Towards a colimit-based semantics for visual programming

Jeremy Gibbons

University of Oxford

Abstract. Software architects such as Garlan and Katz promote the separation of *computation* from *coordination*. They encourage the study of *connectors* as first-class entities, and *superposition* of connectors onto components as a paradigm for component-oriented programming. We demonstrate that this is a good model for what *visual programming tools* like IBM's VisualAge actually do. Moreover, Fiadeiro and Maibaum's categorical semantics of parallel programs is applicable to this model, so we can make progress towards a formal semantics of visual programming.

1 Introduction

There are signs that the popularity of pure object-oriented programming is receding, to be replaced by component-oriented programming [16]. One motivation for this trend is the desire for gaphical tools to support visual assembly by third parties of independently-developed software components. The code generated by such visual programming tools has the kind of architecture promoted by researchers such as Garlan [1] and Katz [9], who argue for the separation of actual computational behaviour from the coordination of these computations.

Their approach encourages the study of connectors as first-class entities, and superposition of connectors onto components as a paradigm for application assembly. Moreover, Fiadeiro and Maibaum's colimit-based categorical semantics of parallel programs [3], a descendant of Goguen's General Systems Theory [5], is applicable to this paradigm, so we can put the two together to make progress towards a formal semantics of visual programming.

In this paper we build on the above-mentioned existing work on superposition and on colimit-based semantics of system assemblies. Our contributions are two-fold: to show that superposition is a good model of the action of visual programming tools such as IBM's VisualAge, and to make the first steps in applying the colimit approach to semantics to superposition of coordinators.

The remainder of the paper is structured as follows. In Section 2 we motivate the movement from object-oriented to component-oriented and visual programming, and describe how visual programming tools operate, with the help of a simple example. In Section 3 we outline the components-and-connectors approach to software architecture, and summarize Goguen's General System Theory, and Maibaum and Fiadeiro's development of it as a model of concurrent systems. Although we are not yet at a stage to apply this theory directly to arbitrary

software components, we provide in Section 4 a simplified illustration in terms of concurrent processes.

2 Components and visual programming

Object-oriented programming is losing its shine: despite its undoubted benefits, we are coming to the realization that it is not a silver bullet for the problems of software construction. In particular, *inheritance breaks encapsulation* [14]: in order reliably to define a subclass, one needs to see not just the public interface but also the private implementation of the superclass, and so a revision of the latter, even without changing its public interface, may break the former.

This observation has led some [16] to propose *component-oriented programming* as an improvement on object-oriented programming. The emphasis is on *object composition* rather than *class inheritance*, and *delegation* rather than *overriding*. This avoids the problem of broken encapsulation alluded to above, and paves the way for a programming methodology based on *third-party assembly* of *black box* components. Assemblers need only know the public interface of a component, not its private implementation.

2.1 Visual assembly of components

This in turn allows component assembly without looking at code at all. Assemblers can literally treat components as black boxes, placed on a canvas and connected by lines. Supporting software can interpret the lines as connections between operations in the component interfaces, and can automatically generate the coordinating code.

This is exactly what visual programming tools like IBM's VisualAge for Java [7] do. The details differ from tool to tool, but the idea is fairly consistent. There will be a *palette* of computational components, and a *canvas* on which these components can be dropped. Two components are connected by dragging a rubber band line from one and dropping it on the other; this connection corresponds to some code, which is generated automatically by the tool. The possible interpretations of a 'connection' will depend on the underlying language, but typically they all boil down to the invocation of some method on the target component, given some suitable triggering condition on the source component.

The Java language [6], and in particular the JavaBeans coding specification [10], was specifically designed to permit this kind of component assembly without stepping outside the language. Reflection can be used on compiled Java code to determine what methods are supported, but the Java class libraries were constructed in such a way as to avoid having to do this wherever possible. The code generated by the tool is source-level Java, which can be examined and if necessary modified afterwards.

2.2 A 'counter' example

To illustrate, consider a little Java applet providing a counter. It consists of three components: a button that can be pressed, a counter that is incremented

on button presses, and a label that displays the value of the counter. One may expect to find these three components in a software component library. A visual assembly tool can automatically generate two new classes: an *event adapter* class, whose instances have a method to be called on button presses that will step the counter, and a *property binder* class, whose instances have a method to be called on counter value changes that will update the label. In addition, the tool will generate one more class, with a main method than instantiates the other five classes and hooks up the instances appropriately. This is illustrated in Figure 1.

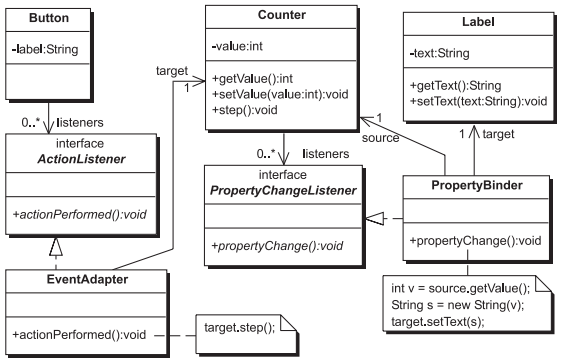

Fig. 1. The class structure of the counter applet

2.3 But what does it mean?

Visual programming tools like IBM's VisualAge, Borland's JBuilder, Microsoft's Visual C++ and so on usually provide only informal descriptions of the meaning of the gestures by which programs are assembled; it can be difficult to predict what the outcome of a particular series of gestures will be. Usually the outcome is in fact a source-level program, albeit not necessarily a pretty one; so one can in principle discover the meaning after the fact by examining this generated code — but that doesn't help much with predictability. Without precise, and preferably formal, descriptions of outcomes, no analysis of the visual assembly as a construction in its own right is possible, and programming is reduced to a trial-and-error process, or at best a 'black art'.

It would be much better to have a formal semantics of the visual assembly as a construction in its own right. As the construction is a diagrammatic entity, this strongly suggests a categorical semantics: category theory provides a denotational semantics for diagrams. (This is in contrast to formalisms such as graph grammars, which specify a syntax for diagrams, and graph rewriting, which provides an operational semantics.) It turns out that one can give a clean, simple semantics of such a visual construction as the *colimit* of a *diagram* of components and connectors; we show how in the remainder of the paper.

We envisage that components in the component library will come provided with their semantics, and that coordination is achieved through a small, fixed

collection of (customizable) connectors, each again with its own (parametrized) semantics. Program assembly is a matter of constructing a diagram of components and connectors, as described below. The semantics of the construction is the colimit of the diagram.

3 Connectors and superposition

As software systems become larger and more complex, the difficult problems and the focus of interest migrate from the small-scale algorithmic aspects of individual components to the large-scale organizational aspects of the system as a whole — the *software architecture* [13]. As a step towards formalizing software architectures, Allen and Garlan [1] propose the separation of *computation* (sequential, single-threaded algorithmic activities) from *coordination* (the glue that binds computations together), and the study of the *connectors* by which coordination is achieved as first-class entities in their own right.

In this proposal, Allen and Garlan were following in the footsteps of a similar journey made a few years earlier by Katz [9], who had proposed *superimposition* or *superposition* of connectors (for coordination) onto components (for computation) as a paradigm for distributed computing. The parallels between large-scale software architecture and distributed computing should come as no surprise: in both fields, the components from which a system is assembled are of a fundamentally different character than the system itself, and the developer is concerned with complex behaviour emerging from the interactions between (relatively) simple independent units.

According to this view, the 'counter' application developed in Section 2.2 consists of three *components* (the button, the counter itself, and the label) coordinated by superposing two *connectors* (the event adapter and the property binder). The components came 'off the shelf'; the connectors were generated automatically by the visual programming tool from gestures made by the assembler. In order to generate this code, the tool needs to know nothing about the components beyond their publicly-advertised interfaces. (Note that there is essentially no inheritance in the system; everything is achieved through object composition, as suggested by Szyperski [16].)

3.1 Categorical semantics of superposition

Fiadeiro and Maibaum [3] provide a semantics for Allen and Garlan's notion of connector, building on Goguen's categorical *General Systems Theory* [5]. Goguen based this on the following slogan:

> *given a category of widgets, the operation of putting a system of widgets together to form a super-widget corresponds to taking a colimit of the diagram of widgets that shows how to interconnect them.*

We believe that this approach can be taken to give a precise semantics for Java applications assembled from JavaBeans, and for similar visual program development methods. We cannot yet justify this belief, though; for one thing, we

would have to identify a category of JavaBeans, and this is still a subject of much research (see for example [8], and other work from the LOOP project). What we can do is illustrate the approach in a simpler setting, and trust that the reader will accept at least that the approach is worth exploring.

In the remainder of Section 3, we explain the General Systems Theory slogan in the context of superposition. In Section 4, we illustrate its application to the superposition of simple processes.

3.2 Superposition via colimits

In this section we present formal definitions of the categorical definitions leading up to colimits. We have tried to present these in as elementary a manner as possible, so in some cases the definitions are non-standard, although equivalent to the standard definitions. (For example, the usual definition of a diagram in a category $\mathcal{C}$ is as a functor from an indexing category to $\mathcal{C}$; the definition given here avoids having first to define functors.) Space considerations preclude us from going into too much detail, so we summarize briskly; for more detail, see a standard text on category theory, such as [12].

A *category* consists of collections of *objects* and *arrows*, each arrow between two objects. For each object, there is an *identity arrow* from that object to itself. Two arrows that meet (the target of the first is the source of the second) may be *composed*; composition is associative and identity arrows are units.

A *diagram* in a category consists of a subcollection of the objects and of the arrows, for which both endpoints of each included arrow are included. A *cocone* of a diagram with objects A_i and arrows f_k consists of an object X and arrows $g_i : A_i \to X$ coherent with the f_k — that is, $g_i = f_k \,;\, g_j$ for each $f_k : A_i \to A_j$. A *colimit* of a diagram is a cocone through which any other cocone uniquely factorizes; that is, a cocone (X, g) such that for any other cocone (Y, h), there is a unique arrow $\alpha : X \to Y$ such that $g_i ; \alpha = h_i$ for every object A_i. Intuitively, colimits capture least upper bounds.

We will be concerned with categories in which the objects are components, and the arrows embeddings of smaller components into larger ones. Two components are synchronized by embedding both in a common super-component. A diagram of components models a system composed from those components, and the arrows in the diagram indicate the synchronizing interconnections between the components. A colimit of a diagram is the least common extension of all the components in the diagram, or equivalently, the minimal system synchronizing all the components; Goguen's slogan of General Systems Theory dictates that this is the appropriate meaning of the individual compositions.

4 A simplified example of the categorical semantics

In order to complete our plan to provide a categorical semantics for visual programming (for instance, visual composition of JavaBeans), we first need to choose a category in which JavaBeans are objects. As observed above, the right such

choice is still an open question. We postpone that choice to further research, and resort here to a rather simpler setting: instead of JavaBeans, we will consider *traced processes*. (This illustration is based on [3].)

4.1 Traced processes

Consider processes of the form $\mathit{Toggle} = \mathit{step} \rightsquigarrow \mathit{step}, \mathit{rollover} \rightsquigarrow \mathit{Toggle}$. The intention is that at each time step, a process in a particular state engages in some actions and moves into a different state. At some time steps, the process may engage in no actions (and stay in the same state).

A process or component C may be modelled as a pair $\langle \Sigma_C, T_C \rangle$, where Σ_C is the alphabet (a set), and T_C is the set of traces (a set of streams of finite subsets of Σ_C). The events in each trace are subsets rather than elements of the alphabet in order to capture absent and simultaneous actions. Trace sets are *closed under stalling*: whenever $\langle s_0, s_1, \ldots \rangle$ is in T_C, so also is $\langle s_0, s_1, \ldots, s_i, \{\}, s_{i+1}, \ldots \rangle$, for each value of i.

For example, the component *Toggle* has alphabet $\Sigma_{\mathit{Toggle}} = \{\mathit{step}, \mathit{rollover}\}$, and a trace set T_{Toggle} that consists of all possible stallings of the basic trace $\langle \{\mathit{step}\}, \{\mathit{step}, \mathit{rollover}\}, \{\mathit{step}\}, \{\mathit{step}, \mathit{rollover}\}, \ldots \rangle$. (From now on, we'll abbreviate '*step*' to '*s*' and '*rollover*' to '*r*'.)

We define the category $\mathcal{TP}\mathit{roc}$ to have traced processes as objects, and *embeddings between processes* as arrows. An embedding is a witness as to how one process is *simulated by* another. Intuitively, a process C is embedded within, or simulated by, a process D if, by renaming some of the actions of D and ignoring the others, it is possible to make D look like C; the embedding is simply the renaming function.

Formally, the arrow $f : \langle \Sigma_C, T_C \rangle \to \langle \Sigma_D, T_D \rangle$ is a partial function f from Σ_D to Σ_C such that

> for every trace $t' \in T_D$, there exists a trace $t \in T_C$ such that, for every event $t'_n = \{d_1, \ldots, d_m\}$ of t', the corresponding event t_n of t is the set of actions $\{c \in \Sigma_C \mid \exists i \,.\, 1 \leq i \leq m \wedge d_i \in \mathrm{dom}\, f \wedge f\, d_i = c\}$, the image of t'_n under f.

For example, consider the simpler process $\mathit{Clock} = \mathit{tick} \rightsquigarrow \mathit{Clock}$, with alphabet $\Sigma_{\mathit{Clock}} = \{\mathit{tick}\}$, and trace set T_{Clock} all stallings of $\langle \{\mathit{tick}\}, \{\mathit{tick}\}, \ldots \rangle$. (From now on, we'll abbreviate '*tick*' to '*t*'.) The process *Clock* is simulated by the process *Toggle*, if (for example) one looks only at the r actions of the latter and thinks of them as t actions. So the category $\mathcal{TP}\mathit{roc}$ contains a morphism $f : \mathit{Clock} \to \mathit{Toggle}$, where the function f from Σ_{Toggle} to Σ_{Clock} is the one-point function $\{r \mapsto t\}$.

4.2 Assembling components

It is well-known that toggles can be chained to make asynchronous counters. The rollover event r of one toggle should be coordinated with the step event s of the next.

In the spirit of superposition, this *coordination* should be coordinated by a separate *connector*. In this case, the right connector to use is just *Clock*; it acts as a *channel* between the two toggles, or an *adapter* between their interfaces. This is captured by the diagram in Figure 2(a).

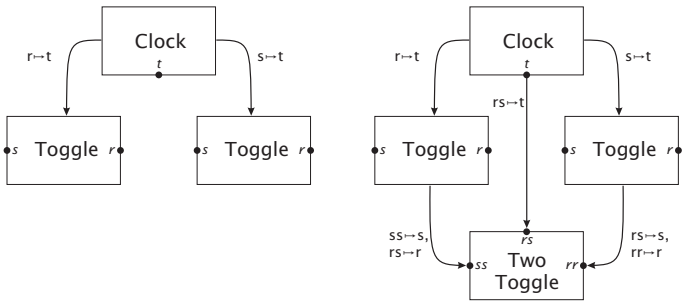

Fig. 2. (a) Interconnection of toggles; (b) Colimit of toggle system

What does this collection of boxes and lines mean? Just the colimit of the diagram, as illustrated in Figure 2(b). We claim that the colimit of the connection diagram in Figure 2(a) is the process *TwoToggle*, with alphabet $\Sigma_{TwoToggle} = \{ss, rs, rr\}$, and trace set $T_{TwoToggle}$ all stallings of the basic trace $\langle\{ss\}, \{ss, rs\}, \{ss\}, \{ss, rs, rr\}, \ldots\rangle$. This is indeed the least common extension of the three connected components, and it does indeed model the two-bit counter formed from two chained one-bit counters.

5 Conclusion

It is probably clear from the above presentation that these are very preliminary ideas — more of a research proposal than a research report. There are many questions still to be answered and directions in which to explore, including:

- What standard process algebra (as opposed to the home-grown one used here) is best suited to describing the kinds of connection involved in visual program assembly?
- Is the corresponding category actually *finitely cocomplete*, which is to say, does every diagram of interconnected components possess a colimit and hence a meaning? Perhaps some healthiness conditions need to be placed on the diagram in order to guarantee the existence of the colimit.
- What is a suitable category for visually-composed components, such as JavaBeans?
- Although this approach was originally envisioned as simply providing a semantics for visual program assembly, can it in fact provide more? For example, the colimit approach has been used in systems such as the *SpecWare* program synthesis tool [15], in which it captures the notion of 'completion of theories'; SpecWare can automatically construct colimits. Can this construction form the basis of a visual programming tool?

- Other applications of the approach can be envisaged. For example, one problem in the use of *design patterns* [4] is that of tracing the instantiation of the design pattern through the development of a piece of software [2]. Superposition can perhaps capture the sense in which a design pattern is 'a component of' a software system, in which case there is hope of formally recording this information throughout the development process.

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering Methodology*, 6(3), 1997.
2. S. Bünnig, P. Forbrig, R. Lämmel, and N. Seemann. A programming language for design patterns. In *Arbeitstagung Programmiersprachen*, Paderborn, October 1999.
3. J. L. Fiadeiro and T.S.E. Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming*, 28(2–3):111–138, 1997.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
5. J.A. Goguen. Categorical foundations of general systems theory. In F. Pichler and R. Trappl, editors, *Advances in Cybernetics and Systems Research*, pages 121–130. Transcripta, 1973.
6. J. Gosling, B. Joy, and G. Steele. *Java Language Specification*. Addison-Wesley, 1996.
7. IBM. VisualAge for Java. http://www.ibm.com/software/ad/vajava.
8. B. Jacobs. Objects and classes, coalgebraically. In B. Freitag, C. B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer, 1996.
9. S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, 1993.
10. Sun Microsystems. JavaBeans specification. http://java.sun.com/products/javabeans/docs/spec.html, 1997.
11. L. Mikhajlov and E. Sekerinski. A study of the Fragile Base Class problem. In *European Conference on Object-Oriented Programming*, pages 355–382, 1998.
12. B.C. Pierce. *Basic Category Theory for Computer Science*. MIT Press, 1991.
13. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
14. A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In M. Meyrowitz, editor, *Object-Oriented Programming, Systems, Languages and Applications*, pages 38–45, 1986. SIGPLAN Notices 21(11).
15. Y. V. Srinivas and R. Jüllig. SpecWare: Formal support for composing software. In Bernhard Möller, editor, *LNCS 947: Mathematics of Program Construction*. Springer-Verlag, 1995.
16. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.