Some Ingredients of Trusted Components

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

Abstract

Components must be trustworthy if they are worth deploying at all. In this position paper, we identify some ingredients that we think are essential for enabling components to be trustworthy.

1 Introduction

Software components must be trustworthy; otherwise there would be no incentive to deploy them whatsoever. The Trusted Components Initiative [13] defines trust as a social phenomenon, and proposes to build trust from a combination of formal and informal approaches.

In this short paper we focus on the formal aspects of component contracts and their fulfilment. However, even in this context, trustworthiness means not just contract fulfilment but also how, when and where it can be guaranteed. We identify some ingredients that we think are essential for enabling components to be trustworthy in this regard.

By components we do not mean binaries without any specifications, such as COTS, or distributed objects in a current component technology such as CCM [1] or EJB [17]. Rather, we mean generic components, not necessarily objects, where not only specifications but also source codes may be available.

2 Formal Specifications and Contracts

A formal specification is the basis for contracts [12], and should therefore specify all the necessary ingredients of a component and its contracts. To this end, we specify a component as follows [8]:

Name	
CONTEXT($\Pi_1, \Pi_2, \ldots$)	
Signature:	$\ldots$;
Axioms:	$\ldots$;
Constraints:	$\ldots$;
INTERFACE	
Operations:	specifications;
	op1(π_1), op2(π_2), $\ldots$;
Dependencies:	$\Pi_1, \Pi_2, \ldots, \pi_1, \pi_2, \ldots$;
	constraints;
CODE	
Code for op1, op2, $\ldots$	

Figure 1: Specification of a component.

The *context* defines the theory underpinning the component. This theory is defined by the *signature* and the *axioms*. The theory may be parametric, with parameters $\Pi_1, \Pi_2, \ldots$; the *constraints* define the conditions on admissible parameters.

The *interface* defines the *operations* $op1, op2, \ldots$ (together with their specifications) and the *context dependencies* of the component.[1] The former are the services the component *provides* (or *exports*). The contracts for these services are the specifications of the operations.

[1] This differs slightly from [18], where an interface contains only the operations.

The context dependencies consist of the context parameters $\Pi_1, \Pi_2, \ldots$ together with the constraints on them, and the operation parameters $\pi_1, \pi_2, \ldots$. They define what the component *requires* (or *imports*) in order to deliver its provided services.

This approach to specifying components is very generic. The component may be just a class or ADT. It may be a module, in particular what Meyer [12] calls an *abstracted module*, which is the basic unit of reuse in RESOLVE [16]. It may be an object model [2] as in OMT [14] or UML [15]. It may yet be an OO Design framework, i.e. a group of interacting objects [6], such as frameworks in *Catalysis* [3, 5]. It could even be a design pattern or schema [4].

3 A Priori Compositional Reasoning

Given components specified in the above manner, we can reason about their *reuse behaviour*, i.e. when deployed, or composed with other components, in all compatible contexts. In particular, we can reason about the fulfilment of their contracts in these contexts, i.e. their trustworthiness. However, the nature of component reuse demands that such reasoning should also be *a priori*, i.e. users should be convinced that the components will always be trustworthy *before* deploying or reusing them.

We call this kind of reasoning *a priori reasoning* [7] because it is supposed to take place *before* the reuse or composition occurs, i.e. contract fulfilment should be guaranteed *a priori*.[2] In other words, *a priori reasoning* requires that it be possible to show *a priori* that the individual components in question will be correct with respect to their (possibly parametric) specifications in *all* compatible contexts.

The corollary of this is that *a priori reasoning* is able to not only provide an assembly guide for choosing the right components for building a specified composite, but also to predict the result of an assembly and use the resulting composite as a component in further assembly. In other words, with *a priori reasoning* we can do component *certification* as well as *assembly prediction*, which means that we can do *predictable component assembly*.

4 Predictable Component Assembly

Consider Figure 2. Two components A and B each have their own interface and code.[3] If the composition of A

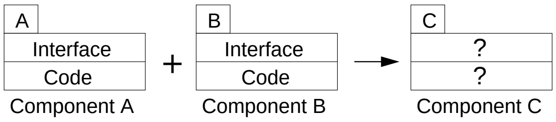

Figure 2: Predicting component assembly.

and B is C, can we determine or deduce the interface and code of C from those of A and B? The answer lies in component certification.

4.1 Component Certification

Certification should say what a component does, i.e. what its contracts are, in terms of its context dependencies, and should guarantee that it will fulfil its contracts, for all contexts where its dependencies are satisfied. A certified component, i.e. its interface, should therefore be specified properly, and its code should be verified against its specification. Therefore, when using a certified component, we need only follow its interface. In contrast, we cannot trust the interface of an uncertified component, since it may not be specified properly and in any case we should not place any confidence in its code.

In the context of *a priori reasoning*, a certified component A is *a priori correct*. This means that:

- A is guaranteed to be correct, i.e. to meet its own specification;
- A will always remain correct even if and when it becomes part of a composite.

This is illustrated by Figure 3, where component A has been certified, so we know how it will behave in the composite C.

[2] As opposed to *a posteriori*, as exemplified by verification-based methods for program construction, which take the *posit-and-see* approach to composition.

[3] For simplicity, we ignore the context, since it plays a less major role here.

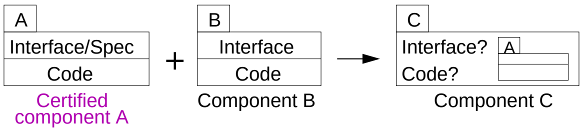

Figure 3: Component certification.

However, we do not know how B will behave in C, since it is not certified. Consequently, we cannot expect to know C's interface and code from those of A and B, i.e. we cannot predict the result of the assembly of A and B.

4.2 Assembly Prediction

For assembly prediction, obviously we need *all* constituent components to be certified (*a priori correct*). Moreover, for any pair of certified components A and B whose composition yields C:

- *before* putting A and B together, we need to know what C will be;
- and furthermore, we need to be able to certify C.

This is illustrated by Figure 4. The specification of C must be predictable prior to composition. Moreover, we

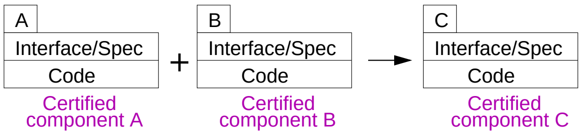

Figure 4: Assembly prediction.

need to know how to certify C properly, and thus how to use C in subsequent composition. *A priori correctness* is just what we need in order to do assembly prediction.

5 A Priori Correctness

In our work the basis for a priori reasoning is *a priori correctness*. Specifically, we consider a notion of a priori correctness of the operations in a component, that we call *steadfastness*.

A *steadfast* operation (program) Op is one that is correct (with respect to its specification) in each intended model of the context $\mathcal{C}$ of the component. Since the (reducts of the) intended models of its specialisations and instances are intended models of $\mathcal{C}$, a steadfast program Op is correct, and hence correctly reusable, in all specialisations and instances of $\mathcal{C}$.

A formalisation of steadfastness is given in [9], with both a model-theoretic, hence *declarative*, characterisation and a proof-theoretic treatment of steadfastness. For lack of space, we do not discuss these theoretical details here.

6 Steadfast Components

Since a steadfast program is correct, and hence correctly reusable, in all specialisations and instances of its context, a component with steadfast operations, which we will call a *steadfast component*, when composed with another steadfast component will also be steadfast. In other words, steadfastness is not only *compositional*, but is also preserved through *inheritance* hierarchies.

Consequently, in the context of assembly prediction (Figure 4) when composing steadfast components, not only can we be sure that the composite will be steadfast, but we can also predict the specification of the composite. This is illustrated in Figure 5.

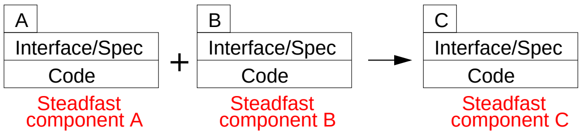

Figure 5: Composing steadfast components.

This stands in contrast to current approaches to modular (formal) specification and verification, e.g. [10, 16] which use *modular reasoning*. This is specification-based reasoning that tries to say before running the software whether it will behave as specified or not (subject to relevant assumptions). This is illustrated in Figure 6. Before a composite module C is deployed, we can predict whether it will work according to its specification. For

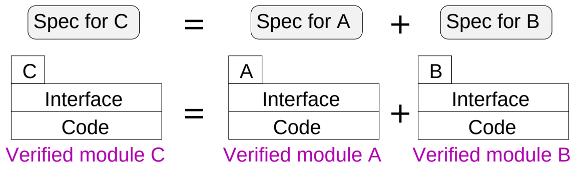

Figure 6: Module composition.

example, if component modules, say A and B, are to be used in C, the correctness of C is established based on the specifications of A and B (even before A and B have been implemented). The components A and B are then verified independently. The contexts of A and B are taken in account when using and verifying A and B.

Thus modular reasoning predicts correctness, from specification. This kind of prediction is we believe subtly different from the prediction that we intend to convey in Figure 4, which predicts specification (and correctness), from (certified) correctness.

In other words, modular reasoning needs to know the specification of the composite before predicting if the composite will work according to its specification. In contrast, steadfastness offers the advantage of being able to predict the specification (and correctness) of the composite prior to composition. Thus, with steadfast modules, we can do assembly prediction as shown in Figure 5. This is illustrated in Figure 7.

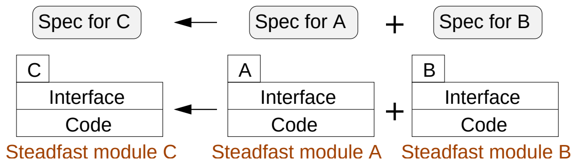

Figure 7: Assembly prediction with steadfast modules.

7 Prove Once, Reuse Often

The most important potential benefit of a priori reasoning about reuse is that the a priori correctness of each component needs to be proved only *once*.[4] This should mitigate the main reason for the lack of impact of formal methods on industrial practice, viz. that formal techniques are very costly and difficult to apply [13]. In principle, at least, with a priori reasoning, we should be able to adapt the reuse motto 'build once, reuse often' into 'prove once, reuse often'.

8 Conclusion

We have identified some ingredients that we think are essential for the development of trusted components, all underpinned by the notion of a priori reasoning about contract fulfilment during component reuse. The key benefit of this approach is that it can potentially contribute to the reconciliation of formal methods with reuse [11], viz. that the only way that a breakthrough can be achieved in trusted components is for the two to work in harness.

References

[1] BEA Systems *et al.* CORBA Components. Technical Report orbos/99-02-05, Object Management Group, 1999.

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

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

[4] 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.

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

[4] Of course, for each deployment, it may be necessary to perform compatibility checks.

[6] 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.
[7] 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, *Proceedings of the Fourth ICSE Workshop on Component-based Software Engineering*, pages 80–83. IEEE Computer Society Press, 2001.
[8] K.-K. Lau and M. Ornaghi. A formal approach to software component specification. In D. Giannakopoulou, G. Leavens, and M. Sitaraman, editors, *Proceedings of Specification and Verification of Component-based Systems Workshop at OOPSLA2001*, pages 88–96, 2001. Tampa, USA, October 2001.
[9] K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. Logic Programming*, 38(3):259–294, March 1999.
[10] G. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, pages 72–80, July 1991.
[11] B. Meyer. The next software breakthrough. *IEEE Computer*, July 1997.
[12] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, second edition, 1997.
[13] B. Meyer, C. Mingins, and H. Schmidt. Trusted components for the software industry. http://trusted-components.org/documents/tc_original_paper.html.
[14] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Sorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
[15] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
[16] M. Sitaraman and B. Weide, editors. *Component-based software using RESOLVE*. Special feature, ACM Sigsoft Software Engineering Notes 19(4): 21-65, October 1994.
[17] Sun Microsystems. Enterprise JavaBeans Specification. Version 2.0, 2001.
[18] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.