[7]

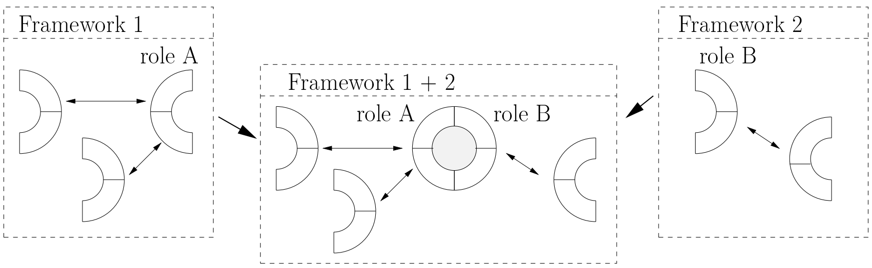

Fig. 7. Objects with multiple roles in different frameworks.

For example, a person can play the roles of a driver and of a guest at a motel simultaneously. These roles are shown separately in the PersonAsDriver and PersonAsGuest frameworks in Figure 8. If we compose these two frameworks,

[7] Double arrows denote interactions between (partial) objects.

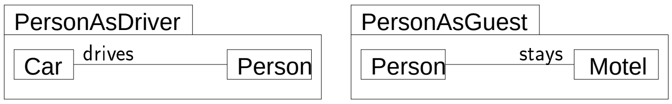

Fig. 8. PersonAsDriver and PersonAsGuest frameworks.

then we get the PersonAsDriverGuest framework as shown in Figure 9. In this

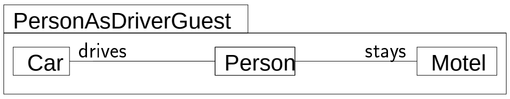

Fig. 9. PersonAsDriverGuest framework.

framework, a person object plays two roles, and is a composite object of the kind depicted in Figure 7. Frameworks are more reusable in practice than objects with fixed roles.

It should be noted that most of the so-called *patterns*, as defined in [24], are frameworks. Also, in UML, it is supposed to be possible to model frameworks and patterns (see e.g. [35]), even though in UML it is not clear how frameworks and patterns (or components) are defined.

Software Architecture. Software Architecture [5] is a relatively new, and as yet immature discipline (see [54]). Although there are architecture description languages, e.g. Wright [2], and architectural design tools, e.g. Rapide [43], research in software architecture has largely been overtaken by the universal adoption of UML (and the Unified Methodology) by software designers.

One difficulty of combining software architecture and current CBD is that whereas the former is top-down, the latter is essentially bottom-up (see e.g. [7]). There is also a conflict between software architecture and current CBD over component reuse. The former prefers components to fit in with the architecture, whereas the latter prefers pre-defined pre-implemented components (see e.g. [50]). Another problem with software architecture is the so-called Architectural Mismatch problem [25] underlying the composition of existing components, viz. that it is in general very hard to build systems out of existing parts if these parts have architectures that do not match. In [25] the authors describe their experience of spending five years on a case study where they failed to put three existing subsystems together and eventually decided to re-design and re-implement these subsystems in order to achieve their composition.

Formal Methods. General-purpose formal methods such as Z [55], VDM [30] and B [1] lack suitable semantics for components, even though they may have been 'upgraded' into versions with objects (e.g. Object Z). They lack semantic characterisations of specifications, objects, components, patterns, frameworks, etc. So they cannot provide good (component) interface specifications.

These methods also do not have meaningful notions of correctness for objects, components, patterns, frameworks, etc, or their composition and reuse. So they cannot provide a good assembly guide.

Existing (semi-)formal OOD methods such as Fusion [11, 17] and Syntropy [12] suffer from the same problems with semantics as the above general-purpose formal methods. Besides, they also use classes or objects as the basic unit of design, and as we saw in the previous section, this is not the best approach for next-generation CBD.

4.2 What can Logic Programming Contribute?

Correct reusability of steadfast programs means *a priori* correctness of their composition. This is a very important feature of steadfastness, and contrasts favourably, for the purposes of CBD, with *a posteriori* correctness normally employed in verification-based approaches to program construction.

Open frameworks containing steadfast programs are suitable as software components in CBD, and we will call them *steadfast components*, since they meet the prerequisites for CBD: *declarative formal semantics*, *interfaces* and *assembly guide* for reuse and composition.

A framework has a model-theoretic, hence declarative, semantics for all its constituents. The constituents of an open framework $\mathcal{F}(\Pi)$ can be identified as the following (see Example 2):

- the problem *signature* Σ, made up of sort symbols SORTS, function symbols FUNS and relation symbols RELS;
- the problem *axioms* AXS (consisting of constraints C-AXS and definitions D-AXS);
- a set SPECS of specifications;
- a set PROGS of programs.

To reflect this, we shall write $\mathcal{F}(\Pi) = \langle \Sigma$, AXS, SPECS, PROGS$\rangle$. The meaning of the problem signature Σ is given by the problem axioms AXS, according to an intended model semantics. Specifications SPECS define program predicates in terms of Σ. An open program $P : \delta \Leftarrow \pi$ (belonging to PROGS) with specification (S_δ, S_π) is interpreted by its j-models, where j ranges over the intended models of $\mathcal{F}(\Pi)$, expanded by the specifications S_π.

An open framework $\mathcal{F}(\Pi) = \langle \Sigma$, AXS, SPECS, PROGS$\rangle$ has a two-level *interface* with formal semantics:

- The *framework interface* is the problem signature Σ, together with the axioms AXS. It provides a set of known properties that define the meaning of the signature and provide a way of reasoning about the problem domain. The semantics of the framework interface is given by the completion operations and the semantics of the corresponding theory morphisms.
- The *method interface* is the set $\{(S_{\delta_i}, S_{\pi_i})\}$ of specifications of the programs (methods) in PROGS. They state precisely what methods do, in a concise and declarative way. The semantics is based on steadfastness, i.e. correctness within a framework, and specification reduction.

In addition, the interface of a framework $\mathcal{F}(\Pi) = \langle \Sigma, \text{AXS}, \text{SPECS}, \text{PROGS} \rangle$ also contains the following *context dependencies*, that constitute an assembly guide for framework and method reuse and composition:

- At the *framework* level, context dependencies are given by the open symbols Π and their constraints C-AXS $\subseteq$ AXS; only the open symbols can be closed, according to suitable completion operations, that must satisfy the constraints.
- At the *method* level, the specifications S_δ of the defined predicates of a program $P : \delta \Leftarrow \pi$ are a guide for program reuse: P can be correctly reused for a specification S_r of a relation r if δ contains a predicate d with specification S_d such that $S_d[d/r]$ reduces to S_r, where $[d/r]$ indicates the renaming of d by r.
  Similarly, the specifications S_π of the open predicates are a guide for program composition: a program $Q : \gamma \Leftarrow \ldots$ correctly composes with P if Q can be correctly reused for the specification S_p of some $p \in \pi$. For the sake of program composition, we can also contextualise specification reduction to the call positions of the open predicates in the clauses of programs, as, for example, in [21]. Contextual reduction is powerful, but it is no longer implementation independent: it can be used to compose two specific programs, while non-contextual reduction applies to any pairs of programs implementing the same specifications.
  Of course, reuse and composition may involve suitable renamings of program predicates.

Thus, context dependencies are a guide for correct reuse. An interesting feature is that specification reduction allows us to control inheritance polymorphism, as follows. A framework $\mathcal{G}$ extends a framework $\mathcal{F}$ if it adds new signature, axioms, specifications and methods (i.e., steadfast programs). Framework extension works as a subclassing mechanism, and allows program overriding. In the general case, $P_\mathcal{G}$ correctly overrides $P_\mathcal{F}$ if $S_{\mathcal{G}_\delta}$ reduce to $S_{\mathcal{F}_\delta}$ (i.e., $P_\mathcal{G}$ is also correct wrt $S_{\mathcal{F}_\delta}$) and $S_{\mathcal{F}_\pi}$ reduce to $S_{\mathcal{G}_\pi}$ (i.e., programs that correctly compose with $P_\mathcal{F}$ also correctly compose with $P_\mathcal{G}$). We can also have more flexible overriding mechanisms. For example, it may happen that $P_\mathcal{G}$ works more efficiently, but requires open operations that are not needed by $P_\mathcal{F}$, or have specifications different from those of $P_\mathcal{F}$. In this case, it is reasonable to replace $P_\mathcal{F}$ by $P_\mathcal{G}$, and provide the new required operations. That is, the use of specifications in the context of the problem domain allows us to treat inheritance in a rather flexible way, while maintaining correctness.

Open frameworks have another important property, viz. they can be used to represent classes, where attributes are the open symbols, methods are the steadfast programs, and class invariants are the constraint axioms. A framework $\mathcal{F}(\Pi)$ representing a class will be called a *class framework*. To build objects of a class framework $\mathcal{F}(\Pi)$, we close its open symbols Π. We may have multiple closures, i.e. many objects of the same class can coexist. Class frameworks have rich features. Attributes may be sorts, functions or relations; for example, relations as attributes can be used to model OO data bases. Framework axioms allow us to specify the abstract data types needed to close the attributes, and to model our general knowledge of the problem domain. Specifications give a formal declarative semantics of method-interface.

This is illustrated below:[8]

Correct framework	+	Steadfast programs	=	ADT $\oplus$ class invariants	+	Methods $\oplus$ specs	=	Correct (composite) class

A framework $\mathcal{F}(\Pi)$ can also be used to define a class diagram with possible constraints (as defined, e.g., in UML), and its instances can be seen as the object diagrams instantiating the class diagram. Constraints are satisfied by all the instances, by the way the open symbols are closed. Steadfast programs are like methods that satisfy their specifications in all the instances because of the *a priori* nature of correctness that steadfastness embodies.

In our explanation, we have implicitly assumed that frameworks representing classes and class diagrams have open symbols that can be closed by internalisation, without introducing new signature or axioms. But we can also assume the existence of only partially specified external entities, like data types, frameworks or objects. This is needed to model *OOD frameworks* [39] as components, to be closed by reusing them in different contexts, which have to provide the required external entities.

To illustrate the reuse of frameworks with steadfast methods as steadfast components, and the role of such frameworks as OOD frameworks, we show a simple example.

Example 3. Consider the following open framework $\mathcal{WLIST}(X, wt)$ for computing sums of weights in weighted lists, i.e. lists with weights associated with their elements. We assume that common ADT's, like reals, parametric lists, etc. have been pre-defined (by frameworks). $List(X)$ is the sort of lists with elements of sort X, and a non-empty list l has elements at positions 0 to $len(l) - 1$ (the empty list $[\,]$ has no positions).[9]

Framework $\mathcal{WLIST}(X, wt)$;

$$
\begin{array}{ll}
\textsc{funs:} & wt \;:\; [List(X), Nat] \rightarrow Real; \\
 & sum : [List(X), Nat] \rightarrow Real; \\
\textsc{c-axs:} & \forall i : Nat \;\; \forall l : List(X) \,.\, i \geq len(l) \rightarrow wt(l,i) = 0; \\
\textsc{d-axs:} & \forall l : List(X) \,.\, sum(l,0) = 0; \\
 & \forall k : Nat \;\; \forall l : List(X) \,.\, sum(l,k+1) = sum(l,k) + wt(l,k); \\
\textsc{specs:} & lsum(l,w) \leftrightarrow w = sum(l, len(l)); \\
 & lwt([x|a], b, w) \leftrightarrow w = wt(rev(b)|[x|a], len(b)); \\
\textsc{progs:} & P_{lsum} : \quad lsum(l,w) \leftarrow sumrev(l, [\,], w) \\
 & \quad sumrev([\,], b, 0) \leftarrow \\
 & \quad sumrev([x|a], b, w) \leftarrow sumrev(a, [x|b], u), \\
 & \qquad lwt([x|a], b, v), w \; is \; u + v
\end{array}
$$

[8] In the diagram, $a \oplus b$ signifies 'a always satisfies the accompanying specifications b'.
[9] For an axiomatisation, see [39].

The framework $\mathcal{WLIST}(X, wt)$ is designed to be reusable with different weighting mechanisms for lists, so we do not fix the weighting mechanism. Instead, we just assume that every position i in a list l has a weight $wt(l, i)$, where $wt(l, i)$ is an open function, to be instantiated by different weighting mechanisms, in different contexts.

Besides $wt(l, i)$, the framework-level interface of $\mathcal{WLIST}$ contains the open sort symbol X of list elements and the defined function $sum(l, i)$).

Relevant properties of $sum(l, i)$, e.g.

$$sum(l, 2) = wt(l, 0) + wt(l, 1) \ ; \ \ldots$$

can be proved as theorems. Theorems are an important component in a framework, as they can help both to explain the meaning of the defined symbols and to reason about specifications and programs. We could also informally but rigorously state explanatory meta-theorems like

$$sum(l, k) = \Sigma_{i=0}^{k-1} wg(l, i)$$

where the (meta) operator Σ is (meta) defined as usual.

The specification-level interface of $\mathcal{WLIST}$ specifies a (defined) program predicate $lsum(l, w)$, which means $w = sum(l, len(l))$, i.e., by the previous meta-theorem, $w = \Sigma_{i=0}^{len(l)-1} wt(l, i)$. The corresponding program P_{lsum} is designed for situations where $wt(l, i)$ may depend on the positions close to i. Thus it uses the open predicate lwt, specified using the weight $wt(rev(b)|[x|a], len(b))$,[10] where $l = rev(b)|[x|a]$ is the list whose weights are to be summed, $i = len(b)$ is the current position in a computation, x is the element at position i in l, and the elements at positions $i+1, i+2, \ldots$ are in a, while those at positions $i-1, i-2, \ldots$ are in b.

The specifications in the interface are the basis for correct reuse through specification reduction.

Here we show a reuse of $\mathcal{WLIST}(X, wt)$ (and $\mathcal{WGRAPH}(X, arc)$, which is a similar framework in which a list represents nodes of a graph and the weights represent arcs between adjacent nodes). Since we can have multiple (partial or total) framework completions, we will rename framework symbols and use the usual dot notation, to avoid confusion.

Consider the following completions:

COMPLETION TOWNS OF $\mathcal{WGRAPH}(X, arc)$:
CLOSE X by $\{LON, MAN, MI, \ldots\}$

COMPLETION FLTS OF TOWNS:
CLOSE arc by $arc(x, y, dis) \leftrightarrow (x = MAN \wedge y = MI \wedge dis = 2000)$
$\qquad\qquad\qquad\qquad\qquad\qquad \vee \ldots$

COMPLETION KMS OF $\mathcal{WLIST}(\text{TOWNS}.X, wt)$:
CLOSE wt by $wt(l, i) = w \leftrightarrow \text{FLTS}.arcwt(l, i, i+1, w) \vee$
$\qquad\qquad\qquad\qquad\qquad\qquad (w = 0 \wedge \neg \exists z \,.\, \text{FLTS}.arcwt(l, i, i+1, z))$

[10] $rev(x)$ means the reverse of x.

TOWNS has been partially completed to set X to a list of towns. FLTS uses arc to link pairs (x, y) of towns that are connected by a flight, and sets the weight of the link (arc) to be the distance between x and y. It also defines useful specification symbols, like $arcwt(l, i, j, w)$, which indicates that positions i and j in a list l are connected by an arc of weight w.

KMS is used to compute the cumulative distances of connecting flight paths, and to build it we choose $\mathcal{WLIST}(\text{TOWNS}.X, wt)$ because, looking at its interface specification, we see that $sum(l, len(l))$ is the cumulative distance, if we choose wt as indicated in the closure above, i.e., $wt(l, i)$ is the distance from the town at position i to the town at position $i+1$ if they are connected by a flight, and is 0 otherwise.

We can reduce the specification of lwt to

$$\begin{aligned} &lwt([x], b, 0), \\ &lwt([x, y|a], b, w) \leftrightarrow \text{FLTS}.arc(x, y, w) \vee \\ &\qquad\qquad\qquad\qquad (w = 0 \wedge \neg \exists z \,.\, \text{FLTS}.arc(x, y, z)) \end{aligned}$$

and derive a correct program for it. Steadfastness will guarantee that it correctly composes with P_{lsum}.

We can see the similarity between the closures TOWNS, FLTS and KMS in the example and objects in OO programming. This similarity allows us to consider an open framework as a way to dynamically build specialisations and instances, to be used as objects. A first study of this approach has been given in [34], where we introduce temporal operators. In this way we can specify static methods (they do not change the current instance) and dynamic methods (they may change the current instance). For example, we can model a window, with attributes describing its current dimensions and content. We can specify rows, rectangles, and so on, by first-order formulas, and give static methods to compute such figures. We can also use temporal formulas to *concisely* specify methods that change the current state, e.g. the window dimensions or the figures currently contained (and drawn) in the window. Our notion of steadfastness (*a priori* correctness) applies to static methods, but we do not have yet a satisfactory notion of *a priori* correctness for dynamic methods. Since *a priori* correctness plays an important role in correct reusability, we are currently investigating the possibility of axiomatising explicitly timed objects and programs at a (first-order or higher-order non-temporal) metalogical level where we can define steadfastness, and to use meta-level steadfastness to model steadfastness at the object level of temporal logic.

In summary, steadfastness is defined in terms of interfaces and context dependencies only. It means correct reusability because it is preserved through *inheritance* and *compositionality*. Steadfast components would therefore provide a good assembly guide.

Moreover, because steadfastness is preserved through inheritance and compositionality, steadfast components can be used for both bottom-up and top-down composition. Therefore steadfast components would allow third-party assembly and, we believe, can provide the semantics of components and component composition and reuse, missing from existing object technology, making the latter more declarative and top-down, and thus enhance OOD with suitable semantics for component interfaces.

Epilogue

So where do we go from here? We conclude this paper by outlining our perspective on the role of our approach in next-generation CBD.

Frameworks containing steadfast methods can be used as the basis for the construction of correctly reusable components. We can define various reuse operations, where constraints and specifications are used to ensure correct reuse of the inherited methods.

The development process of a reusable component for a general problem domain is illustrated in Figure 10.

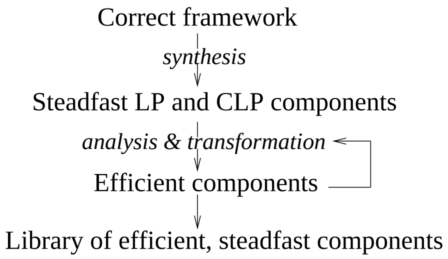

Fig. 10. Developing steadfast components.

In a first phase, we construct an open framework for the chosen problem domain, where we can specify a collection of reusable component operations, and then develop steadfast pure standard logic programs (LPs) and constraint logic programs (CLPs), from their specifications, using logic program synthesis techniques (see [16, 36]). In this phase, we can reuse existing developed frameworks.

In the second phase, we apply, iteratively, logic program analysis (e.g. [13]) and transformation techniques (see [51]) to ensure termination and to improve the efficiency of these programs. To be consistent with CBD, we need open termination (informally, an open program $P : \delta \Leftarrow \pi$ with specification (S_δ, S_π) has to terminate in all the pre-interpretations that satisfy S_π in the framework) and transformations that preserve steadfastness. In contexts different from ours, modular termination analysis, e.g., in [14, 49], and modular transformations, e.g., in [20], have been studied.

Although the steadfast component operations are implemented by (standard and constraint) logic programs, we can extend the notion of steadfast programs to other programming paradigms, and translate in phase 2 the synthesised logic programs into other (imperative) languages.

Different combinations of these programs then together with the framework form different steadfast components. These components will provide the basic library units for constructing libraries of bigger components, be they patterns or frameworks.

In the wider context of CBD, we see our methodology fitting in with the software development cycle in the manner depicted in Figure 11. The role of

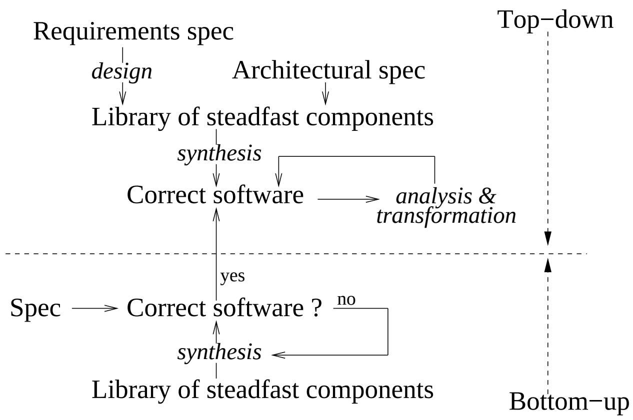

Fig. 11. CBD using libraries of steadfast components.

our methodology will be to provide a library of steadfast components in some chosen programming language, following the process described above. The key contribution of such a library to CBD is that it allows both *top-down* and *bottom-up* styles of development, and indeed a combination of both.

Top-down development will follow the so-called *waterfall model*: given the requirements specification, a design will be made, and software will be synthesised accordingly from the library components in order to meet the requirements. Alternatively we may follow the *software architecture* approach and start with an architectural specification, and synthesise software from the library components. The resulting software is guaranteed to be correct, but it may need to be analysed and transformed to improve efficiency.

Bottom-up development would start from the library of components, and some specification of either the requirements or the architecture. There is no design as such, but instead the development is iterative, in 'pick and mix' style, until the software constructed is seen, or can be verified, to meet the specification. Again, this style is possible because of steadfastness. Composition of steadfast components can show the specification of the composite, and therefore the specification of any software constructed can be compared with the initial specification for the whole system. Guidance as to which components to 'pick and mix' can also be provided by specifications and constraints, as we have discussed before.

If the specifications and the software system under construction have to evolve, then the *spiral model* of software development would be more appropriate. We can achieve this by combining the top-down and the bottom-up development styles described above. In each cycle of the spiral, top-down development can be used to develop software for specifications that have been finalised, whereas bottom-up development can show the gap between interim specifications and the current software system, thus enabling the developer to evolve the specifications or the system appropriately.

The general picture we have painted here represents just a perspective, and not yet a realised solution. In this context, we believe that frameworks and steadfast programs are good foundations for a formalisation and realisation of reusable components, and we are working towards turning CBD into reality in the LP community. Our next effort in this direction will be to extend the specification language (in particular, by introducing the possibility of expressing change) and the programming language (by considering extensions of standard and constraint logic programs), while preserving a suitable notion of steadfastness. We do not pretend to have *the* solution in our hands, but we are convinced that LP can play an important role and we hope that our ideas will stimulate interest in frameworks together with a pertinent theory of specifications, correctness and modularity.

In summary, we believe that despite missing the Software Engineering boat first time round, Logic Programming is in a good position to play an integral part in the Industrial Revolution for IT! Let's not pass this second chance by.

Acknowledgements

We wish to thank the volume editors Fariba Sadri and Tony Kakas for kindly inviting us to contribute to this volume. We are also indebted to the reviewers for their detailed and constructive comments that have considerably improved this paper.

References

1. J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. R.J. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie-Mellon University, 1997.
3. J.-M. Andreoli and R.Pareschi. Linear objects: logical processes with built-in inheritance. *New Generation Comp.*, 9:445–473, 1991.
4. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundations of Systems Specifications.* Springer, 1999.
5. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, 1998.
6. BEA Systems *et al.* CORBA Components. Technical Report orbos/99-02-05, Object Management Group, 1999.
7. J. Bosch and P. Molin. Software architecture design: evaluation and transformation. In *Proc. 1999 IEEE Engineering of Computer Based Systems Symposium*, 1999.

8. D. Box. *Essential COM.* Addison Wesley, 1998.
9. A.W. Brown and K.C. Wallnau. The current state of CBSE. *IEEE Software*, Sept/Oct 1998:37–46, 1998.
10. R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In R. Reddy, editor, *Proc. IJCAI'77*, pages 1045–1058, 1977.
11. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.
12. S. Cook and J. Daniels. *Designing Object Systems*. Prentice-Hall, 1994.
13. A. Cortesi, B. Le Charlier, and S. Rossi. Specification-based automatic verification of Prolog programs. In J. Gallagher, editor, *Proc. LOPSTR 96, Lecture Notes in Computer Science 1207*, pages 38–57. Springer-Verlag, 1997.
14. N. Dershowitz. Termination of rewriting. *J. Symbolic Computation*, 3:69–116, 1987.
15. Y. Deville. *Logic Programming: Systematic Program Development*. International Series in Logic Programming. Addison-Wesley, 1990.
16. Y. Deville and K.-K. Lau. Logic program synthesis. *J. Logic Programming*, 19,20:321–350, 1994. Special Issue: Ten Years of Logic Programming.
17. D.F. D'Souza and A.C. Wills. Extending Fusion: practical rigor and refinement. In R. Malan *et al*, editor, *Object-Oriented Development at Work*. Prentice-Hall, 1996.
18. D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.
19. H.-D. Ehrich. Object specification. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications*, chapter 12, pages 435–465. Springer, 1999.
20. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *TCS*, 166(1&2):101–146, 1996.
21. P. Flener, K.-K. Lau, and M. Ornaghi. Correct-schema-guided synthesis of steadfast programs. In *Proc. 12th IEEE Int. Automated Software Engineering Conf.*, pages 153–160. IEEE Computer Society Press, 1997.
22. P. Flener, K.-K. Lau, M. Ornaghi, and J. Richardson. An abstract formalisation of correct schemas for program synthesis. *Journal of Symbolic Computation*, 30(1):93–127, July 2000.
23. J.H. Gallier. *Logic for Computer Science: Foundations for Automatic Theorem Proving*. Harper and Row, 1986.
24. E. Gamma, R. Helm, R. Johnson, and J. Vlissades. *Design Patterns – Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1994.
25. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proc. ICSE'95*, pages 179–185, 1995.
26. M. Gelfond and V. Liftschitz. The stable model semantics for logic programming. In R. Kowalski and K.Bowen, editors, *Proc. 5th Int. Conf. and Symp. on Logic Programming*, pages 1070–1080. MIT Press, 1988.
27. J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.
28. Object Management Group. The Common Object Request Broker: Architecture and specification Revision 2.0, 1995.
29. R. Helm, I.M. Holland, and D. Gangopadhay. Contracts — specifying behavioural compositions in oo systems. *Sigplan Notices*, 25(10), 1990. *Proc. ECOOP/OOPSLA 90*.
30. C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, second edition, 1990.

31. N. Kobayashi and A. Yonezawa. Asynchronous communication model based on linear logic. *Formal Aspects of Computing*, 7:113–149, 1995.
32. R. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.
33. R. Kowalski. The relation between logic programming and logic specification. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 11–27. Prentice-Hall, 1985.
34. J. Küster Filipe, K.-K. Lau, M. Ornaghi, and H. Yatsu. On dynamic aspects of OOD frameworks in component-based software development in computational logic. In A. Bossi, editor, *Proc. LOPSTR 99, Lecture Notes in Computer Science*, volume 1817, pages 43–62. Springer-Verlag, 2000.
35. G. Larsen. Designing component-based frameworks using patterns in the UML. *Comms. ACM*, 42(10):38–45, October 1999.
36. 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 Symposium*, pages 543–557. MIT Press, 1995.
37. K.-K. Lau and M. Ornaghi. Forms of logic specifications: A preliminary study. In J. Gallagher, editor, *Proc. LOPSTR 96, Lecture Notes in Computer Science 1207*, pages 295–312. Springer-Verlag, 1997.
38. K.-K. Lau and M. Ornaghi. The relationship between logic programs and specifications — the subset example revisited. *J. Logic Programming*, 30(3):239–257, March 1997.
39. K.-K. Lau and M. Ornaghi. OOD frameworks in component-based software development in computational logic. In P. Flener, editor, *Proc. LOPSTR 98, Lecture Notes in Computer Science 1559*, pages 101–123. Springer-Verlag, 1999.
40. K.-K. Lau and M. Ornaghi. Isoinitial semantics for logic programs. In J.W. Lloyd *et al*, editor, *Proceedings of the First Int. Conf. on Computational Logic, Lecture Notes in Artificial Intelligence 1861*, pages 223–238. Springer-Verlag, 2000.
41. K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. Logic Programming*, 38(3):259–294, March 1999.
42. J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.
43. D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Trans, Soft. Eng.*, 21(4):336–355, April 1995.
44. V. Matena and B. Stearns. *Applying Enterprise JavaBeans – Component-based Development for the J2EE Platform.* Addison-Wesley, 2000.
45. R. Mauth. A better foundation: Development frameworks let you build an application with reusable objects. *BYTE*, 21(9):40IS 10–13, September 1996.
46. D. Miller. Forum: A multiple-conclusion specification logic. *TCS*, 165(1):201–231, 1996.
47. G. Nadathur and D.A. Miller. An overview of λProlog. In R.A. Kowalski and A. Bowen, editors, *Proc. 5th Int. Conf. on Logic Programming*, pages 810–827. MIT Press, 1988.
48. L.C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
49. D. Pedreschi and S. Ruggieri. Verification of logic programs. *J. Logic Programming*, 39(1-3):125–176, 1999.
50. D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. *ACM Software Engineering Notes*, 17(4):40–52, 1992.

51. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *J. Logic Programming*, 19,20:261–320, 1994. Special Issue on "Ten Years Of Logic Programming".
52. F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
53. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
54. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
55. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
56. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
57. M.H. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, October 1976.
58. M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.