A Formal Model for Component-Based Software

Philip T Cox* Baoming Song†

*Dalhousie University, Halifax, Canada
† Telecommunications and Informatics Services, Government of Canada

Abstract

In an effort to manage increasing complexity and to maximise the reuse of code, the software engineering community has in recent years put considerable effort into the design and development of component-based software development systems and methodologies. The concept of building software from existing components arose by analogy with the way that hardware is now designed and built, using cheap, reliable standard "off-the-shelf" modules. Because of the analogy with wiring hardware components, component-based software development is a natural candidate for visual expression. Various component software technologies have emerged as a result of this attention, but their evolution has been rather ad hoc. In fact, some systems are defined purely by their implementation with little or no precise definition. In an attempt to address this shortcoming, we propose a well-defined syntax and semantics for a component software model that captures the essential concepts.

1 Introduction

Software reuse has long been one of the major issues in the world of software engineering, where code reuse is seen as the key to many benefits, such as increased productivity, improved reliability, and ease of maintenance. As a result, many software reuse technologies have been developed over the past few years, see for example [8,9].

Software reuse was first realised in the late 50's with the development of libraries of pre-compiled subroutines such as large numerical libraries for engineering and scientific computation. Reuse via subroutines has limited applicability, however, because of the difficulty of encapsulating high-level functionality in subroutines.

A major step forward was made with the advent of object-oriented programming (OOP), which provided inheritance as its primary reuse mechanism. As OOP research and practice has progressed, other reuse techniques have been devised, such as object composition. More significantly, object-oriented application frameworks have emerged, providing the means to capture very large application design patterns in the form of "inverted libraries" which call the code supplied by the application developer, rather than the other way round as with traditional libraries [10].

Another popular reuse technique involves design patterns, which provide guidance during the design phase of software by suggesting high-level organisations for the necessary abstractions [6]. Design patterns provide methodology but not tools.

A recent trend in software engineering is towards software development using components. This development methodology is based on the observation that hardware design and development has evolved well beyond the "build-from-scratch" era. Hardware these days is built from "off-the-shelf" components which are customisable to a degree, cheap because they are manufactured in large quantities, are well tested and reliable, and have a well defined interface. The aim of component-based software development is to attain reuse, economy, reliability and so forth by creating large catalogues of software components for assembling into software systems [19]. This approach has some other desirable consequences as well, such as standardisation, resulting in software products from different developers working correctly together.

The analogy with hardware components on which the notion of software components rests, leads naturally to a visualisation of component-based software as a network of boxes communicating with each other via connecting wires. In recent years several software development tools have appeared that provide visual programming based on components. Visual Age for Java [3], Parts for Java and Java Studio use Java Beans as the underlying component mechanism. In Visual Age and Parts, the visual programming by wiring components together is largely limited to programming the user interface. Components that implement user interface items are represented in the visual program by their usual interface appearance, and wires connect them to indicate the flow of messages between these visible components and others which have no visual representation. A more direct representation of the box-and-wire metaphor is provided by Java Studio (no longer available), in which icons representing the components are wired together.

Another visual manifestation of components can be seen in Microsoft Office products where "objects" of differing types (word processor document, spreadsheet, picture) can be embedded in each other. The objects are components, and are linked together and pass messages to each other according to a standard protocol, Object Linking and Embedding (OLE).

Apple Computer spent several years designing and building a similar but more flexible system called OpenDoc, in which documents consisting of communicating components could be built by dragging components into a workspace [2].

Like many other good ideas, the concept of components has developed in a rather *ad hoc* way. Even though the various component technologies have very similar capabilities, they are all different in detail, and few of them are defined in clear, concise terms the way programming languages are (or can be).

In the following, we attempt to address this shortcoming by providing a formal definition of the syntax and semantics of components as exemplified by the above systems. This definition

- captures the essence of component-based software at a high level;

- provides a simple but precise characterization of components which may help dispel confusion resulting from the rapid evolution of many subtly different component technologies;
- describes an underlying structure on which the syntax of component-based languages can be based, as well as a semantics for such languages;
- provide a basis for a formal testing and verification methodology;
- allows recursive components which none of the existing component technologies provide;

The definition has been implemented in a prototype development environment in which simple component-based programs can be built. The pictures used to illustrate the definitions were generated by this prototype.

Before presenting our definitions in Section 3, we give a brief overview of component models.

2 Component models

Just as there are a variety of similar but not identical component technologies, so are there many similar but not identical definitions of *component*. Two examples are:

"A component is a coherent package of software that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected, unchanged, with other components to compose a larger system." [5]

and:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [19]

Although these definitions differ in detail, they both assert that a component is an independent software package that provides functionality via well-defined interfaces. They also emphasise the "black box" nature of a component: that is, a component can be incorporated in a software system without regard to how it is implemented.

Given these approximately equivalent definitions, there are two questions to be answered: how is a component developed and how is the component applied in software development? A component model should address both questions.

The three main component models currently in commercial use are CORBA, COM, and JavaBeans.

2.1 CORBA

The Common Object Request Broker Architecture (CORBA), was proposed by the Object Management Group (OMG) as a standard for communication between components. It forms part of a larger model, Object Management Architecture (OMA) that defines at a high level of abstraction a context within which components operate and interact, including standard services that components can rely on.

A CORBA component provides functionality via an interface defined in the CORBA Interface Definition Language, and interacts with other components only via that interface.

Other CORBA-related component models are IBM's System Object Model (SOM) which adds some extensions, and OpenDoc based on COM.

2.2 COM

The Component Object Model (COM) due to Microsoft is implemented only on Microsoft platforms [12].

COM specifies a standard for communication between COM components, consisting of COM interfaces and a system for registering and passing messages between components via these interfaces. COM interfaces are defined in Microsoft Interface Description Language (MIDL).

COM defines incoming interfaces as those interfaces that receive calls from other components and outgoing interfaces as those interfaces through which other components are called. COM uses outgoing interfaces to define events.

Other component models have been developed using COM. OLE deals with compound documents as mentioned in the previous section. Active X, like Java Applets, supports Internet and distributed computing. ActiveX components communicate with each other via events.

2.3 JavaBeans

According to the JavaBeans specification [18], a Java Bean is *"a reusable software component that can be manipulated visually in a builder tool."* A Java Bean component may have a visual representation, for example, a user-interface component, or may be non-visual, for example, a database connectivity component .

Java Bean components communicate with each other by sending events. The interface between components is therefore provided by the event model in the Java language. Components have properties which can be set by the programmer to achieve some customisation.

Enterprise JavaBeans (EJB) is an extension of the Java Bean model that provides a mechanism for encapsulating a Java Bean component as a CORBA component.

2.4 Common characteristics

The table below summarises the similarities and differences between the above three component technologies.

Ignoring differences in technical details such as implementation language, component implementation restrictions, parameterisation mechanisms, interface definition methodologies and self-documentation capabilities, we see that these technologies have essentially similar architectures and functionality. Each defines a component as a self-contained "black box" sending messages to and receiving messages from other components via a well defined interface, and performing its computation in response to the receipt of a triggering message (event).

Although these component technologies all have some relationship with object-oriented programming, object-orientation is orthogonal to the key concepts, as the table below shows.

2.5 Formalisations

Some attempts have been made to formally specify component models. A formal specification of CORBA using the Z specification language is used by the OMG member companies to ensure that facilities added to CORBA are correct.

There appears to be no the formalisations of the Java Bean component model, although there is a formal specification of the subset of the Java language on which JavaBeans is based [4].

There have been two attempts to formalise the COM component model. One of them, a language called COMEL (Component Object Model Exemplary Language), provides a formal syntax and semantics embodying COM's informal and complex rules [7]. The other, developed in an industrial

	JavaBeans	COM	CORBA
Component	Module containing multiple classes	Module containing multiple classes or other implementation	Module containing any implementation
Interface	Java language	OLE IDL, which defines interfaces as collection of functions	OMG IDL
Connection	Via event and listener.	Via interface pointers	Via Interface Definition Language
Variability mechanism	Inheritance and aggregation	Genericity, containment and aggregation	Inheritance and aggregation
Platform	Multiple platforms	Windows	Multiple platforms
Implementation Language	Java	Any languages, but primarily use C++ and Visual Basic	Any languages
Distribution Mechanism	EJB, Internet, RMI (remote method invocation)	DCOM, Internet	An ORB
Self-description	Support via introspection	No	No

setting to ensure the design integrity of a commercial system employing COM components, consists of a model based on first-order set theory, expressed in Z [17].

Each of the above formalisations describes a particular component model, and is intended for a particular purpose. None of them serves as a general definition of the key concepts summarised in section 2.4.

The component technologies described above arose from focussed development aimed at solving specific practical problems in industrial software development. For example, COM and OLE arose from a need to make common end-user applications operate seamlessly together.

Broader software engineering research by a somewhat different route has arrived at a similar point. Module Interconnection Languages (MILs) have long been studied as a means to specify the architecture of software systems [13], and have more recently evolved into Architecture Description Languages (ADLs) for formalising the syntax and semantics of high-level software design patterns [15].

Two examples of ADLs, are Darwin [11] and Wright [1], both of which provide general mechanisms for specifying software architectures in terms of connected "components", where a component in these languages is a module that uses the services of, and provides services to, other components. As general ADLs, neither provide a specific characterisation of the commercial component technologies that we address here, although they may well admit such characterisations, as would general software modelling languages such as Z [16].

3 A component model

3.1 Notational conventions

When an entity X is defined as a tuple, we can refer to the constituents of X in various ways. For example, consider definition 3.2.5 below. If X is a simple component, we can refer to the first constituent of X as **inports**(X), or by the phrase "the inports of X". If X is understood in context, we can omit the X, and simply write **inports** or refer to "the inports". If an item is a set, we will usually give it a plural name so that we can refer to its members in the singular. For example, if X is a simple component, we can use phrases such as "an inport of X".

If f is a function with domain S and range T, and S' is a subset of S, then f(S') denotes the subset { f(s) | s ∈ S' } of T. If S' is an *n*-tuple or sequence of elements of S for some *n*, then f(S') denotes the *n*-tuple or sequence of elements of T obtained by applying f to each element of S'. When it is convenient to do so, we will treat a tuple or sequence as if it were a set, in which case we mean the set consisting of all elements occurring in the tuple or sequence.

3.2 Syntax

3.2.1 Definition - domain

The *domain* D is a set which does not include the element *none*. A is an arbitrary but fixed infinite set called the set of *attributes*, each element x of which is associated with a subset of D denoted $\tau(x)$ called the *type* of x.

Three important concepts are introduced in this definition: *domain*, *attribute*, and *type*. The domain D might include values from any data type such as integers, strings, or instances of classes. Attributes can be thought of as variables or data fields. τ is a function that defines the set of all elements of D that can be values of an attribute. For example, suppose D includes the set **Z** of all integers, and for some attribute x of A, $\tau(x) = \mathbf{Z}$, then τ defines the type of x as integer.

3.2.2 Definition - component

A *component over* A is either a source over A, a sink over A, a simple component over A, a compound component over A, or a prototype over A.

3.2.3 Definition - source

A *source* over A is an attribute. If X is a source, **attributes**(X) and **outports**(X) are both defined to be the set consisting of the single attribute X.

Figure 1: Visual Representation of a Source Component

In general, a component produces data at its outports in response to the arrival of data at its inports. A source component has no inports, so from the point of view of the application in which it is embedded, a source component spontaneously generates data. For example, a

button component generates an output value when the button is clicked rather than in response to receiving an input.

Pictorially, a source can be represented as a box with the outport represented by an arrow on the perimeter pointing out from the box as shown in Figure 1.

3.2.4 Definition - sink

A *sink* over A is an attribute. If X is a sink, **attributes**(X) and **inports**(X) are both defined to be the set consisting of the single attribute X.

A sink is a component with no outports as shown in Figure 2. It receives its input data via its inports. Just as sources are active components, sinks are passive ones. The visual representation of a sink is similar to that of a source except that the single attribute is represented by an arrow pointing into the box. An example of a sink is a text field which displays the data it receives on its inport.

Figure 2: Visual Representation of a Sink Component

3.2.5 Definition - simple component

A *simple component over* A is a 4-tuple X of the form (**inports**, **outports**, **function**, **triggers**) where:

- **inports** is an n-tuple $(a_1, \ldots, a_n)$ of attributes for some integer $n \geq 0$;
- **outports** is an attribute distinct from $a_1, \ldots, a_n$;
- **function** is an n-ary function from $\tau(a_1) \times \ldots \times \tau(a_n)$ into τ(**outports**);
- **triggers** is a set of pairs of the form (**target**, **relation**), where **target** is an attribute distinct from the outport, and **relation** is a binary relation on τ(**target**);
- **attributes**(X) is defined to be the set of attributes consisting of the inports, the outport and the targets of X.

The role of simple components in a component-based system is analogous to that of primitive functions such as arithmetic operations or library routines in a programming language: that is, they provide services implemented using a different formalism or language. Ports (**inports** and **outports**) provide the interface through which the component interacts with other components. The functionality of a component is implemented through **function**. Triggers provide the mechanism for initiating execution of the component, the importance of which will be seen in the following definitions.

Figure 3 shows the visual representation of a simple component, represented by a box with inports and triggers located on the left and outports on the right. Since a trigger may or may not be an inport, there are three possible combinations on the left of box. An inport is represented by an arrow pointing into the box; a trigger is represented by a small gun-like icon; and a combined inport and trigger is represented by an icon with an arrow on one end and a gun-like icon on the other.

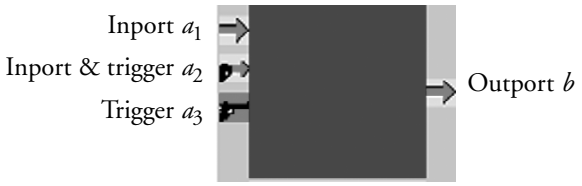


Figure 3: Visual Representation of a Simple Component

An example of a simple component is one that sums two integer values, and could be the one depicted in Figure 3. In this simple example, **inports** is a pair of attributes (a_1, a_2) and **outports** is an attribute b. **function** adds two integer values together and assigns the result to the outport b. Two triggers exist for this simple component. One trigger is a_2 which has the relation $\{(m, n) \mid m \neq n\}$ meaning that the function will be executed if the new value if a_2 is not equal to its old value. Another trigger is a_3 with relation $\{(m, n) \mid m = 1\}$, meaning that the function will be evaluated if the new value of a_3 is equal to 1.

The function of the simple component will be evaluated if and only if the relation for one of its triggers is true. In the above example, the sum of a_1 and a_2 will be calculated only if the relation for trigger a_2 is true, which will be the case if a new value arrives at this inport, or the relation for the trigger a_3 is true, which occurs if the value 1 arrives at a_3. This may happen if, for example, a button linked to this trigger is clicked. Precise semantics will be given in a later section.

3.2.6 Definition - compound component

A *compound component over* A is a 4-tuple X of the form (**components**, **inports**, **outports**, **connections**) such that:

- **inports** is a sequence of distinct attributes.
- **outports** is a sequence of distinct attributes.
- **components** is a set of components, not including X.
- **attributes**(X) is defined as the set **inports**(X) $\cup$ **outports**(X) $\cup$ { x | $\exists$Y $\in$ components(X) such that x $\in$ **attributes**(Y) }.
- The sets **inports**(X), **outports**(X), **attributes**(Y) and **attributes**(Z) are pairwise disjoint for any Y, Z $\in$ **components**(X).
- **connections** is a set of pairs of the form (**origin**, **destinations**) such that if K is a connection, then
  - **origin**(K) is either an inport of X or an outport of a component of X;
  - **destinations**(K) is a set of attributes of X not containing **origin**(K);
  - For each destination d of K, τ(**origin**(K)) $\subseteq \tau$(d).

A compound component is a network of connected components. A connection associates an outport of one component, the origin of the connection, with the inports of one or more other components, the destinations of the connections. A connection indicates the passage of data from origin to destinations. Each destination must be able to accept any value it receives from the origin, so its type must be a subset of the type of the origin.

Figure 4 shows the visual representation of a compound component as it appears within another compound component. It is similar to a simple component except for its colour and the fact that it can have several outports and cannot have triggers.

Figure 5 depicts the internal structure of the compound component in Figure 4. In this example, the compound component consists of four inports, two outports, and two simple components f1 and f2 each of which has two inports and a trigger. f1 is used to calculate the sum of two integers and f2 is used to calculate the difference between two inte-

gers. This compound component can provide the sum of or difference between the two integers arriving on inports a_2 and a_3, depending on which of the trigger relations of the two components holds.

Compound components provide an abstraction mechanism for dealing with the complexity of a component-based program. Note that the concept "compound component" realises the notion of "program": that is, we can build a component-based software system by constructing a compound component.

Figure 4: A Compound Component

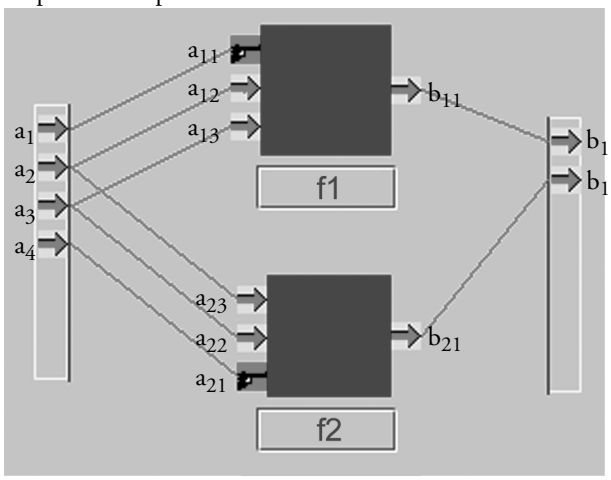


Figure 5: Internal Structure of a Compound Component

3.2.7 Definition - equivalence

Two components X and Y are *equivalent* iff there exists a bijection Φ: **attributes**(X) → **attributes**(Y) such that

- for all x ∈ **attributes**(X), $\tau(x) = \tau(\Phi(x))$.
- **inports**(Y) = Φ(**inports**(X)).
- **outports**(Y) = Φ(**outports**(X)).
- if X and Y are compound components, then
  - **connections**(Y) = Φ(**connections**(X)).

  and there is a bijection Ψ: **components**(X) → **components**(Y) such that for each component Z of X
  - Ψ(Z) is equivalent to Z;
  - Φ(**attributes**(Z)) = **attributes**(Ψ(Z)).
- if X and Y are simple components, then
  - **function**(Y) = **function**(X);
  - **triggers**(Y) = { (Φ(**t**), **C**[x_1, x_2]) | (**t**, **C**[x_1, x_2]) ∈ **triggers**(X) }.
- if X and Y are prototypes, then **class**(X) = **class**(Y).

According to this definition, components are equivalent if they are syntactically identical. It is easy to show that the relation "equivalence" is, in fact, an equivalence relation on components, which leads to the next definition.

3.2.8 Definition - class

If U is the set of all components, then U is partitioned into equivalence classes by the equivalence relation defined above. Each such class is called a *component class*.

Since the semantics of components, provided below, ascribes identical behavior to equivalent components, the partitioning provided by equivalence provides a basis for procedural abstraction, as follows.

3.2.9 Definition - prototype

A *prototype over* A is a triple X of the form (**class**, **inports**, **outports**) where

- **class** is a component class the elements of which are compound components.
- **inports** and **outports** are mutually exclusive sequences of distinct attributes of the same lengths respectively as **inports**(Y) and **outports**(Y) for any Y ∈ class.
- **attributes**(X) is defined to be the set **inports**(X) ∪ **outports**(X).

Note that since prototypes can occur in compound components, the procedural abstraction mechanism provided by prototypes naturally includes recursion, which is not normally a feature of component software models.

Figure 6 shows the representation of a prototype of the class of which the compound component in Figure 5 is a representative. Its colour (green) distinguishes it from simple components (blue) and compound components (yellow).

3.3 Semantics

3.3.1 Definition - occurrence

A component Y is said to *occur in* a component X iff either X = Y or X is a compound component and Y occurs in a component of X.

A connection K is said to *occur in* a component X iff either X is a compound component and K is a connection of X, or K occurs in a component that occurs in X.

Figure 6: A Prototype

3.3.2 Definition - state

A *state* of the set A of attributes is a function σ: A → D such that $\sigma(x) \in \tau(x) \cup \{none\}$ for each x ∈ A.

A state is an assignment of values to all attributes. As we shall see, an execution of a component assumes a certain starting state, which is transformed as the execution proceeds.

3.3.3 Definition - execution

If X is a component and σ is a state, an *execution of* X *from* σ is a sequence of the form (σ_0, X_0, S_0), (σ_1, X_1, S_1), (σ_2, X_2, S_2), … where

- for each i ≥ 0, σ_i is a state, X_i is a component, and S_i is a subset of the simple components occurring in X_i.
- $\sigma_0 = \sigma$; $X_0 = X$; $S_0 = \varnothing$.
- for each i ≥ 1
  - (σ_i, X_i, S_i) is a propagation of (σ_{i-1}, X_{i-1}, S_{i-1}) if one exists;
  - otherwise, (σ_i, X_i, S_i) is an expansion of (σ_{i-1}, X_{i-1}, S_{i-1}) if one exists;

- otherwise, (σ_i, X_i, S_i) is an evaluation of $(\sigma_{i-1}, X_{i-1}, S_{i-1})$ if one exists.

This definition divides execution of a component into three phases: propagation, expansion and evaluation defined below. Note that if none of the three rules apply, the execution sequence terminates.

The propagation rule, as its name implies, moves values that have been generated by a component along connections from the component's outports to other components. During this process, some of these values may arrive at triggers of some simple components, which may become ready to execute as a result.

The third item in each of the elements of an execution is the set of all simple components which are ready to execute. During evaluation, the function of each simple component in this set is evaluated and the result is passed to the outport of the simple component.

The expansion rule operates on prototypes. During expansion, a prototype will be replaced by a compound component that is a member of the class of the prototype.

3.3.4 Definition - propagation

If σ and σ' are states, X is a component, and S and S' are subsets of the simple components occurring in X, then (σ', X, S') is a *propagation* of (σ, X, S) iff for all $x \in A$

- if x is a destination of a connection K occurring in X such that $\sigma(\textbf{origin}(K)) \neq$ *none*, then $\sigma'(x) = \sigma(\textbf{origin}(N))$, where N is a connection of which x is a destination and $\sigma(\textbf{origin}(N)) \neq$ *none*.
- if x is the origin of a connection occurring in X such that $\sigma(x) \neq$ *none* then $\sigma'(x) =$ *none*.
- otherwise $\sigma'(x) = \sigma(x)$.
- $S' = S \cup \{ Y \mid Y$ is a simple component occurring in X
  $\wedge$ Y has a trigger τ
  $\wedge$ τ is a destination of a connection K occurring in X
  $\wedge$ $\sigma(\textbf{origin}(K)) \neq$ *none*
  $\wedge$ $(\sigma'(\textbf{target}(\tau)), \sigma(\textbf{target}(\tau))) \in \textbf{relation}(\tau) \}$.

3.3.5 Example

Consider the compound component X shown in Figure 7, consisting of two source components **Sr1** and **Sr2**, a simple component **Sc**, and a sink component **Si**, connected as shown. All attributes are of integer type. The function f of **Sc** is defined by $f(x) = x^2$ for any integer x, and the relation of the trigger of **Sc** is $\{ (x,y) \mid x \neq y \}$. Assume the current triple characterizing the execution is (σ, X, S) where $\sigma(a_1) = 2$, and $\sigma(a_2) = \sigma(b) = \sigma(c) = \sigma(d) =$ *none*. Propagation produces the triple (σ', X, S), where $\sigma'(b) = 2$ since a_1 has a value other than *none* and the value of a_2 is *none*, and $\sigma'(a_1) = \sigma'(a_2) = \sigma'(c) = \sigma'(d) =$ *none*. Since $\sigma'(b) \neq \sigma(b)$ the relation of the trigger of **Sc** holds so **Sc** is ready to execute; therefore S = { **Sc** }. From this example, we can see that propagation assigns new values to attributes that occur in a component, as well as randomly choosing one of the incoming values from among those which are not *none*.

3.3.6 Definition - expansion

If σ and σ' are states, X and X' are components, and S is a subset of the simple components occurring in X, then (σ', X', S) is an *expansion* of (σ, X, S) iff

either X is a compound component, $Y \in \textbf{components}(X)$ and X' is a compound component identical to X in every respect except that $\textbf{components}(X') = \textbf{components}(X) - \{Y\} \cup Y'$ where $(\sigma', Y', \varnothing)$ is an expansion of $(\sigma, Y, \varnothing)$.

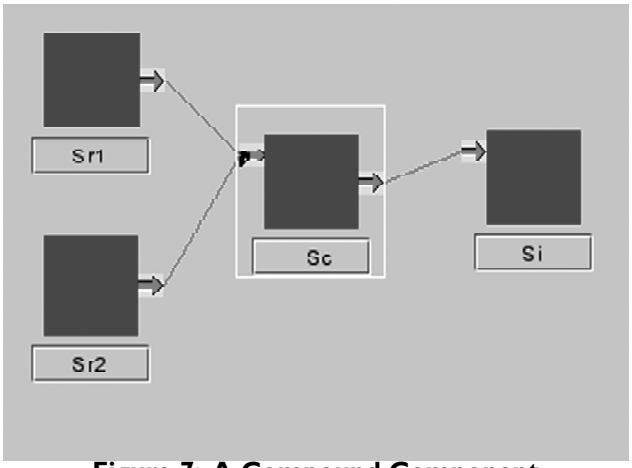


Figure 7: A Compound Component

or X is a prototype, $\sigma(y) \neq$ *none* for some inport y of X, and X' is a compound component such that $X' \in \textbf{class}(X)$, $\textbf{inports}(X') = \textbf{inports}(X)$ and $\textbf{outports}(X') = \textbf{outports}(X)$ and $\sigma'(x) = \sigma(x)$ if $x \in \textbf{attributes}(X)$ and $\sigma'(x) =$ *none* otherwise.

Expansion replaces a prototype with a compound component which is an instance of its class. As we have already mentioned, this process is the equivalent of procedural abstraction in programming languages, and therefore provides the basis for recursion. This is illustrated by Example 3.4 below.

3.3.7 Definition - evaluation

If σ and σ' are states, X is a component, and S is a subset of the simple components occurring in X, then $(\sigma', X, \varnothing)$ is an *evaluation* of (σ, X, S) iff for all $x \in A$

- if x is an outport of Y for some $Y \in S$, then $\sigma'(x) = \textbf{function}(Y)(\sigma(a_1), \ldots, \sigma(a_n))$ where $\textbf{inports}(Y) = (a_1, \ldots, a_n)$;
- if x is an outport of some source occurring in X, then $\sigma'(x) \in \tau(x) \cup \{\textit{none}\}$;
- otherwise $\sigma'(x) = \sigma(x)$.

The evaluation rule describes the execution of a simple component. Consider Example 3.3.5 where applying propagation resulted in the triple $(\sigma', X, \{ Sc \})$ where $\sigma'(b) = 2$, and $\sigma'(a_1) = \sigma'(a_2) = \sigma'(c) = \sigma'(d) =$ *none*. Applying evaluation leads to the triple $(\sigma'', X, \varnothing)$ where $\sigma''(b) = 2$, $\sigma'(a1) = \sigma''(a2) = \sigma''(d) =$ *none* and $\sigma''(c) = \sigma''(b)^2 = 4$.

3.4 Discussion with an Example

In this section we work through an example, the recursive computation of factorial, which is clearly not intended to illustrate the value of component-based software, but to illustrate the above definitions.

We assume domain D includes integers and that all attributes in this example have integer type. The compound component **fact** shown in Figure 8 is comprised of three simple components, one prototype of **fact** itself, one inport x, one outport y, and five connections.

Formally, the component **fact** is defined as follows.

$\textbf{fact} = (\{ \textbf{f}_1, \textbf{f}_2, \textbf{f}_3, \textbf{factPro} \}, (x), (y), \{(x, \{t_1, t_2, i_3\}), (o_2, \{t_f\}), (o_f, \{t_3\}), (o_1, \{y\}), (o_3, \{y\})\})$

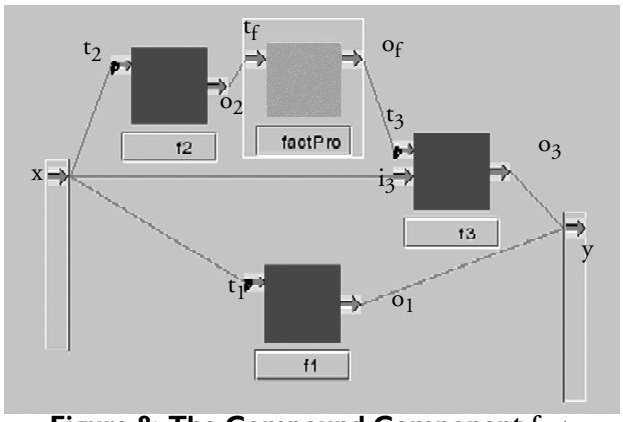


Figure 8: The Compound Component fact

The three simple components $\mathbf{f}_1$, $\mathbf{f}_2$, $\mathbf{f}_3$, are as follows.

$\mathbf{f}_1$ = ((t_1), o_1, F_1, {(0,*m*), (1,*m*) }) where $F_1(i) = 1$ for all integers *i*

$\mathbf{f}_2$ = ((t_2), o_2, F_2, {(*n*,*m*) | *n* > 1}) where $F_2(i) = i - 1$ for all integers *i*

$\mathbf{f}_3$ = ((t_3, i_3), F_3, {(*n*,*m*) | either $n \neq m$ or $n = 1$}) where $F_3(i,j) = i*j$ for all integers *i* and *j*

The prototype factPro is defined as follows.

factPro = (**factClass**, t_f, o_f)

Where **factClass** is the class that contains the compound component **fact**.

These three components not only perform the basic computations, but also control the execution flow of program. The function of $\mathbf{f}_1$ is to decide whether or not to invoke the base case, and the function of $\mathbf{f}_2$ is to decide whether or not to invoke the recursive case, passing a value to $\mathbf{f}_3$ only if the incoming integer is greater than 1. The function of $\mathbf{f}_3$ is to calculate the result as the product of the original integer and the factorial computed by the prototype.

To illustrate the above definitions, we now describe the execution of **fact** given a starting value of 2. The paragraphs that follow describe successive triples in the execution, and show how each triple is derived from the previous one. In each paragraph, we describe the new state by giving only the attribute values which have changed: attributes not mentioned have the same values as in the previous state.

(σ_0, **fact**, ∅)

- $\sigma_0(x) = 2$, and $\sigma_0(a) =$ *none* for all attributes $a \neq x$.
- Propagation applies since $x \neq$ *none* and x is the origin of three connections.

(σ_1, **fact**, {$\mathbf{f}_2$})

- $\sigma_1(x) =$ *none*, and $\sigma_1(t_1) = s_1(t_2) = \sigma_1(i_3) = 2$.
- Since $\sigma_1(t_1)$ is not 1 or 0, $\mathbf{f}_1$ is not triggered. Since $\sigma_1(t_2) > 1$, $\mathbf{f}_2$ is triggered.
- Propagation does not apply since there is no origin with a value other than *none*.
- Expansion does not apply since the third element of the triple is not ∅.
- Evaluation applies, yielding the following triple.

(σ_2, **fact**, ∅)

- $\sigma_2(o_2) = 1$.
- Propagation applies since $o_2 \neq$ *none* and is the origin of a connection.

(σ3, **fact**, ∅)

- $\sigma_3(t_f) = 1$, and $\sigma_3(o_2) =$ *none*.
- Propagation does not apply since there is no origin with a value other than *none*.
- Expansion applies since $\sigma_3(t_f) \neq$ *none*.

(σ_4, **fact'**, ∅)

- **fact** is replaced by **fact'** as shown Figure 9(a). The component **fact**" occurring in **fact'** is shown in Figure 9(b). Like **factPro**, **factPro'** is a prototype for the class of which **fact** is a member.
- $\sigma_4(a) = \sigma_3(a)$ if a is an attribute occurring in fact and $\sigma_4(a) =$ *none* otherwise.
- Propagation applies since $t_f \neq$ *none* and is the origin of three connections.

(σ_5, **fact'**, {$\mathbf{f}_1$'})

- $\sigma_5(t_f) =$ *none*, and $\sigma_5(t_1') = \sigma_5(t_2') = \sigma_5(i_3') = 1$.
- Since $\sigma_5(t_1')$ is 1, $\mathbf{f}_1$' is triggered. Since $\sigma_5(t_2') = 1$, $\mathbf{f}_2$ is not triggered.
- Evaluation applies, yielding the following triple.

(σ_6, **fact'**, ∅)

- $\sigma_6(o_1') = 1$.
- Propagation applies since $o_1' \neq$ *none* and is the origin of a connection.

(σ_7, **fact'**, ∅)

- $\sigma_7(o_1') =$ *none*, $\sigma_7(o_f) = 1$.
- Propagation applies since $o_f \neq$ *none* and is the origin of a connection (see Figure 9(a)).

(σ_8, **fact'**, {$\mathbf{f}_3$})

- $\sigma_8(o_f) =$ *none*, and $\sigma_8(t_3) = 1$.
- Since $\sigma_8(t_3)$ is 1, $\mathbf{f}_3$ is triggered.
- Evaluation applies, yielding the following triple.

(σ_9, **fact'**, ∅)

- $\sigma_9(o_3) = \sigma_1(i_3)* \sigma_8(t_3) = 2*1 = 2$.
- Propagation applies since $o_3 \neq$ *none* and is the origin of a connection.

(σ_{10}, **fact'**, ∅)

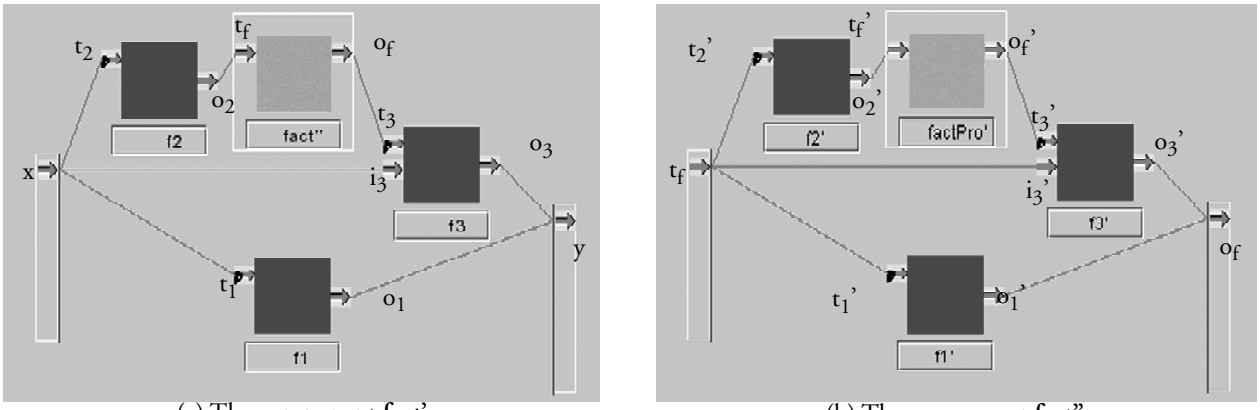
- $\sigma_{10}(o_3) =$ *none*, and $\sigma_{10}(y) = 2$.
- No rules are applicable so the execution sequence terminates.

The output, provided via outport y, is 2 which is the factorial of 2.

Although the above example is rather trivial, it does demonstrate most features of our definitions. From the example, we see that the notion of "recursive component" is a natural extension of the usual concept of component, and that the concept of compound component includes the notion of "program".

4 Further investigation

As mentioned previously, the definitions presented above have been implemented in a prototype software development environment consisting of a visual editor for building components by directly manipulating visual representations, and an interpreter.



(a) The component **fact'** (b) The component **fact"**

Figure 9: The expansion of fact

Although the prototype is implemented in Java, it currently does not take advantage of that fact by generating beans or by producing any other form of executable Java code. However, Java is used to code the textual parts, that is, the functions of components and the relations of triggers.

In further development of the prototype, we will investigate the possibility of generating components according to different component standards, starting with JavaBeans. Although our definition covers the fundamental common features of the various component models, each model also has its own unique features. We need to find a way of "plugging in" modules that will accommodate these features.

Another visual language project that we are involved with is implementing the visual dataflow language, JGraph, that provides the features of Java and supports both the visual development of Java programs, and visualisation of programs written in Java [14]. Component software development systems require an underlying programming language to build the algorithmic portions of systems not addressed by component models. For example, Visual Age relies on Java (or other languages); COM components interface with Visual Basic, Visual C++ and other languages. Since JGraph and the component model described here provide two complementary software development methodologies via visual programming interfaces, we feel that merging the two could result in a much more consistent and natural combination of components and algorithmic programming than that achieved in products such as Visual Age.

5 Acknowledgements

This research was supported by Natural Sciences and Engineering Research Council of Canada Research Grant RGPIN124-99.

6 References

[1] Allen R., Garlan D., (1997), A Formal Basis for Architectural Connection, *ACM Trans. on Software Engineering and Methodology*, v6, n3, 213–249.

[2] Apple Computer Inc., (1993), *OpenDoc – Shaping Tomorrow's Software* White paper.

[3] Carrel-Billiard, M.; Akerley J., (1998), *Programming with VisualAge for Java*, IBM redbook.

[4] Drossopoulou S. and Eisenbach S., (1998), *Towards an Operational Semantics and Proof of Type Soundness in Java* [online]. Available: http://www-dse.doc.ic.ac.uk/projects/slurp/pubs.html#towards

[5] D'Souza D. F. and Wills A.C., (1997). *Objects, Components, And Frameworks with UML – the Catalysis Approach*, Addison-Wesley, Reading, Mass.

[6] Gamma E., Helm R., Johnson R. and Vlissides J., (1994). *Design Patterns – Elements of Resuable Object-Oriented Software*, Addison-Wesley, Reading, MA.

[7] Ibrahim R. and Szyperski C., (1998). Formalization of Component Object Model (COM) – The COMEL Language, *Proc. 8th PhD Workshop, ECOOP'98*

[8] Jacobson I., Griss M., and Jonsson P., (1997). Software Reuse, *Architecture Process and Organization for Business Success*, ACM Press, Addison-Wesley Longman, NJ.

[9] Leach R. J., (1997). *Software Reuse - method, models, and costs, McGraw-Hill*, NJ.

[10] Lewis T., Rosenstein L., Pree W., Weinand A., Gamma E., Caulder P., Andert G., Vlissides J. and Schmucker K., (1995). *Object Oriented Application Frameworks*, Manning Publications Co., Greenwich, CT.

[11] Magee J.,Dulay N., Eisenbach S., Kramer J., (1995), Specifying Distributed Software Architectures, *Proc. 5th European Software Engineering Conf.*, Spain, Springer-Verlag, Berlin, 137-153.

[12] Microsoft Corporation, (1995). *The Component Object Model Specification, Version 0.9* [online]. Available http://www.microsoft.com/Com/resources/comdocs.asp>.

[13] Prieto-Diaz R., Neighbours J., (1986), Module Interconnection Languages, *Journal of Systems and Software 6, 4* , 307-334.

[14] Risley, C.C., JGraph: *A Java Compatible Visual Language*, MCompSci Thesis, Dalhousie University.

[15] Shaw M., Garlan D., (1996), *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall., NJ.

[16] Spivey J.M., (1989), *The Z Notation, a Reference Manual*, Prentice Hall, Englewood Cliffs, NJ.

[17] Sullivan K. J., Socha J., and Marchukov M., (1997). Using Formal Methods to Reason about Architectural Standards, *Proceedings of 19th International Conference on Software Engineering*, Boston, USA

[18] Sun Microsystems, (1997). *JavaBeans for Java Studio: Architecture and API*, White paper.

[19] Szyperski C., (1998). *Component Software, Beyond Object-Oriented Programming*, ACM Press, Addison-Wesley, NJ.