

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.

Keywords: Object-oriented programming, fragile base class problem, component software, coordination, connectors, visual programming, colimits, pushouts.

Contribution and relevance: This paper concerns applications of coordination, theoretical models and foundations of coordination, tools for the development of coordinated applications, and industrial relevance of coordination. We build on existing work on superposition and on colimit-based semantics of system assemblies; our contributions are 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 this domain.

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 graphical 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 explain the move from object-oriented to component-oriented programming. In Section 3 we describe how visual programming tools operate, with a simple worked example. In Section 4 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 5 a simplified illustration in terms of concurrent processes.

2 Component software

Like many fields, programming methodology goes through fashions. From time to time new techniques and paradigms arise, promising to alleviate the problems of software development. Researchers and practitioners invest time, money and energy in each new trend, hoping to make progress in the difficult task of constructing high-quality software at a reasonable cost. However, after a while disillusionment sets in, as the realization dawns that each new approach either fails to solve the original problems, or introduces new problems of its own, or both.

For a relatively long period of time, *object-oriented programming* was the height of software development fashion. The encapsulation of code and data together as *classes of objects* increases modularity, encouraging separation of concerns and reducing coupling between units of software; moreover, the notion of *subclassing* to capture specialized classes of objects, *inheriting* some code from the superclass and *overriding* the rest, promotes flexibility and reuse.

As usual, however, once the novelty had worn off, it became apparent that the object-oriented approach was not the last word in programming paradigms. 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. This leads to the *fragile base class problem* [11]: because the subclass (or ‘derived class’) depends on the implementation as well as the interface of the superclass (or ‘base class’), if the latter is revised to update the implementation — even without changing the public interface — the former can break.

For example, consider the Java class `CharArray` defined in Figure 1, which maintains a string (as a zero-terminated array of characters, as it happens). The public interface of this class reveals only the signatures of the methods

```
public class CharArray {  
  
    protected char[] chars;  
  
    public CharArray() {  
        chars = new char[100];  
        chars[0] = '\0';  
    }  
  
    public String getString() {  
        int i; for (i = 0; chars[i] != '\0'; i++);  
        return new String(chars, 0, i);  
    }  
  
    public void insert (char c, int pos) {  
        int i; for (i = 0; chars[i] != '\0'; i++);  
        for (; i>=pos; i--) chars[i+1] = chars[i];  
        chars[pos] = c;  
    }  
  
    public void prefix (char c) {  
        insert (c, 0);  
    }  
}
```

Fig. 1. The class CharArray

`getString`, `insert` and `prefix` and of the constructor; the implementations should be hidden. Now consider the subclass defined in Figure 2, which adds an attribute `length`, initialized in the constructor, updated in the method `insert`, and accessed by the method `getLength`. If the writer of the subclass can see

```
public class CharArrayLength extends CharArray {

    protected int length;

    public CharArrayLength() {
        super();
        length = 0;
    }

    public int getLength() {
        return length;
    }

    public void insert (char c, int pos) {
        super.insert(c,pos);
        length++;
    }
}
```

Fig. 2. The subclass `CharArrayLength`

the implementation of the superclass, they can see that the method `prefix` is defined there in terms of `insert`, and deduce that it is sufficient to override only the latter method; but this deduction might be invalidated by a revision of the superclass. If they cannot see the implementation, they cannot rely on this aspect of the design, and must also override `prefix`. If one is to respect encapsulation, much potential code reuse is lost. Note also that it would be wrong to override `prefix` in the same way by copying its definition from the superclass and adding a statement to increment `length`, as shown in Figure 3: the overridden `insert` will still be called, and `length` will in the end be incremented twice — a fact that cannot be deduced from the public interface of the superclass alone.

```
public void prefix (char c) {
    super.prefix(c);
    length++;
}
```

Fig. 3. An incorrect overriding of `prefix`

These problems have 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*. Thus, an instance of `CharArrayLength` would ‘have a’, rather than ‘be a’, `CharArray`, and would forward messages to this separate object rather than relying on overriding to determine which class supplies the implementation of a particular method.

The motivation for this development, of course, is to avoid the problems described above that are introduced by inheritance and overriding. This paves the way for a programming methodology based on *third-party assembly of black boxes*: an instance of `CharArrayLength` can safely rely on an instance of `CharArray` to maintain the array of characters, without having to know how that is actually achieved. To assemble these two instances, one need only know their public interfaces. Component assembly can in principal be done *visually*, using drag-and-drop gestures to indicate how one component is to be connected to another, as we describe in the next section.

In this paper, we take for granted the above arguments in favour of component-oriented programming, and also the virtues of visual programming. Our aim is not to question whether these are indeed wise developments; they are both unquestionably in use in industry. Rather, we aim to explain what is really going on behind the scenes in visual programming tools, in order better to understand how to use them, and to learn how to use them better.

3 Visual assembly of components

Once one has a coding style that allows components to be combined based only on their public interfaces, independently of the private implementations, it becomes possible for third parties to assemble these components into applications without needing to look at their implementations. Indeed, component assemblers need not look at code at all: they can literally treat components as black boxes on a canvas, and connect components by drawing lines between the boxes. Supporting software can interpret the lines, perhaps asking the assembler for clarification when multiple interpretations are possible, and automatically generate the code that the assembler is avoiding having to write.

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.

3.1 A ‘counter’ example

To illustrate the kinds of assembly that are possible, consider the little Java applet in Figure 4. When one clicks on the button at the left, the number in the

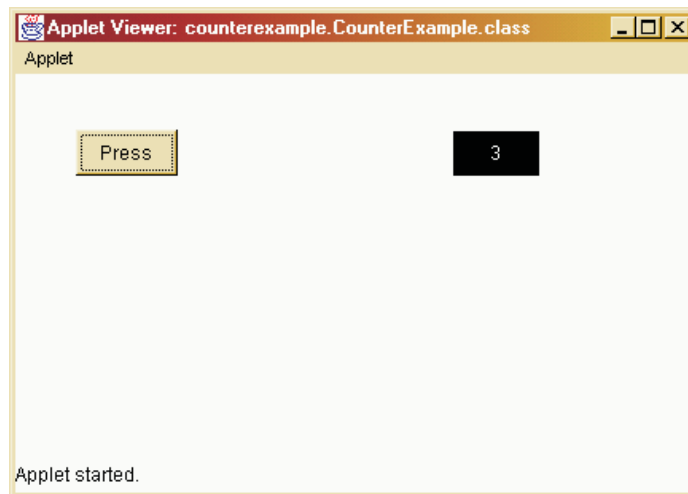


Fig. 4. A counter applet

label at the right increments. Given a suitable library of components (a button, a counter, a label), one might reasonably expect to construct this applet without actually writing any code. This is the promise of visual program assembly.

This applet was actually constructed with IBM's VisualAge for Java (but many other visual programming tools would be just as good). A snapshot of the visual programming editor is provided in Figure 5. The main window contains the *canvas*, on which *component instances*, chosen from the *palette* at the left, are placed. Here we see three component instances, the button and label (both visible, so inside the dotted box), and the counter (invisible, so outside). The behaviours of the component instances are coordinated by means of the two *connectors*, drawn as lines in the editor. The lines are in different styles (although this might be difficult to see in the diagram), because the connectors are of different kinds: the connector between the button and the counter is an *event adapter*, adapting the counter's interface to the format required to be a listener

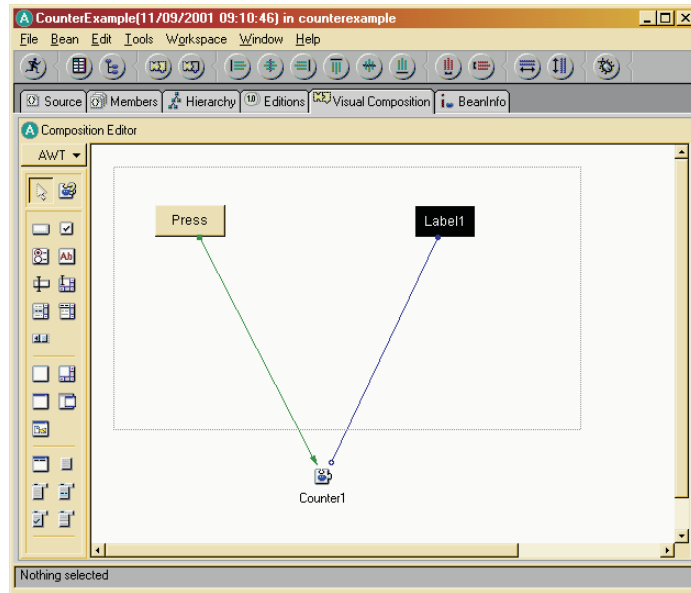


Fig. 5. A visual programming editor

for the button's events, and the connector between the counter and the label is a *property binder*, maintaining the invariant that the *text* attribute of the latter equals the *value* attribute of the former. The architecture is illustrated informally in Figure 6.

From this construction, the editor automatically generates a program with the class structure illustrated in Figure 7. The *Button*, *Counter* and *Label* classes already exist, as do the *ActionListener* and *PropertyChangeListener* interfaces with which the first two interact. The tool automatically generates two new classes: *EventAdapter*, which implements the *actionPerformed()* method of the *ActionListener* interface to call the *step()* method of a *target*; and *PropertyBinder*, which implements the *propertyChanged()* method of the *PropertyChangeListener* interface to call the *setText(String)* method of a *target*. (In fact, VisualAge generates a single class, an inner class of the applet, which implements both interfaces at once; but the effect is the same.) The tool also generates a class for the applet as a whole, which creates one instance each of the three components and the two connectors and hooks them up appropriately.

3.2 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

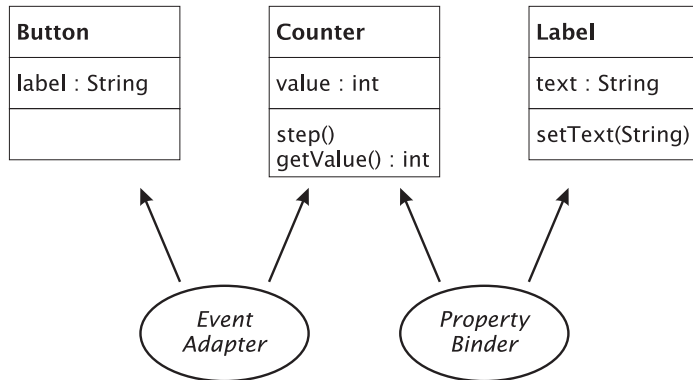


Fig. 6. The architecture of the counter applet

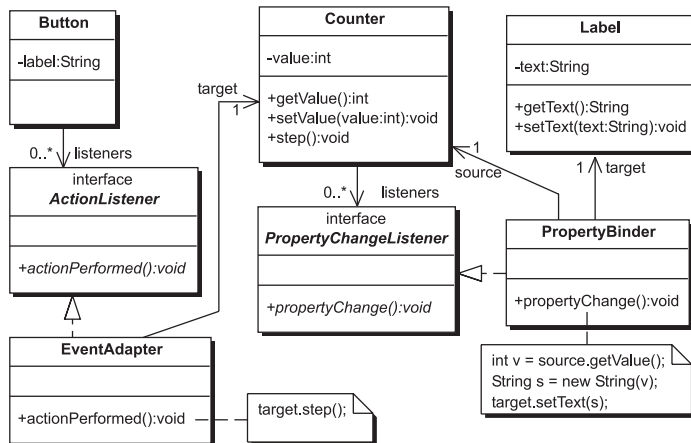


Fig. 7. The class structure of the counter applet

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.

4 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 3.1 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].)

4.1 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 4, we explain the General Systems Theory slogan in the context of superposition. In Section 5, we illustrate its application to the superposition of simple processes.

4.2 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.)

Definition 1 (Category). *A category \mathcal{C} consists of a collection $Obj(\mathcal{C})$ of objects and a collection $Arr(\mathcal{C})$ of arrows, together with mappings src, tgt (‘source’ and ‘target’) from $Arr(\mathcal{C})$ to $Obj(\mathcal{C})$ and id (‘identity’) from $Obj(\mathcal{C})$ to $Arr(\mathcal{C})$, such that:*

- $id(A) : A \rightarrow A$ for every $A \in Obj(\mathcal{C})$;
- whenever $f : A \rightarrow B$ and $g : B \rightarrow C$, there is a composite $f ; g : A \rightarrow C$;
- composition is associative;
- $id(A) ; f = f = f ; id(B)$ when $f : A \rightarrow B$;

where we write ‘ $f : A \rightarrow B$ ’ as a shorthand for ‘ $src(f) = A$ and $tgt(f) = B$ ’. (Note that there is no direct relationship between the use of the term ‘object’ here and its use as in ‘object-oriented programming’.)

Remark 2 (Components). *The categories with which we are concerned will have system components as objects. The example in Section 5 uses a simple kind of process for the objects; we intend to construct a category in which JavaBeans and connectors are the objects. In each case, when the actions of one component can be synchronized with those of another, there is an arrow between the corresponding objects witnessing the synchronization relationship.*

Definition 3 (Diagram). *A diagram in a category is a subcategory, that is, subcollections of the objects and the arrows, such that for every arrow included, its source and target objects are included.*

Remark 4 (Systems composed from components). *In our case, a diagram of components models a system composed from those components. The arrows in the diagram indicate the synchronizing interconnections between the components.*

Definition 5 (Cocone). *A cocone (X, g) of a diagram with objects A_i and arrows f_k is an object X and a family of arrows $g_i : A_i \rightarrow X$ coherent with the f_k — that is, $g_i = f_k ; g_j$ for each $f_k : A_i \rightarrow A_j$. See Figure 8.*

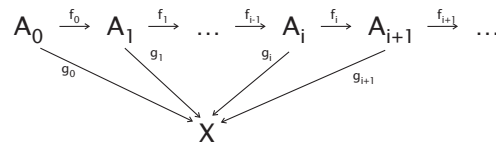


Fig. 8. A cocone

Definition 6 (Colimit). *A colimit of a diagram is an initial cocone, one through which any other cocone uniquely factorizes. That is, a colimit is a cocone (X, g) such that for any other cocone (Y, h) , there is a unique arrow $\alpha : X \rightarrow Y$ such that $g_i ; \alpha = h_i$ for every object A_i . See Figure 9.*

Remark 7 (Pushout). *One sometimes hears of the ‘colimit approach’ to semantics, and sometimes of the ‘pushout approach’. A pushout is simply a colimit of a span diagram — a diagram of two arrows with a common source. See Figure 10.*

Remark 8 (Colimits of compositions of components). *Intuitively, colimits capture least upper bounds. In the case of a diagram representing a system composed from components, with arrows witnessing synchronizations between pairs of components, the colimit is a super-component with which all the components synchronize, or equivalently, the smallest common extension of all the components. Goguen’s slogan of General Systems Theory dictates that this is the appropriate meaning of the individual compositions.*

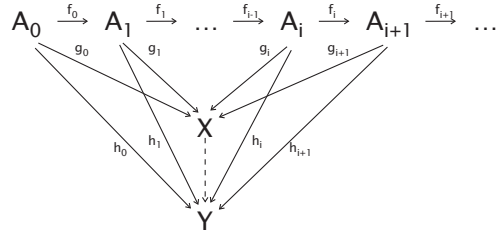


Fig. 9. A colimit

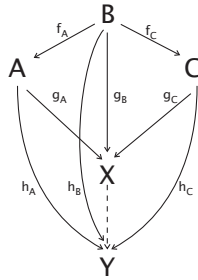


Fig. 10. A pushout

5 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].)

5.1 Traced processes

Consider processes of the form

$$Toggle = step \rightsquigarrow step, rollover \rightsquigarrow Toggle$$

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).

5.2 Modelling processes

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 $\langle s_0, s_1, \dots \rangle$ is in T_C , so also is $\langle s_0, s_1, \dots, s_i, \{\}, s_{i+1}, \dots \rangle$, for each value of i .

For example, the component *Toggle* has alphabet

$$\Sigma_{Toggle} = \{step, rollover\}$$

and a trace set T_{Toggle} that consists of all possible stallings of the basic trace

$$\langle \{step\}, \{step, rollover\}, \{step\}, \{step, rollover\}, \dots \rangle$$

(From now on, we'll abbreviate 'step' to 's' and 'rollover' to 'r'.)

5.3 Arrows between processes

We define the category \mathcal{TProc} 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 $f : \langle \Sigma_C, T_C \rangle \rightarrow \langle \Sigma_D, T_D \rangle$ 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 $t'_n = \{d_1, \dots, d_m\}$ of t' , the corresponding event t_n of t is the set of actions $\{c \in \Sigma_C \mid \exists i. 1 \leq i \leq m \wedge d_i \in \text{dom } f \wedge f d_i = c\}$, the image of t'_n under f .

For example, consider the simpler process

$$Clock = tick \rightsquigarrow Clock$$

with alphabet $\Sigma_{Clock} = \{tick\}$, and trace set T_{Clock} all stallings of $\langle \{tick\}, \{tick\}, \dots \rangle$.

(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{TProc} contains a morphism $f : Clock \rightarrow Toggle$, where the function f from Σ_{Toggle} to Σ_{Clock} is the one-point function $\{r \mapsto t\}$.

5.4 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 11.

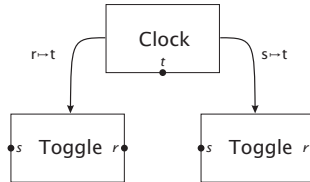


Fig. 11. Interconnection of toggles

What does this collection of boxes and lines mean? Just the colimit of the diagram, as illustrated in Figure 12. We claim that the colimit of the connection

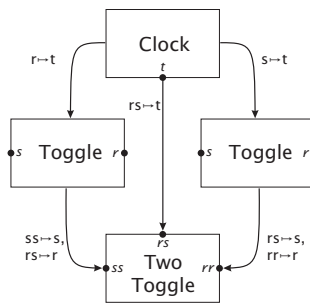


Fig. 12. Colimit of toggle system

diagram in Figure 11 is the process *TwoToggle*, with alphabet

$$\Sigma_{TwoToggle} = \{ss, rs, rr\}$$

and trace set $T_{TwoToggle}$ all stallings of the basic trace

$$\langle \{ss\}, \{ss, rs\}, \{ss\}, \{ss, rs, rr\}, \dots \rangle$$

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.

6 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 Java-Beans?
- 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-Oriented Programming 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.