Component-based Software Development and Logic Programming

Kung-Kiu Lau

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

Abstract. Component-based Software Development (CBD) represents a paradigm shift in software development. In the tutorial I will explain what CBD is about, briefly survey current component technology, and posit that Logic Programming can play a role in next-generation CBD. In this abstract, I give an overview of the material that I plan to cover in the tutorial.

1 Introduction

Software Engineering is entering a new era: the *component* era. Building on the concepts of Object-Oriented Software Construction (e.g. [18]), Component-based Software Development (CBD) aims to move Software Engineering into an 'industrial age', whereby software can be assembled from components, in the manner that hardware systems are constructed from kits of parts nowadays.

Thus CBD represents a paradigm shift in software development: from building monolithic, single-platform, purpose-built-from-scratch systems to constructing assemblies of ready-made components that are platform-independent and supplied by third-parties.

2 What are Software Components?

The first thing to say is that there is as yet no universally accepted standard definition of what a component is. Equally, there is no standard terminology for CBD at present.

Nevertheless, the idea of using components to build software systems is as old as Software Engineering itself. Indeed it is the motivation behind any program units that purport to enable programmers to structure and to reuse their code. These range from sub-routines, modules, packages, etc., that have been familiar for a long time, to commercial off-the-shelf software, or COTS (e.g. [28]), and web services, that are relatively recent. It may be too simplistic to describe such components as just 'reusable blocks of software', but this does capture the essence of what a component is.

Despite the lack of a standard definition and terminology, there is consensus that to qualify as a software component, a piece of software should have the following characteristics:

Encapsulation. The inner workings of a software component must be hidden from the outside world that it exists in.

Use of an Interface. A software component must present and make use of a useful *interface* which the outside world can interpret so that other components know what services it gives or *exports*.

The interface should also specify what the component needs to *import* in order to carry out its operations.

The interface must conform to the accepted standard used in the world which the component works in. Such a standard is sometimes called a *component model*. It specifies, among other things, how component interfaces are constructed, as well as the *middleware* that allows components to interact with one another via their interfaces.

Reusability. A software component must be *reusable* so that it can be used in multiple application contexts.

This enables developers to reduce the effort in developing new applications.

Replaceability. A software component must be able to *replace*, or be replaced by, other existing components.

The new component must at least be able to provide the same services as the old component so that components relying on the services of the old component will still work.

It must also have an interface that other components can still recognise and utilise.

Inter-operability. The services of a software component must be accessible on any platform.

Therefore software components must adhere to standards for component interfaces and middleware services.

A widely accepted definition of a software component is the following, due to Szyperski [26]:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only.
> A software component can be deployed independently and is subject to composition by third parties."

This definition requires additionally that interfaces be specified contractually, i.e. with obligations and rights for both the component and its clients, and *context dependencies*, which define how and where the component can be deployed, be specified explicitly. Furthermore, components can be produced, acquired and deployed independently, and Szyperski argues that this in turn requires a component to be in binary form.

Heineman and Councill [10] define a component as:

> "A software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."

This definition is thus predicated on the existence of a component model.

3 Current Component Technology

A component model is a specification. For implementation, it has to be realised by a technology that provides the 'wiring' standard, or plumbing, for integrating software components according to the standards defined by the component model. In other words, a component technology provides the middleware that allows components to interact with one another (in accordance with the component model).

Implementing middleware requires enormous resources, and not surprisingly is the preserve of large industrial enterprises. Currently, there are 3 main industrial component models (and technologies):

- CORBA (Common Object Request Broker Architecture) [20, 7]
  This is a standard defined by OMG (www.omg.org), the Object Management Group.
- COM (Component Object Model) [6, 2]
  This is the standard marketed by Microsoft (www.microsoft.com).
- EJB (Enterprise JavaBeans) [25, 9]
  This is marketed by Sun (www.sun.com).

All these are essentially wiring standards for integrating objects, in the sense of Object-Oriented Programming, distributed over networks, and therefore collectively constitute a distributed object technology. This technology is based on the *remote procedure call* (RPC) and *remote method invocation* (RMI) mechanisms. As a consequence, although the component models differ in various aspects, they can all access objects (and services) remotely and across platforms.

The details of these component models are extremely complicated. In the tutorial I will attempt to explain the basic underlying ideas and principles, and to outline the key characteristics, of each approach.

Naturally these component models are constantly evolving, CORBA is developing into the CORBA Component Model (CCM) [1], also called CORBA 3, and COM has metamorphosed into .NET [19, 22].

4 Component-based Software Development

For software development, the so-called Unified Methodology is the industry standard. It combines the Unified Software Development Process [11] with the Unified Modeling Language (UML) [23, 27] which is used for specifying and designing software systems. UML also has its roots in Object-Oriented Software Construction, with objects as units of design and reuse.

Whether the Unified Methodology, in particular UML, can be used successfully for CBD remains to be seen. The notion of a component in the original UML was only an artefact at implementation time rather than design time, so strictly speaking it was not part of UML at all.

To introduce components into UML, i.e. to introduce them at design time, Cheesman and Daniels [4] define a component as a UML stereotype which consists of a class with an interface. However, it is not clear how this stereotype can be used to define components other than single classes, e.g. Catalysis frameworks [8] which are groups of interacting objects. The problem is that the Object Constraint Language (OCL) [29], part of UML, cannot deal with components that are not single objects, since it defines object constraints in the context of a single class.

However, here things are in flux, with a major upgrade of UML to UML 2.0 in the pipeline, and the introduction of OMG Model Driven Architecture (MDA) [21], which definitely aims to compose components. So we should 'watch this space'.

5 Is there a Role for Logic Programming?

Given that Logic Programming (LP) has never really made an impact on Software Engineering (SE), it would seem odd to posit that LP has a role to play in CBD [13, 16], but that is precisely what I would like to do.

Firstly, as we said in [17], we believe LP's role in SE is to co-exist with the predominant imperative, in particular the OO, paradigm. Moreover, we believe LP's role in this co-existence is to address the critical kernel of a software system, for which there is no doubt that LP would be superior, for the reason that LP can deliver software correctness.

It is generally accepted that the critical kernel of a software system usually consists of some 20% of the code (see Figure 1), and it is this code that needs

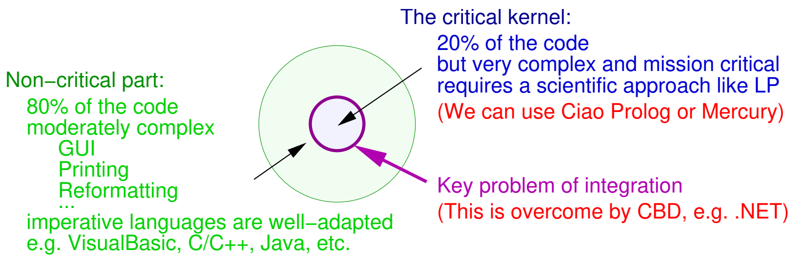

Fig. 1. The integration barrier between LP and predominant paradigms in SE.

a scientific approach such as LP affords. However, even if LP was used for the critical code, the problem of integrating (critical) LP code with (non-critical) code in the predominant (imperative) languages would at best be difficult. For example, we could use a foreign function interface, usually in C, but this is often difficult. Thus, as also shown in Figure 1, there is an integration barrier between LP and the predominant paradigm in SE. This barrier would have to be overcome even if we were to use LP just for the critical kernel.

We contend that CBD provides just the right impetus for removing this integration barrier. Since the the ultimate goal of CBD is third-party assembly of independently produced software components, which must be possible on any platform, it means that the components must be platform- and paradigm-independent. As a consequence, CBD offers a *level playing field* to all paradigms. Current approaches to industrial SE cannot address all the needs of CBD [3], so the playing field is level and LP is not at any disadvantage.

By definition, component technology supports the level playing field, and we believe that a feasible practicable approach is to interface a suitable LP language, such as Ciao Prolog [5] and Mercury [24], to a current component technology. For example, we think that .NET, Microsoft's new component platform, could give LP the necessary component technology (see Figure 1). As any language on a .NET platform can seamlessly interoperate with any other language on .NET (at least at a very low level), we have, for the first time, the possibility to build the critical components using the LP paradigm, while the non-critical, more mundane, components are still OOP-based.

In particular, we believe the declarative nature of LP, and Computational Logic in general, can play a crucial role in addressing key pre-requisites for CBD to succeed, such as component certification, predictable assembly [15], a priori reasoning [12], trusted components [14], etc.

References

1. BEA Systems *et al.* CORBA Components. Technical Report orbos/99-02-05, Object Management Group, 1999.
2. D. Box. *Essential COM.* Addison Wesley, 1998.
3. A.W. Brown and K.C. Wallnau. The current state of CBSE. *IEEE Software*, Sept/Oct 1998:37–46, 1998.
4. J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-based Software.* Addison-Wesley, 2001.
5. The Ciao Prolog Development System. http://clip.dia.fi.upm.es/Software/Ciao/.
6. COM web page. http://www.microsoft.com/com/.
7. CORBA FAQ web page. http://www.omg.org/gettingstarted/corbafaq.htm.
8. D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, 1999.
9. EJB web page. http://java.sun.com/products/ejb/.
10. G.T. Heineman and W.T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley, 2001.
11. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.
12. K.-K. Lau. A priori reasoning for component-based software development. In M. Carro, C. Vaucheret, and K.-K. Lau, editors, *Proc. 1st CoLogNET Workshop on Component-based Software Development and Implementation Technology for Computational Logic Systems*, pages 5–19. Technical Report CLIP4/02.0, School of Computer Science, Technical University of Madrid, 2002.
13. K.-K. Lau. The role of logic programming in next-generation component-based software development. In G. Gupta and I.V. Ramakrishnan, editors, *Proc. Workshop on Logic Programming and Software Enginering, London, UK, July 2000.*

14. K.-K. Lau. Some ingredients of trusted components. In *Proc. Workshop on Trusted Components*, pages 1–5. January 2003, Prato, Italy.
15. K.-K. Lau. Component certification and system prediction: Is there a role for formality? In I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau, editors, *Proc. 4th ICSE Workshop on Component-based Software Engineering*, pages 80–83. IEEE Computer Society Press, 2001.
16. K.-K. Lau and M. Ornaghi. Logic for component-based software development. In A. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond*, LNAI 2407, pages 347–373. Springer-Verlag, 2002.
17. K.-K. Lau and M. Vanden Bossche. Logic programming for software engineering: A second chance. In P.J. Stuckey, editor, *Proc. 18th Int. Conf. on Logic Programming*, LNCS 2401, pages 437–451. Springer-Verlag, 2002.
18. B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, second edition, 1997.
19. Microsoft .NET web page. http://www.microsoft.com/net/.
20. Object Management Group. The Common Object Request Broker: Architecture and specification Revision 2.2, February 1998.
21. OMG Model Driven Architecture. http://www.omg.org/mda/.
22. D.S. Platt. *Introducing Microsoft .NET*. Microsoft Press, 3rd edition, 2003.
23. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
24. Z. Somogyi, F. Henderson, and T. Conway. Mercury – an efficient, purely declarative logic programming language. In *Proc. Australian Computer Science Comference*, pages 499–512, 1995.
25. Sun Microsystems. Enterprise JavaBeans Specification. Version 2.0, 2001.
26. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.
27. Introduction to OMG UML. http://www.omg.org/gettingstarted/what_is_uml.htm.
28. K. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. Addison-Wesley, 2001.
29. J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 1999.