Piccola White Paper

Franz Achermann

Software Composition Group[1]

Draft: December 6, 1999

Abstract. This paper lists domains where using Piccola will be helpful. It summarizes topics to explore in more detail.

1 Introduction

Piccola is a general purpose composition language. It provides basic abstractions to compose components, to invoke services, and to define new services. The runtime model of Piccola is that of communicating agents. Piccola's small and expressive syntax comes from the fact that several concepts are unified by forms. Forms are immutable mappings from labels to values. They represent:

- **Interfaces** to Components. A component offers a set of named services.
- Program fragments or **Scripts**. An agent evaluates a script to a form. The set of bindings made public is the form defined by the script. Bindings associate labels with some values.
- **Contexts**. Agents in Piccola have two separate namespaces. The root namespace represents the lexical scope, the dynamic namespaces represents the environment passed at invocation time. Agents have full control over these two forms.
- **Services**. The unification of forms and services allows us to treat functions as first class values and thus adapt idioms known from functional programming. In addition, services may carry additional information like documentation with them.
- **Arguments** for services. Forms provide a natural way for keyword-based argument passing. This makes abstractions more reusable and extensible than tuple-based signatures.

Piccola is implemented on top of Java. The language is implemented by translating Piccola agents into the πL-calculus, which is the π-calculus with Forms. The πL-agents are interpreted by an abstract πL-machine.

The topics identified in the following sections come from a diversity of domains. This will support our claim that Piccola serves as a vehicle for studying composition in general.

Application development in Piccola follows the multi-paradigm design. For a problem domain, we look for a solution domain that most naturally allows us to script an agent to fulfil the requirement. Such a solution domain can be characterized by a archi-

1. *Author's address:* Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, CH-3012 Berne, Switzerland. *Tel:* +41 (31) 631.3313. *Fax:* +41 (31) 631.3965. *E-mail:* acherman@iam.unibe.ch. *WWW:* http://www.iam.unibe.ch/~acherman.

tectural style that consist of components and connectors [18]. For simple applications a single style may be enough. More complex applications will employ several styles at different level of complexity. For instance in 3-tier applications the user interface may best be scripted in an event based style, the business logic is implemented as a set of business rules, and the persistency layer can be expressed in terms of relational database.

We use the idea of architectural styles to present areas for future work in Piccola. The next section explores different styles and Section 3 mentions applications built within these styles. In Section 4 we strive a number of meta-problems that can be addressed within the Piccola framework. Section 5 describes questions related to the Piccola language including extensions and type system. Finally, Section 6 points to interesting implementation aspects. Section 7 concludes the paper.

2 Styles

One of Piccola's design intention was not to favour any component or object model, nor favour a specific module approach in the language. Traditionally scripting languages incorporate one or more style to program. For example, Unix scripts have pipes and filters, awk-scripts have rules that fire actions depending on the currently processed tuple, and Visual Basic scripts COM components using procedures. In contrast to these languages, Piccola has a very flexible computation mechanism, namely that of agents exchanging forms. This allows us to implement various component types and give a user of these types the feeling he is programming in a scripting language which has built-in support for these types.

An architectural style defines a component model and specifies rules governing composition of these components [17][18]. The rules may exclude certain configurations. Connectors implement, guard, and control the cooperation between components. Architectural description languages formally specify the behaviour of connectors. Sticking to certain connectors allows the application programmer to guarantee that certain system invariants hold. Ideally, these invariants hold *by construction* [1] (see for example Section 2.7).

Connected components should be components again [12]. At the very end, we can describe an architectural style as an *algebra*. There are operators over components yielding new components. A style can also be compared to a object-oriented *framework*. A black-box framework defines a set of components and connectors. Making this composition algebra more explicit in the top-level script makes the concrete usage of the framework more explicit. This simplifies the steep learning curve normally associated with frameworks.

The object oriented paradigm provides two composition mechanisms. These mechanisms are abstract base classes with template methods or classes, prototype-, or delegate objects which are plugged into other hosting objects. In contrast to this paradigm, Piccola provides a much richer and easier-to-use set of composition mechanism, namely that of first class agents, user-defined operators, and context.

In the following, we present a list of styles. Sometimes the style is interesting per se, sometimes the style leads to applications (see Section 3) which are more compelling that the plain style.

2.1 Event Based Architecture

Event based system integration has the advantage that neither the client nor the server need to know each other in advance [2]. The connection is established by subscribing an observer to an observable. This approach makes it possible to develop the provider of a service independent from its client and vice versa. The main drawback of this approach is that the wiring of subscribers and listeners gets obscured. We cannot know statically what will happen when a certain event occurs since we do not know which observers are currently subscribed. Contrast this to the normal procedure call, where we precisely know the effect of a procedure invocation.

In Piccola it is possible to provide abstractions that attach a listener to a publisher in one line, for example by: "`Button ? Action(do: ...)`". This makes the listener-informer connections more explicit. We currently lack a compact notion to remove a listener. The listeners attached to informers specifies the architecture. But when we start to add and remove listeners dynamically, we have a dynamic architecture which is more difficult to understand. The compositional, algebraical view only makes the static configuration explicit.

We can see events as the *gotos of component wiring*. Attaching a listener to a publisher far away is possible but obscures the understanding of the system. A solution would be if we can build higher level abstractions that would minimize necessity to use plain events. Using these abstractions the connections of subscribers to observers will disappear in top-level scripts. An example of such an abstraction is checkbox groups. Individual checkbox items can be added to the group, exactly one checkbox is marked at any time. This behaviour can be implemented as follows: The checkbox group has a slot to store the currently selected button. Clicking on a button causes that button to be written to the slot and the previously selected button to be de-selected. The behaviour of a checkbox group is encapsulated by a class in almost all graphical user interface frameworks. Can we derive similar abstractions to minimize the need to use (low-level) event bindings? Can we describe the overall structure of an event based application as a a *dataflow network*?

In Section 4.3 we discuss the notion of groups of objects. We can relate the individual objects of such a group by events and only provide higher level services to the outside of the group. Event can be encapsulated.

The spaghetti of event subscriptions can also be addressed by terminology of contracts (see Section 4.1).

Alternatively we can associated *obligations* with informers. We could, for instance, require that at least (and only one) listener is attached to a given event type.

Another question using event based integration is how to transport notifications. In particular when observers are distributed on the whole internet, network traffic becomes an issue [3].

2.2 GUI Composition

The GUI-Composition style provides abstractions to do the graphical layout. It may also provide abstractions for dragging, dropping, pointing, pasting, etc.

Reasoning about the layout is difficult. A simple approach would be to adapt the layout managers of known frameworks. What would be gained by formally specifying the layout?

2.3 Actions and Transactions

Piccola is a system to build new services using primitive ones. We can define on top of the builtin notion of services a notion of actions. Such an *algebra of actions* may allow us to combine primitive actions into new ones. For example actions may be defined like:

1. Actions have a precondition. Combined actions may only be executed when all preconditions hold. Additionally, actions must not raise exceptions.
2. Actions have names. Actions without required arguments can be associated to buttons or menu items. Furthermore actions can be associated with arbitrary GUI components, taking for example the selection as input and be invoked by a popup menu. The dynamic context represents the event information, the associated GUI component etc.
3. Actions can be enabled or disabled.
4. Actions can be composed with synchronization policies.
5. Actions have no result. Therefore we need also queries. Queries can block, but they must not invoke any actions.

An example for a composition language with builtin (high level) services is CLAM [14]. In this language, service estimate their invocation costs, can complete partially, and can be invoked synchronous or asynchronous.

Each action has a well defined effect [REF Effect system, Talpin].

The composition with synchronization policies also provides means to have pluggable transactions for pessimistic (low-level) and optimistic (high-level) coordination.

2.4 Mobile forms

We identify a number of domains in which notations similar to nested forms occur, for example XML, the any-framework (used in Sniff) [9]. Piccola provides static nested forms and offers a way to read them as plain scripts. In addition, it is possible to associate agents to forms. Though it is possible to associate behaviour with XML, this functionality is implemented by foreign language constructs like Applets, Java Script and Live Connect.

The style has some similarities with middleware. It defines how forms are communicated and what properties communicable forms need. Traditionally, objects can be transferred when they can be represented as streams. This means the object must know how to serialize itself and how to de-serialize itself from a stream. A Piccola script (a plain ASCII text) can be communicated between *Piccola servers*. Forms encapsulate agents (i.e. behaviour) and data fields. A server receiving a form can either manipulate it directly or invoke some of the received agents within certain contexts.

In a mobile form style, we can explain web-based applications (e.g. active server pages for Microsoft, JavaServer Pages). Furthermore the simple model could serve as a tool to detect (i) inconsistencies or (ii) redundant concepts in these technologies. This may lead to *a conceptual framework for web-based applications*. Ideally, we could describe architectural mismatches or incompatible uses of these technologies or components by an explanation in the Piccola framework. See also Section 5.4. for issues on distribution and Section 3.2 for applications based on mobile forms.

2.5 Multiset Style

In this style everything is a multiset (aka. collection) of forms. This style is closely related to RDBMS. In database management systems, behaviour is stored in data dictionaries and control tables. In contrast to these, Piccola forms may contain behaviour at arbitrary tables.

Querying operators for forms must be further elaborated.

In traditional database-business applications, the user interface is programmed as a set of data-sheet forms, extended with many-one notions to represent the relation. The behaviour of such a particular edit-form is implemented in a 4-GL language. In Piccola, each form can carry its own editor (see Section 3.1).

This style is using set theory, such as union, intersection, etc. to solve some types of problems. Set theory can be used in SQL, for example, to solve complex processing problems.

The Multiset style can fruitfully be combined with the Pipes and Filters style. In fact, a stream can represent a (ordered) multiset. Consequently, multisets may provide sources for pipe and filter architectures. The multiset style may also provide a central, persistent store for documents. This store may be a tuple or form space. Agents communicate with this form space.

In contrast to mobile scripts where any two Piccola machines exchange scripts, in this style clients only communicate with the form space server.

2.6 Rule Based

This style organizes behaviour by a list or pool of rules that are constantly cycled through. The rules can have priorities and may be “switched-off” under some circumstances and conditions. Rules trigger actions (see Section 2.3).

2.7 Pipes and Filters

This is probably the most well known style. Stream paradigms provide solutions by linking the input and output of smaller programs or utilities, sort of like a bunch sausages linked together. Although streaming is highly useful for batch operations, they are more limited in highly interactive systems.

This style also provides the classical example of system invariant that hold by construction. Pipe and Filter systems will not contain cycles, (therefore be deadlock free) provided that each filter only has one input and one output connection and there is not means of chaining filters backwards.

2.8 Objects and Classes

In this style, everything is organized by objects or similar categories (subclasses) and a predetermined set of operations that operate on them. Schneider [16] has implemented a meta-model for object-based programming using forms.

2.9 Run-time Models

Another example of a style is the runtime model of programming languages. For instance in Java, the runtime models consists of method calls (i.e. functions), exceptions, threads, and the synchronization primitives for these threads. It is possible to give a mapping of these abstractions to Piccola. This can lead to a π-model of Java programs. Is there more to say? What kind of reasoning can we do on Java programs? Can we for example detect nested monitors?

3 Applications

This section describes applications that can be built using the styles of the previous section.

3.1 Form Editor

We can implement a generic *form editor* in Piccola, with import and export facilities to common nested form languages like XML. Alternatively, we can use XML editors to edit forms. The form editor may also use form spaces (Section 2.5) to work on.

3.2 Using mobile scripts

There are numbers of technologies that can implemented on top of an infrastructure in which Piccola servers (and browser) exchange scripts.

- **Applets**. The applet is requested by a browser. On receiving an applet, the browser executes it in its own context. The context gives access to services of the browser etc. (see Section 4.4).

  In contrast to applets, normal HTML pages have only data fields (they are nested forms). Currently there are additional technology and notions required to access the HTML-page that contains the applet-tag (Java Script, Life connect etc.). In Piccola, all the required terminology for active web pages will collapse on the concept of mobile forms.
- **iMail**. A mail is represented as an mobile (active) form. Such a form makes use of the default form editing properties (see above) In addition, a mail-form could have a home-server (e.g. the initiator of the mail) and also use services from its home.

  The scripts (agents) attached to the mail (the form) specify the workflow logic.
- **hyperNotes**. Notes modelled as textual nested forms. Related Work: wdb, bib, gedcom, CDIF, XML etc.

Lotus Notes is an example of a system that uses documents as the base for work-flow and other business tasks.

3.3 Email Filter Library

Provide abstractions for scripting email filters. The run-time contains stream of email messages passed through series of Rule filters.

```
Source (stdin) | (Rule(regexp) > file)).
```

Actions for such a system include moving and copying mails to folders, to forward them, or to auto-reply Emails. Conditions are given as regular expressions matching subject, time, body etc. For example, newsgroup mails may be deleted when the remain several days unread.

Needs further analysis and integration with MIME types.

3.4 Piccola ULC

Ultra light clients exchange Piccola scripts with the server.

3.5 Composition Environment

Provide a generic, scriptable composition environment. The environment supports different views on software: design view, architectural view [6].

3.6 Demos

Here is a list of demos examples:

- Scribble Applet. The example from the Java Nutshell book in Piccola.
- Turtle graphics.
- Examples from the CP course.

4 Insights

This section discusses areas where Piccola helps us to understand the issue of software composition in general.

4.1 Component Models

We are faced with a variety of component models in use: COM (and friends), Java Beans (and EJB), CORBA components etc. Are we able to develop these models on top of Piccola? What insights do we get?

Piccola's formal model can be used to reason about requirements for certain components models. As an example, we can study plug-ins and ask ourselves questions like: What is required by a hosting environment so that components act as *plug-ins*? These plug-ins can be (1) installed and (2) de-installed. Plug-ins have properties (3) which are customized. An installed and enabled plug-in has a effect on the hosting environment: it may enhance (4), replace (5), forbid (6) certain services. Finally plug-ins may be dependent (7) on other plug-ins. This dependency may be version specific. We can also speak of a framework for plug-ins or a plug-in style.

4.2 Contracts

Contracts are established and enforced between cooperating agents. The agents agree on a certain contract. Different kinds of contracts studied in the literature are:

- types
- pre & post conditions
- Helm & Holland contracts
- temporal constraints
- reuse contracts

In Piccola, some of these contracts can be expressed, and possibly also derived using the semantics of Piccola (see Section 5.5). How can we specify and reason about:

- Interfaces (operations, input- and output data types, exceptions),
- Dependencies (a la reuse contracts),
- Behaviour: (pre- and post conditions, invariants, protocols),
- Properties (safety and liveness guarantees, fairness, hard or soft real-time guarantees?

4.3 Coordination Rules

Minsky [10] uses coordination rules to enforce or prohibit coordinated objects to do some actions. Such rules can be expressed as Piccola abstractions. We can plug in different agents (as components) and the rule will govern the context for these components. A simple example may intercept all incoming requests to the component and provide a trace facility.

4.4 Dynamic context and Aspects

Piccola provides the notion of dynamic context. In general, this context represents the set of services offered by the hosting programming environment. For instance, the dynamic context of applets gives access to the browser running the applet. In Java, the `system` class gives access to the console output stream provided by the JVM.

In Piccola we currently use the dynamic context to build an exception handling mechanism. There are other areas where the notion of dynamic environment is used. Within transactions we need a transaction identifier associated with some transaction manager. In contemporary OO-design we extend the method signatures with a transaction identifier so that this identifier gets passed along. This extension can be achieved by an aspect [5]. Java now also provides a class `java.lang.ThreadLocal` that allows the programmer to associate a variable with the current thread. In Piccola, such environmental information can be put into the dynamic context.

There are a number of container technologies (or frameworks) evolving (We only mention Java related, but Delphi, COM, Smalltalk, CORBA provide similar approaches (?)):

1. Applets: the context is the browser, providing the graphical embedding.
2. Enterprise Java Beans: The context provides transaction management.
3. Java Server Pages: *Scriptlets* are embedded into HTML-like pages. These scriptlets are evaluated in a context which gives access to beans in the web-application.

The notion of context in Piccola can also be used to implement inheritance. The object (i.e. self) would be stored in the dynamic context. In the current implementation of object in the script `classes.picl`, self is defined as a fixpoint. It would be interesting to represent self in the context and compare the result with context calculi [REF Sands ...].

4.5 Scripting Frameworks

Piccola can be used to wrap available frameworks and sell them as individual scripting languages. Such languages are also called little languages [4] as they are targeted to a specific application domains. Examples include `Makefile`, lex and yacc tools.

Here is a list of frameworks and their corresponding scripting tool:

1. Regular Expression Framework: awk, Perl.
2. AWT, Swing: GUI toolkit.
3. XML/HTML DOM Parsers: Web scripting.

Can we take such frameworks, wrap them into Piccola, and get styles with pragmatics similar to that of the original little languages?

4.6 Specification

We need a specification language for Piccola components. A style can the be defined in terms of this language. The language may have similarities to WRIGHT [1], but closely represent π-semantics. In WRIGHT, the semantics is CSP.

A particular question of interest is how to specify and implement styles. For that purpose we establish the following terminology:

- a script is given in a particular style and yields a component.
- a style defines a set of components types, and composition rules for these.

The question is: how can we (formally) specify the behaviour of a component type. One approach is to give a set of black box tests that validate whether a given form belongs to a component type. But there are some properties which are not caught by such a test. For these we may need additional operators over our specification language. For example, we can specify serializability as:

```
a | b | c -> exists b; c; a
```

which should be read as: given services `a`, `b`, and `c` that are invoked concurrently, then there exists a sequence of these actions (e.g `b` then `c` then `a`) which yields the same final state. Furthermore, this applies for any sets of actions that are elements of the acid set, not just on elements a, b, and c. Here the "then it exists" operator cannot be expressed as a test.

5 Piccola - the Language

The language of Piccola may be extended in different ways. This sections describes this extensions. Finally we want to assign a formal semantics to Piccola. The semantics of Piccola is used to derive properties of scripts, applications, components, or connectors *by analysis*.

5.1 Towards first class labels

In the current version of Piccola there are some features implemented as builtin that go beyond the πL semantics. These primitives are `forEachLabel` and `isEmpty` and they are used to introspect and iterate over the labels of a form.

The current approach builds Piccola by layers:

- Piccola. Uses Strings, Numbers, πL-Forms.
- πL-calculus. Defines Forms with plain extension, polymorphic extension [8].
- π-calculus.

With this approach, forms are mapped to the π-calculus. Labels are translated to tuples of names. Forms are primitive π-objects that provide projection. New extended forms are built by factory processes that implement the look-up specification for plain- and polymorphic extension. The crucial design is how to represent labels. Labels need to be elements that are comparable for equality tests.

Another possibility is to introduce strings on top of the π-calculus. Then forms can be encoded as primitive objects, with labels as individual strings. The comparison is handled by the strings. This allows us to iterate over forms and would also enable to inspect forms. However this seems to be a very permissive approach, as it allows to add principle infinitely many labels to a form. E.g. a list may be represented by:

```
val0=a, val1=b, val2=c, val3=..., ... valn=...
```

With this approach, a user could introduce new labels on the fly. He defines abstractions that check for the presence of a label, create a new string-representation of a label, if it would already be present, and continue this process until a fresh label is found. This would go much beyond what current object based systems allow us to do.

A more restricted (but realistic) approach is to only encode labels as some comparable structures in the π-calculus. A user cannot create new labels out of strings. But he can use a `map` like abstractions for forms. The representation of labels is still hidden from the user.

5.2 Direct Semantics

Between the πL-calculus and Piccola, there is an intermediate calculus or language. The semantics of this layer can be expressed directly. This layer may contain services, channels, nested forms, definitions, assignment. The direct (natural) semantics may be given by action semantics [REF Mosses] or by SOS [REF Plotkin...].

5.3 An algebra of Forms

The notion of forms has evolved on top of the π-calculus. However, forms can be explained as an abstract datatype, independent of names. The goal is to separate Piccola into an algebraic part with forms, and into a higher order part. The higher order part provides abstractions and calculation.

5.4 Distribution

Currently Piccola runs on top of a single πL-machine, which in turn corresponds to a JVM or operating system process. There are essentially two possibilities to enable real distributed systems:

- Make the channels location aware. This approach requires location-aware channels to be implemented in the JPict system. Related Work: Nomadic Pict [15]. This means replacing πL in Piccola with a distributed variant. It would allow us to reason about distribution aspects [11].
- Use external abstractions like HTTP servers, a Piccola Script MIME type to set up a framework for distributed Piccola. These abstractions can be embedded into Piccola without extending the language. But the distribution aspect itself cannot be reasoned about.

5.5 Type Systems

Services can be characterized for several aspects. Here is a list of such aspects:

- Blocking vs. non Blocking.
- Referential transparency.
- Spawning one or more new agents.

These aspects can be combined in a more or less obvious way. For example, two non-blocking services can be sequential composed and the resulting service is non-blocking again.

Subtyping, polymorphism, and substitutability are closely related. In general we substitute a component by one of its subtype. Most notions of subtyping are based on interface matching. However, there are other areas where a notion of subtyping would allow us to formally define plug-compatibility, like (non-)blocking services. Related Work: Behavioural Subtyping [7].

5.6 Visualization

Visualization of communicating agents is useful for understanding and debugging parallel programs.

An idea is to represent a program as nested boxes. Such a box represents the context for an agent, the services of the agent itself, or a channel. In addition there are some more graphical elements for queues, waiting agents etc.

Lab View is a product that uses a circuit board-like layout to create program logic. Although the product was originally designed for electronic applications, it is used for many different ap-

plication types. Subroutines are represented by a chip-like rectangle and parameters are fed to these subroutines by “wiring” output from one “chip” to the input terminals of another. Special constraints can be set up to control the sequence and mode of the “chips”.

6 Implementation Aspects

This section describes implementation design and discusses interesting and non-trivial optimization approaches.

6.1 Implementing Forms

Currently forms are implemented as immutable Java objects. Flow analysis would help to reduce overhead and allow more garbage to be reclaimed by the garbage collector. Ideally, we seek efficient and compact implementations for forms in a language without garbage collection like C++. Reference counting on accessible bindings is needed. We could then update forms instead of cloning them when only one agent has access to the binding. For related work on ownership see [13] for an overview.

Piccola is implemented by translating any statement into an agent that communicates three forms with its predecessor and also with its successor agent. These three forms represent root, dynamic, and the current form. However, it would be much more efficient to store identifiers etc. in the πL-context of the agents. For instance:

```
a = 3
b = a
```

is translated to an agent that extends a current and also the root form with a binding for `a`. The second agent reads **`root`** (and dynamic and the current value) and, looks up `a` in **`root`**`.a` and uses this value. A much more compact agent would directly have `a` as a free form variable and use this value.

Assume a script uses several times the same identifier. Now this identifier is repeatedly looked up in **`root`** as a projection. However, since forms are immutable, projections on them always yields the same value. We could store these identifiers in form variables.

Techniques of *partial evaluation* could be applied. In principle, we could pack forms and optimize or inline agents that represent plain functions.

6.2 Garbage Collection

Distributed garbage collection.

6.3 Piccola Machine

Generate JVM Bytecode directly out of Piccola scripts. A script corresponds to a class file. Alternatively, implement Piccola on top of other languages, like Smalltalk (highly dynamic), oblique (distributed by default), or Haskell (functional style).

7 Conclusion

This document gives an overview how Piccola supports software composition. Piccola provides the direct representation of styles. Using Piccola’s formal semantics we can derive properties

for individual components and connectors. These properties are combined to assert certain properties of applications which are implemented in the style.

8 References

[1] Robert J. Allen, “A Formal Approach to Software Architecture,” Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[2] Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr and Alexander Wise, “A Framework for Event-Based Software Integration,” IEEE Transactions on Software Engineering, vol. 5(4), October 1996, pp. 378-421.

[3] Antonio Carzaniga, Elisabetta Di Nitto, David S. Rosenblum and Alexander L. Wolf, “Issues in Supporting Event-Based Architectural Styles,” Third International Software Architecture Workshop, Orlando, Florida, November 1998, pp. 17-20.

[4] James O. Coplien, Multi-Paradigm Design for C++, Addison-Wesley, Reading, Mass., 1999.

[5] 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

[6] Philippe B. Kruchten, “The 4+1 View Model of Architecture,” IEEE Software, vol. 12, no. 6, November 1995, pp. 42-50.

[7] Barbara Liskov and Jeannette M. Wing, “A New Definition of the Subtype Relation,” Proceedings ECOOP’93, O. Nierstrasz (Ed.), LNCS 707, Springer-Verlag, Kaiserslautern, Germany, July 1993, pp. 118-141.

[8] 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.

[9] P. Marsura and D. Riehle, “Design and Implementation of the Java Any Framework,” Ubilab technical report 98.5.1, Union Bank of Switzerland, 1998.

[10] Naftaly Minsky and Victoria Ungureanu, “Regulated Coordination in Open Distributed Systems,” Proceedings COORDINATION’97, David Garlan & Daniel Le Mètayer (Ed.), LNCS 1282, Springer-Verlag, Berlin, Germany, September 1997, pp. 81-97.

[11] Rocco de Nicola, Gian Luigi Ferrari and R. Pugliese, “Klaim: a Kernel Language for Agents Interaction and Mobility,” IEEE Transactions on Software Engineering (Special Issue on Mobility and Network Aware Computing), Catalin Roman and Ghezzi (Ed.), 1998.

[12] Oscar Nierstrasz and Theo Dirk Meijler, “Requirements for a Composition Language,” Object-Based Models and Langages for Concurrent Systems, P. Ciancarini, O. Nierstrasz and A. Yonezawa (Ed.), LNCS 924, Springer-Verlag, 1995, pp. 147-161.

[13] James Noble, John Potter and Jan Vitek, “Flexible alias protection,” Proceedings ECOOP’98, Eric Jul (Ed.), LCNS 1445, Springer-Verlag, Brussels, Belgium, July 1998.

[14] Neal Sample, Dorothea Beringer, Laurence Melloul and Gio Wiederhold, “CLAM: Composition Language for Autonomous Megamodules,” Proceedings of Coordination’99, LNCS 1594, Paolo Ciancarini, Alexander L. Wolf (Ed.), 1999, pp. 291-306.

[15] Peter Sewell, Pawel Wojciechowski and Benjamin Pierce, “Location-Independent Communication for Mobile Agents: a Two-Level Architecture,” technical report, University of Cambridge.

[16] 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.

[17] Jean-Guy Schneider and Oscar Nierstrasz, “Components, Scripts and Glue,” Software Architectures -- Advances and Applications, Leonor Barroca, Jon Hall and Patrick Hall (Ed.), Springer, 1999, pp. 13-25.

[18] Mary Shaw and David Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, 1996.