VICTORIA UNIVERSITY OF WELLINGTON

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Modelling the Checkability of Reusable Code

Ewan Tempero and Robert Biddle

Technical Report CS-TR-96/20
1 November 1996

Abstract

An important aspect of reusing code is making sure it is used correctly. The easier it is to do this, the easier it is reuse the code with confidence. However techniques developed to do such checking, such as type systems, are not universally accepted because they are regarded as being too difficult to use or too restrictive. We discuss a model of reusability in which the study of techniques for the "checkability" of code might be carried out.

Keywords: Reusability, Programming Languages, Type systems, Specifications

Workshop Goals: Develop models for reusability, Analyse and understand programming language support, Understand impact of specification on reusability

Working Groups: Rigorous Behavioral Specification as an Aid to Reuse

1 Background

Many people regard the only way to get the benefits from reuse is through systematic reuse programs. One aspect of such programs is the existence of repositories of reusable code — as Tracz says, "In order to reuse software, there needs to be software to reuse" [Tra95]. One possible inhibitor to acceptance of reuse programs is the cost of the creation and maintenance of repositories of reusable code. Studies have shown that the cost of developing reusable code is typically two times the cost of developing code for single use. Our interest is in reducing the cost of creating reusable code.

2 Position

There have been many proposals and discussions about how to create reusable code, from guidelines for its creation [JF88], to new language features and disciplines with support tools and formalisms [LSLX94, SW94]. There are many ideas, and it is difficult to understand how they are related, and exactly how they truly improve the development of reusable code. To help us relate the different ideas, we have developed a model that helps us understand how language features impact the reusability of code [BT96b]. One aspect of our model is *checkability*, which is about checking that code is being used correctly. There seems to be a tradeoff between checkability of code and the cost of creating and using it. We would like to refine our model to better describe existing checkability techniques so as to understand what the tradeoff really is.

2.1 Model foundations

Our model is based on two fundamental concepts: *assembly* and *dependency*.

An assembly is the unit of reusability. It is any set of descriptions of source-level code. This allows us to describe any pattern that associates pieces of code together, and, since one possible description of source-level code is an assembly, the description can be hierarchical. This allows us to describe any source-level construct that can be reused, such as "basic blocks", functions, classes, class hierarchies, C++ namespaces, or Java packages. Having such a broad definition of reusable unit means that, for example, our model unifies the concepts of component and generative reuse [BT96a].

Assemblies interact in different ways, such as through passing data from one to another, through control passing from one to another, or through one referring to another. Each of the ways assemblies interact represents a *use* of one by the other. This use has directionality, namely the direction that the control or data passes, or the reference to one by the other, so we refer to one assembly *using* another. The assembly doing the using is the *context*; the used assembly is the *component*.

In order for a context to use a component, conditions have to be met by one or both assemblies. Any such condition is a *dependency*.

Dependencies will usually apply to both the context and component. The context will have to meet certain conditions before it can correctly use the component (such as pass parameters of the correct type) and the component will also have to meet certain conditions before the context can use it (such as have member functions with the right name).

2.2 Component and Context Reuse

Our model is intended to reflect the kind of code arrangements depicted in figure 1. The two assemblies on the left use the assembly in the middle. In this situation, we have multiple contexts using the same component, that is, the component is being reused. We call this

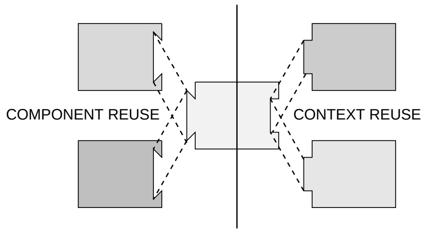

Figure 1: *Component Reuse, as shown on the left of the diagram, involves several different contexts invoking the same component. Context Reuse, as shown on the right of the diagram, involves one context invoking different components using the same interface.*

component reuse. An important aspect of this structure is that it also works in reverse: we have a context, and use it with different components, as shown on the right of the figure. For example, we might have context code that calls a procedure, and in a different setting we may need the same context code, but want a different procedure to be called. We call this *context reuse*.

Context and component are really *roles* played by assemblies. As the centre of the figure shows, an assembly can play both roles: the component role because it is invoked by other contexts, and the context role because it invokes other components. This structure reflects the programming language mechanisms that have supported reusability for many years: macro definition and expansion, and procedure definition and call. However, it also applies to more recent concepts, such as the definition and instantiation of classes in object-oriented programming.

2.3 Dependencies and reusability

Dependencies can adversely affect the reusability of an assembly, because they limit the number of contexts that can invoke it, or the number of assemblies it can invoke. However, there are dependencies that are beneficial to reusability, by allowing generality and checkability of assemblies. Generality is about the number of contexts that can use an assembly as a component, or the number of components that can be invoked by an assembly in the context role. Increasing generality can be achieved by removing, changing, or even adding appropriate dependencies. Checkability is the degree to which it can be checked that an assembly's dependencies have been met.

2.4 Compositional Reuse and Generative Reuse

So far all our discussion about support for reusability has concerned what is often called "compositional reuse". This term is usually used to enable a distinction with another approach to reuse: "generative reuse". Typically compositional reuse involves direct reuse of code in the form it exists; generative reuse involves indirect reuse of code by transforming the code in some way for the reuse. Although generative reuse is a more complex approach, it is regarded as making reuse possible in circumstances where simple compositional reuse would be difficult, or tedious, or both. However, the simplicity of compositional reuse can be an advantage, because it allows more direct support in programming languages and environments.

Many discussions about generators focus on dealing with the input specifications and the process of transformation. However, there is another critical element involved, because neither the input specifications nor the transformation algorithm can create the necessary output code. Involved somewhere are *code fragments* that can form the basis for the output code, subject to transformation according to input specifications.

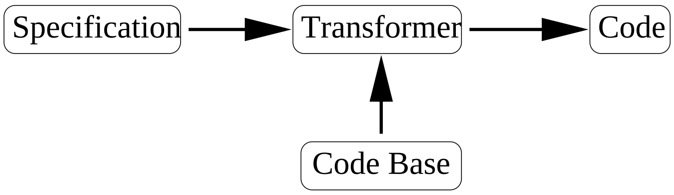

Figure 2: *Generator structure, showing basic elements.*

The process involved in generative reuse is thus typically regarded as having four elements, as shown in figure 2. The input consists of specifications, and the output consists of usable code. The generator takes the input specifications, accesses the code base, and produces the output code. In terms of our model, we see the code base as a set of assemblies. The assembly concept covers a range of code structures, and different generators involve different points along this range. The different kinds of assembly used in generators correspond to other differences in generators.

In terms of our model, the prime benefit of generative reuse concerns generality. Because the transformation process can modify the assemblies in arbitrarily complex ways, the generality of the assemblies can be increased as desired, and with fine precision. In a way, the generality possible means that generative reuse resembles "white-box" reuse, because the transformation system has access to the internal structure of the code.

As well as improved generality, generative reuse also allows related improvements in checkability. As the transformation of the code is carried out, it is possible to do sophisticated checking of dependencies between assemblies. Sophisticated checking requires a more complex transformation system, however, and it will be more difficult to check dependencies involving invocation beyond the set of assemblies involved at transformation. Moreover, some dependencies will involve run-time invocation, and so cannot be checked at an earlier time of transformation. These points are discussed in more detail elsewhere [BT96a].

2.5 Checkability

Knowing that an assembly has been used correctly makes it easier to reuse code with confidence. The easier it is to check this — the more checkable it is — the better reuse of the assembly is supported.

Several features of programming languages directly address this need for checkability, from simple matching of names, to more sophisticated checking of types. In our model, we see such features as introducing new dependencies (on names, on types, etc.) precisely because these dependencies can be easily checked, so supporting checkability. However, supporting checkability is a complicated matter. To start with, checkability becomes more difficult to support where assemblies have more generality — where they can be correctly used in conjunction with a larger number of other assemblies. For example, support for increased generality such as generics and polymorphism also requires more sophisticated approaches to type checking.

In addition to the difficulties in supporting checkability across a range of generality, there are also other practical concerns. We must be concerned with the *time* checking can be done. For example, run-time checking allows great generality, but can be problematic. Not only does run-time checking reduce performance, but it comes late in development, so delaying the confidence in reuse arbitrarily long. On the other hand, static checking can not only affect the generality of code, but also the usability. For example, where programmers must work hard to match types in order to reuse code, they may decide to write new code instead of reusing at all. Moreover, typical type-checking only provides a limited amount of checkability, and does not fully address whether an assembly is being used correctly.

Full checking requires extra information, that is, some kind of "specification", in order to be able to carry out the necessary checks. However there is a continuing debate as to how useful such checks are. Even a technique as simple as type checking is regarded by some as too restrictive, while the more sophisticated checking available through the use of formal methods are regarded as too difficult to use by most programmers. Thus there seems to be a conflict — increasing checkability appears to require decreasing other properties that make code reusable.

Work needs to be done to determine what the relationship is between the expressiveness of the specification for an assembly and the assembly's reusability. This will require refining our model to better explain how dependencies and checkability interact.

3 Comparison

The most common form of specification available in programming languages are type systems. There are a number of ongoing efforts to improve the expressiveness of type systems [AC96, CL95, Mey95]. As yet, no attempt has been made to evaluate the results of these efforts on reusability. The RESOLVE project is investigating the impact of more rigorous behavioural specifications on the reusability of code [SW94, Edw96]. Other efforts involve specifications for new kinds of reusable units, such as GenVoca Realms [BG96].

References

[AC96] Martin Abadi and Luca Cardelli. On subtyping and matching. *ACM Transactions on Programming Languages and Systems*, 18(4):401–423, 1996.

[BG96] Don Batory and Bart J. Geraci. Validating component compositions in software system generators. In Murali Sitaraman, editor, *Proceedings of the Fourth International Conference on Software Reuse*, pages 72–81. IEEE Computer Society Press, April 1996.

[BT96a] Robert Biddle and Ewan Tempero. Modeling units of reusability. Technical Report CS-TR-96/19, Victoria University of Wellington, November 1996.

[BT96b] Robert Biddle and Ewan Tempero. Understanding the impact of language features on reusability. In Murali Sitaraman, editor, *Proceedings of the Fourth International Conference on Software Reuse*, pages 52–61. IEEE Computer Society Press, April 1996. Also available as Technical Report CS-TR-95/17.

[CL95] Craig Chambers and Gary Leavens. Typechecking and modules for multi-methods. *ACM Transactions on Programming Languages*, 17(9), November 1995.

[Edw96] Stephen Edwards. Representation inheritance: A safe form of "white box" code inheritance. In Murali Sitaraman, editor, *Proceedings of the Fourth International Conference on Software Reuse*, pages 195–204. IEEE Computer Society Press, April 1996.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.

[LSLX94] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, pages 94–101, May 1994.

[Mey95] Bertrand Meyer. Static typing and other mysteries of life, 1995. Keynote address at OOPSLA'95.

[SW94] Murali Sitaraman and Bruce Weide. Special feature: Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):21–67, October 1994.

[Tra95] Will Tracz. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Addison-Wesley, 1995.