Component Certification and System Prediction: Is there a Role for Formality?

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

Abstract

In this paper, we specify an open problem: predictable component assembly, *and state a position on* compositional reasoning techniques *that we believe are necessary for this purpose.*

1 A Rhetorical Question?

Yes, naturally, the question in the title is meant to be rhetorical. However, by formality I do not mean that which is exemplified by existing formal methods. Rather, I mean the use of *a priori reasoning*, instead of the prevalent *a posteriori reasoning* used for verification-based software construction.

2 What is A Priori Reasoning?

Verification-based methods take the '*posit-and-see*' approach: given the specification for a program, *first* posit a program, *then* see if the program is correct (wrt to the given specification); similarly, to construct a specified composite, *first* posit the components and their composition, *then* see if their composition meets the given specification. This is what I call *a posteriori reasoning*: reasoning about correctness (or other properties) takes place *after* the program or component has been constructed. In other words, it does not offer any help with the construction beforehand.

By contrast, *a priori reasoning* takes places before the construction takes place. In the context of CBSE, *a priori reasoning* assumes that it is possible to show *a priori* that the individual components in question are correct (wrt their own specifications). It then offers help with reasoning about the composition of these components, both to guide their composition in order to meet the specification of a larger system, and to predict the precise nature of any composite, so that the composite can in turn be used as a unit for further composition.

In other words, whereas *a posteriori* reasoning takes the *posit-and-see* approach, *a priori reasoning* tries to leave out the *posit* element altogether, and replace the *see* element by precise *prediction*.

3 A Priori Reasoning Demands Formality

Unlike *a posteriori* reasoning, which may or may not be formal (the former corresponds to *posit-and-see*, and the latter *posit-and-prove*), *a priori reasoning* demands formality because it is just not possible otherwise, as will become plain in Section 6.3.

4 Can Formality Succeed in CBSE?

But why should anyone believe that formality can be employed successfully in CBSE when existing formal methods have not made an impact on current software engineering practice? Well, obviously I cannot be absolutely certain that it can be, but I am reasonably convinced that *a priori reasoning* is practicable (and hence the associated formality) and it can help CBSE succeed,

5 Why does CBSE Need A Priori Reasoning?

My answer to this question is simply that I believe that *a priori reasoning* can deliver the key pre-requisites for CBSE to achieve its ultimate goal of *third-part assembly*.

6 What are the Key Pre-Requisites for CBSE?

I believe they are the following:

- a *standard semantics* of components and component composition (and hence reuse);

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

6.1 Standard Semantics

Without a standard 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.

6.2 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 and where we can deploy the component, i.e. its *context dependencies*. Otherwise, third-party assembly would not be possible. Therefore, (in contrast to [21]) 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.

6.3 Assembly Guide

The interface of a reusable component contains a collection of operations. In order to have a good assembly guide, we need to know what each operation does (correctly) and what its reuse within another component (after component composition) will yield. 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. This implies that the specification of component operations in its interface should be compositional.

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*, and it is the key to providing a good assembly guide.

7 What A Priori Reasoning can Deliver

As stated above, I believe *a priori reasoning* can deliver the pre-requisites for the success of CBSE.

7.1 Standard Semantics

As we have seen in Section 6.3, *a priori reasoning* requires formal semantics. If we can come up with a suitable semantics, then this semantics will (automatically) provide definitions for components and ther composition and reuse. Thus, achieving *a priori reasoning* will necessarily also provide a *standard semantics* for components and their composition and reuse.

7.2 Interfaces

Such a semantics in turn will include suitable declarative semantics for *interfaces*.

7.3 Assembly Guide

As we have also seen in Section 6.3, *a priori* correctness can provide an *assembly guide*.

7.4 A Spiral Model for CBSE

Another significant consequence of *a priori reasoning* is that *a priori* correctness can provide a hybrid approach that is both top-down and bottom-up for CBSE, as illustrated in Figure 1. First a library of *a priori correct* components has

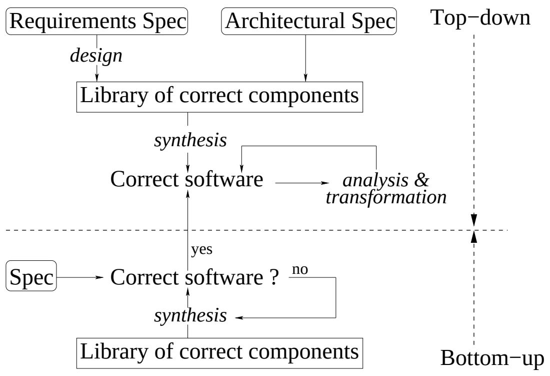

Figure 1. CBSE using a library of correct components.

to be built. The nature of *a priori correctness*, coupled with the use of *a priori reasoning*, then allows these components to be composed into larger systems in either a top-down or bottom-up manner, or indeed a combination of both.

Top-down development can follow the traditional *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 *a priori correctness*. Composition of correct 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.

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.

8 Predictable Component Assembly

I believe that *a priori reasoning* addresses an open problem, viz. *predictable component assembly*. It does so because it deals with component certification and system prediction.

8.1 Component Certification

Certification should say what a component does (in terms of its context dependencies) and certify that it will do precisely this (for all contexts where its dependencies are satisfied).

A priori correctness means that a component is guaranteed to meet its own specification, and will always remain correct even if and when it becomes part of a composite. It will therefore provide a basis for component certification. Such certification will tell us what behaviour to expect of a component in any appropriate context, and it should therefore engender confidence in certified components.

8.2 System Prediction

A priori correctness means we know before composition takes place what the result of composition will be, e.g. putting A and B together (with proper certification for A and B), we know what the result C will be. That is, the specification of C must be predictable prior to composition. Moreover, we will know how to certify C properly, and thus how to use C in subsequent composition.

For system prediction, such a compositional property forms the basis of *predictable component assembly*.

9 Existing Methods

Existing CBSE methods are not capable of predictable component assembly. In other words, they do not meet the key pre-requisites for CBSE as outlined in Section 6.

9.1 Object Technology

Existing CBSE is based on current object technology, i.e. UML-based tools together with middleware such as CORBA, COM and Enterprise Java Beans. In my view, the first problem here is the lack of a standard semantics. As yet these do not exist for components in general, and for specific kinds of components like *patterns* and *frameworks* in particular (and even *objects*?). Neither do they exist for component composition or reuse.

Moreover, existing object technology is, in my view, too low-level and bottom-up (and therefore hard to do in the absence of a clearly defined methodology). It is too much based on objects, and therefore not very reusable (as compared to frameworks, see e.g. [12, 17, 10]).

For CBSE 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 pre-requisite for this evolution would be *a priori reasoning*.

9.2 Formal Methods

General-purpose formal methods such as Z [20], VDM [13] and B [1] lack suitable semantics for components. They lack semantic characterisations of 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 [6, 9] 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 mentioned in the previous section, this is not the best approach for next-generation CBSE.

9.3 Software Architecture

Software architecture (see e.g. [19, 3]) would seem to be a good technique for component assembly. Already there are architecture description languages, e.g. Wright [2], and architectural design tools, e.g. Rapide [8]. However, software architecture is top-down, but current CBSE is essentially bottom-up (see e.g. [4]).

There is also a conflict between software architecture and current CBSE 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. [18]).

Another problem with software architecture is the so-called Architectural Mismatch problem [11] 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.

10 So What Kind of Formality do we Need?

I believe that we need a new approach to, and hence a new generation of, formal methods. For one thing, we must build formal semantics into tools with good user interfaces, in order to obviate the need for non-expert users to grapple with impenetrable mathematics. Mastering these should not be a pre-requisite, just like in Engineering, where tools are based on a mature science, but apprentices can use tools without mastering the underlying theory. In other words, we need a formality that can be put into practical use, e.g. by being captured in a standard library of certified components.

11 A First Step Towards A Priori Reasoning

So what have we done to give us any confidence in our belief that *a priori reasoning* is practicable? We have characterised a notion of *a priori correctness*, that we call *steadfastness* [16], in the context of Computational Logic (see e.g. [15].), and we are applying this to frameworks in the CBSE methodology *Catalysis* [10], see e.g. [14].

12 Summing Up

The goal of CBSE is to provide the engineering science and tools for constructing software products by plugging components together, like building hardware from kits of component parts. However, at present the key pre-requisites for CBSE to succeed have not been met (see e.g. [5]).

In the long run, for CBSE to ultimately succeed, I believe software components must become like hardware components, with universal standards and component labelling. To achieve this, we need an "Industrial Revolution for IT", and I believe *a priori reasoning* can play a crucial role in this revolution.

References

[1] J. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie-Mellon University, 1997.

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

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

[5] A. Brown and K. Wallnau. The current state of CBSE. *IEEE Software*, Sept/Oct 1998:37–46, 1998.

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

[9] D. D'Souza and A. Wills. Extending Fusion: practical rigor and refinement. In R. Malan *et al*, editor, *Object-Oriented Development at Work*. Prentice-Hall, 1996.

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

[11] 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.

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

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

[14] J. Küster Filipe, K.-K. Lau, M. Ornaghi, K. Taguchi, A. Wills, and H. Yatsu. Formal specification of Catalysis frameworks. In *Proc. 7th Asia-Pacific Software Engineering Conference*, pages 180–187. IEEE Computer Society Press, 2000.

[15] 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 *Proc. LOPSTR 99, Lecture Notes in Computer Science*, volume 1817, pages 43–62. Springer-Verlag, 2000.

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

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

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

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

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

[21] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Harper and Row, 1986.