

# Forms - A Flexible Notion for Software Composition\*

MARKUS LUMPE

Iowa State University

Department of Computer Science

113 Atanasoff Hall, Ames, IA 50011-1041, USA

lumpe@cs.iastate.edu

Tel: +1 515 294 2410, Fax: +1 515 294 0258

JEAN-GUY SCHNEIDER

Swinburne University of Technology

School of Information Technology

P.O. Box 218, Hawthorn, Victoria 3122, Australia

schneidr@it.swin.edu.au

Tel: +61 (0)3 9214 8189, Fax: +61 (0)3 9819 0823

## Abstract

The development of flexible and reusable programming abstractions has suffered from the inherent problem that reusability and extensibility are limited due to position-dependent parameters. To tackle this problem, we have been working on the definition of a general-purpose composition language based on a variant of the  $\pi$ -calculus as formal semantics, in which agents communicate by passing immutable extensible records, called *forms*, rather than tuples. Using this approach, we are able to define compositional abstractions in a more natural and robust way. In this position paper, we will extend the notion of forms and illustrate that forms may serve as a unifying concept for component-based software development.

## 1 Introduction

Now, more than ever, it is not enough for applications to fulfill only their functional requirements, but modern applications must be *flexible*, or “open” in a variety of ways in order to cope with the advances in computer hardware technology, rapidly changing requirements, and maintenance issues. The kinds of flexibility required by open systems are presently best supported by component-based software technology: components, by means of abstraction, support portability, interoperability, and maintainability. Extensibility and configurability are supported by different forms of binding technology: application parts, or even whole applications, can be created by wiring together software components. This ensures that applications stay flexible by allowing components to be replaced or reconfigured, possibly at run-time.

However, component-based application development substantially benefits not only from using components, but as important as components are the concepts of *architectures*, *scripts*, *co-*

---

\*In *Proceeding of the Third Australasian Workshop on Software and System Architectures (AWSA '00)*, John Grundy and Jun Han (Eds.), Sydney, Australia, November 2000, pp. 24–36.

*ordination*, and *glue* [27]. Hence, component-based application development must always be conducted in a conceptual framework which consciously applies the paradigm

$$\text{Applications} = \text{Components} + \text{Scripts.}$$

Object-oriented programming languages and design techniques go a long way in offering the required flexibility and extensibility, but current practice shows that the technology is often applied in a way that hinders the development of open systems [29]. Although object-oriented languages are well-suited for implementing software components, they fail to shine in the construction of component-based applications, largely because object-oriented design tends to obscure a component-based architecture [19]. Using a *composition language* that allows us to express applications as compositions in terms of components, scripts, and glue would be a major step in overcoming these problems.

However at present, there does not exist a general-purpose composition language that (i) supports application configuration through a structured, but nevertheless flexible wiring technology and (ii) enforces a clear separation between computational elements (i.e. components) and their relationships. In addition, most available systems mainly focus on special application domains and offer only rudimentary or no support for the integration of components that were built in a system other than the actual deployment environment. The reason for this situation is not only the lack of well-defined (or standardized) component interfaces, but also the ad-hoc way the semantics of the underlying language models are defined.

In order to solve the problems of present-day component technology, we argue that it is necessary to define a composition language based on an appropriate semantic foundation. In particular, if we can understand all aspects of software components and their composition in terms of a small set of primitives, then we have a better hope of being able to cleanly integrate all required features for software composition in one unifying concept.

We have been working on the definition of such a composition language, called PICCOLA, that provides means to support the paradigm of component-based development. Amongst others, this composition language supports the following features (refer to [20] for further details):

**Active objects:** Objects are computational entities that provide services based on an encapsulated state. Objects may be active (concurrent), distributed, mobile, and may live in different environments. In either case, an object can be viewed as a kind of server, or *process*.

**Components:** Components are black-box entities that encapsulate both provided and required services behind well-defined interfaces [19]. Components may be fine-grained, when used to build individual objects, or coarse-grained, when used to build compositions of objects.

**Connectors:** Connectors define how associated components interact with each other [28]. A composition language must support the specification of new kinds of connectors.

**Architectures:** A composition language must allow programmers to make the architecture of an application explicit in the corresponding source code. It must offer support for common architectural styles, for combining multiple, heterogeneous architectural descriptions, and be open enough to support new (user-defined) architectures [5].

**Glue:** A composition language must offer abstractions to overcome so-called *compositional mismatches* (e.g., to bridge the gap between different object and component models) [25]. Components that cannot be separated from their individual runtime environments must still be able to communicate. Glue must achieve the mappings between these different models.

**Reflection:** Glue abstractions are often realized by intercepting messages between objects, and performing some (reflective) transformations on these messages [6]. Reflection is also important for exercising run-time control on configurations [13]. Metaobjects are active objects that control the creation, instantiation, and composition of other objects, and can be used to realize various forms of reflective behaviour.

**High-level syntax:** The specification of an application as a composition of components must be highly readable and compact. The syntax should favour a declarative style of programming in order to reflect the architecture of an application.

We have been exploring two approaches to define the composition language PICCOLA: (i) an imperative language style [15, 26] defined in the tradition of the PICT programming language [23] and (ii) a language emphasizing a more functional and declarative style of programming [2]. By combining classical programming concepts with concepts especially defined for our approach of “Applications = Components + Scripts”, we hope to (i) discover the right higher-level abstractions and (ii) define a unified paradigm to facilitate component-based application development.

Common to both approaches mentioned above is the fact that all language features are defined by transformation to a core language - the  $\pi\mathcal{L}$ -calculus [15], an inherently polymorphic variant of the  $\pi$ -calculus [17], in which the communication of tuples is replaced by communication of *forms*, immutable extensible records. By this approach, we address the inherent problem that reusability and extensibility of abstractions are limited due to position depended parameters.

Besides forms, which have their analogues in many existing programming languages and systems (such as HTML, Visual Basic, and Python to name just a few), the  $\pi\mathcal{L}$ -calculus also incorporates *polymorphic form extension*, a concept that corresponds to asymmetric record concatenation [9], as a basic composition operation for forms. Both forms and polymorphic form extension are the key mechanisms for extensibility, flexibility, and robustness as (i) clients and servers are freed from fixed, positional tuple-based interfaces, (ii) abstractions are more naturally polymorphic as interfaces can be easily extended, and (iii) environmental arguments (such as communication policies or default I/O-services) can be passed implicitly.

However, the expressive power of the  $\pi\mathcal{L}$ -calculus is limited when modeling sophisticated higher-level compositional abstractions like mixins [8], encapsulation abstractions [30], alias free references [21], or coordination styles [1]. In fact, when modeling higher-level abstractions, the emphasis is on forms, and not on concurrency or distribution. Moreover, the underlying operators of the calculus have solely been used to define the behaviour of the abstractions and put an extra burden on their definition, since we have to explicitly cope with communication aspects.

In this paper, we present a first-order system of forms. More precisely, we extend the form operations of the  $\pi\mathcal{L}$ -calculus with (i) restriction facilities first proposed in [26] and (ii) *form nesting*, which has proven to be the most important structuring tool. In contrast to our previous work, we will study forms in isolation, i.e. we separate forms from the underlying process calculus. This will allow us to specify compositional abstractions in general and architectural styles in particular in a more convenient and direct way. The reader should note that the communication primitives of the underlying process calculus can be represented as designated higher-level abstractions [2].

This paper is organized as follows: in section 2, we present a first-order system of forms. In section 3, we show how first-order forms can be used to model basic compositional abstractions. In section 4, we analyze higher-level concepts that require a more sophisticated notions of forms. We conclude with a summary of the main observations and a discussion about future work in section 5.

## 2 First-order forms

Forms are finite mappings from an infinite set  $\mathcal{L}$  of labels to an infinite set  $\mathcal{V}^+ = \mathcal{V} \cup \{\mathcal{E}\}$ , the set of values extended by  $\mathcal{E}$  that denotes bottom, an undefined value. We use  $F, G, H$  to range over forms and  $l, m, n$  to range over  $\mathcal{L}$ . The syntax of forms is defined as follows:

$F ::= \langle \rangle$	<i>empty form</i>	$V ::= v$	<i>atomic value</i>
$F\langle l=V \rangle$	<i>binding extension</i>	$\mathcal{E}$	<i>bottom</i>
$F \cdot F$	<i>polymorphic extension</i>	$F$	<i>nested form</i>
$F \setminus F$	<i>polymorphic restriction</i>	$F_l$	<i>projection</i>

Every form is derived from an *empty form*  $\langle \rangle$ , which does not define any bindings. In fact, an empty form denotes an empty component interface (i.e. a component that does not provide or require any services).

*Binding extension* is used to extend a given form with exactly one binding and service, respectively. With this operation we can either add a fresh service or override an existing one.

With *polymorphic extension* we can add or override a set of services. In combination with projection, polymorphic extension corresponds to asymmetric record concatenation [9]. The concept of polymorphic extension is used to (i) combine two distinct sets of services (e.g., to model COM-aggregation [24]), (ii) combine two overlapping sets of services giving precedence to the rightmost specifications, (iii) specify default arguments (the default values given in the left form may be overridden by new values in the right form), or (iv) wrap an existing set of services by putting this set between two forms, *pre* and *post*, that define the corresponding pre- and post-operations, respectively.

*Polymorphic restriction* can be considered as the “inverse” operation of polymorphic extension. There is, however, as subtle difference: while polymorphic extension may add a new binding for a given label, say  $l$ , and preserving the original definition, polymorphic restriction physically removes all occurrences of a given label. Therefore, if we have two forms  $F$  and  $G$  with  $F \equiv \langle l=V_1 \rangle \langle m=V_2 \rangle \langle l=V_3 \rangle$  and  $G \equiv \langle n=V_4 \rangle \langle l=V_5 \rangle$ , the polymorphic restriction  $F \setminus G$  yields  $\langle m=V_2 \rangle$ .<sup>1</sup>

*Atomic values* and *bottom* denote simple form values. Atomic values may be both classical values like *Integers* or *Strings* or opaque values denoting abstract data types like Visual Basic/COM *VARIANT*'s.

*Nested forms* and *projections* are complex form values. Nested forms provide additional structure for forms. In general, nested forms are used to separate services and arguments. A typical application of nested forms is a wrapper that takes a service and its arguments as two separate parameters.

Projection selects for a given label  $l$  the corresponding value from a form  $F$ . The meaning of projection is twofold: first, projection selects a value from a given form, and second, projection

<sup>1</sup>In order to facilitate the specification of forms, a form  $\langle \rangle \langle l=V \rangle$  is just written as  $\langle l=V \rangle$ .

$$\begin{aligned}
\langle \rangle_l &= \mathcal{E} \\
(F\langle l=V \rangle)_l &= V \\
(F\langle m=V \rangle)_l &= F_l \quad \text{if } m \neq l \\
(F \cdot G)_l &= \begin{cases} F_l, & \text{if } G_l = \mathcal{E} \\ G_l, & \text{otherwise} \end{cases} \\
(F \setminus G)_l &= \begin{cases} F_l, & \text{if } G_l = \mathcal{E} \\ \mathcal{E} & \text{otherwise} \end{cases}
\end{aligned}$$

Table 1: Projection.

enables keyword-based position-independent parameter passing, which is a key concept for extensibility and robustness (refer to [12, 15, 26] for a detailed discussion). The definition of projection is given in Table 1. Note that if a form provides multiple bindings for label  $l$ , then the definition given in Table 1 ensures that a projection always yields the *rightmost binding*.

The extension of a form  $F$  with an empty binding for a label  $l$  (i.e.  $F' \equiv F\langle l=\mathcal{E} \rangle$ ) deserves special attention. From a different perspective,  $F'$  can be considered as a form where all bindings for label  $l$  have been removed. In fact, we cannot distinguish between a form, say  $G$ , which does not contain a binding for a given label  $l$ , from a form  $H$ , which has a binding  $\langle l=\mathcal{E} \rangle$  as its rightmost binding for label  $l$ . Since the extension of a form with an empty binding is frequently used, we shall use the notation  $F \setminus l$  for  $F\langle l=\mathcal{E} \rangle$  throughout the rest of this work.

The reader should note that the definition of polymorphic extension implies a different meaning of  $G \equiv F\langle l=\mathcal{E} \rangle$  and  $G' \equiv F \cdot \langle l=\mathcal{E} \rangle$ : if we apply the label  $l$  to  $G$ , the result is  $\mathcal{E}$  whereas the result for an application to  $G'$  is the projection  $F_l$ .

In order to define a complete theory for forms, it is necessary to address further aspects of forms such as equivalence of forms, normalization of form expressions etc. However, a detailed discussion of all these aspects is beyond the scope of this paper and has, therefore, been omitted here (refer to [16] for details).

### 3 First-order form applications

In this section, we will demonstrate how first-order forms facilitate the definition of different symbol import mechanisms. In general, most programming languages provide a fixed set of language features to resolve external symbols like the “import” statement of both Java and Python, the “#include” directive of C/C++, or the “uses” declaration of ObjectPascal.

Assume there exists a service `resolve` that implements symbol resolution. This service expects as argument a form that denotes a module name and yields a form containing bindings for every symbol defined in the specified module. Using this approach, it is possible to model different import schemes. For example, in order to model the “#include” directive of C/C++, we simply have to specify:

```
resolve(\langle module="stdio.h" \rangle)
```

Using polymorphic extension, a sequence of include directives can be specified as follows:

```
resolve(<module="stdio.h">).resolve(<module="stdlib.h">)
```

With this scheme, all imported symbols collapse into one single namespace. If we, however, want to maintain the symbols in different namespaces, then we simply invoke the resolve service within a binding extension (this scheme corresponds to *fully qualified names* approach found in Ada):

```
<stdio=resolve(<module="stdio.h">)><stdlib=resolve(<module="stdlib.h">)>
```

Selective import can be specified using projection. For example,

```
resolve(<module="System">)Writeln
```

denotes the selective import of the symbol "Writeln" from the System unit of ObjectPascal. Furthermore, it is possible to model restricted import using form restriction, e.g. import all symbols except symbols a, b, and c:

```
resolve(<module="X">)\a\b\c
```

or import all symbols from module "X" not defined in "Y"

```
resolve(<module="X">)\resolve(<module="Y">)
```

In order to employ these scheme in a real system, forms have to be used to model *dynamic contexts* for symbol lookup. This approach has been implemented in version 2 of PICCOLA [2].

## 4 Towards higher-order forms

A general purpose composition language like PICCOLA [2, 3] is designed to be a language for composing software components which may be written in a separate implementation language. Therefore, a composition language must provide both means to express (i) how components are configured (i.e. scripts) and (ii) connectors, coordination as well as glue abstractions needed to configure individual components.

The challenge is to find the right set of language elements that meet our requirements. We argue that we need an approach that allows us to represent concepts like namespaces, contexts, interfaces, scripts, and objects in an uniform fashion and have chosen the maxim

Everything is a form.

Such an unification leads to an extremely simple language, and allows us to abstract uniformly over all related language concepts. In the following, we will present some *higher-order forms* that will facilitate the specification of these concepts. The reader should note, however, that we will use a tentative syntax to define the new higher-order elements.

In order to provide better support for higher-level dynamic abstractions (such as *events*, *repositories*, and *blackboards*), higher-order forms need some notion of functional abstraction and contexts. A first approach to represent functions and contexts as forms is shown in [4]. Based on this approach, we define functions as follows:

```

Form-FunctionDeclaration ::= 'def' FunctionName FunctionArguments '=' Body
FunctionArguments      ::= '(' FormVariable ')' [ FunctionArguments ]
Body                   ::= [ Pattern ] Form [ Body ]
Pattern                 ::= FunctionName Form-Pattern ':'
Form-Pattern           ::= '(' Form ')' [ Form-Pattern ]

Form-FunctionCall      ::= FunctionName FunctionArguments
FunctionArguments     ::= '(' Form ')' [ FunctionArguments ]

```

Functions are defined in the tradition of functional programming languages (we use a Haskell-like syntax) and can have one or more arguments. The arguments are specified in a *curried* style, which promotes the separations of concerns (i.e. it enables the *grouping* of related aspects or features) and is a prerequisite for pattern matching. Within a function declaration one can use pattern matching to select a specific branch of the function. A branch is selected if the corresponding pattern matching succeeds.<sup>2</sup> If a branch could be matched, the corresponding code is evaluated and the function returns. Otherwise, the evaluation continues at the next statement. The reader should note, however, that the function body is also a form, which is defined as a sequence of subforms (i.e. extensions). The order of the branch patterns is therefore significant. Furthermore, it is possible to add additional code (e.g., bindings, function declarations, or function calls) between branches. As a consequence, the evaluation of a function may have side effects even though no branch could be matched at all.

For example, assume we have defined a service `eval` that evaluates a given form and an abstraction `newSlot` that implements a storage cell with two methods `read` and `write`, respectively. Then a future<sup>3</sup> can be specified as follows:

```

def Future (Service) (ServiceArguments) =
  slot = newSlot ()
  eval (slot.write (Service (ServiceArguments)))
  slot.read ()

```

When called, the future returns a *ticket* for the result and evaluates the specified service concurrently (i.e. the result may not be available at the time the ticket is returned to the callee).

First-order forms provide only limited support for dynamic abstractions. There is no operation available either to enumerate the labels of a form or to discover a new set of labels. Such a feature, for example, will allow us to query the provided services of a component, to analyze the methods of a COM-component that implements the `IDispatch` interface, or to apply a generic wrapper to all services of a component without knowing the actual number of services in advance.

In order to address this issue, higher-order forms need to provide means for *first-class labels* and *set of labels*, respectively. We use the notation  $@F$  to denote the set of labels of a form  $F$ .<sup>4</sup> To extract a label from the set, we use a list notion  $l : s$  found in functional programming languages, where the label  $l$  denotes the head and  $s$  denotes the tail of the list, respectively. The reader should note that lists are not really forms, but there exists an encoding of lists as forms [26]. Form iteration (i.e. enumeration on all elements of a form) can then be defined using the function `map`:

<sup>2</sup>Rule selection can be implemented using *input-guard choice* [18].

<sup>3</sup>Futures are proxies that mimic “wait-by-necessity” [14].

<sup>4</sup>All labels in the set  $@F$  are distinct.

```

def map (Form) (Abstraction) =
  def apply (List) (Form) (Abstraction) =
    apply ([]) (F) (A) : ⟨⟩
    apply (l:s) (F) (A) : ⟨l=A (F1)⟩ (apply (s) (F) (A))
  apply (@Form) (Form) (Abstraction)

```

The function `map` is implemented using the locally declared function `apply`, which uses both pattern matching and first class labels. Using functions and pattern matching, we get a quite compact specification for `map`. These concepts also provide means to view forms more algebraically and will allow us, for example, to define a *form data model*.

Besides the notion of the set of labels, we also need some operators to test whether two labels are equal or not. Predicates will not only be used to check properties of labels, but also to check for or to guarantee certain conditions of forms. Properties are defined as follows:

$$\textit{Property} ::= \textit{' ( Form | Label ) '-> ' Condition '}$$

In a property declaration, a test (i.e. a condition specification) is applied to either a form or a label. If the test succeeds, the predicate yields *true*, otherwise *false*. Form predicates have been proposed in [26] to test a form for a given label.

One of the main benefits of a composition language is that it allows programmers to make the architecture of an application explicit in the corresponding source code. Therefore, it is essential that a form-based language like PICCOLA enables the explicit representation of architectural styles. In the following, we will address this aspect by illustrating the implementation of a component framework for stream and filter composition similar to the Bourne Shell [7]. We will also show that a form-based approach enhances the extensibility and robustness of the corresponding connector implementations. Due to lack of space, we will only discuss the main aspects of the framework (refer to [26] for details).

The component framework consist of three kinds of components: (i) data sources (producing data elements on their output-port denoted by `Out`), (ii) filters (accepting data elements on their input-port, denoted by `In`, processing the data, and producing results on their output- and/or error-port, denoted as `Out` and `Err`, respectively), and (iii) sinks (representing the end of data processing, having an input-port denoted by `In`). All three kinds of components can be distinguished by the types of services (i.e. the ports) they provide and require, respectively. From a component point of view, an output- or error-port can be considered as a provided service whereas an input-port is a required service, respectively.

The connectors of the framework are defined by the set  $\{<, |, >, | \&, > \&\}$ . Each of the connectors connects a set of free output-ports with a free input-port. The connector `|`, for example, connects the port `Out` of a component (either a source or a filter), say  $C_1$ , with to the port `In` of another component, say  $C_2$ . If both  $C_1$  and  $C_2$  have a free port `Err`, then the output of both ports are merged if either of the two ports will be connected in a further composition (i.e. the port `Err` of the composite component  $C_{1|2} = C_1|C_2$  acts as a gateway to both `Err` of  $C_1$  and  $C_2$ , respectively). The other connectors of the framework are defined similarly as the corresponding counterparts of the Bourne Shell.

The composition rules for the connector `|` is given in Table 2. The presence of a character `I`, `E`, or `O` indicates that the corresponding I/O-port of a component is free (e.g., `IOE` denotes a component where all the I/O-ports are free whereas `O` denotes a component where only the

IOE   IOE	→	IOE	⇒	XOE   IYE	→	XYE
OE   IOE	→	OE				
IOE   IE	→	IE				
OE   IE	→	E				
<i>(binding of O and I, merge of error)</i>						
IOE   IO	→	IOE	⇒	XO   IY	→	XY
OE   IO	→	OE				
IOE   I	→	IE				
OE   I	→	E				
IO   IOE	→	IOE				
O   IOE	→	OE				
IO   IE	→	IE				
O   IE	→	E				
IO   IO	→	IO				
O   IO	→	O				
IO   I	→	I				
O   I	→	∅				
<i>(binding of O and I, no error merge)</i>						

Table 2: Composition rules for connector ‘|’.

port Out is free). The other connectors of the framework have similar composition rules and, therefore, have been omitted in Table 2 (refer to [26] for details).

All connectors of the framework are defined in a way that applying a connector to two components leads to a composite component which may be used for further composition, unless all its I/O-ports are connected. Therefore, the component ‘∅’ cannot be used for further composition. This is in contrast to Bourne Shell scripts where ‘cat infile | sort >& outfile’ cannot interact with any other Unix filter (all I/O-streams are connected), but can be used as a component in another pipe and filter chain.

The implementation of the connector ‘|’ uses an approach based on predicates, polymorphic extension, and restriction. As indicated in the code given below, we simply connect the output-port of the left-hand side component, denoted by `Left`, with the input-port of the right-hand side component (using the abstraction `connect`) and restrict access to the corresponding ports `In` and `Out`, respectively. Then we test whether both arguments have an unbound port `Err`. If this is the case, the two error-ports are merged (using the abstraction `merge`), and the resulting composite component is expressed as a polymorphic extension of the forms representing the interfaces of both components and the result of `merge`. Note that it is not necessary to restrict access to `Err` of both arguments as the binding for `Err` is overridden by the merging of the error-ports.

```
def _| (Left) (Right) =
  connect (<Out=Left.Out, In=Right.In>)
  if ([ Left -> Err ] && [ Right -> Err ])
    (Left\Out) · (Right\In) · merge(Left)(Right) # then branch
    (Left\Out) · (Right\In) # else branch
```

The two underscores in the abstraction above are used for declaring infix operators and the expression `a | b` is just syntactic sugar for `_| (a) (b)`. Note that the abstraction `if` implements the standard *if-then-else* statement and `&&` the boolean disjunction operator.

The implementation of the stream and filter framework illustrates how composite components

are expressed using polymorphic form extension and restriction whereas the required run-time information is retrieved by using the higher-order form abstractions previously defined in this section. The resulting composition operators are much more generic and robust in comparison to “classical” approaches as they benefit from the extensibility of polymorphic form extension. As an example, consider the case where we extend the framework with (i) a new sort of filter component which provides an additional input-port (denoted by  $\text{In}'$ ) and (ii) a corresponding set of new connectors. The existing connectors of the framework are not affected by this extension (they do not depend on the presence or absence of a free port  $\text{In}'$ ) and can be reused as is.

The framework is defined in a way that it applies an *algebraic view* of composition: a composition of components can be viewed as a term of a *many-sorted algebra*, where the components are the operands and the connectors the corresponding operators, respectively. Like in any many-sorted algebra, operators cannot be applied to any operand, and terms always have a well-defined structure. In addition, an application of an operator to a set of operands (i.e. a composition of components using a connector) is, by definition, again an operand of the algebra. By using an appropriate implementation technique and syntax for the framework, this approach helps an application programmer in making the underlying architecture of an application explicit. The reader should note that an algebraic approach cannot only be applied to stream-based frameworks, but also used for more sophisticated architectural styles [1].

## 5 Conclusions and future work

We have presented a basic framework and some higher-order notions of forms, a special notion of immutable extensible records. Forms are the key concepts for extensibility, flexibility, and robustness in component-based application development. Furthermore, forms enable the definition of a canonical set compositional abstractions in an uniform framework.

In section 4, we have introduced the concept of labels as first class values. However, first-class labels make polymorphic form extension obsolete as it could be replaced by an abstraction (based on binding extension and label introspection) with the same behaviour. This observation suggests a simplification of our theory of forms, motivating the definition of a *form algebra* containing orthogonal concepts only (i.e. none of the concepts can be expressed in terms of the others).

In our view, components are inherently concurrent and, therefore, it is essential to provide support for coordination abstractions [22]. In order to use forms for Linda-like data-driven coordination mechanisms [11], we will need support for sophisticated *form matching*. With functions we already have the basic mechanisms available to define form matching, but how to efficiently implement matching and how to define *prototype forms* that will be used as patterns in Linda primitives such as  $\text{in}(\tau)$  or  $\text{rd}(\tau)$  is still an open question.

Another area of future work will include the definition of a typing scheme for forms, as types impose constraints which help to enforce the correctness of a program [10]. This is of particular importance in the context of composition as we would like to detect compositional mismatches based on mismatched types as early as possible. Initial work in that area has been done in the context of the  $\pi\mathcal{L}$ -calculus [15]. This typing scheme lacks, however, recursion and parametric polymorphism. Furthermore, we will need more support to represent non-functional properties of components. Current type systems usually do not address this issue.

Our long-term goal is the development of a robust formal model to specify applications as compositions of distributed components. A *form calculus* will provide higher-level forms, communication facilities, and agents as core abstractions. A composition language based on this calculus

will then provide means to define flexible composition scripts and user configurable architectural styles.

In this work, we have mainly focussed on technological issues, but there are just as many, and arguable equally important, *methodological issues*: component software focuses on software solutions, not problems, so how can we drive analysis and design so that we will arrive at flexible component-based frameworks and applications? Given a problem domain and a body of experience from several applications, how do we re-engineer existing software into a component framework? As we develop a component framework, how do we select a suitable architectural style to support black-box composition? Finally, what kind of higher-level syntax is suitable in order to reflect the underlying architectural style of an application?

### Acknowledgements

We thank all members of the Software Composition Group for their support of this work, especially Oscar Nierstrasz for helpful comments on an earlier draft. This work has been partially supported by the Swiss National Science Foundation under Project No. 20-53711.98, “A framework approach to composing heterogeneous applications”, and the Swiss Federal Office for Education and Science under Project BBW Nr 96.00335-1, within the Esprit Working Group 24512: “COORDINA: Coordination Models and Languages”.

### References

- [1] Franz Achermann, Stefan Kneubühl, and Oscar Nierstrasz. Scripting Coordination Styles. In António Porto and Gruia-Catalin Roman, editors, *Coordination Languages and Models*, LNCS 1906. Springer, September 2000.
- [2] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola – a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing: An Object-Oriented Approach*, chapter 18. Cambridge University Press, 2000. to appear.
- [3] Franz Achermann and Oscar Nierstrasz. Application = Components + Scripts – A tour of Piccola. In Mehment Aksit, editor, *Software Architectures and Component Technology*. Kluwer Academic Press, 2000. to appear.
- [4] Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages*, LNCS 1897, pages 77–89. Springer, September 2000.
- [5] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.
- [6] Lodewijk Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, NL, June 1994.
- [7] S.R. Bourne. An Introduction to the UNIX Shell. *Bell System Technical Journal*, 57(6):1971–1990, July 1978.

- [8] Gilad Bracha and William Cook. Mixin-based Inheritance. In Norman Meyrowitz, editor, *Proceedings OOPSLA/ECOOP '90*, volume 25 of *ACM SIGPLAN Notices*, pages 303–311, October 1990.
- [9] Luca Cardelli and John C. Mitchell. Operations on Records. In Carl Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*. MIT Press, 1994.
- [10] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
- [11] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, April 1989.
- [12] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. PhD thesis, Centre Universitaire d’Informatique, University of Geneva, CH, 1994.
- [13] Stéphane Ducasse, Mireille Blay-Fornarino, and Anne-Marie Pinna-Dery. A Reflective Model for First Class Dependencies. In *Proceedings OOPSLA '95*, volume 30 of *ACM SIGPLAN Notices*, pages 265–280, October 1995.
- [14] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. The Java Series. Addison-Wesley, October 1996.
- [15] Markus Lumpe. *A  $\pi$ -Calculus Based Approach to Software Composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.
- [16] Markus Lumpe and Jean-Guy Schneider. A Theory of Forms. Draft manuscript, University of Bern, Institute of Computer Science and Applied Mathematics, August 2000.
- [17] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, Part I/II. *Information and Computation*, 100:1–77, 1992.
- [18] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. Technical Report 392, Computer Laboratory, University of Cambridge, UK, January 1996.
- [19] Oscar Nierstrasz and Laurent Dami. Component-Oriented Software Technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.
- [20] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a Composition Language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer, 1995.
- [21] James Noble, John Potter, and Jan Vitek. Flexible alias protection. In Eric Jul, editor, *Proceedings ECOOP '98*, LCNS 1445, pages 158–185, Brussels, Belgium, July 1998. Springer.
- [22] George A. Papadopoulos and Farhad Arbab. Coordination Models and Languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*. Academic Press, August 1998.

- [23] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language based on the Pi-Calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, March 1997.
- [24] Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.
- [25] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.
- [26] Jean-Guy Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.
- [27] Jean-Guy Schneider and Oscar Nierstrasz. Components, Scripts and Glue. In Leonor Barroca, Jon Hall, and Patrick Hall, editors, *Software Architectures – Advances and Applications*, chapter 2, pages 13–25. Springer, 1999.
- [28] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.
- [29] Jon Udell. Componentware. *Byte*, 19(5):46–56, May 1994.
- [30] Marc Van Limberghen and Tom Mens. Encapsulation and Composition as Orthogonal Operators on Mixins: A Solution to Multiple Inheritance Problems. *Object-Oriented Systems*, 3(1):1–30, March 1996.