Dynamically reconfigurable software components

Samuel Kamin & Lars Clausen
University of Illinois at Urbana-Champaign
1304 W. Springfield
Urbana, IL 61821
{kamin,lrclause}@cs.uiuc.edu

October 31, 2001

1 The need for small, flexible components

Two forces in the modern software engineering environment suggest a style of program development in which small functional units — which we call *components* — are separately deployed and combined to create complete applications. These forces are the intense time-to-market pressures which require great reusability of software, and the decreasing size and increasing ubiquity of computers with communication capabilities. Traditional components — subroutine libraries, COM or CORBA components — are highly deployable but are too large. They provide extremely useful reusability at a certain level, but they do not account for the most common case: when a deployable unit is not *quite* what is needed, and needs to be modified. For these cases, class libraries are better, but they are too bulky for some applications.

The new world of ubiquitous, interacting devices will require that code be highly mobile. In a collection of mutually-communicating devices, there may be no standard configuration that all members can assume, except for the most generic services. Just as installation of a large application on a current workstation can involve the installation, or re-installation, of other applications, so the small devices of the future will frequently need to install small services on other devices with which it is attempting to collaborate.

Thus, we envision the need for small functional units of software. They will be easily deployable, like COM components. But they must also be very flexible to allow for reusability — like class libraries, but much more

so. Our best examples of components that have this level of generality are *macros*, as in Lisp or the C++ template library. Unfortunately, macros do not quite pass the deployability test, because they require re-compilation. Indeed, some experts deny that macros are components at all.

2 Components are program generators

Thus, deployability is in direct conflict with flexibility, yet both are needed. A way out of this dilemma is to use *program-generating programs*, a.k.a. *two-level programs*. Since they are themselves programs, they are highly deployable, but they can be arbitrarily sophisticated in the ways they choose to generate programs and therefore are highly flexible.

Self-installing applications typically seen on PC's are an example of this, albeit at a rather gross level. They can take advantage of certain detectable parameters of the system — e.g. the presence of some software packages, user preferences indicated in a dialogue — to customize the installation. Unlike COM components, the executable installers can produce very different installations at different times.

The model of components as two-level programs is quite general. Because programs are generated late (at load time or run time), the program generator can take into account numerous local conditions. Run time constants can be propagated. Large and small components are both included in the model, since the generated code may be arbitrarily efficient, avoiding function calls, etc. (The latter is a benefit often claimed for the C++ template library.)

Thus, in our view, software of the future will be distributed as two-level code, to be run on the client's computer (with appropriate parameters provided) to produce the executable application. This will permit highly abstract, parameterized, reusable code that is, at the same time, highly efficient.

3 Producing program generators

There is one gap in our presentation: we haven't said how to produce such program-generating components. This is ordinarily done in one of two ways: manually or automatically. Manually written programs generators usually produce their result in the form of a string or abstract syntax tree that must be compiled after it is generated; producing machine code directly is too difficult. Program generators can be produced automatically by, for example,

partial evaluation; however, there is a limit to the kind of program generators that can be produced this way. Thus, manually written generators are source-based and therefore fail the deployability test, while automatically produced generators have limited functionality.

In our research, we have used the concept of *compositional compilation*, drawing on the concept of *compositionality* from denotational semantics, to produce flexible, binary code generators using a two-level language. That research is described in [1]. Compositionality allows programs to be split into almost arbitrarily small pieces, then combined to produce relatively efficient code. Furthermore, a comparatively simple notation, using the well-known technique of quotation and anti-quotation (just as is often used in macro systems) can produce these program generators. And, finally, the notion of compositionality allows for a great variety of data to be included in a component, data that can be used to generate more efficient target machine code, or to support some other service such as security or regenerability.

4 Standardization of data representations

To support code generators, we anticipate the standardization of certain data representations that have not been standardized as of yet. *Machine language* itself will, we predict, be substantially standardized in the future (whether on an actual machine architecture or an abstract machine we do not say). The need to componentize software will make portability among different target architectures appear to be a needless burden.

Thus, we would anticipate development of a common code exchange format (CCE), which in our current work we take to be Java virtual machine code. This will already imply a common representation for primitive types and a common function-invocation protocol. Beyond that, we will see standardized representations for lists, tuples, and objects. Such standardization will be needed not to make programs themselves portable, but to make program-generating *components* portable.

5 Components are higher-order values

In [1], we defined a "component of type τ" to be a value of type τ, where types are formed from this grammar:

$$\begin{aligned}\tau \in \text{Type} = \quad & \text{int} \mid \text{float} \mid \ldots \text{ other primitive types } \ldots \mid \text{CCE} \\ & \mid \tau_1 \times \tau_2 \mid \tau_1 \text{ list} \mid \tau_1 \rightarrow \tau_2\end{aligned}$$

A component of type τ has "clients of type τ," meaning values of type $\tau \rightarrow$ CCE. Thus, a client is a function that, when applied to a component, produces a piece of code[1]

We note that the use of higher-order values (values taking functions as argument and/or returning them as results), previously confined mainly to the functional programming world, is critical here. For example, consider a component that generates code for finite state machines. The client will need to pass a high-level definition of a finite state machine to the component, including actions to be taken at appropriate points. The type of the component would be FSMDef $\rightarrow$ CCE, and the type of a client of this component would be (FSMDef $\rightarrow$ CCE) $\rightarrow$ CCE.

6 Components are objects

For this discussion, we will modify the above definition slightly, in that types will be represented by classes, and values by objects. We encode the constructors above in classes whose details are not of interest, so long as they implement appropriate interfaces:

> C is a $\tau_1 \times \tau_2$-*class* if it implements the methods $\pi_1 :\rightarrow C_1$ and $\pi_2 :\rightarrow C_2$, where C_1 is a τ_1-class and C_2 is a τ_2-class.
>
> C is a τ-*list-class* if it implements methods $hd :\rightarrow C_1$ and $tl :\rightarrow C$, where C_1 is a τ-class.
>
> C is a *($\tau_1 \rightarrow \tau_2$)-class* if it implements the method $apply : C_1 \rightarrow C_2$ where C_1 is a τ_1-class and C_2 is a τ_2-class.

We have switched from functions to objects to help us make our next point. We take it that the primary responsibility of a component is, under the right circumstances, to enable the client to generate code. Thus, if a component is an object of a τ-class, then the client must be an object of a $(\tau \rightarrow CCE)$-class. We are speaking of objects only, which is to say, packages containing data and *executable* code; thus, we are operating entirely at the binary level.

To achieve true mobility, components will need to carry more data and supply more services. Here is where the use of objects helps. Examples are:

[1] Actually, we assume a type *Code* such that every client has a standard coercion function **genml**:*Code*$\rightarrow$CCE. Then, "CCE" would be replaced by *Code* in the discussion above. This distinction is not important at the level of abstraction of this presentation.

- **Produce documentation.** Build the set of types again, but using *DocString* in place of CCE. A component may be a τ-class (in the above sense) and also a $\tau[CCE \mapsto DocString]$-class[2]; that is, it implements both interfaces. Again, a client contains a method **gendoc**: $\tau[CCE \mapsto DocString] \to DocString$, so that when applied to a component, appropriate documentation is produced.

- **Produce proofs.** Proof-carrying code [] is a method of guaranteeing certain properties of a piece of code by providing a proof — which the client may verify — of that property. Our components can contain *parameterized* proofs; that is, they can be objects of $\tau[CCE \mapsto Proof]$-classes, and clients can contain **genproof**: $\tau[CCE \mapsto Proof] \to Proof$.

- **Reconstruct.** A component may provide a method of reconstructing its "provenance," that is, the original source of its sub-components. Suppose that by "source" we mean a script containing URL's. Again, generate types as above, using *URL-script* in place of CCE. The **genProvenance** method obtains the parameterized script of the component and produces an actual script, from which the component could be reconstructed (with up-to-date versions of its constituents).

Each of these services require a great deal of study, and undoubtedly other, even more difficult, services will be created.

References

[1] Sam Kamin, Miranda Callahan, and Lars Clausen. Lightweight and generative components I: Source-level components. In K. Czarnecki and U.W. Eisenecker, editors, *Generative and Component-Based Software Engineering (GCSE'99)*, volume 1799 of *LNCS*, pages 49–62, September 28–30 1999.

[2]The notation reads, "τ with occurrences of CCE replaced by *DocString*."