Explicit Namespaces

Franz Achermann, Oscar Nierstrasz

Software Composition Group, University of Berne[1]

Abstract. A namespace is a mapping from labels to values. Most programming languages support different forms of namespaces, such as records, dictionaries, objects, environments, packages and even keyword-based parameters. Typically only a few of these notions are first-class, leading to arbitrary restrictions and limited abstraction power in the host language. Piccola is a small language that unifies various notions of namespaces as first-class *forms*, or extensible, immutable records. By making namespaces explicit, Piccola is easily able to express various abstractions that would normally require more heavyweight techniques, such as language extensions or meta-programming.

1 Introduction

Virtually all programming languages support various notions of namespaces, or sets of bindings of labels to values. These include:

- **Interface**. Objects have a set of named methods.
- **Scopes**. Identifiers are bound in the enclosing static or dynamic scope.
- **Package**. A package provides a set of named services or components.
- **Keyword-based parameters**. Arguments to services are bound by keywords instead of position.

Typically, however, these notions are supported in different ways by a language, and each carries its own restrictions. This leads to a number of problems like *inflexible namespaces*, *frozen scoping rules*, and *limited abstraction*.

Inflexible Namespaces. An inflexible namespace can lead to name clashes. In open systems where components may be added or replaced at runtime, name clashes between components from different applications, domains, or vendors can cause system failures. The following lists symptoms that are due to inflexible namespaces:

- **Flat namespace**s. In older versions of Smalltalk, all classes must have unique names. To avoid name clashes, developers must follow naming conventions. Smalltalk Agent was one of the first Smalltalk implementations that provided namespaces. Now, most Smalltalk systems support namespaces. Similarly, clas-

---

1. *Authors' address:* Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland. *Tel:* +41 (31) 631.4618. *Fax:* +41 (31) 631.3965. *E-mail:* {acherman, oscar}@iam.unibe.ch. *WWW:* http://www.iam.unibe.ch/~scg.

This work has been funded by the Swiss National Science Foundation under Project No. 20-53711.98, "A framework approach to composing heterogeneous applications".

sic C++ has one static namespace. Standard C++ [6] introduces namespaces as an additional language feature.

- **Fixed namespace**s. In Java, each package has its own namespace for classes. Packages are nested, but inflexible. Two frameworks which — by chance — use the same package names, cannot be merged. The "solution" is to propose (internet wide) unique package naming conventions.
- **Restricted Scoping**. Python [14] has only three kinds of namespaces: one for global objects, one for class scope, and one for local block invocations. Although functions are first class values in Python, nested functions do not have closures. However, closures can be simulated by specifying values as default arguments.
- **Static Services**. Normally the run time environment (or the operating system) provides some static services. These services include printing to the console or accessing the local disk. These services normally operate within an implicit context. For example, the context defines where standard output should go (to the console or to a file), or the GUI context contains the look and feel of the user interface. It is in general not possible to adjust this context only for certain parts of an application. For instance, a developer might wish to redirect output of some threads to the console, while other threads may output to the null device.

Frozen Scoping Rules. Most modern languages use static scoping. Identifiers are visible within the block where they are declared and may also be visible in blocks that are statically (i.e. textually) nested within that block. Identifiers in the scope of a module or package can be exported to be used in other modules.

In contrast to these statically scoped languages there exist languages with dynamic scoping, like Postscript. Identifiers are looked up following the call stack. Dynamic scoped language are often considered less safe to use and require more care to program in. However, there are also abstractions implemented in dynamically scoped languages that are hard or clumsy to implement in statically scoped languages. Such abstractions include properties that do not align with the functional structure and cannot be localized in modular units. Examples include failure handling, synchronization and coordination.

Limited abstraction. The fact that namespaces are not available at runtime limits arbitrarily the expressive power of abstractions. A typical symptom of limited abstraction is programmers having to write a lot of boilerplate code. Examples of desirable abstractions include:

- A generic synchronization wrapper that wraps all the methods of an object to run in mutually exclusive mode. The inability to abstract over all methods of an object (in Java, for example) forces us to define a subclass that overrides each method to include the same synchronization code.
- An abstraction to generate proxies. A common use of proxies is to make distribution transparent. The proxy has the same interface as the original server object, but delegates all calls to the server object over the network. The proxy has similar code for all methods: it transfers arguments over the network, invokes the remote service, and waits for the result. For instance in Java RMI, the tool `rmic` automatically creates RMI proxies for remote objects out of their object code. But it is not possible to program the functionality of this tool directly in Java, without reading

and writing Java bytecode. The reflection support in `java.lang.reflect` only allows one to inspect code, but not to change it.

We address these problems by unifying namespaces as *forms*. In section 2 we briefly present Piccola, a small language that introduces explicit namespaces as forms. We illustrate how forms overcome the problems we have listed. In section 3 we present two applications of dynamic namespaces that demonstrate how the uniform treatment of explicit namespaces allows simple abstractions to be implemented in Piccola that would require more heavyweight approaches such as metaprogramming or compiler extensions in other languages. Finally, the last two sections present related and future work.

2 Piccola

Piccola is designed to be a general purpose "composition language" [1][2]. That is, it is designed as a language for composing software components which may be written in a separate implementation language. Piccola's job is to express how components are configured, and to provide the connectors, coordination abstractions and glue abstractions needed to configure components. As such, the problems listed in the introduction are especially important for Piccola. We tackle these problems by unifying all related notions of namespaces as *forms* (immutable, extensible records):

Everything is a form: Namespaces, contexts, interfaces, parameters, abstractions, scripts and objects are all modelled as forms. This unification leads to an extremely simple language, and allows us to abstract uniformly over all these related concepts.

Static and dynamic namespaces: Both client and server contexts are explicitly named, giving abstractions a fine degree of control over both static and dynamic scoping.

Explicit namespaces: Namespaces can be explicitly manipulated and composed, making it quite a simple matter to combine, rename and compose packages or modules.

Keyword-based parameters: Abstractions are monadic, always taking a single form as a parameter, and returning a form (which possibly encapsulates an abstraction). First class arguments extend the expressiveness of abstractions.

2.1 Separation of Concerns

Structure in Piccola is modelled by *forms*. *State* is modelled by *channels*, which are used to store forms. *Behaviour* is modelled by *agents*, which communicate by sending and receiving forms through shared channels. *Abstraction* is provided by *services*, which are implemented by agents and channels.

Forms. Forms are finite mappings from labels (identifiers) to values. Forms are immutable. The primitive operators on forms are *extension*, *projection*, and *iteration* over the labels of a form. Form extension concatenates a form with either a single binding or another form, yielding a new form as a result. Projection looks up a value bound by a label in a form. Iteration over a form returns the set of defined labels in a form. (Sets are objects, which are encoded as forms.)

A form in Piccola is defined by a *script*, which is a sequence of bindings and form-expressions. Form-expressions are structured using parentheses or indentation, and separated using commas or newlines, in the style of Python. The comma or newline stands for the extension operator. Bindings declare either nested forms or service definitions. The empty form is written as `()`. For example:

```
aForm =
    aSubForm = ()              # a nested form
    aService(X): X             # service definition
    r(count = 3)               # form expression
```

The form `aForm` contains the labels `aSubForm`, `aService`, and all the labels that are returned by invoking the service `r`. If `r()` returns a form with label `aSubForm` or `aService`, these bindings will hide the bindings that precede the invocation. The service `r` is invoked with the argument form `count = 3`.

Channels. State is represented by *channels*. Channels have the semantics of locations in the asynchronous π-calculus [16]. Using channels, we can model blackboards, locks, reference cells etc. The semantics of Piccola is given in terms of the πL-calculus [13], a variant of the π-calculus in which agents communicate forms instead of tuples.

Agents. *Agents* implement the behaviour of a Piccola program. Agents communicate along channels and exchange forms. Unlike forms, agents and channels do not appear in the syntax of Piccola, but they can be directly instantiated, if necessary, by means of the predefined services `run` and `newChannel`.

Services. A service represents a function or procedure. It is represented by a replicated agent that reads from a channel (the service location) and evaluates a form as its result. The service-protocol specifies how the result channel gets passed from the caller to the callee [15][21]. Piccola has only four keywords, two of which are needed to define services. The value returned by a service may be denoted by **`return`**. A recursively-defined service must be declared with **`def`**, which constructs a fixpoint.

2.2 Static and Dynamic Namespaces

Piccola is statically scoped, and the static context of an agent is always explicitly accessible as a form called **`root`**. The dynamic namespace of a calling agent, however, is also available to the service invoked as a form called **`dynamic`**. (**`root`** and **`dynamic`** are the other two keywords of Piccola.)

Labels used in a script are normally looked up in the **`root`** form, and bindings will extend the **`root`** form. For example, this binding defines a service `newDocument`:

```
newDocument(X): wrap(newBasicDocument(X))
```

Agents evaluating form expressions textually below this binding have the identifier `newDocument` in their **`root`** form. More explicitly, we could also extend the **`root`** form to include the definition of the service `newDocument`:

```
(1) root =
(2)     root
(3)     newDocument(X): wrap(newBasicDocument(X))
```

This statement is read as follows: Replace the **`root`** form with a new form (Line 1). The new form is indented. It is the current **`root`** form (Line 2) extended with the service `newDocument` (Line 3).

Lookup of identifiers is done in the **root** form. Therefore, the body of the `newDocument` service is equivalent to:

```
root.wrap(root.newBasicDocument(root.X))
```

This more clumsy notation stresses the fact that these identifiers are looked up in the root namespace of the agent implementing the body of the service. Note that the argument label `X` is only defined in the **root** form of the service body.

The static scoping offered by these conventions is fine for most purposes, but some kinds of abstractions can only be conveniently implemented with the help of dynamic scoping. The **dynamic** namespace of an agent contains whatever is explicitly put there, and is passed automatically whenever the agent invokes a service. The following `myPrintln` service includes the current user in its output:

```
myPrintln(Text): println(dynamic.user + ":" + Text)
```

A caller of this service may change its dynamic namespace to include the current user:

```
dynamic = (user = "John")          # change dynamic
myPrintln("Hello")                 # invoke service
```

Note that the dynamic namespace does not break encapsulation. Values that are not put into this form remain local. The dynamic namespace is useful for passing implicit information between agents, but it should not be misused as an alternative to explicit passing of parameters.

2.3 Explicit, First-Class Namespaces

The possibility to explicitly read and assign the **root** namespace enables us to directly support the various importing facilities found in other languages, like the *import package* statement of Java or the *from package import* facility of Python. The service `load()` locates a file containing a script, evaluates it, and returns the form defined by the script. Assume we have a script "`hello.picl`" with the contents:

```
# File: hello.picl
info: println("This is the hello script")
```

The script defines a form with a service bound by `info`. We can now:

- import all the bindings of the hello script and extend our **root** with them:

  ```
  root = (root, load("hello"))
  info()                           # invoke it
  ```

  This is equivalent to importing all names from a given module. If the service `info` is already defined, it will be overridden.

- import all the bindings but keep them in a separate nested form `helloFile`. This prevents our **root** namespace from getting cluttered up:

  ```
  helloFile = load("hello")
  helloFile.info()                 # and use it
  ```

- import only the `info` service under a different name:

  ```
  helloInfo = load("hello").info
  helloInfo()                      # and use the service
  ```

The reader should note that these mechanisms can be combined. For example we can import a module, store it under a new name and rename selected services within. By using first-class forms to represent packages, language-specific import statements or

namespace qualifiers become superfluous. We thereby overcome the problems related to rigid namespaces mentioned in the introduction.

2.4 First-Class arguments

Services in Piccola are monadic, taking a single form as a parameter. Keyword based arguments are transferred as nested forms. Since arguments are forms, form extension allows us to easily model *default arguments*. For instance, the following generic wrapper adds pre- and post- services to a given service:

```
myDefaults =                # a form with two (empty) services
     pre: ()
     post: ()
wrap(X)(Args):
     (myDefaults, X).pre()  # invoke pre() in X or myDefaults
     res = X.service(Args)  # invoke main service
     (myDefaults, X).post()
     return res
```

The service `wrap` is curried. It first expects a form X with three labels: `pre`, `service`, and `post`. Invoking the service `s = wrap(..)` with a form `Args` calls `pre()`, then invokes the service with the passed `Args` form and finally calls `post()`. Observe how the `pre` and `post` service have a default. We prefix the argument form `X` with default bindings encapsulated in the form `myDefaults`. The projection `(myDefaults, X).pre` will extract the service bound to `pre` in `X`, if it exists. Otherwise the default service defined in `myDefaults` will be used.

3 Dynamic abstractions

This section will outline two applications using dynamic namespaces that typically could not be implemented without either language extensions or meta-programming. The first example implements an exception handling mechanism as a library abstraction in Piccola, using dynamic namespaces to pass the exception handler to the context in which exceptions are raised.

The second example implements an ownership abstraction, realised as a wrapper for arbitrary forms and an evaluation context that may own certain objects. Only the owner can execute services of the wrapped objects. This is an example which is not commonly found as language construct. We conclude the section with some recommendations for disciplined use of the dynamic namespace.

3.1 Exceptions

An exception is raised during program execution as a reaction to some erroneous situation. The part of the program that detects the erroneous situation cannot handle it. Instead, it signals this situation and terminates execution. We say the program *raises an exception*. An exception handler, which was installed at an earlier point during program execution, catches the error and handles the exception, i.e. brings the system back to a consistent state.

The problem is how to transmit the flow of control from the place where the exception is detected to the appropriate handler. A simple approach would be to define some global exception-holding variables. After invocation of a service, the client is obliged

to check this error state and handle it if appropriated. This solution is clumsy since each function call must be followed by an error check. It also does not work in a concurrent system, since all processes would share the same error slot. Another possibility is to extend the returned value to contain a flag that indicates whether the returned value is valid or an error occurred during its computation. This approach requires that we adapt all return values to reflect the change. Furthermore it assumes that all services have a reply, which, for example, may not be necessary for distributed notifications.

Our solution is to use the dynamic namespace to transmit the exception from the raising point to the appropriate handler. The exception handler is set as follows:

```
try
    do: ...                    # use exception handler
    catch(E): ...              # handle an exception
```

The service `try` takes a form containing two services. The first is the `do:` service. Its body represents the scope of the exception handler. The handler itself is specified as a service `catch(E)` where `E` is the formal exception value. Whenever an exception occurs during the execution of the `do:` service, this handler is invoked instead of the normal continuation. Here is the implementation of the `try` and `raise` services:

```
(1)  raise(E): dynamic.raise(E) # delegate to dynamic raise
(2)  try(block):
(3)      exception = newChannel()
(4)      return OrJoin          # start agents left and right
(5)          left:
(6)              block.catch(exception.receive())
(7)          right:
(8)              raise(e):     # local raise abstraction
(9)                  exception.send(e)
(10)                 stop()
(11)             dynamic = (dynamic, raise = raise)
(12)             return block.do()
```

The `OrJoin` service (Line 4) takes two services (`left` and `right`) and executes them concurrently. It returns the result of whatever service first terminates. Consider first the scenario in which a block is executed that does not lead to an exception:

1. Two agents passed to `OrJoin` are started. The left agent has no impact as it is blocked on the local exception channel. This agent finally gets garbage collected, since no one ever will write to the exception channel.
2. The right agent runs `block.do()` (Line 12).
3. `OrJoin` receives the result of the right agent and returns this as the result of the `try` statement.

Next, consider the case where the block raises an exception:

1. The two agents are started. The left agent waits on the exception channel.
2. The right agent runs the `block.do()` (Line 12).
3. To raise the exception in the `do()` block, the global `raise(..)` (defined on Line 1) is invoked.
4. The global `raise()` delegates the exception to **`dynamic`**`.raise()` which is the local `raise` abstraction (Line 8).

5. The local `raise` sends the exception value along the exception channel (Line 9) and silently halts using the `stop()` service. This means that OrJoin will not see this service terminating.
6. The left agent is the only one to continue, fetching the exception value `E`, invoking `catch(E)` and returning (Line 6).

The `raise` service can be considered as an implicit additional argument passed during invocation. This resembles the idiom used when programming with exceptions. The signature of a service that may raise an exception looks like `aService(..., ExceptionHandler e)`. Compared to this approach, the explicit dynamic namespace has several advantages. First, it supports the separation of functional aspect from the error handling aspect. It seems more appropriate to directly relate the formal argument of a method to its functional aspect, instead of blurring it up with contextual arguments. It makes code more readable (thus maintainable) when unnecessary parameters are not visible. Imagine a function which does not raise an exception itself, but is required to pass the handler down to all services it uses. Finally, dynamic namespaces allow the programmer to introduce an exception handler later in the project development without rewriting code that neither handles exceptions nor detects erroneous situations.

Observe that the exception abstraction cannot be implemented as a simple wrapper that adds some pre- and post execution code. The reason is that `raise` must be accessible from anywhere *within* the executed block.

3.2 Ownership

In our second example we consider ownership of objects. An *ownable* object belongs to at most one owner. Only the owner can invoke services of the owned object. An ownable object can be *fetched* by an owner, which then has privileged access to it. The owner may release or transfer ownership. A notion of ownership can be used in various areas: for example synchronization for owned objects can be managed by the owner, or the owner can take over garbage collection issues on the owned object. Ownership can guarantee alias free references [17].

To translate an ordinary object into an ownable object, we do the following:

- Add an instance variable to store an owner.
- Add methods to fetch, remove, and transmit ownership. Of course, fetch will only work when the object is not owned for the moment. Remove and transmit are only possible, if the caller owns the object in question.
- Modify each method such that it expects an owner as additional argument. The precondition of the method is strengthened, as it is necessary that the passed owner be the owner of the object. Only when the passed owner owns the object can the method be performed, otherwise an exception is raised.
- All calls to the object methods must reflect the change and also include the owner.

Using explicit namespaces, it is possible to (1) build a generic abstraction `wrapOwnable(Form)` that wraps all services of the form to check for ownership, and (2) to build an evaluator `runAsOwner(Block)` that runs a block of code with an owner. Assume we have object factories to create an owner, and an ownable:

```
newOwner:
     owns(Ownable): ...     # do we own the ownable?
     add(Ownable): ...      # add the ownable
     remove(Ownable): ...   # remove the ownable
     loseAll: ...           # remove all ownables we have
newOwnable:
     addTo(Owner): ...
     release: ...
```

Given an instance `o` of `ownable`, then `o.addTo(Owner)` stores the owner, provided `o` is not already owned, and notifies the owner using `Owner.add(o)`.

Evaluating a block within the context of an owner is now written as:

```
runAsOwner(block):
     # create new Owner
     dynamic = (dynamic, currentOwner = newOwner())
     block.do()                          # evaluate Block
     dynamic.currentOwner.loseAll()      # drop all owned
```

This runs the block within a dynamic namespace with an associated (initially empty) owner. Finally, the generic wrapper that makes a form into an ownable form is:

```
(1)  getCurrentOwner: dynamic.currentOwner
(2)  wrapOwnable(Form):
(3)       ownable = newOwnable()              # delegate
(4)       newForm = wrapAllLambda             # adapt all services
(5)            form =
(6)                 Form
(7)                 releaseThisForm: ownable.release()
(8)            map(service)(Args):
(9)                 if (getCurrentOwner().owns(ownable))
(10)                     then: service(Args)   # invoke service
(11)                     else: raise(NotOwnerException)
(12)      return
(13)           newForm
(14)           ownThisForm: ownable.addTo(getCurrentOwner())
```

The wrapper needs some explanation. Line 3 creates the ownable object as a delegate. Then all services of the wrapped form `Form`, extended with `releaseThisForm` are modified by a map function. The new function (Line 9 - 11) checks if the current (dynamic) owner owns this ownable object. If so it invokes the original service with the given arguments. Otherwise an exception is raised, signalling that the caller does not own the object. The library service `wrapAllLambda` uses form-iteration to get the set of defined labels (i.e. the exported services) of `form`.

Note that we include the additional service `releaseThisForm` (Line 7) into the map to ensure that only the current owner may release it. (Transfer of ownership is omitted in the code). We return the wrapped form (Line 13) extended with the service to acquire ownership (Line 14).

3.3 Observations

We can draw the following lessons from the two examples:

- Each feature requires a label in the dynamic namespace. Exceptions use `raise`, and ownership uses `currentOwner` to store the context sensitive information. We assume that these bindings do not conflict with other usages of the dynamic namespace.
- The users of the contextual abstractions do not need to access **`dynamic`** themselves. Instead it is better to provide static abstractions that access the context sensitive information, e.g. `getCurrentOwner()` in the second example.
- Contextual abstractions are used in pairs: Outside is an abstraction (e.g. `try`) that executes a piece of code (the `do` block) within a extended context. Within this block are clients of the contextual abstraction that invoke the service (e.g. `raise`) provided by the surrounding context. Using the contextual service not within the established context is a type error: it results in looking up a label in **`dynamic`** which is not bound.

4 Related and Future Work

Objects and many different variants of inheritance (e.g. Smalltalk-style vs. Beta-style inheritance [3]) can also be modelled as applications of forms as explicit namespaces [23]. In effect self is represented as a form containing the object's methods. Subclassing corresponds to extending the form representing self. A form is conceptually simpler than an object, since it lacks a notion of inheritance. For instance, in Self [25] objects have a parent link providing inheritance by means of delegation. Therefore, in Self delegation is built into the language, whereas we implement it using the forms.

Many scripting languages provide access to the environment by representing it as a dictionary. Python [14] has built-in functions to return its namespaces as dictionaries to enable introspection. Modification of these dictionaries, however, is undefined. A dictionary gives the programmer much more freedom than is presently possible with forms. In particular, labels of forms in Piccola are not first class values, whereas dictionaries for environments often use strings as keys.

Forms can also be compared to Odersky's variable functions [18]. Variable functions are mappings between sets of arbitrary values (not just from labels to forms), and can be updated to model state changes.

Namespaces play an emerging role in middleware: For instance the Corba naming service [19] uses nested namespaces to identify distributed objects.

Future work is required to clarify the relation between namespaces, and security and authentication issues. In an open system, mobile code runs in two modes: one mode gives unrestricted access to local resources, while restricted access employs a security manager to guard access and use of local resources. In the ambient calculus [4] an ambient corresponds to an administrative domain. An ambient can only access services within its domain. An interesting question to explore is whether we can unify ambients and namespaces.

Pict [20] is a language that takes the π-calculus as a core language and adds functions, assignment etc. as syntactic sugar. We used Pict for experiments in modelling software composition [22]. The πL-calculus is a result of these studies. It replaces tuple communication by form communication. Piccola is formally defined on the πL-calcu-

lus. It adopts the primitives of the πL-calculus (channels, and parallel composition operator) but makes them available as predefined services which can be overridden if necessary. The form-calculus [23] extends the set of core form operators. The additional operators are simple label restriction and form restriction to remove labels, and a matching operator to check for the existence of a label. Lumpe has developed a type system for the πL-calculus [13], but this system cannot be incorporated directly into Piccola, because it lacks parametric polymorphism and recursive types.

Common Lisp [24] allows the programmer to declare "special" variables to be dynamically scoped. Many languages now have features incorporated into their libraries that allow the programmer to create and use dynamic variables. For instance in Java2, the class `java.lang.ThreadLocal` contains a different value for each thread. Programmers use this class to store transaction identifiers or similar constructs.

Applications using dynamic namespaces have many similarities to programming with monads in functional languages. Monads are used to model state in a purely functional world [10][26]. The dynamic namespace builds on the notion of clients and providers of services. It therefore naturally extends to open, distributed systems, whereas monads are closely related to the lambda calculus.

In the area of object oriented languages, there exist several proposals to better support separation of concerns within a program. The proposal that seems the most attracting is aspect-oriented programming (AOP) [8]. In AOP, aspects are explicitly separated from normal classes. The *aspect weaver* merges the aspect into the source code. Using AOP can greatly reduce the complexity of code [11].

Many of the applications possible using dynamic namespaces can also be implemented using metaobjects and message passing control [5][7]. We consider the approach with explicit namespaces to be much more lightweight.

5 Conclusion

Piccola is a small language for composing software components. It is intended to be a general language suitable for expressing many different styles of components and composition abstractions. One way it achieves this is by unifying various notions of namespaces present in other languages, such as environments, interfaces, objects and packages, and making them explicitly manipulable as "forms."

Explicit namespaces make it possible in Piccola to have flexible static and dynamic scoping, to support various module concepts, and to implement generic wrappers that go beyond adding pre- and post methods to services. All this flexibility can be achieved with a minimal set of operators over forms and does not require the use of meta programming facilities.

A stable implementation of Piccola is available from the authors' web site. Work is ongoing in many areas, including experimental development of compositional styles for various application domains, reasoning about compositional properties, visualization, distribution, and flexible type systems.

Acknowledgements

We thank the members of the SCG for stimulating discussions and in particular Stéphane Ducasse and Matthias Rieger for helpful comments on a draft of this paper, and the anonymous referees for providing constructive and valuable suggestions.

References

[1] Franz Achermann and Oscar Nierstrasz, **"Applications = Components + Scripts -- A tour of Piccola,"** *Software Architectures and Component Technology*, Mehmet Aksit (Ed.), Kluwer, 2000, to appear.

[2] Franz Achermann, Markus Lumpe, Jean-Guy Schneider and Oscar Nierstrasz, "**Piccola - a Small Composition Language**," *Formal Methods for Distributed Processing, an Object Oriented Approach*, Howard Bowman and John Derrick. (Ed.), Cambridge University Press., 2000, to appear.

[3] Gilad Bracha and William Cook, "**Mixin-based Inheritance**," *Proceedings OOPSLA/ ECOOP'90*, ACM SIGPLAN Notices, vol. 25, no. 10, Oct. 1990, pp. 303-311.

[4] Luca Cardelli and Andrew D. Gordon, **"Mobile Ambients,"** *Foundations of Software Science and Computational Structures*, Maurice Nivat (Ed.), LNCS, vol. 1378, Springer Verlag, 1998, pp. 140-155.

[5] Stéphane Ducasse, "**Evaluating Message Passing Control Techniques in Smalltalk**," *Journal of Object-Oriented Programming (JOOP)*, vol. 12, no. 6, SIGS Press, June 1999, pp. 39-44.

[6] D. Kalev, *Ansi/Iso C++ Professional Programmer's Handbook*, Que Professional Series, 1999

[7] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[8] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin, "**Aspect-Oriented Programming**," *Proceedings ECOOP'97*, Mehmet Aksit and Satoshi Matsuoka (Ed.), LNCS 1241, Springer-Verlag, Jyvaskyla, Finland, June 1997, pp. 220-242.

[9] Doug Lea, "**Design for Open Systems in Java,"** *Proceedings COORDINATION'97*, David Garlan & Daniel Le Mètayer (Ed.), LNCS 1282, Springer-Verlag, Berlin, Germany, September 1997, pp. 32-45.

[10] Sheng Liang, Paul Hudak and Mark P. Jones, **"Monad Transformers and Modular Interpreters",** *Conference Record of POPL'95*, San Francisco, California, 1995, pp. 333-343.

[11] Martin Lippert and Cristina V. Lopes, "**A Study on Exception Detection and Handling Using Aspect-Oriented Programming,"** Technical Report P9910229 CSL-99-1, Xerox Parc Palo Alto, Dec. 1999.

[12] Markus Lumpe, Franz Achermann and Oscar Nierstrasz, "**A Formal Language for Composition**," *Foundations of Component Based Systems*, Gary Leavens and Murali Sitaraman (Ed.), Cambridge University Press., 2000, pp. 69-90.

[13] Markus Lumpe, "**A Pi-Calculus Based Approach to Software Composition**," Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[14] Mark Lutz, *Programming Python*, O'Reilly, 1996.

[15] Robin Milner, "**Functions as Processes**," Proceedings ICALP'90, M.S. Paterson (Ed.), LNCS 443, Springer-Verlag, Warwick U., July 1990, pp. 167-180.

[16] Robin Milner, "**The Polyadic pi Calculus: a tutorial**," ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh, Oct. 1991.
[17] James Noble, John Potter and Jan Vitek, "**Flexible alias protection**," *Proceedings ECOOP'98*, Eric Jul (Ed.), LCNS 1445, Springer-Verlag, Brussels, Belgium, July 1998.
[18] Martion Odersky, "**Programming with Variable Functions**," *Proc. International Conference on Functional Programming*, Baltimore, 1998.
[19] Robert Orfali, Dan Harkey and Jeri Edwards, *Instant Corba*, Wiley, 1997.
[20] Benjamin C. Pierce and David N. Turner, "**Pict: A Programming Language based on the Pi-Calculus,**" Technical Report, no. CSCI 476, Computer Science Department, Indiana University, March 1997.
[21] D. Sangiorgi, "**Interpreting functions as Pi-calculus processes: a tutorial**," RR 3470, INRIA Sophia-Antipolis, France, February 1999.
[22] Jean-Guy Schneider and Markus Lumpe, "**Synchronizing Concurrent Objects in the Pi-Calculus,**" *Proceedings of Langages et Modèles à Objects'97*, Roland Ducournau and Serge Garlatti (Ed.), Hermes, Roscoff, October 1997, pp. 61-76.
[23] Jean-Guy Schneider, "**Components, Scripts, and Glue: A conceptual framework for software composition**," Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.
[24] Guy L. Steele, *Common Lisp The Language, Second Edition*, Digital Press, 1990.
[25] David Ungar and Randall B. Smith, "**Self: The Power of Simplicity,**" *Proceedings OOPSLA'87, ACM SIGPLAN Notices*, December 1987, pp. 227-242.
[26] Philip Wadler, "**Monads for functional programming,**" *Advanced Functional Programming*, J. Jeuring and E. Meijer (Ed.), LNCS 925.