Towards a formal composition language

Markus Lumpe, Jean-Guy Schneider
Oscar Nierstrasz, Franz Achermann

Software Composition Group, University of Berne,
Institute for Computer Science and Applied Mathematics (IAM),
Neubrückstrasse 10, CH-3012 Bern, Switzerland.
Tel: +41 31 631 46 92
Fax: +41 31 631 39 65
Email: {lumpe,schneidr,oscar,acherman}@iam.unibe.ch

Abstract

When do we call a software development environment a *composition environment*? A composition environment must be built of three parts: i) a reusable component library, ii) a component framework determining the software architecture, and iii) an open and flexible composition language. Most of the effort in component technology was spent on the first two parts. Now it is crucial to address the last part and find an appropriate model to glue existing components together. In this work, we investigate existing component and glue models, define a set of requirements a composition language must fulfill, and report our first results using a prototype implementation of a general-purpose composition language based on the π-calculus.

Keywords: components, component frameworks, composition language, formal glue, π-calculus, modular compiler construction.

Workshop Goals: Analyzing the state-of-the-art of component technology; discussion about composition languages, formal foundations, and implementation techniques.

1 Introduction

Now, more than ever, it is not enough for applications to fulfill only their functional requirements. Modern applications must be *flexible*, or "open" in a variety of ways.

The Concise Oxford Dictionary defines "flexible" as: "that will bend without breaking, pliable, pliant; easily led, manageable; adaptable, versatile; supple, complaisant." Software that "bends without breaking" is *portable* (to different hardware and software platforms), *interoperable* (with other applications), *extendible* (to new functionality), *configurable* (to individual users' or clients' needs), and *maintainable*. These kinds of flexibility are currently best supported by component-oriented software technology: components, by means of abstraction, support portability, interoperability and maintainability. Extendibility and configurability are supported by different forms of binding technology, or "glue": application parts, or even whole applications can be created by composing software components; applications stay flexible by allowing components to be replaced or reconfigured, possibly at run-time.

Currently there is considerable experience in component technology and a lot of resources are spent in defining component models and components. However, much less effort is spent in investigating appropriate glue technology. Specialized scripting languages and 4GLs each provide different forms of glue. Glue can be either highly structured, reflecting a particular architectural style of application composition, or it may be largely unstructured. Structured glue constrains flexibility by limiting the ways in which components may be configured, but supports application maintainability and comprehension by making application architecture explicit [MDEK95]. Unstructured glue is ultimately more flexible, but at the cost of maintainability: Tcl [Ous94], for example, is a simple scripting language that is good for combining arbitrary C functions for simple configuration tasks, but it does not support programming in the large, and, if abused, can lead to large, unmaintainable scripts.

At present, there exists no general-purpose *composition language* that supports both application configuration through structured glue (AKA "connectors" [SG96]), as well as the specification of architectural styles by defining different kinds of connectors. In our view, a composition language would be a combination of an architectural description language and a scripting language and could also serve as the back end to a visual composition environment [NM95].

We are presently prototyping a composition language called PICCOLA ("Pi Calculus-based Composition Language"), whose formal semantics is specified in terms of the π-calculus [Mil89]. In our view, a general-purpose composition language must be formal to support reasoning about component configurations. In particular, a composition language should support the following features, each of which benefit from formal semantics:

- **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. A composition language must be able to instantiate and communicate with active objects.

- **Components:** components are abstractions over the object space [ND95]. Components may be fine-grained, when used to build individual objects, or coarse-grained, when used to build compositions of objects.

- **Glue:** glue mechanisms and operators (connectors) define how associated components interact with each other [SG96]. A composition language must support the specification of new kinds of connectors.
- **Object Models:** a composition language must be able to bridge the gap between different object models. Objects and components that cannot be separated from their individual run-time environments must still be able to communicate. Connectors must achieve the mappings between these object models.
- **Reflection:** glue is often realized by intercepting messages between objects and performing some (reflective) transformations on these messages [Ber94, DBFPD95]. Reflection is also important for exercising run-time control on configurations. Metaobjects are active objects that control the creation, instantiation, and composition of other objects [LSN96] 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. It is, therefore, important to be able to assign a high-level syntax when defining components (as is possible in languages like Smalltalk).

So far we have been using the π-calculus both as a formal specification language for modeling objects, components, and connectors as processes and as metaobjects and as a prototyping language. The π-calculus is admittedly low-level, and working with it is like programming in a kind of "concurrent assembler." On the other hand, it provides the possibility of developing higher-level abstractions which can then be used directly to model the features of interest.

In this paper, we show problems of existing component and glue models and discuss requirements for a general-purpose glue model. We also report on our initial experiences in modeling components and glue with the π-calculus and describe our first steps in developing a platform (called JPICT) for experimenting with the design of PICCOLA. We conclude with some remarks about future work and directions.

2 Problems and requirements

A *software component* is *an element of a component framework*. Although this definition appears to be circular, it captures an essential characteristic of components in general: components have been *designed* to be combined and composed with other components. A software component is *a static abstraction with plugs* [ND95]. Therefore, a component is a software abstraction and can be seen as a kind of *black box entity* that hides its implementation details. It is a static entity in the sense that it must be instantiated in order to be used. Finally, a component has plugs which are not only used to *provide* services, but also to *require* them. A *component framework* is a collection of software components together with a *software architecture* that determines the interfaces that components may have and the rules governing their composition. The essential point is that components are not used in isolation, but according to a software architecture that determines how components are plugged together.

A component is a black box entity with a set of *required* and *provided* services. It is obvious that connections have to be established between required and provided services. This is where the *glue* part of the composition language comes into play. In our terms, the glue acts as a means of communication between the components: it is responsible for component instantiation, configuration, and for correct intercomponent data transmission. Glue can be seen as a kind of *software bus* where components are connected to.

It is important that an application developer should have the choice to specify the control flow of an application either by using a component with a predefined control flow or by using abstractions provided by the composition language itself. We will denote the former kind of application development as *embedding*, the latter as *scripting*.

When combining components written in two different languages, so-called *paradigm clashes* might occur. As a first example, we had the following memory problem when we embedded a C++ program into a Smalltalk application: the Smalltalk application tried to access an object (where it still had a reference to) which already had been destructed by the C++ program. This invalid memory access resulted in a complete crash of the application. An ad hoc solution to this problem is to define a specific glue layer between the two languages which handles shared memory accesses. The disadvantage of this solution is that this glue layer has to be rewritten for each new application using both C++ and Smalltalk.

Second, Delphi is a component-based development environment suitable for Windows and database applications. A major disadvantage of Delphi is that it does not support dynamic linking of components[1] and, therefore, all components of an application have to be already known at compile time. This often results in huge monolithic applications where it is not possible to exchange components at run time.

In order to overcome these kinds of problems, we think that the essential concepts and mechanisms have to be extracted from existing component and glue models, formalized, generalized, and integrated into a general-purpose composition language. It is not yet precisely clear, what the essential concepts and abstractions of composition will be, but in section 1 we have already listed some of the features a composition language must support.

It is clear, however, that we cannot completely separate the glue model from the rest of the composition language; everything has to match each other. Especially the glue and the component model have to work hand in hand, since the glue acts as a kind of telecommunication switch between the components. It is, therefore, not easy to precisely separate the requirements for those two models.

We argue that both the component and glue model of a composition language must be open. The component model needs to be open in order to define new component types (possibly as a composition of other components). Since the glue model has to match the component model, this implies that the glue model also has to be open. But this is not enough. In order to allow a highly readable and compact specification of applications, an application developer should be able to define new abstractions and change the semantics of existing ones. Therefore, the complete language model needs to be open.

It is important for application developers that the component model a composition language offers provides a uniform view of components. It should not be necessary (or even possible) to know in which language a component has been implemented. The interface of a component must contain

[1] This applies only to components completely developed in Delphi.

all the necessary information (e.g., services) in order to use it in a safe and predictable way. This requires that all components follow a specific interface standard. If we want to use components not following this interface standard, we have to map them into the component model. We consider this mapping as a part of the glue model, which is another reason why the glue model needs to be open.

The glue is responsible for correct intercomponent data transmission. If the data format provided by one component does not match the data format required by another component, the glue has to transform the first data format into the second. This has the consequence that i) the glue model must support a basic number of data formats and knows how to transform them into each other, ii) new data formats and transformations can be added to the glue model, and iii) a user must be allowed to plug in special components which will do the necessary transformation.

When building an application, software engineers usually do not think in terms of language constructs, but in a much higher level using architectural guidelines and design patterns. Unfortunately, present day languages only offer very limited support to explicitly represent higher level design elements in the application itself. An explicit representation of design elements will not only result in faster application development, but also ease maintenance and extension of existing applications.

One approach to explicitly represent the architecture of a system in source code is to adapt the concept of *executable connectors* [DR97] to components. Connectors are runtime entities which define a set of rules how connected objects react and interact during external message sending. They change the observable behaviour without modifying the objects themselves. Therefore, connectors can be seen as a kind of higher-level glue for synchronizing and composing objects. We think that connectors are one of the key abstractions for defining *architectural glue* for larger applications and that they are most easily integrated when the composition language supports so-called *message interceptors* [SL97].

If we summarize the requirements discussed above, we can say that a composition language needs abstractions for i) instantiating and configuring components, ii) the definition of control flow, and iii) the specification of component interfaces. We now argue that all these abstractions should be based on a rigorous semantic foundation and be defined using the same mathematical model. What we need is a *composition-calculus*, allowing us to precisely define all abstractions of the composition language.

The π-calculus is a calculus where both communication and configuration are primitives [Mil89] and which has successfully been used to model features of object-oriented programming languages [Wal95]. Our experience shows that the π-calculus is a promising formal foundation for modeling concurrent objects [LSN96] and for the definition of higher-level, reusable synchronization abstractions [SL97]. Other work has proven that the π-calculus can be used to precisely specify the configuration and behaviour of concurrent programs and to deduce behavioural properties [RE94]. Although the π-calculus is rather low-level, it allows for defining and experimenting with higher-level abstractions, which can be easily mapped to the underlying calculus. We believe that the π-calculus is powerful enough to model all abstractions we need for a composition language (especially for glue and scripting), and it is, therefore, a natural step to define our language on top of a π-calculus based formal foundation.

3 Using a Prototype

We have used PICT [PT97], an experimental programming language based on the polyadic mini π-calculus, as an executable specification language to model objects, synchronization abstractions, and interceptors. The main goal of this work was to test if the π-calculus serves as a good formal foundation to define components and glue. It turned out that the language PICT was a good choice in first place, but as the modelled abstractions became more and more sophisticated, we had to realize that PICT was too restrictive and not appropriate any more for our needs. Therefore, we decided to start with an implementation of our own π-calculus based programming language. The domain of this language is primarily component scripting, but it will also be used to evaluate our component-oriented higher level abstractions.

There are several reasons why we started implementing our own language. First of all, we need our own platform in which we are able to control the mapping of language constructs to the underlying calculus and more important to experiment with different type systems. PICT is not an open language and we do not have any hooks which can be used to change the default semantics of the language. But it is crucial for our work to have facilities available that allow us to experiment with new abstractions.

Second, PICT can be seen more or less as a general purpose programming language. But what we need is a language which can be used to glue components supporting the right kind of formal semantics. For this reason our composition language will provide gluing and scripting mechanisms based on the π-calculus.

We started with a dynamically typed language design. Dynamic typing is one of the main features of modern scripting languages [Ous97]. This approach should give us more freedom to compose arbitrary components, but requires more effort in the runtime system. It is not yet clear what is the right type system, but we hope that experimenting with several type systems in PICCOLA will lead to a sound type system suitable for gluing components.

Now, what is the right kind of language or system for a prototype language environment? Should we use a general purpose language or not? Should we use an interpreter or compiler for both the implementation of our system and its execution? Can we use the same system for compilation and execution of programs? We could ask hundreds of these questions.

We have chosen Java as implementation platform. We feel that Java is the right system for prototyping. The Java language is platform neutral and we must not worry about platform specific problems. We can use Java for both compilation and execution of programs. Implementing our prototype language in Java allows us to easily integrate Java objects as rough component abstractions in our system. We have a rich set of predefined classes which makes it easier to implement our language. And the most crucial property of the Java system is that multithreading as well as real distributed execution is available by default.

The current prototype of JPICT, where a first version of PICCOLA is used as the composition language, consists of three parts: scanning and parsing of PICCOLA, translation to a core language, and a runtime system for the core language.

For the scanning and parsing part we have used an object-oriented technique. The use of tools like **Lex** and **Yacc** for Java do not seem to be appropriate for us. If we have to change a part of the language, then it is more difficult to manage all necessary modification when using these tools. In our object-oriented approach we assign each building block of the language a corresponding class. Currently, we have implemented the lexer and parser in one piece, but we are experimenting with a new approach which allows us to add new language constructs to the compiler more or less at runtime. We call this approach *modular compiler construction.* This approach uses metaclasses to configure the compilation. Unfortunately, we currently do not have a good support for this in Java.

The core code consists of the basic π-calculus constructs: sending process, receiving process, create a new channel, and parallel composition of processes. Each of these constructs is implemented in one class. The translation phase maps higher level abstractions like assignment, function definition, and function calls to the corresponding core constructs.

The runtime system heavily uses threads. A π-process is simply mapped to a Java-thread. The synchronization of threads is achieved by calling the corresponding `put` and `get` methods of the channels used in the thread communication.

Unfortunately, a 1:1 mapping of π-processes to a Java thread is not the best solution. It turned out that even for simple programs with some objects more than 100 threads were created. It is not yet clear, what will be the optimal number of threads, but as more threads are active, the execution speed dramatically goes down due to the thread specific overhead.

Fortunately, our first results show that our approach works. Even with the penalties mentioned we got results within days. Implementing the first version of the system took us only two weeks. The first version implemented only the core language set, but is was a solid base for further extensions. A next version was enriched with higher level abstraction like assignment and functions. Additionally, we have implemented a first version of channel interceptors [SL97].

We still have some problems with our implementation. Especially the chosen mapping of a π-process to a Java thread has to be changed in future versions. But the results are promising and we plan to extend our system with additional features like reflection support and a powerful object model. Furthermore, we will also add a visual development environment as a composition workbench.

4 Conclusions and future work

When we talk about components we tend to ignore fact that components are more or less useless without an appropriate framework and gluing mechanisms. Furthermore, we argue that a *single component* is a contradiction in terms. When we talk about components we have to be aware that a specific component always belongs to a specific component framework.

It is the component framework which defines the software architecture of the components and the possible gluing mechanisms. The framework defines the kinds of interfaces that components may have and the rules governing their composition.

What all component models have in common is that they define components more or less as black box entities. Unfortunately, each component model also defines its own composition mechanisms, which usually cannot be applied to components of other component frameworks. Therefore, it is necessary to find basic mechanisms that allow to compose arbitrary components even if they belong to different component frameworks.

Our work addresses exactly this issue. We are trying to combine modern scripting mechanisms with component technology. The use of the π-calculus as the formal semantics seems to be the right direction. A composition language must provide means for configuration and execution of programs. Configuration is described by setting up components as parallel processes. The execution of this configuration takes place with the help of communication and process reduction. The application programmer does not have to use the primitive π-calculus constructs; our composition language provides all necessary higher level abstractions. Furthermore, we intend to add support for meta-level programming. With this metalevel we hope to get a system which is easy extendible in order to add new (possibly not yet known) component models.

We have learned from our modelling of various object-oriented abstractions in the π-calculus and the design of PICCOLA that the π-calculus serves as a good formal basis for our work. But as we have already mentioned, working with the π-calculus (PICT) is like programming in a concurrent assembler. We intend to develop an extented π-calculus which we call a *composition-calculus*. This composition-calculus will constitute the formal basis of our composition language. The definition of this calculus is driven by two directions: i) the application of the π-calculus to find suitable higher level abstractions and ii) the formalization of language constructs which constitute the core set of the composition language.

An important aspect of our next work is the evaluation of different type systems. Currently we use dynamic typing in our system. But dynamic typing also means that only a predefined set of types is allowed to be used for components. This works well in most cases, but when we add a new component model with a richer type system, it will break the existing type system. It is not acceptable to reduce the supported types only to some basic types like Integer and String. We need greater flexibility here. We hope that the metalevel of the composition language will help us to integrate also new types and type representations.

Beside the language design we also experiment with different compilation techniques. As already mentioned, we think that real object-oriented compiler construction is the right way for language prototyping. Unfortunately, Java does not support a rich set of metalevel programming. We have tested other platforms (e.g., Delphi, C++) that have a better support for modelling metaclasses. With these systems we got first results which are promising. We have implemented the lexical and syntactical analysis only by using metaclasses. These metaclasses represent the building blocks of the language. When the compilation starts, all metaclasses are registered in the system. This allows us to change the language and the compiler quite easily simply by adding or removing metaclasses. Each metaclass is fully responsible for the language construct it is implementing. Therefore, a metaclass does not change anything within the compiler. This approach is still in an experimental state and will be presented in future work.

Our prototype language compiler does not address efficiency, but flexibility. Furthermore, we have two systems in one: an interpreter as well as a compiler. We only need to change the mode of our system in order to switch between interpretation and compilation.

When do we call a software development environment a *composition environment*? A composition environment must be built of three parts: i) a reusable component library, ii) a component framework determining the software architecture, and iii) an open and flexible composition language. Most of the effort in component technology was spent on the first two parts. Now it is crucial to find an appropriate model to glue existing components together. In our opinion, the π-calculus is a powerful system in order to fill the missing gap in an elegant and natural way.

Acknowledgements

We thank all members of the Software Composition Group for their support of this work, especially Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar.

References

[Ber94] Lodewijk Bergmans. *Composing Concurrent Objects.* PhD thesis, University of Twente, NL, June 1994.

[DBFPD95] 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.

[DR97] Stéphane Ducasse and Tamar Richner. Executable Connectors: Towards Reusable Design Elements. In *Proceedings ESEC '97*, 1997. to appear.

[LSN96] Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Using Metaobjects to Model Concurrent Objects with PICT. In *Proceedings of Langages et Modèles à Objets '96*, pages 1–12, Leysin, October 1996.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software Architectures. In Wilhelm Schäfer and Pere Botella, editors, *Proceedings ESEC '95*, LNCS 989, pages 137–153. Springer, September 1995.

[Mil89] Robin Milner. A calculus of mobile processes, part I+II. Technical Report ECS-LFCS-89-85, Computer Science Department, University of Edinburgh, UK, 1989.

[ND95] 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.

[NM95] 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.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Ous97] John K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. White Paper, May 1997.

[PT97] 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.

[RE94] Matthias Radestock and Susan Eisenbach. What do you get from a pi-calculus semantics? In *Proceedings of Parallel Architectures and Languages Europe (PARLE '94)*, LNCS 817, pages 635–647. Springer, 1994.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, April 1996.

[SL97] Jean-Guy Schneider and Markus Lumpe. Synchronizing Concurrent Objects in the Pi-Calculus. In *Proceedings of Languages and Models with Objects '97*, Brest, October 1997. to appear.

[Wal95] David J. Walker. Objects in the Pi-Calculus. *Information and Computation*, 116(2):253–271, 1995.

Biography

Franz Achermann works as a research and teaching assistant in the Software Composition Group at the Institute for Computer Science and Applied Mathematics of the University of Berne, Switzerland. His main interests are in the field of object-oriented programming and languages and formal methods for software engineering. He completed his M.Sc. at the University of Berne in 1995. He worked in industry in the area of databases before joining the group at the beginning of 1997.

Markus Lumpe works as a research and teaching assistant in the Software Composition Group at the Institute for Computer Science and Applied Mathematics of the University of Berne, Switzerland. He is interested in the design and implementation of object-oriented and component-oriented languages using fully object-oriented compiler construction techniques. He completed his M.Sc. in 1990 at the University of Dresden. He worked in industry in the area of object-oriented system design and implementation. In fall 1994, he joined the Software Composition Group in Berne.

Oscar Nierstrasz is Professor of Computer Science at the Institute of Computer Science and Applied Mathematics of the University of Berne where he leads the Software Composition Group. He is interested in all aspects of component-oriented software technology, and particularly in the design and implementation of high-level specification languages and tools to support reusability and evolution of open applications. He completed his M.Sc. in 1981 and his Ph.D. in 1984 in the area of Office Information Systems at the University of Toronto. He worked at the Institute of Computer Science in Crete (1985), and in the Object Systems Group at the Centre Universitaire d'Informatique of the University of Geneva, Switzerland (1985-1994).

Jean-Guy Schneider works as a research assistant in the Software Composition Group at the Institute for Computer Science and Applied Mathematics of the University of Berne, Switzerland. His main interests are in object-oriented and parallel programming, scripting languages, and the definition of formal methods for component-oriented software engineering. He received his M.Sc. in Computer Science from the University of Berne in 1992 and has been working for the group since 1994.