The Role of Logic Programming in Next-Generation Component-Based Software Development

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

Abstract

To date Logic Programming has not made any impact on Software Engineering. As Software Engineering moves on from *objects* to *components*, will Logic Programming be simply forgotten forever? In this position paper, I will argue that the answer could be and should be, perhaps surprisingly, negative.

1 Introduction

In the mainstream of the field of Software Engineering, the imperative paradigm (in particular, the object-oriented paradigm) dominates. C/C++ and now Java are the only game in town. Declarative paradigms may be acknowledged by software engineers as elegant and nice in theory, but in practice these paradigms don't get a look in. CASE tools based on functional or logic programming? In your dreams!

Does this mean that Logic Programming will forever be irrelevant to Software Engineering? I believe and hope not. In this position paper, I will try and explain why I think (and hope) so. The reason for my belief (and optimism) lies in the so-called Industrial Revolution for Information Technology — Component-based Software Development (CBD).

2 CBD — "The Next Industrial Revolution"

Component-based Software Development (CBD) has been hailed as the "Industrial Revolution for IT"! This "revolution" aims 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. (This of course requires (cross-platform) *interoperability*, a complex issue that we do not have space to address here.)

2.1 What is a (Software) Component?

There is no universally accepted definition of a component at present. The following definition [37] is probably the most widely used one (and is one that I almost agree with, see later):

> A **software component** is a unit of composition with contractually specified *interfaces* and explicit *context dependencies*.
>
> An *interface* is a set of named operations that can be invoked by the clients.
>
> *Context dependencies* are specifications of what the deployment environment will need to provide, such that the components can function.

[4] defines a component to be "a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces". [11] defines a component to be "a package of software that includes implementation, with a specification of interfaces provided and required".

Like these three examples, most definitions in the literature converge to some degree, but they are not all identical because they use different (informal) semantics for interfaces and context dependencies, and different kinds of components (e.g. binary code, source code, more abstract representations (such as specifications), and even design documents).

Herein lies the first, fundamental problem for CBD: what is the formal semantics of a component? 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.

You might argue that objects still (after all these years) do not have a standard definition, and yet this has never affected their popularity. However, I would counter that it is precisely the lack of formal semantics that has resulted in oo software not being as reusable as industry had been led to believe it would be (see later).

2.2 What are the Key Requirements for CBD?

Apart from formal semantics for components (and their composition), the success of future CBD depends on the following:

- good (component) *interface specifications*;
- good *assembly guide* for selecting the right components.

2.2.1 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, in contrast to [37] (see Section 2.1), I believe that 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* (with only *proof-theoretic* semantics) are not enough for specifying interfaces. Rather, we need *declarative* (e.g. *model-theoretic*) semantics.

2.2.2 Assembly Guide

In order to have a good assembly guide, we need to know what each component does (correctly) and what its composition with another component will yield. That is composition of components should yield the specification of the composite. This implies that the specification of a component and its interface should be compositional.

This is only possible if we have a notion of correctness of components wrt their specifications and of correctness of their 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 wrt its own specification, as well as *pre*-stated conditions that will guarantee the correctness of a composite (of correct components) wrt to its specification. This kind of correctness means *correct reusability* because it preserves *inheritance* and *compositionality*, and it is the key to providing a good assembly guide.

A priori correctness stands in contrast to *a posteriori correctness*. The latter is the usual kind of correctness for algebraic compositions, where correctness of a composite can only be proved for a composite *after* the composition has taken place. It therefore cannot provide a assembly guide.

2.3 Why is Existing Object Technology not Enough?

At present, CBD exists in the form of oo software development, employing current object technology, i.e. UML-based tools together with middleware such as CORBA, COM and Enterprise Java Beans (EJB). So what are the problems? Aren't objects all we need? Isn't the current object technology enough? Well, no, or else there would not be the recognition that we need a "revolution" to deliver Software Engineering into the "industrial age".

Apart from the lack of formal semantics, there are two basic reasons, in my view, why existing object technology is not enough for (next-generation) CBD:

(i) it is too low-level and bottom-up, and therefore hard to do in the absence of a methodology;

(ii) it is too much based on objects, and therefore not very reusable.

Much of current CBD work suffers from being low-level, consisting in what I would call *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.

I would also argue that work using CORBA, COM and EJB is rather low-level too, employing as it does what I would call *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 (ADLs) and/or 'component assembly' languages. However, a prerequisite for this evolution would be *formal* (*declarative*) *semantics* for:

- objects, components, patterns, frameworks
- interfaces (contracts)
- component assembly
- component reuse
- component correctness
- component assembly correctness.

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. [15, 28, 11]). 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 the CBD methodology *Catalysis* [11], a driver may be represented (in UML) as the framework shown in Figure 1. A

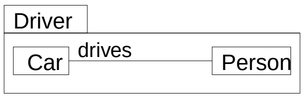

Figure 1: The Driver framework.

driver is a person who drives a car, or 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. [15, 28]). 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 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 2, which regards an object as a closed entity with one fixed role. Such objects are very hard to reuse in practice.

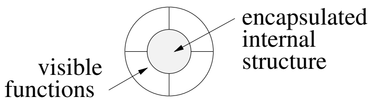

Figure 2: Objects with one fixed role.

On the other hand, frameworks allow objects that play different roles in different frameworks to be composed by composing frameworks. In *Catalysis*, for instance, this is depicted in Figure 3.

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 4. If we compose these two frameworks, then we get the PersonAsDriverGuest framework as shown in Figure 5. In this framework, a person object plays two roles, and is a composite object of the kind depicted in Figure 3. Frameworks are more reusable in practice than objects with fixed roles.

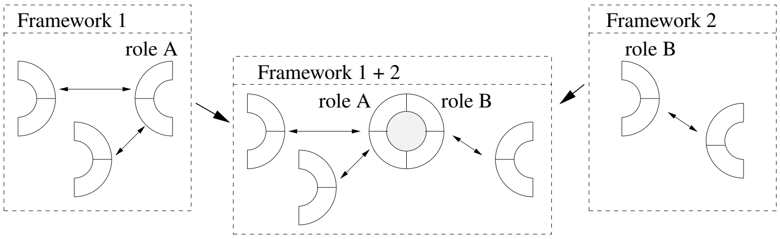

Figure 3: Objects with multiple roles in different frameworks.

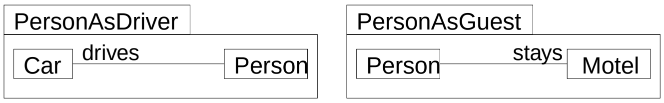

Figure 4: PersonAsDriver and PersonAsGuest frameworks.

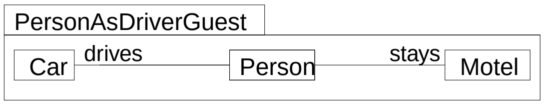

Figure 5: PersonAsDriverGuest framework.

It should be noted that most of the so-called *patterns*, as defined by the Gang of Four [12], are frameworks. Also, in UML, it is supposed to be possible to model frameworks and patterns (see e.g. [18]), even though in UML it is not clear how frameworks and patterns (or components) are defined.

2.4 Why are Existing Formal Methods not Enough?

What about formal methods? Surely there must be some existing formal methods that provide all the formal semantics we need for CBD? Well, in my opinion, no.

General-purpose formal methods such as Z [36], VDM [16] and B [1] lack suitable semantics for components. Even though they may have been 'upgraded' into versions with objects (e.g. Object Z), to my mind these methods largely operate at *syntax* level, often only with *proof-theoretic* or *type-theoretic* 'semantics'. They lack semantic characterisations of specifications, objects, components, patterns, frameworks, etc. So I think 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. They rely on *a posteriori* verification to prove any kind of correctness. So I think they cannot provide a good assembly guide.

What about formal methods for OOD? It is fair to say that the existing (semi-)formal OOD methods (such as Fusion [6, 10] and Syntropy [7]) 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.

2.5 Why is Software Architecture not Enough?

[3] defines software architecture as follows:

> The **software architecture** of a program or computing system is the *structure* (or *structures*) of the system, which comprises
>
> - software components
> - the externally visible properties of these components
> - the relationships between them.

Clearly software architectural design is top-down. So why can't we use this approach for CBD? Well, as I see it, there are 2 main problems:

- software architecture is not as yet a mature discipline;

- there are conflicts between software architecture and current CBD.

Software architecture is a relatively new discipline (see [35]). Although there are architecture description languages (ADLs), e.g. Wright [2], and architectural design tools, e.g. Rapide [27], in my view, research in software architecture has largely been overtaken by the universal adoption of UML 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, as we saw in Section 2.3. The following quote from [5] makes the point:

> "The process of identifying the entities that make up the architecture is different from, for instance, traditional object-oriented design methods. Those methods start by modelling the entities present in the domain and organise these in inheritance hierarchies, i.e. a bottom-up approach.
>
> Our experience is that during architectural design it is not feasible to start bottom-up since that would require dealing with the details of the system. Instead one needs to work top-down.
>
> . . . our experience is that the entities resulting from architecture design are generally not found in the domain . . . "

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. The following quote from [31] illustrates this:

> "The important lesson in reusing components is that the possibilities for reuse are the greatest where specifications for the components are constrained the least — at the architectural level.
>
> Component reuse at the implementation level is usually too late because the implementation elements often embody too many constraints.
>
> Moreover, consideration for reuse at the architectural level may lead development down a different, equally valid solution path, but one that results in greater reuse."

Another problem with software architecture is the so-called Architectural Mismatch problem [13] 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 [13] 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.

3 What can Logic Programming Offer?

Can Logic Programming offer anything for next-generation CBD? Well, I believe it can – it can provide the bits missing from current CBD. Not surprisingly perhaps, the evidence on which I base my belief comes from the work of my colleagues and myself in Logic Program Development.

Logic Programming can provide the semantics of components, component composition, etc missing from existing object technology (see Section 2.3), making the latter more declarative and top-down. In essence, Logic Programming can enhance OOD with suitable semantics for component interfaces.

Logic Programming can also add to current Formal Methods by providing the necessary declarative (model-theoretic) semantics, in particular *a priori* correctness (as opposed to *a posteriori* verification, see Section 2.4), that will provide a good assembly guide.

Software Architecture would benefit from the semantics for interfaces and the assembly guide, since they can ease, if not reconcile, the top-down versus bottom-up conflict between CDB and OOD (see Section 2.5).

3.1 Logic Program Development

The salient features of our work in logic program development, e.g. [21, 23, 17], address the needs of CBD:

- Components are (specification) frameworks (with formal semantics).

  These correspond to frameworks in the CBD methodology *Catalysis* [11, 25, 19] that we saw in Section 2.3.

- We have (semantics for the) correctness of components and their composition and reuse.

- Specifications of components have (formal) declarative semantics;

  Our approach can thus provide good component interfaces.

- Correctness (based on *steadfastness*[24]) is *a priori*.

  Our approach can therefore provide a good assembly guide.

Our approach is based on a three-tier formalism (illustrated in Figure 6). At the top level, we have formalised a problem domain as a full first-order logical theory, which we call a (*specification*) *framework* $\mathcal{F}$. $\mathcal{F}$ has an intended model, i.e. it has model-theoretic (and hence *declarative*) semantics. A *parametric* framework $\mathcal{F}$ is one

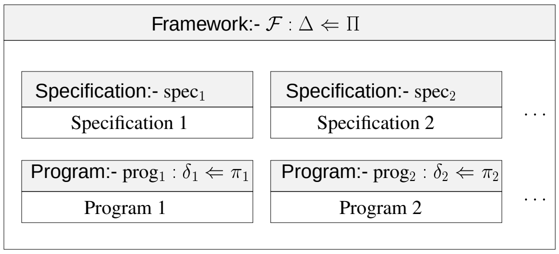

Figure 6: A parametric framework.

with parameters Π, and it defines a set Δ of symbols in terms of Π, and will contain in addition *parameter axioms* for Π.

In the middle, inside a framework $\mathcal{F}$, we have defined *specifications* as certain forms of first-order formulas in $\mathcal{F}$ (see [22]). A specification in a framework $\mathcal{F}$ is a set of axioms that defines a set of new symbols in terms of the symbols of the framework.

At the bottom level, inside a framework $\mathcal{F}$, we have the *programs*, either standard or constraint logic programs. A program prog_i defines a set of symbols δ_i in the clause heads in terms of the program parameters π_i, which may be just any relation symbols in $\mathcal{F}$ that are not yet (completely) defined, or of course they may be relation symbols from the framework parameters Π.

In this three-tier formalism, we can define program correctness in a model-theoretic manner [22]. More importantly, we can also define and reason about the *correctness* of a parametric framework. This leads us to the key result, viz. *steadfastness*, i.e. correctness that is preserved through inheritance hierarchies.

Correctness of a parametric framework $\mathcal{F}$ means the *consistency* of (each instance) of $\mathcal{F}$, and the *correctness* of (each instance of) each prog_i wrt its specification spec_i in (each instance of) $\mathcal{F}$, i.e. the consistency of each instance of $\mathcal{F}$ and the correctness of each prog_i in each instance of $\mathcal{F}$.

Thus a *correct* parametric framework (with parameters Π) is a consistent framework[1] that contains:

(i) specifications $\mathrm{spec}_1, \mathrm{spec}_2, \ldots$;

(ii) *steadfast* programs prog_1, prog_2 (with parameters $\pi_1, \pi_2, \ldots$) wrt to their specifications.

A *steadfast* program is one that remains correct (wrt to its specification) in each instance of $\mathcal{F}$. A steadfast program is thus a correctly reusable unit in all instances of $\mathcal{F}$. A formalisation of steadfastness is given in [24], with both a model-theoretic, hence *declarative*, characterisation and a proof-theoretic treatment of steadfastness.

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 (algebraic) program construction.

In oo terminology, a correct framework is equivalent to a (composite) class in which the class invariants on the ADT are always satisfied because of the way the framework has been constructed; and steadfast programs are like methods that always satisfy their pre-post-condition specifications because of the *a priori* nature of correctness that steadfastness embodies. This is illustrated below:

$$\begin{array}{c}\text{Correct}\\\text{framework}\end{array} + \begin{array}{c}\text{Steadfast}\\\text{programs}\end{array} = \begin{array}{c}\text{ADT}\\\oplus\\\text{class}\\\text{invariants}\end{array} + \begin{array}{c}\text{Methods}\\\oplus\\\text{pre-post-}\\\text{conditions}\end{array} = \begin{array}{c}\text{Correct}\\\text{(composite)}\\\text{class}\end{array}$$

[1] Consistency can be achieved by incremental framework construction and composition, see e.g. [20].

The ADT of a parametric framework is of course polymorphic. Framework composition relies on theory morphism. The semantics of frameworks is given in [23], it is model-theoretic and hence declarative.

Thus, in summary, our methodology provides a formal basis for next-generation component-based software development, by formalising components (and their interfaces), their correctness, and the correctness of their composition.

3.2 Steadfast Components

Correct frameworks are suitable as components, and since they contain steadfast programs, we call them *steadfast components*.

Looking at Figure 6, in a parametric framework $\mathcal{F} : \Delta \Leftarrow \Pi$, the set of (defined symbols Δ and) the (steadfast) programs prog_i provides a set of *operations* (that compute δ_i), whilst the set of framework parameters Π and the program parameters π_i provides a set of explicit *context dependencies*. Thus by the definition in Section 2.1, a parametric framework is a software component.

In addition, the interface of a parametric framework $\mathcal{F} : \Delta \Leftarrow \Pi$ also contains the specifications spec_i of its programs prog_i (together with their context dependencies π_i). Therefore the interface of $\mathcal{F}$ does contain *all we know* of this component. Moreover, since $\mathcal{F}$ is a polymorphic theory, and we have the necessary declarative semantics for theory morphism and composition, we believe that we have a good specification of the interface of a steadfast component (cf. Section 2.2.1).

Steadfastness is defined in terms of interfaces and context dependencies only, in the sense of [37] (see Section 2.1), and in terms of only interfaces in our sense. It means correct reusability because it preserves *inheritance* and *compositionality*. Steadfast components would therefore provide a good assembly guide (cf. Section 2.2.2).

Therefore we believe that steadfast components would allow third-party assembly. Moreover, because steadfastness preserves inheritance and compositionality, steadfast components can be used for both bottom-up and top-down composition (see Section 3.3).

3.3 Developing Steadfast Components

Although we have defined steadfast components in terms of (standard and constraint) logic programs, we can extend the notion of steadfast components to other programming paradigms, by a two-phase process. In phase 1, we develop steadfast components with logic programs, and then in phase 2, we translate these into other (imperative) languages. The entire development process is illustrated in Figure 7.

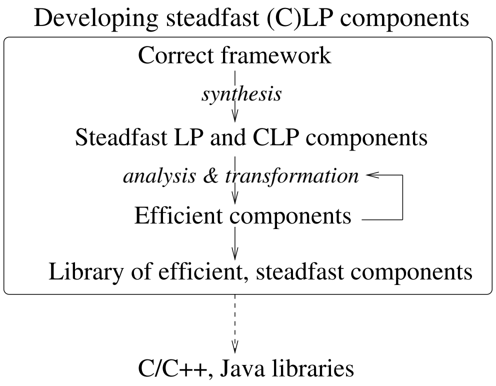

Figure 7: Developing steadfast components

In phase 1, for a chosen problem domain, we first construct a correct framework, and then develop steadfast pure standard logic programs (LPs) and constraint logic programs (CLPs), from their specifications, using logic program synthesis techniques (see [9, 21]), and, iteratively, logic program analysis (e.g. [8]) and transformation techniques (see [32]) to ensure termination and to improve the efficiency of these programs.

In phase 2, we translate the transformed (terminating and efficient) LPs and CLPs into non-Logic Programming programs. To preserve the semantics of LPs and CLPs, we can translate the minimum Herbrand model into its fixed-point equivalent, i.e. the T_P operator (see [26]), and thereby translate minimum model semantics into fixed-point (denotational) semantics for non-Logic Programming programs. 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 non-Logic Programming languages. Of course, the degree of formality depends on that of the semantics available for these languages.

It is worth noting in particular that CLPs have many important industrial applications. By using the above process, we will be able to develop correct CLP libraries in non-Logic Programming languages.

In the wider context of CBD, we see our methodology fitting in with the software development cycle in the manner depicted in Figure 8.

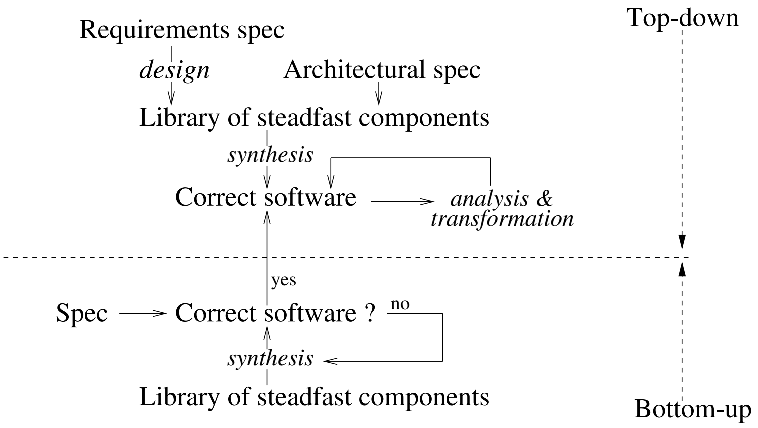

Figure 8: CBD using libraries of steadfast components.

The role of 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. Here, however, techniques for synthesising software from steadfast components, and those for analysing and transforming software, will not be the same as those used in logic program development, although the latter would provide a very good starting point.

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 steadfastness (via *a priori* correctness).

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.

4 Conclusion

In this position paper, I have indicated why I believe Logic Programming can have a second chance of making an impact on Software Engineering. The opportunity is created by the need for declarative semantics for components, as Software Engineering enters the dawn of the Next Industrial Revolution for IT — CBD.

The declarative nature of Logic Programming lends itself to a declarative characterisation of CBD, as has been demonstrated by work in Logic Progam Development. We have outlined how research in logic program synthesis and transformation can be used to characterise and develop steadfast components. The cornerstone of this approach is the notion of steadfastness — correctness that preserves inheritance and compositionality. I believe that steadfast components can meet the key requirements of next-generation CBD, namely good interface specifications and a good assembly guide, to achieve third-party assembly.

However, in this paper, we have focused on functional requirements only, but even here it would be an achievement to be able to do third-party assembly. Furthermore, our frameworks have more than UML classes and *Catalysis* frameworks. The latter describe only object models (but not methods), and our components are therefore more complete, since they contain not only methods/programs but also their specifications (and their correctness).

Finally, I believe that

$$\text{Unified} \neq \text{Ultimate}$$

by which I mean the Unified Modelling Language, and the Unified Software Development Process, are not the ultimate solution to CBD. Logic Programming can and should play an integral part.

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. Ph.D. these, Carnegie-Mellon University, May 1997.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[4] G. Booch. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[5] J. Bosch and P. Molin. Software architecture design: evaluation and transformation. *Proc. 1999 IEEE Engineering of Computer Based Systems Symposium.*

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

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

[8] 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.

[9] Y. Deville and K.-K. Lau. Logic program synthesis. *J. Logic Programming*, 19,20:321–350, 1994. Special Issue: Ten Years of Logic Programming.

[10] 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.

[11] D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissades. *Design Patterns – Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1994.

[13] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. *Proc. ICSE'95*, pages 179–185.

[14] W. Harrison, H. Osher, and H. Mili. Subjectivity in oo systems. In OOPSLA 94 & 95 workshop reports.

[15] R. Helm, I.M. Holland, and D. Gangopadhay. Contracts — specifying behavioural compositions in oo systems. *Sigplan Notices*, 25(10), 1990. *Proc. ECOOP/OOPSLA 90*.

[16] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 2nd edition, 1990.

[17] 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* 1817:43-62, Springer-Verlag, 2000.

[18] G. Larsen. Designing component-based frameworks using patterns in the UML. *Communications of the ACM* 42(10):38–45, October 1999.

[19] K.-K. Lau, S. Liu, M. Ornaghi, and A. Wills. Interacting frameworks in *Catalysis*. In J. Staples, M. Hinchey and S. Liu, editors, *Proc. Second IEEE Int. Conf. on Formal Engineering Methods*, pages 110-119, IEEE Computer Society Press, 1998.

[20] K.-K. Lau, C.D.M. Moss, and M. Ornaghi. Formal development of correct classes in computational logic. In D.J. Duke and A.S. Evans, editors, *Proc. BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing Series. Springer-Verlag, 1997. Abstract on page 3.

[21] 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.

[22] K.-K. Lau and M. Ornaghi. Forms of logic sepcifications: A preliminary study. In J. Gallagher, editor, *Proc. LOPSTR 96, Lecture Notes in Computer Science 1207*, pages 295–312. Springer-Verlag, 1997.

[23] 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* 1559:101-123, Springer-Verlag, 1999.

[24] K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. Logic Programming*, 38(3):259–294, March 1999.

[25] K.-K. Lau, M. Ornaghi, and A. Wills. Frameworks in *Catalysis*: Pictorial notation and formal semantics. In M. Hinchey and S. Liu, editors, *Proc. 1st IEEE Int. Conf. on Formal Engineering Methods*, pages 213–220. IEEE Computer Society Press, 1997.

[26] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[27] D.C. Luckham *et al*. Specification and analysis of system architecture using Rapide. *IEEE Trans, Soft. Eng.*, 1995.

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

[29] B. Meyer. *Eiffel the Language*. Prentice-Hall, 1992.

[30] O. Nierstrasz and D. Tsichritzis, editors. *Object-Oriented Software Composition*. Prentice-Hall International, 1994.

[31] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. *ACM Software Engineering Notes* 17(4):40-52, 1992.

[32] 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".

[33] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[34] T. Reenskaug *et al*. *Working with Objects*. Manning/Prentice-Hall, 1995.

[35] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[36] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[37] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley 1997.