A Typeful Composition Model for Dynamic Software Architectures

Frédéric Peschanski
Laboratoire d'Informatique de Paris 6 (LIP6)
Frederic.Peschanski@lip6.fr

Christian Queinnec
Laboratoire d'Informatique de Paris 6 (LIP6)
Christian.Queinnec@lip6.fr

Jean-Pierre Briot
Laboratoire d'Informatique de Paris 6 (LIP6)
Jean-Pierre.Briot@lip6.fr

ABSTRACT

Future generations of software systems should be able to evolve consistently while they are running. To address this problem, we propose a model and a domain-specific language, named Scope, that offers the dynamic composition of component-based software architectures. The basic constitutive operation available is the establishment of a connection between two given components. Interconnected components can exchange information in the form of anonymous event emissions. Event types are used in our work both to capture programmer's intentions at the conceptual level and to ensure type safety at the operational level. A type inference algorithm maps these two levels. Compositions can be flat or recursive, the result being an architecture or a composite component. Language-level extensions are proposed through type confinement, connection labelling and component refinement.

1. INTRODUCTION

Software architectures describe the overall structure of complex computer systems rather than the details of their implementations [2]. From this point of view, the architectures are considered as interconnections of computational components.

Following these principles but with a focus on operationalization, we present in this paper a model and a language, Scope, which allow the *incremental* composition and modification of *dynamic* software architectures.

Using Scope, components can be added or removed at runtime. The way these components are related to each other can also evolve. To that effect, *connections* between components can be established or removed while the system is running, in a controlled manner.

We relate our work to the emerging field of *Software evolution* which addresses more generally the complex problems related to the design, implementation and execution of computer systems that will evolve over time [1].

Compared to existing work, we see three important contributions highlighted in this paper.

First, the model proposes as foundation a taxonomy of the data exchanged by components - the *events* and their types - without any direct relation to the corresponding computer-level representation. When connecting two given components, a developer describes the *nature* of the exchanged data: temperatures, prices, database requests and so on. These informations correspond to the intentions of the developer, hence their designation as *intentional types*. Then, an efficient type inference algorithm deduces the corresponding *effective types* which describe the categories of data that components will *effectively* pass to each other: integers, tuples, and so on. This way, we foster the compositional issue and its conceptual point of view while preserving the operationalization needs.

Second, the major interest of the *linguistic* approach we adopt is that Scope considers events, event types, components, component definitions as well as intentional connections as first-class entities. For example, we describe in this paper a set of dataflow operators that can be considered as parameterizations (which we will design as refinements) of more abstract component categories. The addition and multiplication operators of the example only differ by the internal operation they realize. Ultimately, they conceptually represent components which receive and emit numbers and should so be usable at this abstraction level.

Third, perhaps the most fundamental contribution of our work also results from our linguistic point of view. Under some conditions, an architecture - a set of interconnected components - can be given the status of a first-class entity : a *composite component*[1] which can be itself reused to compose new architectures. We see this *recursive composition model* as a very expressive and intuitive abstraction tool.

We decided to formalize precisely the type system and all the language primitives introduced in this paper. The formalization itself is based on state-transition semantics owing to the dynamic nature of the proposed model.

The paper begins with the description of the Scope composition model, from the structural point of view. First we introduce in section 2 the event-based type system. Then the component model can be itself elaborated in section 3, as well as the whole flat composition model. We finally introduce the composite components and the recursive composition model in section 4.

The Scope language is then presented in section 5. The example we chose to illustrate this section is a simple dataflow,

[1] Sometimes called a *nested component*.

inspired by [4]. While very simple, it exhibits all the constructs of our language: component definition and use, event management as well as some interesting language-level extensions: confined types, labelled connections and component definitions refinement.

Finally, we relate our project to other research work and conclude the paper.

2. TYPE SYSTEM

We will begin our discussion by the elaboration of the type system which represents the core constituent of our formal model.

2.1 Event types

Types describe the events that components exchange at runtime. The basic event types represent simple data types like numbers or strings. But the type system itself is elaborated incrementally and dynamically so more complex datatypes can be added to the system when needed.

Definition 1. Let $\mathcal{T}$ be the **type system**, i.e. the set of all the event types managed by the system at a certain time.

The elements of $\mathcal{T}$ are not independent, they can be related using different mathematical relations. In addition to the implicit relations $=$ and $\neq$, the most important one is the supertype (resp. subtype) relation.

Definition 2. Let x and y be elements of $\mathcal{T}$. If x is a **supertype** (resp. **subtype**) of y, then we note $x \geq y$ (resp. $x \leq y$).

We will also consider the *strict* variants $>$ and $<$. We will also say that two types x and y are related if and only if $x \geq y$ or $y \geq x$. We will then note $x \sim y$. Unrelated types will be noted $x \not\sim y$.

The supertype relation is a partial order: it is reflexive, transitive and antisymmetric. we say that $\langle \mathcal{T}, \geq \rangle$ defines a *partially ordered set* or *poset*.

We also impose that our type system contains a *unique root* element , i.e. $\exists!\, z \in \mathcal{T} \,/\, \forall x \in \mathcal{T}, z \geq x$

A fundamental property of our type system is that for a given type x in $\mathcal{T}$, there can be multiple associated direct subtypes or supertypes because the $\langle \mathcal{T}, \geq \rangle$ set is only partially ordered. The following functions extract the minimum (resp. maximum) supertypes (resp. subtypes) from a subset T of $\mathcal{T}$:

Definition 3. Let $T \subseteq \mathcal{T}$,
$\mathbf{lub}(T) \equiv T \setminus \{x \in T \mid \exists y \in T, y < x\}$, and
$\mathbf{glb}(T) \equiv T \setminus \{x \in T \mid \exists y \in T, y > x\}$

Figure 1 shows a graphical representation of a basic type system. We can see that while being very simple, this type system exposes some important properties. In many type systems, real numbers and integers are considered as disjoint because they are represented in a different way at the operational level while perhaps from a more intuitive point of view, integers are particular real numbers.

To decouple the structure and the purpose of the type system, we remain deliberately open regarding the interpretation of types. The inference algorithms introduced in this paper only captures the structural properties of the type system. Set-based or logical semantics can be employed to give meanings to the types themselves.

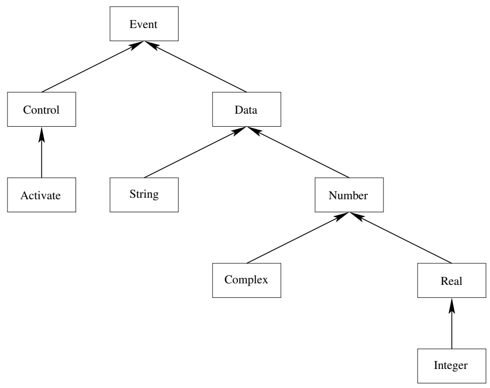

Figure 1: Graphical representation of a type system

Another interesting property is the introduction of control-related types like the *Activate* type. We explored the needs for control links at the interconnection graph level in previous work [3].

2.2 Type system dynamics

The dynamic nature of the type system we define is a fundamental property. Initially, the type system is only composed of the *Event* root type which is as expected related to itself in the $\geq$ relation:

Initial state 1. $\langle \mathcal{T}, \geq \rangle_{init} = \langle \{Event\}, \{Event \mapsto Event\} \rangle$

Because the state of the type system will evolve over time, we want to ensure the correctness of the changes operated on it. To do so, we define an *invariant* of the type system so that we can prove the state modifications by ensuring the preservation of this invariant.

Invariant 1. $\langle \mathcal{T}, \geq \rangle$ is a **Poset** and has a **Unique root**.

We can now introduce the fundamental operation of adding a new type in the system. This operation is described as a transition between the state of the type system *before* the change, $\langle \mathcal{T}, \geq \rangle$ and the corresponding *after* state, $\langle \mathcal{T}', \geq' \rangle$:

$\mathbf{add\_type(t?, super?)} : \langle \mathcal{T}, \geq \rangle \rightarrow \langle \mathcal{T}', \geq' \rangle$
$t? \notin \mathcal{T}$ $super? \neq \emptyset$ $super? \subseteq \mathcal{T}$ $\forall s_1 \in super?, \forall s_2 \in super? \setminus \{s_1\}\ then\ s_1 \not\sim s_2$
$\mathcal{T}' = \mathcal{T} \cup \{t?\}$ $\geq' = (\geq \cup \{s \mapsto t? \mid s \in super?\})^*$

The *add_type* operation takes two parameters[2]: $t?$ which represents the type to add to the type system (input type) and *super*? as the set of direct supertypes of t? (input set of supertypes).

Suppose we want to add a new type *Price*[3] as having *Real* and *String* as direct supertypes. The first precondition explains that the new type added must not preexist in

[2] We use question marks to suffix parameters.

[3] A price could be considered as an amount represented literally or textually.

the type system. We must check that $Price \notin \mathcal{T}$. The direct supertypes (*Real* and *String*) must also be disjoint, as expressed in the last precondition.

The first postcondition, described in the last part of the definition, expresses that if all the preconditions are fulfilled, then the new type $t?$ is a valid type of the resulting type system. In our example, *Price* will be a new element of the type system. Next, we have to add all the mappings in the $\geq$ relation that relate our new event type $t?$. This is the reflexive and transitive closure of the mapping from all the direct supertypes in *super*? to $t?$. *Price* should be related to the set $\{Price, String, Real, Number, Data, Event\}$ which reflects the fact that $Price \geq Price, String \geq Price, Real \geq Price$ and so on.

3. FLAT COMPOSITION MODEL

We call flat composition any assembly of related components which form an architecture. In order to allow flat composition, we have to explain how components can be added to existing architectures and then how to relate components using connections.

3.1 Components

At the structural level, a Scope component is an abstract entity defined by an identifier and a type. The component type is defined as follows:

Definition 4. A **component type** is a couple $\langle In, Out\rangle$ where In (resp. Out) is the **input type** (resp. **output type**) of the component, represented as a set of types.

Informally, In (resp. Out) describes the kinds of event a component is able to receive (resp. likely to emit). It is important to note that there exists no relation between the types of In and those of Out. This way, we avoid an important covariant/contravariant issue as found in most typed object models [7].

Moreover, In and Out are, mathematically speaking, sets so a given type cannot be declared twice in the same set.

However, related but different types can be declared in either In or Out. Consider for example the component type: $\langle\{Data, String\}, \{Integer, Real\}\rangle$

While the input and output types indicated are related in the subtyping relation, these component types are perfectly valid.

So, a component type corresponds to a logical description stating conceptually which kinds of data a component is able to receive or likely to send. Other component models compel "physical" considerations through *pins* [4] or *ports* [8]. We will see in section 5.6 that we can model such physical entities in Scope through *labelled connections*.

A component is, from the type system point of view, defined like this:

Definition 5. A **component** is a couple $\langle Id, Type\rangle$ where Id identifies the component and $Type = \langle In, Out\rangle$ is the component type as defined above.

Let's describe a dataflow operator *add*1 which performs additions of real numbers:
$\langle add1, \langle\{Real\}, \{Real\}\rangle\rangle$

Conceptually, this additioner receives and sends sequences of real numbers. Following a common dataflow behaviour, the corresponding operational component (as described in section *5.2.1*) will perform the addition as soon as possible, that is, when two numbers will be available. Each time an addition is realized, a result event is sent through the component output. A possible graphical representation of this component is depicted in figure 2. Note that we associated a name (*AddReal*) to the component definition, because of its first-class status.

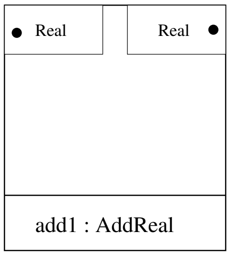

Figure 2: Graphical representation of a dataflow additioner

3.2 Effective connections

Because of their dynamic nature, connections have an operational meaning hence their designation as effective connections.

Definition 6. An **effective connection** is a triplet $\langle Id_1, Id_2, T\rangle$ where Id_1 and Id_2 identify the *source* component c_1 and the *destination* c_2. T is the set of **effective types** of the connection.

The main information to point out in the previous definition is that T denotes a set of types designed as effective. This set explains which kind of events can be sent to c_2 from c_1 using the given connection. From now on, an event of type t_1 can be sent to c_2 using the connection $\xi = \langle Id_1, Id_2, T\rangle$ if and only if $\exists t_2 \in T$ *such as* $t_2 \geq t_1$.

For example, Figure 3 describes the effective connection $\langle add1, add2, \{Real\}\rangle$ between two additioners *add*1 and *add*2, realizing a quaternary additioner dataflow that can work on real numbers or any subtypes (like integers).

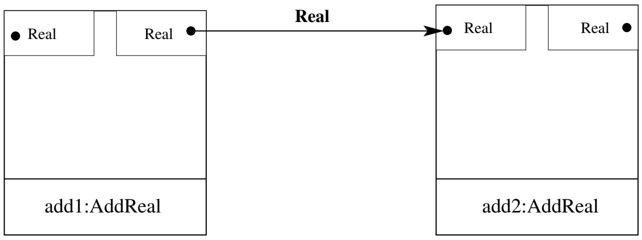

Figure 3: Graphical representation of an effective connection between two components

Note that because of the decoupling between the input and output types, the connections in Scope are *unidirectional* while other approaches introduce bidirectional links [4] [11]. Of course, bidirectionality can be obtained through connection composition.

3.3 Architectures

A Scope architecture is a running system composed of interconnected components. From a formal point of view, an architecture then agglomerates a set of components together with a set of effective connections.

Definition 7. An **architecture** $\mathcal{A}$ is a couple $\langle \mathcal{C}, \Xi \rangle$ where $\mathcal{C}$ is the set of components which compose the architecture and Ξ is the set of effective connections describing how the components in $\mathcal{C}$ are related to each other.

An *operational system* will then be the association of a type system and an architecture:

Definition 8. An **operational system** is a couple: $\langle \langle \mathcal{T}, \geq \rangle, \mathcal{A} \rangle$ where $\mathcal{A}$ is the architecture managed by the operational system.

Initially, the system is empty:

Initial state 2. $\langle \langle \mathcal{T}, \geq \rangle, \mathcal{A} \rangle_{init} = \langle \langle \mathcal{T}, \geq \rangle_{init},\ \langle \emptyset,\ \emptyset \rangle \rangle$.

We then define an invariant for the operational system:

Invariant 2. The type system invariant must be satisfied, and:

- $\forall c_1 = \langle Id_1, Type_1 \rangle \in \mathcal{C}$, $\forall c_2 = \langle Id_2, Type_2 \rangle \in \mathcal{C} \backslash \{c_1\}$, $Id_1 \neq Id_2$
- $\forall \xi = \langle c_1, c_2, T \rangle \in \Xi$,
  - $c_1 = \langle Id_1, \langle In_1, Out_1 \rangle \rangle \in \mathcal{C}$
  - $c_2 = \langle Id_2, \langle In_2, Out_2 \rangle \rangle \in \mathcal{C}$
  - $\forall t \in T$, $\exists t_1 \in Out_1 \,/\, t_1 \sim t$
  - $\forall t \in T$, $\exists t_2 \in In_2,' \,/\, t_2 \geq t$

Added to the expected preservation of the type system invariant, the operational system invariant must also ensure that all the components in the system have different identifiers. The third statement expresses that every established connection references valid source and destination components.

The effective types T must also be "compatible" with the types of the source and destination components. First, this set must contain types that are related to at least one output type of the source component. This way, any event communicated through the connection can be emitted from the source component. Suppose that a *number* connection is connected to a *real* output. Each real number event will be emitted through the connection while unrelated or more generic numbers will be silently rejected.

The final condition says that for every effective connection type, there must exist a supertype in the destination component inputs. This ensures that the destination component can handle every types of events emitted by the source component using the connection. If only real numbers are accepted by a component, then an *Integer* connection is inconsistent because integers have some properties not available in more general numbers (like a successor).

3.4 Operations

After having described the invariant and the initial state of our operational system, we can introduce the different primitive operations allowed on it. The type system itself is not modified by any of these operations except *add_type* which has been described previously. So, from now on, we will only describe the architectural modifications of the system.

3.4.1 Adding and removing components

The *new* operation which allows the introduction of a new component in the system is defined as follows:

$\mathbf{new(c? = \langle Id?, \langle In?, Out? \rangle \rangle)} : \langle \mathcal{C}, \Xi \rangle \rightarrow \langle \mathcal{C}', \Xi' \rangle$
$(In?, Out?) \subseteq \langle \mathcal{T}, \geq \rangle^2$ $\forall c = \langle Id, Type \rangle \in \mathcal{C}, Id \neq Id?$
$\mathcal{C}' = \mathcal{C} \cup \{c?\}$ $\Xi' = \Xi$

As preconditions, we must ensure that the input and output types of the component to add to the system are valid. Its identifier must also be different from the ones already defined by the system. If these preconditions are satisfied, then we can effectively add the component by a simple update of the $\mathcal{C}$ set. The other elements of the system remain unchanged. The removal of a component can then be defined:

$\mathbf{remove(Id?)} : \langle \mathcal{C}, \Xi \rangle \rightarrow \langle \mathcal{C}', \Xi' \rangle$
$\exists!\, c = \langle Id, Type \rangle \in \mathcal{C} \,/\, Id = Id?$ $\forall \xi = \langle c_1 = \langle Id_1, Type_1 \rangle,\ c_2 = \langle Id_2, Type_2 \rangle, T \rangle \in \Xi$ *then* $(Id_1 \neq Id?) \wedge (Id_2 \neq Id?)$
$\mathcal{C}' = \mathcal{C} \backslash \{c\}$ $\Xi' = \Xi$

It is important to point out that this removal transition can only take place when the component to remove is not connected to any other component. We will define a more useful removal operation in section 3.5.3.

3.4.2 Establishing connections

The establishment of a connection between two components expects three informations: the identifiers of the source and destination components and an intentional connection type, which can be under some condition omitted.

We identified three different ways to connect components that, while preserving the system invariant, correspond to specific and meaningful intentions from the programmer's point of view: *specialization*, *generalization at source* and *generalization at destination*. The fundamental property, common to all the connection categories, is that the effective types of the resulting connection are always subtypes of the intentional connection type.

3.4.3 Connection by specialization

In most situations, components will be connected through a *specialization* algorithm. The idea is that we want to describe the connections at a higher level of abstraction than the description of the components themselves.

Suppose for example that we implement different dataflow operators which work on strings or numbers. We would like to describe the connections between these operators at a higher level of abstraction, for example by indicating that in each case, they emit and receive events of type *Data* (a common supertype of *Number* and *String*).

Consider the components $c_1 = \langle Id_1, \langle \emptyset, \{Integer\} \rangle \rangle$ and $c_2 = \langle Id_2, \langle \{Number\}, \emptyset \rangle \rangle$.

If we use *Data* as the intentional connection type, the system should infer *Integer* as effective type when we try to connect c_1 to c_2 by specialization because this is the exact type of the events emitted by c_1 (see Fig. 4).

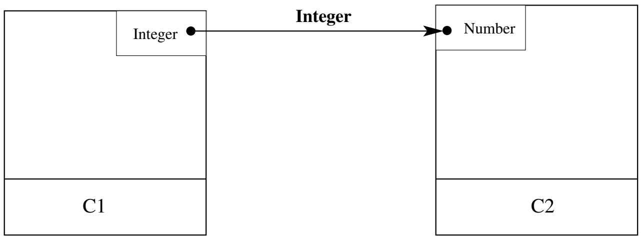

Figure 4: Connection by specialization

The output type chosen at the source component must be, in the specialization case, a large subtype of the destination input type. This is the case in our example since $Integer \leq Number$. Both must be also subtypes of the intentional type, a condition fulfilled by our example.

The result is the connection $\xi = \langle Id_1, Id_2, \{Integer\}\rangle$.

At the operational level, we can then ensure that events of type $Integer$ (and all its subtypes) can now circulate from c_1 to c_2 through ξ.

Sometimes, more than one effective type should be inferred. Suppose that we add a type $Complex$ to the outputs of c_1. Then, when choosing the effective types, both $Integer$ and $Complex$ match so the resulting connection should be:
$\xi = \langle Id_1, Id_2, \{Integer, Complex\}\rangle$

On the other hand, the effective connection types cannot be resolved if we try to connect the same components using a too specialized intentional type, for example $Binary$[4]. There is no output type in c_1 or input type in c_2 which are specializations of $Binary$.

Now that we exposed the principles of the connection by specialization primitive, we can give its definition:

$\textbf{connect\_specialize}(\mathbf{Id_1}?, \mathbf{Id_2}?, \mathbf{t}?) : \langle \mathcal{C}, \Xi\rangle \rightarrow \langle \mathcal{C}', \Xi'\rangle$
$\exists c_1 = \langle Id_1, \langle In_1, Out_1\rangle\rangle \in \mathcal{C} \,/\, Id_1 = Id_1?$
$\exists c_2 = \langle Id_2, \langle In_2, Out_2\rangle\rangle \in \mathcal{C} \,/\, Id_2 = Id_2?$
$T_1 = glb(\{t_1 \in Out_1 \mid t_1 \leq t?\}) \neq \emptyset$
$T_2 = glb(\{t_2 \in In_2 \mid t_2 \leq t?\}) \neq \emptyset$
$T = \{t \in T_1 \mid \exists t' \in T_2, t \leq t'\} \neq \emptyset$
$\xi = \langle c_1, c_2, T\rangle \notin \Xi$
$\mathcal{C}' = \mathcal{C}$
$\Xi' = \Xi \cup \{\xi\}$

The first and second preconditions make sure that the components involved by the connection already exist in the system. The third precondition defines a set of types called T_1 which contains all the largest subtypes of the intentional connection type $t?$ in the output types of the source component, using the glb function defined in section 2. This set, for the connection to work, must be different from the empty set. The set T_2 is then defined as the largest subtypes of $t?$ in the inputs of the destination component which also must be non-empty.

We then define the set T of the effective connection types including all the elements in T_1 that have at least one subtype in T_2. This makes sure that the events sent by the source components can be received by the destination component. If this set is empty, then the connection fails. Finally, as a last precondition, the new connection, with effective types T as defined above, must not preexist in the system[5].

We can note that, from the definitions of T_1, T_2 and T, we have $\forall t \in T,\ t \leq t?$. This is consistent with the property that the effective types of a connection are subtypes of the intentional connection type.

The most important postcondition states that the system now as a new connection established.

3.4.4 Generalization at source

In some cases, an output type can be used as a dispatcher between different intentional subtypes. The idea here is to connect a generic component to more specialized ones, using subtypes to dispatch the events between the different destinations.

Consider the components:
$c_1 = \langle Id_1, \langle \emptyset, \{Number\}\rangle\rangle$, $c_2 = \langle Id_2, \langle \{Integer\}, \emptyset\rangle\rangle$, and $c_3 = \langle Id_3, \langle \{Complex\}, \emptyset\rangle\rangle$.

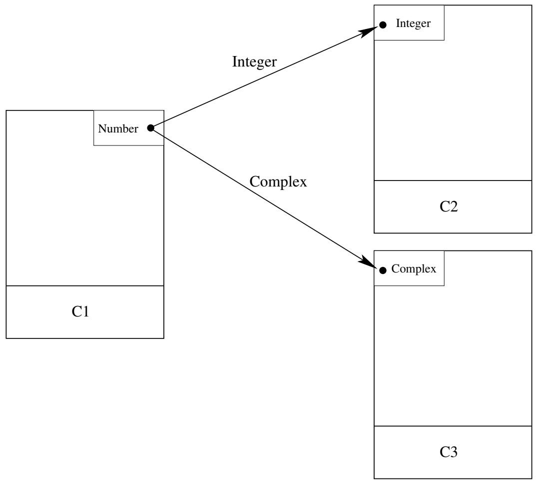

Figure 5: Type-dispatcher component using connections by generalization at source

We would like to connect c_1 to c_2 for the type $Integer$ and c_1 to c_3 for the type $Complex$ as depicted in Figure 5. A possible interpretation would be to associate c_1 to a component that asks the user to enter a number with the keyboard. If an integer is entered, then an event in sent to c_2 while if the user enters a complex numbers, the event is sent to c_3.

Using a specialization connection category would lead to a refusal since $Number$ is both a supertype of $Integer$ and $Complex$. In this situation, where c_1 is identified as a *type dispatcher* for subtypes of $Number$ events, we can use the connection by generalization at source to obtain the following effective connections :

- $\xi_1 = \langle Id_1, Id_2, \{Integer\}\rangle$ using the declared type $Integer$ and
- $\xi_2 = \langle Id_1, Id_3, \{Complex\}\rangle$ using $Complex$.

[4]We suppose $Binary$, representing binary numbers $\{0, 1\}$, as being a subtype of $Integer$.

[5]This case won't lead to an error, we will silently reject the connection since it has already been established.

This variant can be formalized like this[6]:

$\mathbf{connect\_gen\_source(Id_1?, Id_2?, t?)} : \langle \mathcal{C}, \Xi \rangle \rightarrow \langle \mathcal{C}', \Xi' \rangle$
$T_1 = lub(\{t_1 \in Out_1 \mid t_1 \geq t?\})$ $\quad \cup\, glb(\{t_1 \in Out_1 \mid t_1 < t?\}) \neq \emptyset$ $T_2 = glb(\{t_2 \in In_2 \mid t_2 \leq t?\}) \neq \emptyset$ $T = \{t \in T_2 \mid \exists t' \in T_1,\, t \leq t'\} \neq \emptyset$

The main difference with the connection by specialization operation is that now, the lookup phase will also extract all the smallest supertypes (using the *lub* function) of the intentional connection type in the outputs of the source component. Then, as usual, the largest subtypes are searched at both ends of the connection. The resulting set of effective types will then be the types in these inputs that have a supertype in the selected source outputs.

But even if the chosen source types can be generalizations of the intentional connection type, the fact that $T \subseteq T_2$ still leads to the property: $\forall t \in T,\, t \leq t?$.

3.4.5 Generalization at destination

The complementary connection category of the previous one is the connection by generalization at destination.

The idea is now to connect a generic component as destination of some specialized sources.

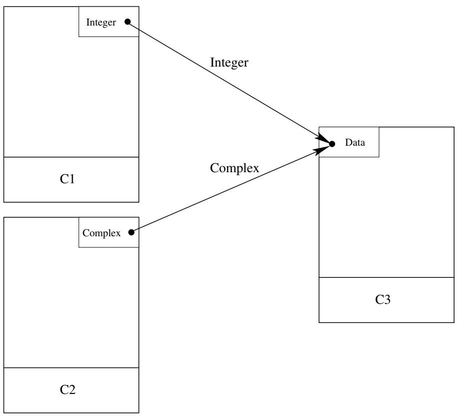

Figure 6: Type-concentrator component using connections by generalization at destination

Suppose we want to connect $c_1 = \langle Id_1, \langle \emptyset, \{Integer\} \rangle \rangle$ and $c_2 = \langle Id_2, \langle \emptyset, \{Complex\} \rangle \rangle$ to $c_3 = \langle Id_3 \langle \{Data\}, \emptyset \rangle \rangle$ (see fig. 6).

Here c_3 can be designed as a *type concentrator* component. It could for example represent a useful generic data inspector.

The definition of the corresponding operation follows:

$\mathbf{connect\_gen\_dest(Id_1?, Id_2?, t?)} : \langle \mathcal{C}, \Xi \rangle \rightarrow \langle \mathcal{C}', \Xi' \rangle$
$T_1 = glb(\{t_1 \in Out_1 \mid t_1 \leq t?\}) \neq \emptyset$ $T_2 = lub(\{t_2 \in In_2 \mid t_2 \geq t?\})$ $\quad \cup\, glb(\{t_2 \in In_2 \mid t_2 < t?\}) \neq \emptyset$ $T = \{t \in T_1 \mid \exists t' \in T_2,\, t \leq t'\} \neq \emptyset$

Now, this variant chooses subtypes of the intentional connection type at source but possibly supertypes at destination. Anyway, the operation still ensures that the final effective types are subtypes of the intentional type.

[6]Only the modified conditions are indicated in the description.

3.4.6 Removing connections

The connection removal operation is, from a structural point of view, very simple. It is defined like this:

$\mathbf{disconnect(Id_1?, Id_2?, t?)} : \langle \mathcal{C}, \Xi \rangle \rightarrow \langle \mathcal{C}', \Xi' \rangle$
$\exists c_1 = \langle Id_1, \langle In_1, Out_1 \rangle \rangle \in \mathcal{C} \,/\, Id_1 = Id_1?$ $\exists c_2 = \langle Id_2, \langle In_2, Out_2 \rangle \rangle \in \mathcal{C} \,/\, Id_2 = Id_2?$ $\Xi? = \{\xi = \langle c_1, c_2, T \rangle \mid \forall t \in T,\, t? \geq t\} \neq \emptyset$
$\mathcal{C}' = \mathcal{C}$ $\Xi' = \Xi \backslash \Xi?$

The effect of the *disconnect* primitive is to remove all the connections from $Id_1?$ to $Id_2?$ for any (large) subtype of $t?$, if they preexist in the system.

A variant *disconnect_all* allows the parameter $t?$ to be omitted. The generic connection type is then associated to *Event*. The result will be the disconnection of all connections from $Id_1?$ to $Id_2?$.

$\mathbf{disconnect\_all(Id_1?, Id_2?)} : \langle \mathcal{C}, \Xi \rangle \rightarrow \langle \mathcal{C}', \Xi' \rangle$
$\Xi? = \{\xi = \langle c_1, c_2, T \rangle\} \neq \emptyset$

We could have defined *disconnect_all* in a simpler way: $disconnect\_all(Id_1?, Id_2?) \equiv disconnect(Id_1?, Id_2?, Event)$

Another variant, *disconnect_component*, is a similar primitive that disconnects all the connections to and from a given component. It is defined as follows:

$\mathbf{disconnect\_component(Id?)} : \langle \mathcal{C}, \Xi \rangle \rightarrow \langle \mathcal{C}', \Xi' \rangle$
$\exists c = \langle Id, \langle In, Out \rangle \rangle \in \mathcal{C} \,/\, Id = Id?$ $\Xi? = \{\langle c_1, c, T \rangle \in \Xi \mid c_1 \in \mathcal{C}\}$ $\quad \cup \{\langle c, c_2, T \rangle \in \Xi \mid c_2 \in \mathcal{C}\} \neq \emptyset$

3.5 Compound operations

Using our primitives defined in the previous section, we can define some other important compound operations.

3.5.1 Compound connection kinds

The choice of a given connection kind is highly contextual. It depends at the same time on the intentional type and on the concerned component types.

Thankfully, this choice can be in most cases inferred automatically through a simple algorithm which is described as the following compound operation:

$$
\begin{array}{l}
connect(Id_1?, Id_2?, t?)) \equiv \\
\quad \text{let } c_1 = \langle Id_1, \langle In_1, Out_1 \rangle \rangle \in \mathcal{C} \,/\, Id_1? = Id_1, \text{ and} \\
\qquad c_2 = \langle Id_2, \langle In_2, Out_2 \rangle \rangle \in \mathcal{C} \,/\, Id_2? = Id_2, \text{ in:} \\
\qquad\quad \text{if } glb(\{t_1 \in Out_1 \mid t_1 \leq t?\}) \neq \emptyset \text{ and} \\
\qquad\qquad glb(\{t_2 \in In_2 \mid t_2 \leq t?\}) \neq \emptyset \\
\qquad\quad \text{then } connect\_specialize(Id_1?, Id_2?, t?) \\
\qquad\quad \text{elseif } \; lub(\{t_1 \in Out_1 \mid t_1 \geq t?\}) \\
\qquad\qquad\qquad \cup\, glb(\{t_1 \in Out_1 \mid t_1 < t?\}) \neq \emptyset \\
\qquad\quad \text{then } connect\_gen\_source(Id_1?, Id_2?, t?) \\
\qquad\quad \text{else } \; connect\_gen\_dest(Id_1?, Id_2?, t?)
\end{array}
$$

In some situation, more than one connection kind can be established. In this case, we impose a priority on the different operations, the specialization case being the most prioritary one.

This choice results from different experiments and follows the intuition that we generally want to specialize the connection types. When it is not the case, dispatchers are generally not connected to concentrators for the same type (and vice versa). It is difficult to think about a component receiving (resp. sending) at the same time very specific and very generic events in the same type branch. So, the priority order between both kinds of generalizations could almost be chosen in a random manner.

In order to reduce the number of connections to declare, another variant establishes all the possible connections between two given components:

$$\begin{aligned}&connect\_all(Id_1?, Id_2?, t?)) \equiv\\ &\quad connect\_specialize(Id_1?, Id_2?, t?)\ ;\\ &\quad connect\_gen\_source(Id_1?, Id_2?, t?)\ ;\\ &\quad connect\_gen\_dest(Id_1?, Id_2?, t?)\end{aligned}$$

The semi-columns here form a sequence of operations where the operations on the left part of the semi-columns must be applied *before* the operation on the right part.

3.5.2 Fully inferred connection kind

Using the most general type in the system, we can completely infer the result of the connection from the structure of both the concerned components:

$$connect\_infer(Id_1?, Id_2?) \equiv connect(Id_1?, Id_2?, Event)$$

Note that this variant is more prone to ambiguities because it does not capture the programmer's intent. However, in many simple situations, this primitive can be used safely[7]. This primitive can also be of great help to discover how two given components could be connected[8].

3.5.3 Compound component removal

By removing first all the connections that relate a component to other ones using the *disconnect_component* primitive, we can then safely use the *remove* primitive to remove the component from the system.

$$remove\_gen(Id?) \equiv disconnect\_component(Id?)\ ;\ remove(Id?)$$

4. RECURSIVE COMPOSITION MODEL

Recursive composition allows architectures to be considered as composite components. The main problem here concerns the type of the resulting component because it should not be independent of the associated architecture.

Consider again the architecture depicted in Figure 3. It represents a double additioner dataflow which realizes the addition of four successive real numbers.

Suppose that we want to create a composite component C which declares that it receives and sends events of type *Activate*[9], and is associated to the double additioner architecture.

[7]For example, when it's obvious that there exists only one way to connect two given components.

[8]Silent operationalizations should be proposed to test connections without realizing them.

[9]Without being activated, the dataflow will pass events silently.

In order to define the composite type, we first select the input and output types declared at the composite level. Here, C declares that it receives and sends *Activate* events. These form the *basic types* of the composite component.

Then, we have to add to these basic input and output types all the types in the inner architecture that are not concerned by any inner connection. For example, *Real* is an input type of the inner component *add*1 and is not in any inner connection. We the say that *Real* is an *unsatisfied input type* of the inner architecture. We can remark that *Real* is also an *unsatisfied output type* of the inner component *add*2.

So finally, the resulting composite will be defined by the triplet $\langle C, \langle\{Real, Activate\}, \{Real, Activate\}\rangle, \mathcal{A}\rangle$ where $\mathcal{A}$ is the associated architecture, as shown in figure 7.

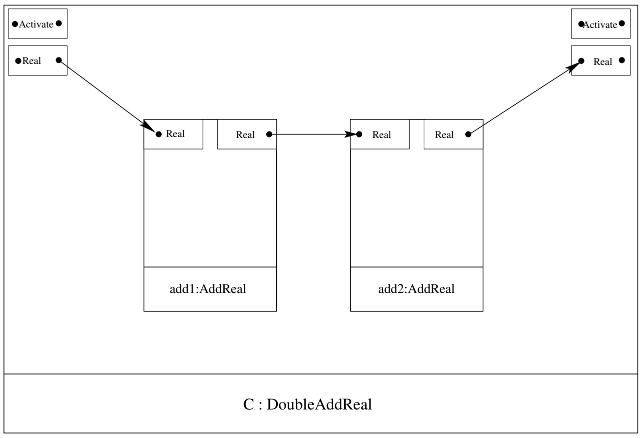

Figure 7: Graphical representation of a composite component

The general rule for defining the type of a composite component is given below:

Definition 9. A **composite component** C is a triplet $\langle Id, \langle In, Out\rangle, \mathcal{A} = \langle I_c, I_\xi\rangle\rangle$ such as:

- Id is the identifier of the component
- $\mathcal{I}_c$ is a set of inner components $\mathcal{I}_c = \{i_c = \langle id_c, \langle in_c, out_c\rangle\rangle\}$
- $\mathcal{I}_\xi$ is a set of inner connections $\mathcal{I}_\xi = \{i_\xi = \langle id_{c_1}, id_{c_2}, T\rangle\}$
- $In = BasicIn \cup InnerIn$ is the input type of the composite component such as:
  - $BasicIn$ is the set of its basic input types,
  - $\forall i_c \in \mathcal{I}_c, \forall t \in in_c, (t \in InneIn)$ iff
    $\forall i_\xi \in \mathcal{I}_\xi\ /id_{c_1} = id_c, \forall c_t \in T, (c_t \not\sim t) \vee (c_t < t)$
- $Out = BasicOut \cup InnerOut$ is its output type such as:
  - $BasicOut$ is the set of its basic output types,
  - $\forall i_c \in \mathcal{I}_c, \forall t \in out_c, (t \in InnerOut)$ iff
    $\forall i_\xi \in \mathcal{I}_\xi\ /id_{c_2} = id_c, \forall c_t \in T, (c_t \not\sim t)$

The most difficult part of the formal definition concerns the definitions of the sets $InnerIn$ and $InnerOut$. In fact, we simply formalize the process of selecting the unsatisfied input types as being those that are not concerned by any connection involving the inner components.

It is important to note that, in our approach, composite components are more than just their associated architecture : they have their own properties. For example, C is participating to an activation protocol though this is not the case of the inner double additioner. However, the composite properties are partially resulting from the associated inner architecture. C can handle real numbers only because the associated architecture is providing this service.

We think this approach corresponds to a real expectation when considered from the real world. Consider for example the model of a car, described intuitively as the composition of different components : body, motor, transmission, tires, and so on. While some properties of the car (such as its horse power) directly result from its composition, other properties are associated to the car itself : average consumption, maximum speed and so on.

5. THE SCOPE LANGUAGE

The Scope language, itself the implementation of the Scope model described previously, is an extension of the Scheme language [5], written in Scheme[10].

A Scope application can be composed of *declarative* and *embedded* statements. The declarative statements are made inside Scope definitions while the embedded ones are regular Scheme expressions. A definition is composed of a sequence of statements which are globally analyzed and interpreted by the Scope interpreter. A program itself is a Scheme program or script, with some extensions. These extensions are embedded constructs like functions or macro-definitions.

5.1 Event management

The main information added to the concept of event at the operational level is a set of denoted values called slots.

The **event** construct is used to define a new event type:

```
(event Data
  (super Event)
  (slot value))
```

This adds a new type called Data which has for unique supertype Event and which defines a unique slot named value.

The **make-event** primitive creates an event instance, represented internally as a closure, that is, a function and an attached environment.

For example, we can declare a new event of type Data like this:

```
(define ev (make-event 'Data))
==> ev: #[closure arglist=(message . args)]
```

After having instantiated an event of a given type, we can interact with it using a set of methods falling in two categories:

- *Accessors* to get the values of the different slots in the event, prefixed by get-.
- *Mutators* to change the slot values. The prefix indicated in this case is set-. We also postfix them with an exclamation point[11].

[10] We see Scheme and its reflective capabilities as an excellent environment for supporting domain-specific languages.

[11] It is usual to use the ! symbol to suffix imperative constructs in Scheme.

We know that the event type Data defines the slot value. To set a new value for this slot and then read it, we can use the corresponding mutators and accessors like this:

```
(ev 'set-value! 10)
(ev 'get-value)
==> 10
```

A restricted form of *introspection* is also provided by some management methods:

```
(ev 'get-type)
==> Data
(ev 'get-supertypes)
==> (Event)
(ev 'get-slots)
==> (value source)
```

The slot source is defined by the root type Event. It identifies the last sender of the event[12].

5.2 Component management

Similarly to events, we distinguish in Scope the descriptions of the components we will use to create the architectures and their instances. In our terminology, a *component* is an instance and a *component definition* is its description.

Compared to our formal model which is mainly focused on the type system, the components we implement provide a lot of features besides their types. Practically, a component[13] consists of:

- A unique identifier,
- the definition name of the component,
- its input and output types,
- a corresponding set of functionalities,
- a set of properties,
- an initializer, and
- a set of local functions

There is at least one functionality for a given input type. It explains how to react to the reception of a compatible event. The component properties describe the internal state of the component while the initializer allows this state to be initialized at runtime. It is interesting to note that in our approach, the instantiation and initialization phases are clearly distinguished, allowing co-instantiation as in [10]. Finally, the local functions define some implementations that the developers would like to attach to a given component while not being functionalities on their own.

5.2.1 *Component definitions*

Our first example will be the definition of the addition operator depicted in Figure 2:

```
(component AddReal
  (receive Real)
  (send Real)
```

[12] A value of null indicates that the event was sent from the top level environment.

[13] For the moment, we are only describing flat components, composites are detailed in section 5.4.

```
  (property readonly op1)

  (when Real (in-event)
    (if (eq? op1 'null)
        (set! op1 (in-event 'get-value))
        (let ((out-event (make-event 'Real)))
            (out-event 'set-value! (+ op1 (in-event 'get-value)))
            (set! op1 'null)
            (send out-event)))))
```

The definition name of this component is AddReal. Using the **receive** and **send** constructs, we declare that the component receives and sends Real events. The property op1 is not initialized, so its default value will be null. The **readonly** modifier indicates that the value of the property is readable from the outside of the component, but not writable. Then, the functionality corresponding to the input type Real is defined in a **when** clause. The body of our functionality is a simple algorithm that expects two successive numbers and then outputs their sum.

It is very important to note that, as expected, the **send** primitive which realizes the effective emission of event does not reference the destination component.

Constant generator is another interesting category of component in the case of dataflow systems. The purpose of such component is to generate a constant value when it is activated. This allows a dataflow to be fed with preprogrammed values instead of user supplied data.

The real number generator is defined like this:

```
(component GenReal
  (receive Activate)
  (send Real Activate)

  (property readonly constant)

  (initialize (iconstant)
    (set! constant iconstant))

  (when Activate (in-event)
    (let ((out-event (make-event 'Real)))
        (out-event 'set-value! constant)
        (send out-event)
        (send in-event))))
```

When a component of type GenReal receives an Activate event, then it sends the constant it contains (as a property) and propagates the activation pulse. The initialize clause defines the component's initializer. When such an initializer is provided, the component has to be initialized before being used.

5.2.2 Component instantiation

Components (or component instances) can be created from a definition name using the **make-component** primitive. For example we can create a number generator in the following manner:

```
(define generator (make-component 'GenReal))
==> generator: #[closure arglist=(message.args)]
```

To be able to communicate with each other using events, two given components must be first connected. We introduce the connection primitive in the next section. However, components also provide some management methods. These functionalities are generally used from the toplevel environment to configure or introspect the components. Let's demonstrate the use of the most common management methods:

```
(generator 'get-def-name)
==> GenReal
(generator 'get-input-types)
==> (Activate)
(generator 'get-output-types)
==> (Real Activate)
```

A method `initialize` is also created to invoke the component initializer if it is defined. Let's give a default value for the constant generator:

```
(generator 'initialize 3.14159265)
```

A specific accessor is associated to each **readonly** property, so we can read the constant value of the generator:

```
(generator 'get-constant)
==> 3.14159265
```

If the property is declared as **public**, then a mutator is also provided.

5.3 Connection primitives

In order to demonstrate the use of the different connection primitives, we will build a simple dataflow example whose purpose is to convert temperatures from Fahrenheit to Celsius degrees.

We can start by defining a simple temperature type:

```
(event Temperature
  (super Real)
  (slot unit))
```

A temperature will be a temperature value (provided by the `Number` type) and a unit symbol: `C` or `F`. The formula to convert a $temp_F$ temperature in Fahrenheit degrees to a $temp_C$ temperature in Celsius degrees is given below :

$$temp_C = (temp_F + -32) \times \frac{5}{9}$$

In order to build our dataflow, we will need an AddReal and a MulReal[14] component as operators as well as two GenReal constant generators. The following script creates and initializes these components:

```
(define add (make-component 'AddReal))
(define mul (make-component 'MulReal))
(define const1 (make-component 'GenReal))
(const1 'initialize -32)
(define const2 (make-component 'GenReal))
(const2 'initialize (/ 5.0 9.0))
```

Now, we can apply the **connect** primitive to connect the const1 component to add and const2 to mul. Since our components manage, at a conceptual level, numbers, we will declare the intentional type Number for the connections:

```
(connect const1 add 'Number)
==> (Real)
(connect const2 mul 'Number)
==> (Real)
```

[14]The definition of MulReal is almost identical to the one of AddReal.

We can remark that there is no ambiguity when connecting a real generator to a real operator, so we could have used the **connect-infer** primitive:

```
(connect-infer const1 add) ==> (Real)
(connect-infer const2 mul) ==> (Real)
```

We also have to connect our add operator to the mul component as its second input with (**connect** add mul 'Number).

The figure 8 shows our partial architecture of the temperature conversion dataflow. This architecture is only partial because we do not know yet how to provide the missing information: the temperature to convert. We also have to use the result of the conversion, for example by printing it on the screen. Finally, the activation of the whole dataflow should be resolved.

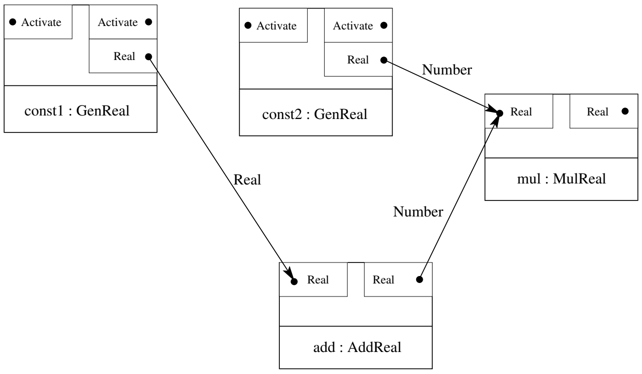

Figure 8: A partial architecture of a temperature conversion dataflow

First, we will define a very generic component whose purpose is to print any kind of Data[15]:

```
(component DataOutput
  (receive Data)

  (when Data (event)
    (print (event 'get-value)))

  (function public print (arg)
    (display arg))

  (function public newline ()
    (newline)))
```

Let's create a `DataOutput` instance:

```
(define out (make-component 'DataOutput))
(out 'print "Hello world")
==> Hello world
```

We can see in this example that every **public** local function (such as print) is associated to a method of the same name.

What we intend to do next is to connect this component as destination from the result of the conversion, emitted by the mul component. We recognize here a typical *generalization at destination* configuration where the specialized events (real numbers) must be communicated to a generic component, accepting all kind of data. However, the category of connection, as we explained in section *3.5.1*, can be omitted :

[15]As long as the value slot is printable.

```
(connect mul out 'Number)
==> (Real)
```

Finally, we will define a generic input component, both responsible of data input and activation:

```
(component DataInput
  (receive Activate)
  (send Data Activate)

  (function public activate ()
    ...)

  (when Activate (event)
    ...))
```

When receiving an Activate event or when the public function activate is invoked, the component will wait for the user to enter a value using the keyboard. Then, depending on the type of this input, a more or less specialized event will be generated.

The distinctive trait of the DataInput component definition is that it is typical of *type dispatcher* components. It generates a generic event type (Data) and is meant to be connected for subtypes. This is a common *generalization at source* configuration. Let us connect this component to the converter:

```
(define in (make-component 'DataInput))
(connect in add 'Number)
==> (Number)
```

Finally, our architecture must be completed by setting the activation links as depicted in figure 9. We can then start a conversion by activating the `in` component:

```
(in 'activate)
? 68
20
```

While our example is very basic, any dataflow architecture with more complex operators might be created using the same principles. It is also important to notice that we created the converter *dynamically* and *incrementally*. At each step of the structuration, the type system ensures that the connections will lead to correct communications of data.

5.4 Composite components

In the previous section, we built a temperature converter as an architecture, composed of a set of interconnected components. If we only consider the logical part of this composition, we see an architecture which receives real numbers (input temperatures) and sends real numbers (output temperatures). The composite component framework, as proposed in Scope, will help us to reuse this architecture to create a stand-alone component.

The **composite** declarative statement allows the creation of a new composite definition. All the constructs available in flat component definitions can be used to describe a composite. The embedded architecture creation and management is handled by a set of extra constructs.

In order to build an actual converter component, we can start by simply stating that this component is exactly the architecture we built in the previous section.

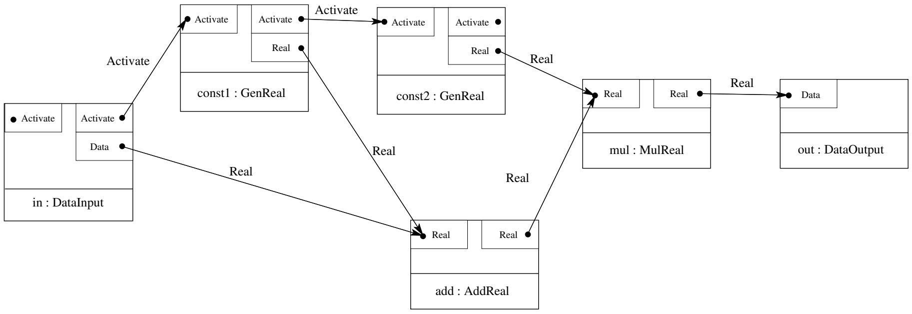

Figure 9: Temperature conversion dataflow: complete architecture

We can express this declaratively in Scope:

```
(composite TempConverter
  (inner GenReal const1)
  (inner GenReal const2)
  (inner AddReal add)
  (inner MulReal mul)

  (connection const1 add Number)
  (connection const2 mul Number)
  (connection add mul Number)
  (connection const1 const2 Activate))
```

The **inner** keyword introduces an inner component, designed by its definition name[16] and an identifier. For example, const2 is an inner component as defined in GenReal. We can also declare the way the inner components are interconnected in the composite.

The resulting input types of the composite component are Activate (unsatisfied in const1) and Real (unsatisfied in add). The unsatisfied output types are also Activate and Real. Before using the composite, the constant generators must be initialized. We can do this by adding an initializer to the composite itself:

```
(composite TempConverter
  ...
  (initialize ()
    (const1 'initialize -32)
    (const2 'initialize (/ 5.0 9.0)))
  ...)
```

As we can see, the inner components are directly accessible through their identifiers from the composite code. Finally, we can create an instance of TempConverter and then connect it to a couple of data input and output components:

```
(define convert (make-component 'TempConverter))
(connect-infer in convert) ==> (Real Activate)
(connect convert out 'Number) ==> (Real)

(convert 'initialize)
(in 'activate)
? 68
20
```

From the outside world, the resulting architecture looks like figure 10. The details concerning the constitution of the TempConverter component is of no importance at this level.

[16]This inner component can be of course a flat or composite component.

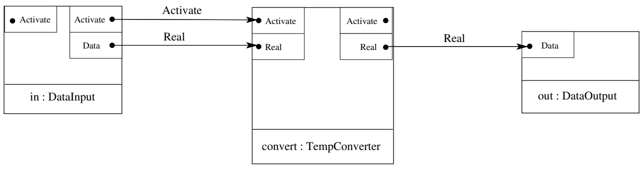

Figure 10: Temperature conversion dataflow updated with a composite

5.5 Type confinement

We introduced the type Temperature in the previous section but we did not use it in our dataflow. The problem is that the operators are "too" generic, they work on real numbers. One way to specialize the converter is to use the type confinement constructs: **confined** and **filtered**. The first one hides an input type from the outside world while the second one hides an output type.

To explain that our temperature converter composite can only manage Temperature events, we simply have to confine and filter the Real types and add the code to handle the temperatures. This implies the following modifications to the composite definition:

```
(composite TempConverter
  ...
  (receive Temperature)
  (send Temperature)
  (confined Real)
  (filtered Real)

  (when Temperature (event)
    (send-explicit add event))

  (filter Real (in-event)
    (let ((out-event (make-event 'Temperature)))
        (out-event 'set-value! (in-event 'get-value))
        (out-event 'set-unit! 'C)
        (send out-event)))
  ...)
```

The **when** clause, as usual, explains what to do when a Temperature event is received by the composite. Here, the event is forwarded to the inner architecture through the **send-explicit** primitive. This event will be accepted as a Real since we defined Temperature as one of its subtypes.

When an inner component generates a Real event, it is intercepted at the composite level by the **filter** clause. The filter here, when invoked, creates a Temperature and then sends it as usual. Finally, we obtain the component depicted in figure 11, that is, a real temperature converter when seen from the outside world.

While *transtyping* is one possible use of the type confinement mechanism, other purposes can be envisaged such as true confinement, conflicting input types resolution[17] and so on.

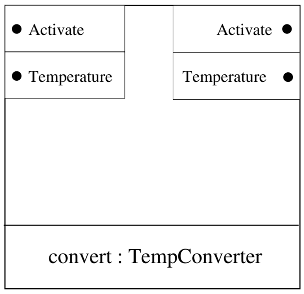

Figure 11: Confined and filtered composite

5.6 Labelled connections

Sometimes, a conceptual description of a component is not enough. Suppose for example that we want to implement a subtraction and a division component. The main problem here is that these operators are not commutative, like additions or multiplications. We need a way to differentiate the left part from the right part of these operations.

To that effect, the Scope language allows connections to be *labelled*. A label is a symbol declared in a **when** clause, as in the following (partial) definition:

```
(component SubReal
  (receive Real)
  (send Real)

  (property readonly op1)

  (when Real left (event)
    ...)

  (when Real right (event)
    ... ))
```

In order to connect a component (for example, a DataInput) to a SubReal, the chosen label must be indicated:

```
(connect in sub 'Number 'left)
==> (Real)
```

We see such *labelled connections* as being a clean way to model physical pins or ports when needed while keeping the more logical type-based descriptions of components. We also investigate control-based *connection guards* but this is beyond the scope of this paper.

5.7 Component refinement

We think that the component model we implement in Scope is powerful enough to help in many situations to reduce *source code redundancy* and foster *reuse*. For example, the dataflow components are very generic and so can be reused in many different contexts. However, when we look more deeply at the level of the implementation, we can see that the code redundancy situation is not optimal. For example, the only difference between the addition and multiplication operators on numbers is the low level operator involved. Using the *refinement framework* proposed in the Scope language, we can define these components more cleanly.

First, let's define an abstract operator component:

```
(component OpReal
  (receive Real)
  (send Real)

  (property readonly op1)

  (function-abstract private operator (o1 o1))

  (when Real (event)
    (if (eq? op1 'null)
      (set! op1 (event 'get-value))
      (let ((sev (make-event 'Real)))
        (sev 'set-value! (operator op1 (event 'get-value)))
        (set! op1 'null)
        (send sev)))))
```

Note the variant **function-abstract** which allows the declaration of an undefined function. Now, we can refine this definition to describe the addition operator:

```
(component AddReal
  (refine OpReal)
  (refine-function private operator (o1 o1))
    (+ o1 o2))
```

The definition of AddReal is exactly the definition of OpReal with the refinement of the local function operator. In the same manner, we can define a multiplication operator:

```
(component MulReal
  (refine OpReal)
  (refine-function private operator (o1 o1))
    (* o1 o2))
```

While it resemble *inheritance* of object-oriented languages, it is very important to note that the refinement framework introduced here has nothing to do with the type system: refinement is not subtyping in Scope.

The refinement framework is only an implementation tool. For example, while multiple-inheritance semantics is generally considered as a major and unresolved issue in modern object-oriented languages, the refinement of multiple Scope component definitions is allowed and properly defined. When a conflict occurs between multiple refined definitions, the refinement process simply fails which means that sharing source code in this case is not possible.

6. RELATED WORK

Our whole approach is primarily inspired by the *software architecture* field of research. Mary Shaw and David Garlan identify in [2] *"an architecture of a specific system as a collection of computational components - or simply components - together with a description of the interactions between the components - the connectors"*. This implies a very generic and conceptual nature for the concept of component. While our component model follows these principles, the Scope connections are largely different from connectors as found

[17] A conflict may occur when a received event can be dispatched to more than one inner component.

in Architecture Definition Languages such as [12]. In our approach, connections have a very fine-grained and operational nature. While complex interactions generally involve complex connectors in ADLs, they will imply complex connection compositions in Scope. This difference was mainly motivated by the dynamic nature of the software architectures we are working on.

Typed composition models, comparable to our approach, have been introduced in previous research work. The Rapide framework [6] also introduces composable typed components for the prototyping of concurrent architectures. As in our work, Rapide components are communicating to each other using typed events, showing both the usefulness and generality of this approach. However, Rapide does not introduce composite components or dynamic composition.

To our knowledge, few research work propose composite components models and recursive composition. Perhaps the most resembling approach is Fabrik [4] which proposes a composition model similar to ours. In both approaches, architectures can be reused as regular components. Dynamic composition is also allowed in Fabrik but the type system does not allow subtyping. Moreover, we do not address specifically the domain of visual programming, hence our linguistic point of view.

7. CONCLUSION

We proposed in this paper a cleanly formalized composition model, supported by a language-level implementation. While our main focus is the (flat and recursive) composition of dynamic software architectures, we think the Scope model is also useful at a more abstract and conceptual level, when designing and prototyping component-based systems.

As a matter of fact, the recursive composition model is in our opinion a real advance, providing at the same time a powerful modelling and abstraction tool, supported by an efficient implementation where flat and composite components are not distinguished.

Our dataflow examples, while interesting, were mainly employed for pedagogical purposes. We are working on more concrete examples which foster the incremental design model encouraged in Scope. For example, we investigate the design of workflow management systems which would allow different runtime reconfigurations : adjunction of new business processes or process components and so on. We also work on Multi-Agent Systems where agent or group adjunction/removal are canonical functionalities. Moreover, we investigated in [3] the modelling of agents as composites. This way, we may also modify the behaviour of specific agents.

The Scope language is based on a Lisp dialect called Scheme. While not being the most used language in the industry[18], Scheme is in our opinion a powerful glue language as well as an invaluable environment for linguistic experiments, education and even more generally for all-purpose software prototyping. In our case, we were for example able to cleanly separate the structural and behavioural issues and in fact remain focused on the first ones while delegating the second to the underlying Scheme semantics.

Together with the flat and recursive composition models, the added properties (such as local functions, properties and initializers) and the language extensions, we think the Scope language foster reuse and reduce source code redundancy, perhaps in a more effective way than regular object-oriented languages.

However, our approach can be considered as complementary to object-oriented environments. To demonstrate this, we propose in [9] an alternative implementation, smoothly integrated in the Java environment. This approach shows that regular classes can be seen as a good implementation tool for flat components. Composite components, however, are more difficult to embed in class-based languages (due to their non-recursive nature).

[18]However, Scheme has had some impact in the domain of structured documentation, as the underlying language of DSSSL.

8. REFERENCES

[1] *International Symposium on Principles of Software Evolution (ISPSE 2000), Kanazawa, Japan*. IEEE computer society, November 2000.

[2] D. Garlan and M. Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing Company, 1993.

[3] A. Guillemet, G. Haik, T. Meurisse, J.-P. Briot, and M. Lhuillier. Mise en œuvre d'une approche componentielle pour la conception d'agents. In *Ingénierie des systèmes multi-agents, JFIADSMA'99*. Editions Hermès, 1999.

[4] D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, and K. Doyle. Fabrik, a visual programming environment. In *Proceedings of OOPSLA'88*. ACM Press, November 1988.

[5] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language scheme. Technical report, February 1998.

[6] D. C. Luckham. Rapide : A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. *DIMACS Partial Order Methods Workshop*, IV, July 1996.

[7] B. Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

[8] Object Management Group. *Corba CCM specification (draft)*. `http://www.omg.org`, 1999.

[9] F. Peschanski. Comet : A component-based reflective architecture for concurrent and distributed programming. In *OOPSLA'99 Workshop on Reflection and Software Engineering*, 1999.

[10] C. Queinnec. Designing meroon v3. In J. Kopp, H. Hohl, and H. Bretthauer, editors, *Proceedings of the ECOOP'93 Workshop on Object-Oriented Programming in Lisp: Languages and Applications*, September 1993.

[11] Sun Microsystems. *JavaBeans 1.01 specification*. `http://java.sun.com/beans`, 1998.

[12] G. Zelesnik. The unicon language reference manual. Technical report, Carnegie Mellon University, 1996.