Workshop on Composition Languages (WCL 2001)

Proceedings, Vienna, Austria
September 11, 2001

Edited by
Jean-Guy Schneider and Markus Lumpe

Co-located with the
8th European Software Engineering Conference
and the
9th ACM SIGSOFT Symposium on the
Foundations of Software Engineering

Workshop on Composition Languages (WCL 2001)

Proceedings, Vienna, Austria
September 11, 2001

Edited by
Jean-Guy Schneider and Markus Lumpe

Co-located with the
8th European Software Engineering Conference
and the
9th ACM SIGSOFT Symposium on the
Foundations of Software Engineering

Copyright ©2002 by Swinburne University of Technology, Melbourne, Australia.
All rights reserved.

ISBN 0 85590 785 1

The copyright of this proceedings as a collection of papers belongs to Swinburne University of Technology. The copyright of the individual papers in this proceedings belongs to their authors.

The paper in this proceedings reflect the author's opinion and are published as presented and without changes. Their inclusion in this publication does not necessarily constitute endorsement by the by the editors or Swinburne University of Technology.

Additional copies may be ordered from

Jean-Guy Schneider
School of Information Technology
Swinburne University of Technology
P.O. Box 218
Hawthorn, Victoria 3122
Australia
Email: jschneider@swin.edu.au

Preface

This is the proceedings of the first workshop on *Composition Languages (WCL 2001)*, held in Vienna, Austria on September 11, 2001. The workshop was affiliated with the *8th European Software Engineering Conference (ESEC 01)* and the *9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*.

The objective of the workshop was to provide a forum to address problems concerning the design and implementation of higher-level languages for component-based software development. More precisely, the workshop primarily aimed at focusing on language aspects, and not on component-based systems in general. Furthermore, the event was intended to initiate discussions about various theoretical and practical issues related to composition languages.

This proceedings summarizes the goals and outcomes of the workshop and includes all accepted position statements. It can be expected that the results of the workshop will assist both researchers and practitioners in outlining collaborative topics for further exploration. Furthermore, we very much hope that the results will also advance the participants' understanding of all aspects related to component composition and serve as a checkpoint in the endeavour to close the gap between research and practice.

The papers in this proceedings are position statements and are included here based on a formal peer-reviewing process. All submissions have been reviewed by at least two independent reviewers (not necessarily by the organizers themselves) and have been accepted based on (i) the quality, (ii) the relevance to component-based development, and (iii) a given set of possible discussion topics.

The proceedings of this workshop and the slides of all presentations are available electronically from the workshop's web-page at the following URL:

http://www.cs.iastate.edu/~lumpe/WCL2001/

We would like to thank Volker Gruhn for encouraging us to organize this workshop at ESEC/FSE, Harald Gall for his help with local arrangements in Vienna, Franz Achermann for his assistance in reviewing all submissions, Barbara Hurst for the proofreading of the camera ready submissions, and all workshop participants for their contributions.

Jean-Guy Schneider and Markus Lumpe
Workshop Co-Chairs

Table of Contents

Workshop Summary 1
Markus Lumpe, Jean-Guy Schneider

CoML: Yet Another, But Simple Component Composition Language 11
Dietrich Birngruber

HotAgent Component Assembly Editor 25
Ludger Martin

Partial Evaluation of Inter-language Wrappers 35
Nathanael Schärli, Franz Achermann

Reflections on the Anatomy of Software Composition Languages and Mechanism . . . 45
Michel Chaudron

Workshop Summary

Markus Lumpe[1], Jean-Guy Schneider[2]

[1] Department of Computer Science
Iowa State University
113 Atanasoff Hall
Ames IA 50011, USA
Tel: +1 515 294 2410, Fax: +1 515 294 0258
lumpe@cs.iastate.edu

[2] School of Infomation Technology
Swinburne University of Technology
P.O. Box 218
Hawthorn, VIC 3122, AUSTRALIA
Tel: +61 3 9214 8189, Fax: +61 3 9819 0823
jschneider@swin.edu.au

In the following, we summarize the goals, presentations, and outcomes of the first Workshop on Composition Languages (WCL 2001). This summary includes the most important thoughts expressed during the workshop sessions and outlines avenues for further exploration.

Workshop Goals

Component-oriented software is an increasingly accepted technology where applications are built as compositions of interacting software components. The objective of this technology is to build applications by simply plugging together elements from a collection of reusable software components, and aims at the production of high-quality software systems with shorter and more cost-effective development cycles.

In the recent years a considerable effort has been spent to define appropriate collections of components (such as CORBA, COM, JavaBeans, Enterprise JavaBeans, and .NET). However, much less effort has been spent in investigating appropriate composition languages and environments, which allow application developers to express applications flexibly as compositions of components. In fact, current practice shows that existing composition environments do not fulfill all expectations and that more research effort is required to achieve the desired flexibility and extensibility for building component systems.

The situation today is that most available composition environments focus mainly on special application domains and offer at best rudimentary support for the integration of components that were built in a system other than the actual deployment environment. Furthermore, these systems do not enforce a clear separation of computational elements (i.e. components) and their relationships, which is needed to address the flexibility and maintainability of component-based systems. The reason for this situation is not only the lack of

well-defined (or standardized) component interfaces, but in particular the ad-hoc way the semantics of the underlying language models are defined.

Various workshops have been held to address problems of existing component technology, but most of these events focused on aspects of component-based programming related to implementation of components, component frameworks, and systems. However, there is an emerging need to specifically address the problems of existing composition systems by focusing on important aspects of the design and implementation of composition languages and environments.

This briefly summarizes the context in which the goals of this workshop were originally defined. More specifically, we wanted to emphasize important issues of (i) the design and implementation of higher-level languages for component-based software development, (ii) approaches that combine architectural description, component configuration, and component composition, (iii) paradigms for the specification of reusable software assets, and (iv) expressing applications as compositions of software components. We particularly encouraged authors to submit position statements focusing on formal aspects of these issues. The following list of suggested topics was included in the original call-for-submissions:

- Higher-level abstractions for composition languages,
- Programming paradigms for software composition,
- Syntactical issues to enhance readability (e.g., making application architectures explicit in the source code),
- Type systems for composition languages,
- Portability issues (i.e. design/implementation strategies for cross-platform development),
- Interoperability issues (i.e. gateways to other systems/languages),
- Implementation techniques for composition languages,
- Scalability and extensibility of the language abstractions,
- Formal semantics of composition languages,
- Case studies of composition language design,
- Case studies of applications using composition languages.

Questionnaire

To assist the organizers in defining the topics of discussion in advance, the authors were asked to include a list of critical questions or some, perhaps provocative, statements in their submissions. The following list summarizes the submitted statements and questions:

– What are distinguishing requirements for a composition language with respect to other languages?
– What new kind of problems appear when we talk about composition that did not appear when reusing object-oriented frameworks?

- Why all this hype about composition when COM and friends work perfectly fine in industry?
- Current component platform developments tend to unify operating system independence and programming language independence by compiling into a virtual machine such as Sun's Java byte code or Microsoft's Intermediate Language. Similar techniques are also used by interpreted languages such as scripting languages. The question is: are cross platform composition languages needed in the future if the integration is already solved by the component platform?
- From our perspective, XML is "trendy." Are the days over where programmers assembling components had to know different composition languages (and syntax)? Is the industry forcing the majority of their developers to use a single (meta) language such as a unified composition language? However, we personally hope that we never have to write XML code manually – because it is not readable and thus tool support play a more important role than composition languages.
- The component-based building of systems can only succeed if a component framework is used that is based on the integral design of a component model and a composition model. In other words, it is of utmost importance that both a component model and its corresponding composition model are developed together.
- Composition languages are the 'language interface' to composition mechanisms – the surface of the mechanisms underneath. For composing software components at higher levels of abstraction, the corresponding composition mechanisms may consist of mechanisms that are of a higher order of (algorithmic) complexity than the mechanisms we are currently considering for software composition. These mechanisms heavily influence the ease of 'composability' – this is not just a language issue.
- Components need to provide separate interfaces for 'plugging' them into a system and for 'playing with' other components in this system.
- The decoupling of the transfer of data from the transfer of control leads to a looser coupling than the joint transfer as it is used in RPC-based interaction styles.
- Coordination languages have focused on the management of control flow and generally ignored the possibility of separately managing data flow.
- The reusability of components can be increased if components are 'style-less' and the interaction style(s) between components are solely determined by the involved composition mechanism(s). Clearly, this requires that components are designed in a consistent manner and that the stakeholders of a particular system trade design flexibility for increased (de)composability.
- Is it possible to find all important characteristics of visual programming with a simple component model?
- What is a suitable visualization of components?
- How can we prove the correctness of visual code?

Workshop Program

The workshop was organized into two morning sessions mainly devoted to the presentations of the accepted position statements and two discussion sessions in the afternoon. Based on the number of participants, it was decided to deviate from the original idea of splitting the participants into focus groups and only hold one discussion group.

09:00 - 09:30	Welcome and introduction
09:30 - 10:30	**Session 1: Languages**
	DIETRICH BIRNGRUBER: *CoML: Yet Another, But Simple Component Composition Language*
	LUDGER MARTIN: *HotAgent Component Assembly Editor*
10:30 - 11:00	Coffee Break
11:00 - 12:00	**Session 2: Mechanisms**
	NATHANEL SCHÄRLI and FRANZ ACHERMANN: *Partial Evaluation of Inter-language Wrappers*
	MICHEL CHAUDRON: *Reflections on the Anatomy of Software Composition Languages and Mechanisms*
12:00 - 12:30	Finalization of discussion topics for afternoon sessions
12:30 - 13:30	Lunch Break
13:30 - 15:00	**Session 3: Discussion I**
15:00 - 15:30	Coffee Break
15:30 - 16:45	**Session 4: Discussion II**
16:45 - 17:00	Conclusions of workshop, suggestions for future events

Presentations

This section briefly summarizes the four presentations given during the two morning sessions of workshop.

The first presentation was given by Dietrich Birngruber who presented CoML, a platform independent composition language based on XML (refer to page 11 for full version of the submission). The main motivation behind his work was the definition of a notation for exchanging component compositions between different development tools. As additional requirements, the notation also had to enable recursive component compositions and reduce the learning process for application developers using the notation. The CoML language was introduced and the corresponding meta-tags were explained using an example of a JavaBeans composition. He concluded his talk with current shortcomings and future development activities in the context of CoML.

The second presentation in this session was given by Ludger Martin who presented HotAgent, a prototype of a visual component assembly editor as well as the underlying component model (refer to page 25 for details). The assembly editor is based on a graph metaphor, contains query facilities to search for components in the component repository, and uses color to distinguish between different composition mechanisms. The textual composition language (based on a Smalltalk-like notation) incorporated into HotAgent which is used to store a composition of components was also presented. He concluded his presentation by raising the question whether visual composition environments can be used for large-scale real-world projects and how a visually composed set of components can be efficiently tested.

Common to both presentations in the languages session was the fact that even if visual development environments are used for software composition, there still is the need for a textual representation. Such a textual presentation has to provide the means to make a composition persistent, reusable, and to enable the interaction with other development tools. Both authors preferred an XML-based approach.

The first paper in the mechanisms session was presented by Franz Achermann (a joint submission with Nathanel Schärli; refer to page 35 for full version). In his presentation, Franz Achermann focused on the efficient representation of inter-language wrappers, a technique used to enable inter-language bridging to access components outside the current composition environment.

The problem was stated in the context of Piccola, a small composition language defined in terms of a process calculus, where almost all abstractions are defined as external components. He gave a brief introduction into the language, its concepts, and the underlying Java-based realization. The concept of inter-language bridging was shown using Piccola scripts, which can be optimized by separating side-effects from services. He concluded his presentation with comments about future work in optimizing inter-language bridging.

Michel Chaudron gave the final presentation of the workshop and reflected on the anatomy of composition mechanisms in existing languages (full version of the submission available at page 45). In particular, he focused on the relationship between data flow and control flow at various levels of abstraction and illustrated problems that may occur when the two flows are (too) strongly coupled.

To solve the induced coupling problems, he proposed an approach in which a component implementation is completely separated from any interaction style (i.e. a component is a purely computational entity). In order to enable the composition of such components, independently designable interaction-style aware composition wrappers have to be used that add the desired interaction style to the resulting composite.

He concluded his presentation with the proposal that a software composition language should provide mechanisms for integrating various forms of control flow, data flow, and information models. He suggested, however, that a composition language should not define a fixed set of models per se.

In this session, both presentations focused on selected aspects of the efficient representation of compositional abstractions. Although different problems were addressed, the essence was the same – composition of software components requires new abstractions, which need to be supported by an underlying composition model and the language used to specify the composition. The concrete requirements on the design and implementation may range from the support of a lazy evaluation scheme, as in the case of inter-language bridges, to independently designable composition wrappers that provide the means for late interaction style specification.

Discussion Sessions

After the presentations and some initial discussions, a list of topics for the two discussion sessions was finalized (based on the questions and statements submitted by the authors prior to the workshop as well as discussion topics arising during the presentation session). This list contained the following topics:

- What are the distinguishing features of architectural description languages, composition languages, coordination languages, and scripting languages?
- Is it possible to define a clear separation of data flow and control flow in component-based software systems?
- Is it possible or even desirable to consider composition languages independently of components and their corresponding component models?
- Do composition languages need computational abstractions?
- Do present-day composition techniques scale up for larger-scale systems and if so, how?
- Where do architectural description languages fall short?
- How does the development of embedded systems benefit from component-based software development?
- Can we define an open component model?
- Is it desirable to have support for both static and dynamic composition mechanisms in a composition language?
- Do we need support to explicitly specify properties of component interaction relationships?
- If needed, what are the appropriate representations of asynchronous composition mechanisms?
- Can we represent non-functional properties in a composition language?

Due to time limitations, it was not possible to address all the topics given in the list above; only time for the first five topics was available. However, there was a consensus amongst the participants that the remaining topics should be addressed in future research and/or considered for further discussions. In the following, we will briefly summarize the results of the discussion sessions.

The main focus in the discussions was on the definition of the term "composition language." In fact, it was not immediately clear what the distinguishing feature of such a language should be and what the differences between a composition language, an architectural description language, a scripting language, and a component integration language were. In the same context the workshop participants also tried to identify the difference between architectural and component specification.

A first characterization suggested that a composition language is a kind of *umbrella* over many activities in the software development process: specification, architectural definition, searching for components, glue, composition, and deployment. Hence, a composition language is not just *one* language, but consists of a *system* of languages.

A second attempt to characterize composition languages was to analyze activities of the development process at different levels of abstraction:

Specification Level	Architectural Description Language
Selection Level	(Ensembles)
Integration Level	Composition Language
Implementation Level	C/C++, Java etc.

A consequence of this characterization is that an architectural description language describes a composition of components at an abstract level and the concrete composition (i.e. the run-time components) is defined at the integration level. However, it was not clear what kind of language is required at the selection level (denoted as ensembles above). The discussion could not give a definite answer to this question, as a simple query language (such as SQL) would not have enough expressive power to bridge the gap between architectural specification and integration. Therefore, the original characterization of composition languages was refined as follows:

Specification Level	Architectural Description Language
Selection Level	Composition Language
Integration Level	Scripting Language
Implementation Level	Programming Language

This characterization formed the end point of the discussion about what a composition language should be and was probably the best consensus, which could be reached within the time limit given.

The second discussion section was devoted to selected aspects of component-based software development. In particular, the focus was on component models and their appropriate representation. All participants agreed on the following working hypothesis

Component Model = Component Standards + Component Framework

where a component standard defines “what it takes to be a software component” and a component framework defines the corresponding services and interaction mechanisms, respectively.

First, component frameworks were addressed. The main issue was whether we can separate data flow from control flow or not. The common understanding was that a component framework has to provide some support to configure and change the control flow “style”, respectively. What are the advantages and drawbacks of such an approach? The participants could not come up with a satisfying answer here. The main problem is that current component models and frameworks do not support such a separation. In fact, a concrete set of components may impose certain restrictions on data and control flow. Therefore, a fully customizable approach may not be possible or even desireable.

But what about “standard” components? The working hypothesis clearly states that a particular component model is built of a set of standard components. In fact, both the component framework and the standard components constitute the rules to build composites. Components do not exist in isolation, but in context with a software architecture that determines how components can be plugged together and more importantly how components can interact. As a result, the participants concluded that that specifications and changes must occur in a consistent manner within a component assembly. Therefore, whether or not the specification of control and data flow can be separated or at least least deferred until deployment time depends on (i) the software architecture, (ii) the particular interaction style, and (iii) the constraints imposed by the desired behavior of the resulting composite.

As far as language support is concerned, in the discussion there was a clearly recognizable tendency: to best enable software composition, we need to move towards an unifying closed component model. Such an unifying model would make language design and language support much easier. As a consequence, composition languages need to evolve into “integration languages.”

Summing up the afternoon sessions it can be noted that a very lively discussion was going on and that various (sometimes quite contradictory) points of view were brought forward. It has become clear that the terms and concepts related to component-based software technology in general and composition languages in particular have not yet matured and that further discussions are needed for clarification. We believe, however, that this workshop was a first step in this direction.

Conclusions, Outlook

Concluding, we are able to state that, despite the small number of participants, the first Workshop on Compostition Languages was quite a successful event that was appreciated by all participants. On the other hand, it was obvious that the issues we originally wanted to address in this workshop have not been exhaustively discussed and that further effort needs

to be spent in clarifying the involved terms and concepts, in defining language models and environments suitable for software composition, and in investigating the applicability of these languages and environments for (larger-scale) industrial applications.

To address these issues further, we believe that it is necessary to enhance and encourage research collaborations between various disciplines (such as component models and frameworks, software architectures, concepts of programming languages, etc.) and bridge the gap between these disciplines. Therefore, we plan to organize further events especially dedicated to the topic of composition languages.

CoML: Yet Another, But Simple Component Composition Language

Dietrich Birngruber

Johannes Kepler University Linz,
Institute for Practical Computer Science
System Software Group
A-4040, Linz, Austria
Tel: +43 732 2468 7133, Fax: +43 732 2468 7138
birngruber@ssw.uni-linz.ac.at

Abstract. Components offer various advantages, such as platform and tool independence but composition languages do not address the same independence. We present a platform independent component composition language called Component Markup Language. CoML offers abstractions for component composition and for meta information. We envision different usage scenarios for CoML - as a resource script or input for software engineering tools, such as development tools, GUI builders, architecture visualizer tools etc.

1 Introduction

The dream of composing software out of prefabricated artifacts is as old as the term software engineer [7]. Despite this long time (or perhaps because of it) the community has not agreed upon a single definition for software component. We have agreed upon not to introduce yet another component definition but to use the definition from [17]:

> A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

With the Component Markup Language (CoML) we try to provide a tool for the last part of this definition: a tool for component *composition by third parties*. Before we present CoML itself in Section 4, we look at general requirements for a composition language in Section 2. We describe our design decisions in Section 3 and explain why we did not use existing approaches in Section 5. We conclude with directions for future work in Section 6.

2 Requirements for a Composition Language

This section briefly sketches requirements for a composition language. Some of the listed requirements are based on the work of Nierstrasz and Meijler (see [11]).

1. *Connection-oriented composition*: components need a way to describe how they are "plugged" or "wired" or composed together. Without this wiring the components can not communicate. An effect of components is that, technically speaking, they can be wired not only at composition-time but also at run-time. This requires a "very late" binding mechanism. Examples of connection-oriented composition mechanisms are events or pipes and filters.
2. *Aggregation-based composition*: besides connection-oriented composition, some component models offer mechanisms for aggregating components to a higher- level component. Examples of aggregation-based composition mechanisms are "container" frameworks or binary aggregation. The framework approach allows the aggregated components to be embedded at run-time or at design-time. The components define a containment hierarchy, such as the containment hierarchy of visual JavaBeans which adhere to the Swing framework [18]. The binary integration approach focuses on implementation reuse, such as the aggregation of COM components. It integrates the interfaces of an aggregated component into the interfaces of the enclosing component, therefore reusing the code of the aggregated component.
3. *Composition code reuse*: besides reusing components it is desirable to reuse already written composition code snippets as well, as long as the reused code results in a component.
4. *Recursive component construction*: the result of composed components is a "new" composed component or an "adjusted" component. The new composed component contains other components. The result of an adjusted component is the customized component itself.
5. *Component model and platform independence*: different component models should be addressed. This is an essential requirement of our project. Without the independence of particular component models (JavaBeans, .NET, CORBA) and of operating systems we would have to provide a component and platform specific language as well as an execution environment for each component model and platform.
6. *Minimize the need for new tools*: software tools are used in every software development process. It seems that without proper tool support, developers are not going to accept a new composition language. A new language often leads to the need for new tools and, therefore, the introduction of a new language increases the effort for presenting and implementing new ideas.
7. *Component framework support*: components have to interact with existing frameworks. In this section, we use the term framework for both object-oriented [5] and component frameworks [17]. For altering or extending their behaviour, both kinds of frameworks provide hot spots (such as abstract classes and interfaces) where new classes or components can be plugged into the corresponding framework [14]. Frameworks are widely

accepted and used tools. The problem is that some components may violate the preconditions of a hot spot syntactically and/or semantically.

8. *Reduce learning process*: the process for learning a new language should be as short as possible. Using already established syntactic and semantic conventions can help to reduce the learning process.
9. *Extensibility support*: it should be possible to extend the language primitives easily in order to support alternative component models.
10. *Versioning support*: components are maintained and (hopefully) improved over time. It is necessary to distinguish between the different component releases because the behavior of a new release may have changed and existing clients break. Some existing component platforms try to address this problem by versioning components and by defining compatibility rules [13]. The component platforms define which versions are compatible and can be used without breaking existing clients. A composition language should address this aspect, too.

3 CoML Design Principles

For our CoPL (Component Plan Language) project we needed a platform independent language which describes the composition of components. CoPL is used both for scripting a CoML generator and for describing component compositions. However, CoPL contains so-called "decision spots" for selecting an appropriate component implementation when a CoPL script is processed by the CoML generator. The output of the CoML generator is a CoML script. CoPL is based on previous work on JavaBeans compositions [3].

In the previous section we presented requirements for a composition language. CoML has been designed to meet many of this requirements - but not all. In this section, we present general design decisions for achieving these requirements which we considered absolutely necessary in the context of the CoPL/CoML project.

XML based. Platform independence is a crucial aspect of CoML. We considered using a binary format instead of a text based format in order to increase the processing speed. However, the description itself shall be transferable to different operating systems and different tools should be able to use our component composition descriptions. Our intention primarily is to have CoML scripts created and used by software tools rather than by human beings. The promising wide acceptance of XML [20] and the integration into software tools convinced us to use a XML based syntax for CoML. Because of the nature of XML, CoML scripts consist of XML elements. These elements define a hierarchy or tree which resembles the component composition. The structure of a hierarchical or aggregation-based composition ideally fits with XML. This design decision allows CoML to fulfill the requirements 5, 7 and 8.

Declarative language. CoML can be used to describe a set of components and their wiring, i.e. to describe a component composition. However, CoML cannot be used to specify the functionality of a component or how it has to be implemented. The latter tasks is left for programming languages such as C++, C#, or Java. In order to understand the roles of different programming languages and techniques during a component oriented development process, we briefly present a typical component development and component usage process (also refer to [19]):

1. *Component construction time*: component developers design and implement the components. At present, mostly object-oriented programming languages are used (a good distinction between an object-oriented library and components can be found in [17]). The binary software components along with a proper documentation are packaged and delivered to third parties - to the component assemblers.
2. *Component composition time*: component assemblers combine the acquired components into an executable application. At present, different technologies for component composition are available. The component assembler uses traditional programming languages, scripting languages [1], and/or visual programming techniques [15]. Component assemblers need not be humans. Software tools such as code generators (e.g., GenVoca generators [2]) can assemble components as well. However, software generators need additional instructions for composing the appropriate components or the generators directly implement a particular domain model.
3. *Run-time*: users execute the composed application. Based upon the output format of the component composition, the application is either interpreted or executed - but the result is the same: a running application.

Scripting languages play a major role in step 2, the component composition time. A CoML script represents the concrete composition structure of a set of components. Typically, a CoML script describes an executable part and not the full application itself. We assume that CoML is used as an independent composition description, which will be further processed by different development and assembly tools. Similar to a C++ resource script which describes the layout of a GUI dialog, CoML describes a component composition and can be exchanged between different tools. However, one requirement of software components is that they are executable units and, therefore, CoML scripts must be considered as executable entities as well.

Meta information. For versioning support or to identify the component library to be used, additional information is required beyond just the composition itself. The problem is that particular information is needed for one target component platform which might be meaningless for other platforms.

As an example, Java Beans components are executed by a Java Virtual Machine (JVM) and use the Java core library (version 1.2). The Java core library shipped with a new JVM release has changed, and the used beans are not compatible with the new version. Intentionally, the Java component platform does not support versioning. Relying on the

target component platform to handle this problem is not sufficient, because not all platforms support versioning or if they support versioning, the different target component platforms use a different version numbering system etc.
CoML addresses this problem by separating the composition declarations from the platform specific information and other meta information. This meta information is extensible and may be used by the software tools which process CoML scripts. This design decision allows CoML to fulfill the requirements 9 and 10.

Connection-oriented programming model. We need a composition description which allows us to describe the composition of components from various component models. The major component platforms CORBA [12], JavaBeans [16], and .NET [8]/COM [9] support a connection-oriented programming model [17]. A common implementation of the connection-oriented programming model is the event mechanism. The event mechanism uses an event source which fires an event when certain conditions are met (like state change), an event sink which reacts upon the fired event, and event data encoding information about the event. This design decision allows CoML to fulfill requirement 1.
Besides using events as the major connection mechanism, we had to define minimum requirements for component models scriptable by CoML:

- a component is accessed via interfaces,
- a component provides strongly typed interfaces,
- a component interface offers methods and/or properties,
- a component uses an event-mechanism as its primary "wiring" technique for plugging components together, and
- the component life-cycle is split into design-time and run-time and, therefore, CoML must make the distinction between wiring components and creating component instances, respectively.

The component platforms CORBA, JavaBeans and .NET/COM fulfill these requirements.

4 CoML Language

In this section we introduce the XML based language CoML. Right at the beginning, we present a list of CoML elements (refer to Table 5) and an example. The elements or CoML specific tags are explained in more detail in the following subsections.

We tried to keep the language as simple as possible and thus CoML has a limited amount of elements. CoML provides elements for

- describing which component library and version are used and additional component platform dependent information (see `meta` element),
- describing a component itself by stating the desired interface and implementation (see `components` and `component` elements),

Element name	Description
coml	root element
meta	contains the meta information
components	list of composed components; contains <component ...> elements
component	declares a component
property	sets or gets a property of a specific component
method-call	calls a method
on-event	describes an event binding; connection-based composition
add	adds another component; aggregation-based composition

Additionally, each element has several attributes.

Table 1. CoML elements (tags)

- describing the customization of a component (see `property` and `method-call` elements), and
- describing the component composition - using a connection-oriented programming model and component aggregation (see `on-event` and `add` elements).

Except for the `meta` element, every CoML element can be regarded as an (abstract) method call or expression. However, the elements abstract from the target platform and typically they are processed sequentially. Generally, the child elements are the arguments of the enclosing parent element. The syntax of some CoML elements is influenced by IBMs Bean Markup Language.

Example

This CoML snippet shows a simple composition of two GUI components - a slider and a progress bar. When the slider is moved, the slider's current value is displayed in the progress bar. Slider and progress bar are connected via the `change` event. The slider is the event source and the method `setValue` of the progress bar the event sink. The target component platform is JavaBeans from Sun.

```
<?xml version="1.0" encoding="UTF-8"?>

<coml id="Slider_bar_Pattern" version="0:0:0:1">
  <meta>
    <info type="java" version="1.2.*"/>
  </meta>
  <components>
    <component id="bar1" class="javax.swing.JProgressBar">
      <property name="value" access="set">
        <int>50</int>
      </property>
    </component>
```

```
    <!--progress bar reacts upon the sliders  event-->
    <component id="slider" class="javax.swing.JSlider">
       <on-event name="change" filter="stateChanged">
        <property name="value" access="set" idRef="bar1">
          <property name="value" access="get" idRef="slider"/>
        </property>
       </on-event>
    </component>
  </components>
</coml>
```

This example shows the structure of each CoML script: The root element `coml` contains exactly two children - `meta` and `components`. The `components` element describes the actual component composition and the `meta` element provides additional information.

Meta Information

The `meta` information element is a collection of `info` elements. An `info` element describes which component library and version are used as well as additional component platform dependent information. Which additional information and therefore which additional elements are provided depends on the target component platform.

In order to distinguish between the component platforms, each `info` element has a `type` attribute. The value of the `type` attribute determines the additional child elements for describing the platform specific information. This approach of providing platform specific information and elements makes CoML flexible for future platforms and usage scenarios. Another attribute of the `info` element is the `version` attribute. The `version` attribute determines the compatible component platform version.

Example 1. This example uses the Java platform and the components are compatible with JDK 1.2 and later:

```
<meta>
  <info type="java" version="1.2.*"/>
</meta>
```

We propose the `type` attribute values “java”, “net” and “corba” for the JavaBeans, .NET, and CORBA platforms. However, this can be extended for other platforms as well.

The `version` and `type` attributes are mandatory for the `info` element. Table 2 summarizes the elements for providing meta information.

Component Declaration

Components are accessed via interfaces and have an implementation. The element `components` describes a list of declared components whereas the element `component` denotes

element	attributes	description
meta	-	collection of info elements
info	version, type	describes platform specific information

Table 2. Element descriptions for meta information

element	attributes	description
components	-	collection of component elements
component	class — idRef, [id], [interface]	declares a component

Table 3. Element descriptions for component declaration

a component declaration whose value (i.e. the component) is to be used as the argument of the enclosing element. The `component` element can be a child element of other elements whereas the `components` element - such as the `meta` element - has to be a child element of the root element `coml`.

If the `component` element has an `id` attribute then the declared component will be added to the global component list. If an `id` attribute is present then the component has a name and can be referred to by an `idRef` attribute. The id or name has to be unique within the current CoML script. If a `class` attribute is present, then a component instance will be created, or if an `idRef` attribute is present, then a component will be retrieved and referenced. The `interface` attribute denotes a type cast - or a similar technique such as COMs QueryInterface. If no interface is given, then the class name is the interface name (for OO based platforms) or the default interface (such as for COM).

Example 2. This example declares a component list with a component with the id “bar1”. The component “bar2” is declared as an additional id for the component “bar1”. The component “bar1” has an interface of type “Foo” is implemented by the class “FooImpl”.

```
<!--begin of the actual composition description-->
<components>
  <component id="bar1" interface="Foo" class="FooImpl"/>
  <component id="bar2" idRef="bar1">
    <!-- ... customize this component -->
  </component>
</components>
```

The child elements of the `component` element “bar2” customize this component or wire it with others. The following subsections describe these child elements.

The `class` or the `idRef` attribute is mandatory for the `component` element whereas the `id` and `interface` attributes are optional. Table 3 summarizes the elements for declaring a component.

element	attributes	description
property	name, access, [idRef]	for setting and getting properties
method-call	name, [idRef]	method call
add	[idRef]	aggregation-based composition
on-event	name, [filter]	connection-based composition

Table 4. Element descriptions for property, method, aggregation and event binding.

Properties

Some component models allow the modification of component *instances* via properties (for a detailed discussion of the differences between objects and components please refer to [17]). A label of a GUI-button is typically modeled as a property. Properties can be read and set.

Example 3. This example sets the “Amount” property to “50”. The property has an integer type.

```
<component id="bar1" class="FooImpl">
  <property name="Amount" access="set">
    <int>50</int>
  </property>
</component>
```

The `property` element declares a property modification (setter access) or a property read access (getter access). The `name` attribute declares the name of the property. In case of a read access the value of the property element can be used as an argument of the enclosing element and the `access` attribute has the value “get”. The property element of a declared read access has no child elements. The “set” value of the `access` attribute declares a property modification - a new property value will be set. The values of the new property are its child elements. Every element whose value is a primitive type value or a component can be used as a property child element. No type checks are performed. Syntactically it is allowable to assign values with incompatible types to a property.

Typically, the component to whom the property belongs is the value of the parent `component` element. However, it is also possible to access the properties of another component as the parent by using the `idRef` attribute. The value of the `idRef` attribute is a declared identifier of a component (see the `id` attribute of the `component` element).

The `name` and `access` attributes of the `property` element are mandatory and the `idRef` attribute is optional. Table 4 summarizes the `property` element.

Methods

The `method-call` element declares a method call. Its `name` attribute declares the method name. If an `idRef` attribute is not present, the callee is the parent component, otherwise

the callee is the named component. The child elements are the input arguments. Every element whose value is a primitive type value or a component can be used as a property child element. In the current version of CoML, a described method can have one or zero return values which can be used as an input argument to the enclosing element.

Example 4. This example shows a simple method call of the method “AsString” and has the integer “10” as argument.

```
<method-call name="AsString">
  <int>10</int>
</method-call>
```

Allowing only one output parameter is a considerable limitation but allows CoML to have a limited amount of elements and, therefore, to remain a simple language. Furthermore, CoML does not integrate error or exception handling mechanisms as some component platforms use exceptions (e.g., CORBA) whereas other platforms use return codes (e.g., HRESULTs in COM) to signal erroneous execution. Both aspects (i.e. finding a platform-independent way of integrating error handling and supporting more than one output parameter) will be part of future work. CoML does not provide special tags for distinguishing call by reference or call by value. This is clearly defined by the formal parameters of the described method itself and does not need to be modeled by CoML.

The `name` attribute of the `method-call` element is mandatory and the `idRef` attribute is optional. Table 4 summarizes the `method-call` element.

Aggregation

The element `add` allows the aggregating of components to a higher-level component by defining a containment hierarchy. The child elements are the added components along with arguments. Which arguments are provided depends on the used containment framework. The `idRef` attribute is optional. Table 4 summarizes the `add` element.

Events

The event mechanism allows components to be plugged together based on the connection--oriented programming model. Some of the characteristics of event based systems are that components can be composed via events both at design-time and at run-time. The collaborators of an event mechanism are the event source, the event data and the event sink or handler. Only components can be event sources whereas a method call, setting a property, or aggregating a component can take the role of an event sink.

The `on-event` element declares an event connection. The `name` attribute describes the event name. Only `component` elements can act as an event source and they are the only allowed parent elements of an `on-event` element. Optionally, a `filter` attribute can be

used to prevent an event sink from receiving all events. The required single child element describes the event sink/handler. The elements `component`, `property`, `method-call`, and `add` can be used for describing an event sink.

Example 5. The event example given on page 16 shows the wiring between two graphical components - a slider and a progress bar. The slider "slider" is the event source and when the event "change" is fired the property "value" of the component "bar1" is set. This property setter acts as an event sink.

```
<component id="slider" class="javax.swing.JSlider">
   <on-event name="change" filter="stateChanged">
    <property name="value" access="set" idRef="bar1">
      <property name="value" access="get" idRef="slider"/>
    </property>
   </on-event>
</component>
```

CoML does not handle concurrency and in particular race conditions, e.g., it is legal to further wire the component "bar1" with "slider" the other way round. Lets consider such a situation: whenever the slider "slider1" is moved, the value property of the progress bar is set. This state change of the progress bar would again set the "value" property of the "slider1" and so forth. In this particular situation the race condition would be broken because when the event comes back to the slider, it would not fire a "change" event again. The target component platform stops the event race. However, other scripting languages such as Piccola, which is based on the piL calculus [6], have shown a way to handle concurrency issues in a composition language.

The `name` attribute of the `on-event` element is mandatory and the `filter` attribute is optional. Table 5 summarizes the element `on-event`.

5 Related Work

Sun and IBM have developed their own composition language based on the meta syntax XML. However, both Sun's JavaBean Persistence [10] and IBM's Bean Markup Language (BML) [19] are tailored for JavaBeans and do not support meta information as a first class abstraction (refer to Table 5).

The main goal of Sun's approach is to have a proprietary standardized format for exchanging mainly GUI JavaBeans compositions between different Java IDEs. This is an approach similar to CoMLs primary purpose. At the beginning of the project, we tried to use Bean Persistence. Unfortunately, the expressiveness of Bean Persistence for composing components via events is too limited. Bean Persistence uses special proxy classes to build an event connection and does not provide XML elements to clearly mark an event connection. Bean Persistence depends on the JavaBean mechanism for detecting event sources -

	BML	Bean Persistence	CoML
Execution Environment	Java compiler, Java interpreter	API for serializing GUI JavaBeans	Java interpreter, .NET interpreter
Main Purpose or Intention	scripting language for JavaBeans	JVM and IDE tool independent persistence mechanism	tool and component platform independent composition description
XML Syntax	Java Bean oriented	Java Bean oriented	component platform independent
Missing Composition Concepts	does not offer meta information	does not offer abstractions for events, aggregation, properties and meta information	does not offer custom glue code

Table 5. BML, Bean Persistence and CoML comparison

i.e. method naming conventions - and does not abstract the actual event-gluing. The original JavaBeans specification does not define a logical containment/hierarchy relationship among connected beans and Bean Persistence does not support it either.

CoML is influenced by BML. The main differences are that CoML is not focused on JavaBean composition, that CoML supports interfaces where BML allows explicit type conversions, that CoML does not allow embedding of foreign scripting code - such as JavaScript [4] - in order to remain platform independent, and that CoML supports meta information.

6 Conclusions

With CoML we introduced an XML based component composition language which provides component platform independent syntax and behavior suitable for exchanging component compositions between different development tools. With an abstraction for meta information, CoML provides means for versioning, for handling differences of the component platforms, and for extensibility.

CoML is an ongoing research project and is not finished, yet. Some of future research efforts will cover how to handle different formal parameter behavior of method calls and how to provide convenient error handling mechanisms. It is still not resolved which information should be part of the meta information and whether we will address cross platform composition as well.

References

1. Franz Achermann and Oscar Nierstrasz. Application = Components + Scripts – A tour of Piccola. In Mehment Aksit, editor, *Software Architectures and Component Technology*. Kluwer Academic Press, 2001.
2. Don Batory. Subjectivity and GenVoca Generators. In *Proceedings of the 4th International Conference on Software Reuse (ICSR '96)*. IEEE, 1996.

3. Dietrich Birngruber and Markus Hof. Using Plans for Specifying Preconfigured Bean Sets. In Li Qiaoyun, editor, *Proceedings of TOOLS USA 2000*, Santa Barbara, California, 2000.
4. European Computer Machinery Association. *ECMAScript Language Specification*, June 1997.
5. Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.
6. Markus Lumpe, Franz Achermann, and Oscar Nierstrasz. A Formal Language for Composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 4, pages 69–90. Cambridge University Press, March 2000.
7. M.D. McIlroy. Mass Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering*. NATO Science Committee, January 1969.
8. An Introduction to Microsoft .NET. White Paper, Microsoft Corporation, 2001.
9. Microsoft Corporation. *The Component Object Model Specification*, 1995.
10. Philip Milne and Kathy Walrath. Long-Term Persistance for JavaBeans. White Paper, Sun Microsystems, November 1999.
11. 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.
12. Object Management Group. *OMG: Event Service Specification (Version 1.1)*, March 2001.
13. Mat Pietrek. Avoiding DLL Hell: Introducing Application Metadata in the Microsoft .NET Framework. *MSDN Magazine*, October 2000.
14. Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.
15. Stefan Schiffer. *Visuelle Programmierung: Grundlagen und Einsatzmöglichkeiten*. Addison-Wesley, 1998.
16. Sun Microsystems. *JavaBeans Specification*, July 1997.
17. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
18. Kathy Walrath and Mary Campione. *The JFC Swing Tutorial: A Guide to Construction*. Addison-Wesley, 1999.
19. Sanjiva Weerawarana, Francisco Curbera, and Matthew J. Duftler. Bean Markup Language: A Composition Language for JavaBeans Components. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems*, San Antonio, Texas, January 2001.
20. World Wide Web Consortium. *Extensible Markup Language (XML) 1.0. W3C Recommendation*, February 1998.

HotAgent Component Assembly Editor

Ludger Martin

Darmstadt University of Technology
Department of Computer Science
Wilhelminenstr. 7, D-64283 Darmstadt, Germany
Tel: +49 (0)6151 16 6710, Fax: +49 (0)6151 16 6707
lumartin@gkec.informatik.tu-darmstadt.de

Abstract. Many recent development environments support the visual construction of applications. Some of them use the term component to explain the composition process. In this paper, the assembly editor of the component development environment HotAgent is presented to show a possible solution to visually construct agents for electronic commerce. The requirements of the composition process are described and it is shown how to use the HotAgent assembly editor.

1 Introduction

For a few years, there have been various kinds of visual development environments. More recent ones are specially designed to write programs using the advantages of component technology. De Mey and Gibbs [4] argue that components are needed to construct applications in a visual composition editor. They claim that it is not possible to compose programs visually without some kind of components.

In this paper, the HotAgent [9] development environment is presented. It is a component framework to construct agents for electronic commerce. The framework provides a special set of components to support the easy construction of agents. The HotAgent development environment offers different tools to develop components, to test components and sets of components, and to compose agents using components. This paper focuses on the composition of components with the HotAgent assembly editor.

Lüer and Rosenblum [6] present seven requirements for component-based development environments:

- The components need a modular design; in particular private parts and public parts of a component need to be separated.
- The components need to be self-descriptive.
- The components are anchored in a global name space.
- There are two parts of the development process: the first is the component development and the second is the application composition.
- The application composition is done by connecting and adapting components.
- An environment needs multiple views; e.g., for component development, for application composition, or for type checking:

– For better reuse, components should be used by reference.

Most of the requirements are taken into account by the HotAgent development environment, as shown in the next sections.

In Section 2, the component model used by HotAgent is presented. In Section 3, the composition process is explained and related work is discussed. In Section 4, other parts of the HotAgent development environment are presented. The paper concludes with a summary and a discussion of future work in Section 5.

2 Component Model

The HotAgent development editor is not based on one of the well-known component models like EJB (Enterprise Java Beans), CCM (CORBA Component Model), or COM+ (Common Object Model). Instead, a small component model is used. This has the advantage that HotAgent is easier to use with a lower overhead, which allows a user to focus on component usage. Inter-component communication only relies on a single common interface each component has to implement. Such an approach eases the connection of components and simplifies visual composition.

The component model is a modified version of a model previously presented by Siemon [13]. It is based on the considerations of Stritzinger [14] and is implemented in VisualWorks Smalltalk. A component relies only on one common interface to communicate with other components. The interface is the sole public part of all components. All other classes that are a part of each component are private. This approach enables all components to communicate with each other independently of their tasks.

Lüer and Rosenblum [6] demand the self documentation of components. The HotAgent component model offers three kinds of documentation. The first is a short unique description name of the component, e.g., eMail, which is stored in a global name space. To have the opportunity to sort the components into different categories, every component has one category name, e.g., data management. This is also used to locate components in an easy way. If a component is chosen, it is necessary to check that it is the right one for the task at hand. To do this, a component has a description text to give an impression of its features.

If an instance of a component is created, it is important to assign a unique instance name. This name is only known in the composed agent and can be used to identify the component during runtime.

The interface of a component ensures the communication between the components. Every component provides so-called entrances and exits. Lüer and Rosenblum [6] call the entrances *required ports* and the exits *provided ports*, respectively. An entrance can receive data and produce a result, depending on the component functionality. An exit is activated if a component has to notify its environment about a change in its internal state. It supplies all necessary data and can process a possible result. Since Smalltalk is an untyped programming language, the data can be of any type. If a wrong type is provided by an exit

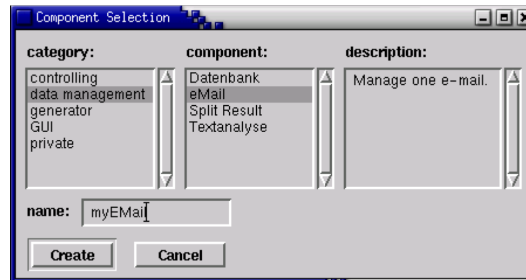

Fig. 1. Creating Components

or an entrance needs other data, an error will occur during runtime. It is only possible to communicate using these public entrances and exits. To compose components, exits can be connected with entrances. Similar to the component, every entrance and exit has a name and a description text to allow self documentation. It is important to describe the data provided by an exit and needed for an entrance, respectively.

3 Component Assembling

As mentioned above, components can be composed by connecting exits with entrances. It is possible to connect one exit to several entrances and vice versa. Wren [6] allows only the connection of one entrance to one exit. Bühler et al. [2] pose two important questions: (i) what kind of visual interaction model is chosen and (ii) what does the textual representation look like? These questions will be considered throughout this section. As an illustration, the example of an agent for a medical advisory service (originally presented by Clausius [3]) will be discussed. A patient writes an email to the medical agent belonging to a medical practice and describes his symptoms. The agent diagnoses the disease and replies with an email containing a recommendation for a medicine.

3.1 HotAgent Assembly Editor

One requirement for component-based development environments is that there are two development processes: (i) component development and (ii) application composition. At present, the HotAgent assembly editor fulfills the second requirement only and, therefore, can only be used to visually compose agents using predefined components.

In previous work [8,11] we have shown that a graph metaphor is the best method for visual composition of components: components are represented as nodes and connections are represented as edges, respectively. Most state-of-the-art visual programming environments are based on this metaphor. Therefore, the HotAgent assembly editor is based this metaphor.

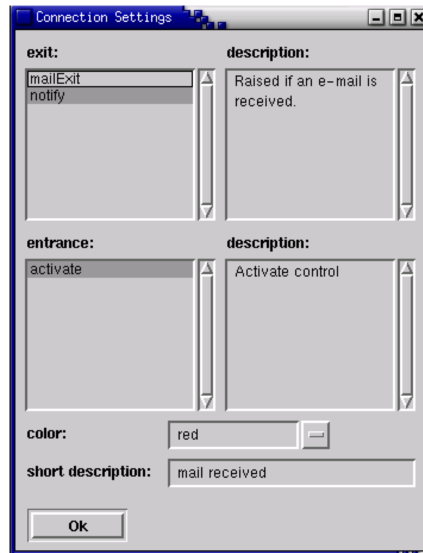

Fig. 2. Creating Components: Entrance/Exit Selection

The first thing to do when designing an application is to choose a position for a new component on a workspace. Components can be classified into two different types: (i) visible components representing an element of a graphical user interface (GUI) and (ii) invisible components like a database without any visual representation. The workspace of the Hot-Agent assembly editor consists of two parts. Any invisible component can be put on the main part. Moreover, there is a frame to position visible and invisible components. The two types of components (visible and invisible) should not be mixed up with the category name (e.g., data management).

After choosing a position, a dialog window (see Figure 1) appears to choose a component. As mentioned before, components are sorted into several categories. The Dialog shows a list from which to choose the category and if this is done, a component can be selected in a second list. To determine if the component does the desired task, there is a field to read the description of the selected component. Every component instance needs a name, which has to be unique in the composed agent, so it is necessary to define one in this dialog. In contrast to the component names, the component instance names are stored in the name space of the composed agent. After clicking the button *create*, the component will be placed on the workspace. In the example, an eMail component with instance name myEMail is added to the workspace.

Once all required components have been placed onto the workspace, it is necessary to connect them. In order to create a connection, a source component needs to be selected. Once this has been done, the middle handle of the selected component can be dragged onto

the target component. After finishing the drag, HotAgent checks whether there is at least one unconnected source component exit and one unconnected target component entrance. If this is the case, another dialog window (see Figure 2) pops up which can be used to select the corresponding exits and entrances. In this window, the exit and entrance can be chosen from a list. Relative to the selection, a description of the exit or entrance is displayed. In addition, connections can be labeled and colored to support a secondary notation[1]. The label can be any text to give a short description of the component. Figure 2 shows how to create a connection from the notify exit of the eMail component to the activate entrance of the control component.

Figure 3 displays the workspace with the complete Medical Advisory Service agent. It consists of an entry field, a button to emulate an email client, and several invisible components such as eMail, control, database, etc. to realize the agent.

The connections have different colors to distinguish their purpose. Every connection has a short text to describe its task. By choosing the position of the components, the application developer has a good opportunity to use secondary notation and to obtain a composed agent.

If the components and the connections are dense in some areas of the workspace, the view can become unclear. To solve this problem, it is possible to zoom into the workspace. After zooming, the positions of a component are changed but not the dimensions. It creates more free space between the components. Using this technique, it is possible to place more invisible components, e.g., between visible components. It is also possible to turn off the display of invisible components, connections, and labels. This supports a clear view of the workspace.

In order to store the visual specification, a textual language is needed. It is useful to separate the textual representation of a visual specification into two parts: (i) a part to store the component specification like name and position and (ii) a second part is to store the information needed for the connections. Using this textual language, the composed agent can be executed. Both languages are represented using an array. In Smalltalk, an array definition is wrapped by `#( ... )` and every element is separated by a blank. An array can contain any type of data, e.g., a string `'eMail'`, a symbol `#name:`, or a number. An example of an array with the symbol `#name:` and the string `'eMail'` is `#(#name: 'eMail')`.

Component Specification: The data structure of the array for the component specification consists of two entries for each component: (i) the component description name (e.g., Window) and (ii) an array containing all settings related to this component. It includes the component instance name, the component origin, and dimension. In this array, the setting specifier (e.g., `#origin:`) is followed by its value. Visible components need to specify a dimension, invisible ones do not. The example given below shows a part of the component

[1] A secondary notation denotes extra information, e.g., color, or position, in other meaning than the program syntax

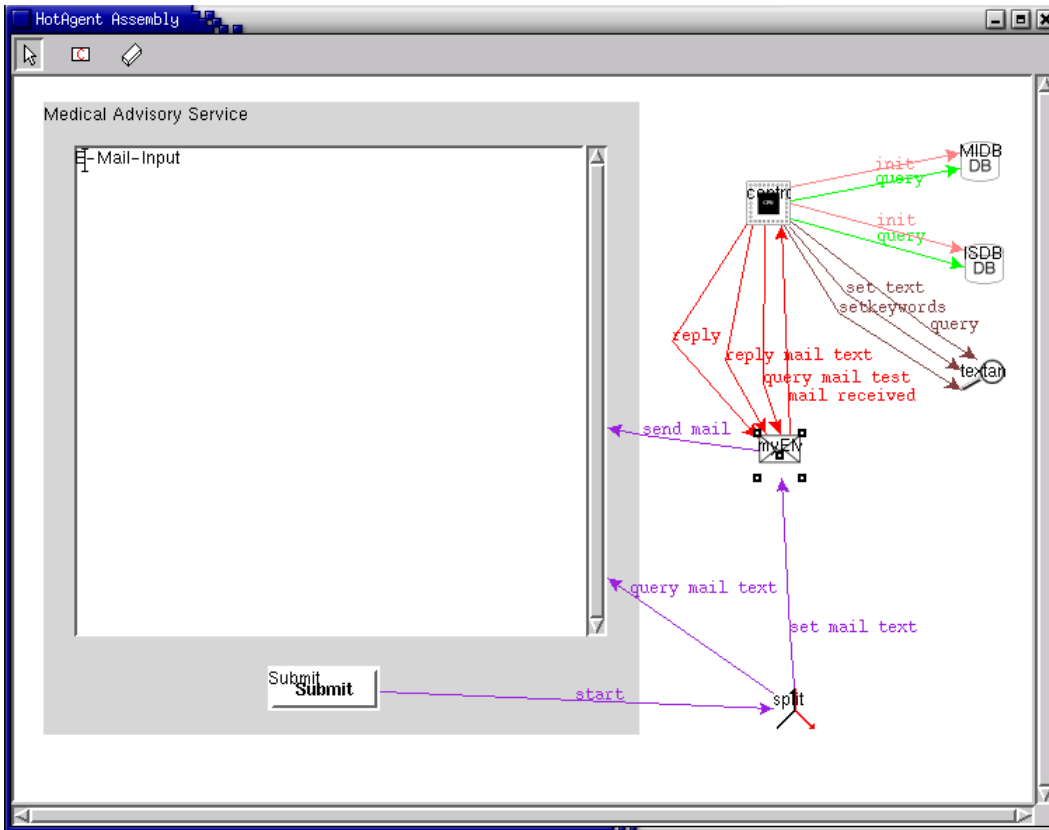

Fig. 3. Workspace with Components and Connections

specification of the Medical Advisory Service. The first two lines define a window to hold an entry field and the third specifies an invisible component.

```
#('Window' #(#name: 'Medical Advisory Service' #origin: #(21 18)
             #dimension: #(425 451))
  'eMail' #(#name: 'eMail' #origin: #(530 254))
  [...]
)
```

Connection Specification: The connection specification is also an array. Every element of this array corresponds to one connection. Every connection is described in an additional array with six elements. The first element is the source component name. Its exit name follows in the second element. The third element is the target component and the entrance name, followed by the specification of the color and the connection label. The first line of the example given below describes the connection of the eMail component to the control

component. The connection starts at the notify exit and ends at the activate entrance. It is painted in red color and has the label mail received.

```
#(#('eMail' 'notify' 'control' 'activate' #red 'mail received')
  #('control' 'queryMailText' 'eMail' 'queryText' #red
    'query mail test')
  [...]
)
```

3.2 Related Component Assembly Systems

The visualization used by Visual Age is similar to the one used by the HotAgent assembly editor [11]. In Visual Age, the color of the connections depends on the type of the connection and the connections are not labeled. This can cause some confusion if a program gets more complex. In addition, any kind of class can be placed on the workspace. The mixture of classes and components on the workspace leads to an unclear concept. The user does not need to follow the requirements of components and can run into problems because classes do not have the same behavior as components.

The tool presented by Hull et al. [5] supports distributed components. All compatible modules can be connected. Components and connections can be labeled. One advantage of their tool is that the components can be placed onto an unrestricted number of levels. Another advantage is the automated orthogonal arrangement of components and connections.

Agentsheets [12] is a tool to create domain specific development environments. The components are agents with special behavior. They are organized in grids and automatically connected to their neighbours on the grid. The behavior of a component depends on its neighbours. Agentsheets can, for example, be used to construct biospheres.

4 Other HotAgent Parts

In the previous section programs for component compositions were discussed. HotAgent offers some additional tools [7] to work with components.

The simplest method to create new visual components is to use the HotAgent pattern component editor (under development). The first step is to place graphical elements into a workspace. The elements can have different shapes and colors. The second step is to assign behavioral patterns to the elements, e.g., an element can be moved on a line or plane. The editor creates all necessary classes for the new component on its own. Using this editor, new control mechanisms for a user interface can be created.

The component editor is an all-purpose editor. It is good for constructing visible and invisible components. It supports the programmer by creating component entrances and exits, classes, variables, and methods. It is possible to connect methods with each other and entrances and exits with the corresponding methods. Using this editor, a programmer can construct components without knowing very much about the underlying component model.

The component technique is known for its reusability. This property in particular requires well tested components to ensure reliable applications. HotAgent offers the possibility to test single components or groups of components using user defined test cases (see [10]). To realize this, the test framework SUnit [1] is extended to handle entrances and exits of components. The programmer can write a test case by defining and checking the data flow between the test framework and the components.

5 Summary and Future Work

In this paper, the functionality of the HotAgent assembly editor is described and an illustration of how to compose agents visually is given. By using a secondary notation while connecting the components, it is possible to get a clear composition model. It still has to be shown that the method of the assembly editor is usable in real-world projects.

In Lüer and Rosenblum [6], seven requirements for component-based development environments are presented. Most of them are addressed by HotAgent. HotAgent offers only a component instance view to compose agents. Further experiments have to be done in order to include a type view for an untyped component model into HotAgent.

Mey and Gibbs [4] mentioned that the type of entrances and exits has to be checked before connecting. This is not done in HotAgent yet because the component model is untyped. Investigations are needed to find a way to include it in HotAgent.

In Section 4, a test framework for components is introduced. Further experiments are needed in order to define a way to test visually composed programs or to check the correctness of visually assembled code.

References

1. Kent Beck. Simple Smalltalk Testing: With Patterns. *Smalltalk Report*, October 1994. <http://www.xprogramming.com/testfram.htm> (July 2001).
2. Frank Bühler, Mike Callaghan, and Paul Luker. VOODE/VOOPL-1: The Visual Construction of CORBA Components. In *Proceedings of the IEEE International Symposium on Visual Languages*, pages 61–62, 2000.
3. Thorsten Clausius. Komponenten zur Konstruktion von Agenten. Master's thesis, Department of Computer Science, Darmstadt University of Technology, 2001.
4. Vicki de Mey and Simon Gibbs. A Multimedia Component Kit. In *Proceedings of the First ACM International Conference on Multimedia*, pages 291–300, 1993.
5. M. Elisabeth C. Hull, Peter N. Nicholl, Philip Houston, and Niall Rooney. Towards a visual approach for component-based software development. *Software Concepts & Tools*, 19(4):154–160, 2000.
6. Chris Lüer and David S. Rosenblum. Wren – An Environment for Component-Based Development. Technical report, Department of Information and Computer Science, University of California, September 2000.
7. Ludger Martin. Visual development environment based on component technique. accepted at the 2001 IEEE Symposium on Human-Centric Computing Languages and Environments.
8. Ludger Martin. Visualisierung von Komponenten für Benutzungsoberflächen. Master's thesis, Department of Computer Science, Darmstadt University of Technology, 2000.

9. Ludger Martin. *HotAgent Homepage*. <http://www.gkec.informatik.tu-darmstadt.de/HotAgent/>, July 2001.
10. Ludger Martin and Elke Siemon. Extending SUnit to Test Components. Accepted at the Workshop on Component-based Application Engineering in Research and Practice.
11. Ludger Martin and Elke Siemon. Component Visualization Based on Programmer's Conceptual Models. In *OOPSLA ´00 Companion*, pages 73–74, 2000.
12. Alex Repenning. Agentsheets: A Tool for Building Domain–Oriented Visual Programming Environments. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 142–143, 1993.
13. Elke Siemon. *Über den Entwurf von Benutzungsschnittstellen technischer Anwendungen mit visuellen Spezifikationsmethoden und Werkzeugen*. PhD thesis, Darmstadt University of Technology, 2001.
14. Alois Stritzinger. *Komponentenbasierte Softwareentwicklung*. Addison-Wesley, 1997.

Partial Evaluation of Inter-language Wrappers

Nathanael Schärli, Franz Achermann

University of Bern
Institute of Computer Science and Applied Mathematics
Software Composition Group, Schützenmattstr 14, CH-3012 Bern
Tel: +41 31 631 33 13, Fax: +41 31 631 39 65
{schaerli|acherman}@iam.unibe.ch

Abstract. Wrapping external components by scripts can be a performance bottleneck if inter-language bridging is frequent. Piccola is a pure composition language that wraps components according to a specific composition style. This wrapping must be efficient, since even arithmetic operations are done by external components. In this paper, we present how to use partial evaluation to overcome much of the overhead associated with the wrapping. It turns out that Piccola scripts can be highly optimized since form expression exhibit the right kind of information to separate side-effects from services and resolve internal dependencies.

1 Introduction

Piccola [2] is a pure composition language that has a generic mechanism to use external components. It allows the user to adapt these components according to a specific composition style using wrappers. We argue that a composition language must allow the user to specify this glue code in a flexible and high-level way. On the other hand, efficient execution of the glue code is an important issue for inter-language bridging in general and for a composition language in particular.

We propose the use of partial evaluation to optimize the wrapping abstractions. Piccola is well-suited for partial evaluation due to its simple syntax and semantics based on forms. Forms are extensible records and have light-weight introspection facilities. Piccola has no built-in datatypes or objects that would complicate reasoning.

In this paper we present a partial evaluation technique that allows lazy evaluation. We represent the result of wrapping – and of service invocations in general – by lazy forms. Lazy forms have the advantage that only necessary expressions are evaluated. The technical contribution of this paper is the separation of side-effects from services to make them referentially transparent.

This paper does not give an introduction to Piccola itself; an overview of Piccola and its use of forms can be found in [2] or in the forthcoming thesis of the second author [1]. We explain the necessary syntax and semantics of Piccola when discussing the code examples.

This paper is organized as follows: in Section 2, we present a wrapper for Piccola components and examine the introduced performance loss. In Section 3, we present the concept

```
addComparison X:
    _==_ Y: asBoolean(X._=_ Y)
    _!=_ Y: asBoolean(X._~=_ Y)      # Squeak uses ~= for inequality
    _<_ Y: asBoolean(X._<_ Y)
    _>_ Y: asBoolean(X._>_ Y)
    _<=_ Y: asBoolean(X._<=_ Y)
    _>=_ Y: asBoolean(X._>=_ Y)

def asNumber X:
    peer = X.peer
    addComparison X                  # add all comparison operator bindings
    -_: asNumber X.negated()         # arithmetic operators...
    _+_ Y: asNumber(X._+_ Y)         # plus
    _-_ Y: asNumber(X._-_ Y)
    _*_ Y: asNumber(X._*_ Y)
    _/_ Y: asNumber(X._/_ Y)
    abs: if (_<_ 0)                  # if this is smaller than 0
        then: -_()                   # return -X
        else: asNumber X             # else return self
    trunc: asNumber X.truncated()
```

Fig. 1. The `asNumber` wrapper of SPiccola

of lazy forms and illustrate the partial evaluation strategy. Section 4 contains a set of examples. Section 5 concludes the paper and addresses related and future work. The partial evaluator for core Piccola is given as an appendix.

2 Profiling a SPiccola wrapper

SPiccola is the implementation of Piccola on top of Squeak. In Piccola, infix operators can be defined as services in one component. `A + B` invokes the service _+_ bound in `A` with the argument of `B`. The wrapper `asNumber` in Figure 1 receives a form `X` containing a peer that refers to the native Squeak number object. It returns a form of service bindings according to the composition style for numbers. It does so by building a form that contains the necessary operators and services.

Some services such as arithmetic plus do have a corresponding Squeak method. The implementation of the plus operator (_+_) takes the right-hand side operand `Y`, calls the native Squeak plus (available through `X._+_`) and wraps the result using the number wrapper. Other services like `abs()` are specified using previously defined bindings. Last but not least, the comparison operators are factored out. The invocation `addComparison X` adds the operators `==, !=, ...` to the resulting number form.

Let us consider when this wrapper gets triggered. The evaluation of an expression `a = 1 + 2` causes the following invocations of the wrapper:

1. The external object representing the number 1 gets wrapped by `asNumber`.
2. A projection on the label `_+_` of the wrapped form is performed.
3. The external object representing the number 2 gets wrapped.
4. The wrapped form is passed as an argument to the `_+_` service. Executing this service sends the Squeak message + to the object 1 with a projection on the label `peer`[1] object of 2. The result (the number 3) is again wrapped.

The wrapping service is invoked three times and each time constructs a form with 14 bindings. But for the forms built in step 1 and 3, only one of these bindings (`_+_` or `peer`) is used while all the other bindings are discarded.

Examination of Piccola scripts show that on average, we only use about 10% of the bindings created by the Piccola wrappers. Assuming that the time used for setting up such an interface is uniformly distributed over the created bindings, this means that the overhead for building such an interface could be reduced by 90% if we only created the bindings that are actually used.

In a reference implementation of Piccola, the introduction of this number wrapper leads to approximately six times slower performance. It should be noted that our current requirements for number specify only a small number of bindings per number. In contrast, the Squeak v2.9 `SmallInteger` object understands more than 400 messages. If the wrapper consists of so many bindings, the performance penalty would be much worse.

3 Lazy Forms

Lazy evaluation is used in certain functional languages like Haskell. It means that beta reduction only gets performed when a value is actually used. In Piccola, evaluation is in general strict, since the language has side-effects. In order to avoid the evaluation of expressions in unused bindings, we can split up service invocation into the creation of a lazy form and its usage. A lazy form defers evaluation of its bindings and only evaluates the bindings that are effectively needed. As forms are immutable, it is possible to cache the projected value.

At *invocation time*, we only execute the side-effect specified by the service and return a lazy form. When a binding of the lazy form is *used*, only the bound expression gets evaluated. The bound expression may refer to the argument of the lazy form.

There are two critical requirements for services to invoke them lazily: the services need to be separated into a functional and a side-effect part, and the functional part may not contain internal dependencies.

[1] The projection on `peer` does not appear in the SPiccola code since it is part of the generic inter-language bridge.

Separated side-effect. It is necessary to execute the side-effect part on invocation time. Consider the following service:

```
chfact channel:
    value = channel.receive()      # wait for a channel
    factorial = fact value         # heavy calculation without
                   side-effect
```

It contains a side-effect receiving a value from a channel and an invocation of a referentially transparent function `fact` to calculate the factorial. The side-effect part can be represented by the following anonymous service:

```
\channel: s1 = channel.receive()
```

and the functional part by:

```
\channel Sideeffect:
    value = Sideeffect.s1
    factorial = fact value
```

The functional part is a higher order abstraction taking the result of the side-effect as an additional argument. Note that a service can contain different side-effects which are bound by different fresh labels, like `s1`. The original service `chfact` is the functional composition of the above two anonymous services.

Independent bindings. In order to evaluate only a single binding when a projection occurs, it is necessary that it does not refer to other bindings in the same lazy form. In our example, the `factorial` binding refers to `value`. This dependency can be resolved by replacing the use of `value` by its definition. Note that substitution of referentially transparent bindings does not introduce side-effects.

```
\channel Sideeffect:
    value = Sideeffect.s1
    factorial = fact Sideeffect.s1       # No reference to value
```

The reader may want to notice the following implementation aspect: although we present de-serialization by substitution, it is common to use references in order to avoid multiple expensive computations. This technique is used to implement call-by-need evaluation.

In the next section, we give some more examples of the separation. The formal translation of core Piccola is given in appendix 5. Due to lack of space, we only present the core of Piccola omitting syntactical sugar like user-defined operators or dynamic namespaces.

4 Transformation examples

Lazy closures are a triple $\mathbf{Cl}(x, P, S)$ where x is the formal argument, P is the purely functional part, and S is the side-effect. Below we present lazy services in tables.

A referentially transparent service. As a first example, consider the invocation of `addComparison` service presented in section 2. The service `addComparison` is transformed into a lazy closure:

service	functional part	side-effect

`addComparison(X)`	`_==_ Y: asBoolean(X._=_ Y)`
	`_!=_ Y: asBoolean(X._~=_ Y)`
	`_<_ Y: asBoolean(X._<_ Y)`
	...
		ϵ

Since it does not contain side-effects, we can inline its body into the code. Thus `asNumber` becomes:

```
def asNumber X:
    peer = X.peer
    _==_ Y: asBoolean(X._=_ Y)        # in-lined addComparison
    _!=_ Y: asBoolean(X._~=_ Y)
    _<_ Y: asBoolean(X._<_ Y)
    ...                               # rest as in Figure 1
```

Optimize compound service. As a more involved example consider the service `f` which contains an invocation of `chfact` defined in the previous example. Written in tabular form the service `chfact` is

service	functional part	side-effect
`chfact(ch)`	`value =` *eff*`.s1` `factorial = fact` *eff*`.s1`	`s1 = ch.receive()`

where *eff* denotes the result of the evaluation of the side-effects. We only write the body of the anonymous abstraction and use the identifier *eff* as placeholder for the argument. Now assume a service `f` invoking `chfact`:

```
f ch:
    result = chfact ch
```

The service `f` is split up into functional and side-effect part as:

service	functional part	side-effect

`f(ch)`	`result =`
	  `value =` *eff*`.s1`
	  `factorial = fact` *eff*`.s1`
		`s1 = ch.receive()`

Default Arguments. The last example shows the use of wrappers to define default arguments. The glue code adding default values is optimized by partial evaluation. Assume a service `newBox` creating a new peer box with default arguments and a client service `c` as follows:

```
newBox X:
    '(width = 10, heigth = 15, X)
    ''println "Width: " + width
    newPeerBox(w = width, h = height)

c: box = newBox(height = 20)
```

In Piccola, a quoted expression 'E is syntactic sugar for $\mathbf{root} = (\mathbf{root}, E)$. Thus a quoted expression modifies only the local namespace denoted by the keyword **root** and does appear in the resulting form. Consequently, double quoted expressions extend the namespace with the empty form and denote pure side-effects.

These services `newBox` and `c` get split into lazy closures as:

service	functional part	side-effect

`newBox(X)`	*eff* `.s2`	`s1 = println "Width: " +`
		`(width = 10, X).width`
		`s2 = newPeerBox`
		`w = (width = 10, X).width`
		`h = (height = 15, X).height`

`c ()`	`box =` *eff* `.s2`	`s1 = println "Width: " + 10` `s2 = newPeerBox(w = 10, h = 20)`

Observe that the client takes the default width 10 from the wrapper and specifies its own height 20 for the peer box. The `newBox` wrapper does not add any performance overhead.

5 Related and future work

This work mainly uses partial evaluation to transform Piccola scripts (e.g., [4]). In addition to partial evaluation we also translate services to isolate the side-effect part. Reasoning about side-effects is a necessary precondition to apply our technique to other languages. Sample et al. argue that even more information should be used for composition, like cost, associated network delay, or security requirements [7].

In order to script external components, they have to be imported into the composition environment first. The tool SWIG [3] automatically generates wrappers for components written in C or C++ so that they can be scripted in Perl, Tcl, or Python. Making components available in Piccola can be done generically since both implementation languages of Piccola (Squeak and Java) provide run-time introspection. We are more concerned with the

other side of the coin, namely how the scripting language can raise the level of abstraction without adding too much runtime overhead. Jones et al. [6] use Haskell to script COM components making use of higher-order functions. They use the type system to detect certain composition errors at compile time. However, a language with lazy evaulation makes it hard to reason about concurrency, but this is not in the scope of this paper.

It is interesting to compare partial evaluation with aspect-oriented programming [5]. We *weave* the composition wrappers into existing scripts, thereby making use of the formal foundation of Piccola for reasoning.

We are working on an integrated composition environment in Piccola. The information derived by the translation knows the *current value* of an identifier. This might be helpful in order to detect unwanted corruption of the root namespace. For instance:

```
'size = newVar(17)        # a Piccola variable
'myservice()              # call some other services, extending root
a = size.get()            # return the value stored in size
```

One can think of tool-tips like information when the user selects the identifier `size` in the last line. If the service call returns a binding `size`, we would unexpectedly return the contents of the wrong binding. Type information would also help to improve the static analysis and vice-versa. Due to the lack of space, we omitted the treatment of errors in the current version. Certain type errors like using a form without a service as a functor can be detected by the analyzer.

The appropriate treatment of fixed-points is still a point of debate. **Def** is syntactic sugar for fixed points. The fixed point is encoded by a channel and as such using the fixed point is not a referentially transparent service. However, for mutually recursive services, a lazy fixed point combinator would do the job and it would allow us to use the fixed point without side-effects.

In this work, we introduced a way to split up Piccola services into a side-effect and a referentially transparent part. This split enables the execution of wrappers in constant time and not in linear time with respect to the number of overridden bindings.

Acknowledgement

We thank Oscar Nierstrasz for helpful comments on an earlier draft of this paper.

References

1. Franz Achermann. *Piccola: A language to Script Composition Styles*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, 2001. to appear.
2. Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*. Kluwer, 2001. to appear.
3. David M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, 1996.

4. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 493–501. ACM, January 1993.
5. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, LNCS 1241, pages 220–242, Jyvaskyla, Finland, June 1997. Springer.
6. Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM Components in Haskell. In *Proceedings of Fifth International Conference on Software Reuse*, Victoria, British Columbia, June 1998.
7. Neal Sample, Dorothea Beringer, Laurence Melloul, and Gio Wiederhold. CLAM: Composition language for autonomous megamodules. In Paolo Ciancarini and Alexander L. Wolf, editors, *Proceedings of Coordination '99*, LNCS 1594, pages 291–306. Springer, 1999.
8. Nathanael Schärli. Supporting Pure Composition by Inter-language Bridging on the Meta-level. Master's thesis, University of Bern, 2001. to appear.

A The translation

This appendix contains the formal translation of Piccola services into a functional and side-effect part. The translation is explained in more detail in [8]. Core form expressions E in Piccola evaluate to forms F. Identifiers are ranged over by x. *Lazy forms* F^* are a modification of form expressions:

$$\begin{aligned} E &::= \epsilon \mid x \mid \mathbf{root} \mid E \cdot E \mid E.x \mid E\ E \mid \backslash x : E \mid x = E, E \mid \mathbf{root} = E, E \\ F &::= \epsilon \mid x = F \mid F \cdot F \mid \mathbf{Cl}(x, F, E) \\ F^* &::= \epsilon \mid x \mid x = F^* \mid F^* \cdot F^* \mid \mathbf{Cl}(x, F^*, F^*) \mid \\ &\qquad \mathbf{Pr}(F^*, x) \mid \mathbf{Se}(F^*) \mid \mathbf{Ap}(F^*, F^*) \mid \mathbf{Re}(F^*, F^*) \end{aligned}$$

We use P, Q, R, S to range over lazy forms. For convenience, S denotes a lazy form containing a side-effect, and P a form that specifies a pure function. Lazy forms are: the empty form, an identifier, a singleton binding, an extension, a closure $\mathbf{Cl}(x, P, S)$, an unevaluated projection, an unevaluated service selection, an application $\mathbf{Ap}(R_1, R_2)$, or a substitution $\mathbf{Re}(R_1, R_2)$ where R_2 substitutes identifiers in R_1. We can express substitution by a form, since essentially a form represents a name-value binding.

The translation $[\![E]\!]_R$ defines a tuple (P, S) of lazy forms where P is the functional part and S is the side-effect part of evaluating E in the environment R. In the end, we are only interested in P and in-line S wherever possible. $[\![E]\!]_R$ is defined inductively on the grammar for expressions as follows:

$$[\![\epsilon]\!]_R \stackrel{\text{def}}{=} (\epsilon, \epsilon) \qquad \text{(empty)}$$

$$[\![x]\!]_R \stackrel{\text{def}}{=} (project(R, x), \epsilon) \qquad \text{(ident)}$$

$$[\![\mathbf{root}]\!]_R \stackrel{\text{def}}{=} (R, \epsilon) \qquad \text{(root)}$$

$$[\![E_1 \cdot E_2]\!]_R \stackrel{\text{def}}{=} (P_1 \cdot P_2, S_1 \cdot S_2) \qquad \begin{aligned}(P_1, S_1) &= [\![E_1]\!]_R \\ (P_2, S_2) &= [\![E_2]\!]_R\end{aligned} \qquad \text{(extend)}$$

$$\llbracket E.x \rrbracket_R \stackrel{\text{def}}{=} (project(P,x), S) \qquad (P,S) = \llbracket E \rrbracket_R \qquad \text{(project)}$$

$$\llbracket \backslash x : E \rrbracket_R \stackrel{\text{def}}{=} (\mathbf{Cl}(x,P,S), \epsilon) \qquad (P,S) = \llbracket E \rrbracket_{R\cdot(x=x)} \qquad \text{(abs)}$$

$$\llbracket x = E_1, E_2 \rrbracket_R \stackrel{\text{def}}{=} ((x = P_1) \cdot P_2, S_1 \cdot S_2) \qquad \begin{array}{l}(P_1,S_1) = \llbracket E_1 \rrbracket_R \\ (P_2,S_2) = \llbracket E_2 \rrbracket_{R\cdot(x=P_1)}\end{array} \qquad \text{(assign)}$$

$$\llbracket \mathbf{root} = E_1, E_2 \rrbracket_R \stackrel{\text{def}}{=} (P_2, S_1 \cdot S_2) \qquad \begin{array}{l}(P_1,S_1) = \llbracket E_1 \rrbracket_R \\ (P_2,S_2) = \llbracket E_2 \rrbracket_{P_1}\end{array} \qquad \text{(sandbox)}$$

Due to the lack of space, we only explain a few tranformations. The empty form returns the empty form and has no side-effect. An identifier returns the projection within the current root context and has no side-effect. The keyword $\mathbf{root}$ returns the current root form and has no side-effect. Polymorphic extension returns the extension of its subexpressions, and sequentially composes their side effects. Abstraction returns a closure and has no side-effect. The difficult case is invocation:

$$\llbracket E_1\ E_2 \rrbracket_R \stackrel{\text{def}}{=} \begin{cases} (\mathbf{Re}(Q, x = P_2), & \text{if } P_1' = \mathbf{Cl}(x,Q,\epsilon) \\ \quad S_1 \cdot S_2) & \\ (\mathbf{Re}(Q, (x = P_2) \cdot (\mathit{eff} = y)), & \text{if } P_1' = \mathbf{Cl}(x,Q,Q') \\ \quad S_1 \cdot S_2 \cdot (y = \mathbf{Re}(Q', x = P_2))) & \quad \text{and } Q' \neq \epsilon \\ (\mathbf{Pr}(\mathit{eff}, y), & \text{otherwise} \\ \quad S_1 \cdot S_2 \cdot (y = \mathbf{Ap}(P_1', P_2))) & \end{cases} \qquad \text{(invoke)}$$

where $(P_1, S_1) = \llbracket E_1 \rrbracket_R$, $(P_2, S_2) = \llbracket E_2 \rrbracket_R$, $P_1' = \mathit{service}(P_1)$ and y denotes a fresh identifier. The first case applies when the invoked service does not contain any side-effects. In this case, we return a substitution of the formal argument x by the functional part of the argument P_2 in the body Q of the closure. The side-effect of the result consist of the sequential composition of the side-effects of the functor and the argument. The second case applies when the service has a side-effect Q'. In this case, the substitution of the first case is extended with a fresh identifier y to contain the *result* of the side-effect invocation. Finally, if we do not have enough static information, we return a lazy application. We assume that this translation is behaviour preserving, but we have not carried out a formal proof, yet.

The meta-function $project(F^*, x)$ returns a projection onto x in the lazy form F^*, and $service(F^*)$ returns the service bound in the lazy forms. In the worst case, for instance when F^* is an identifier, these functions denote the lazy projection $\mathbf{Pr}(F^*, x)$ or the lazy service lookup $\mathbf{Se}(F^*)$, respectively.

Reflections on the Anatomy of Software Composition Languages and Mechanism

Michel Chaudron

Technische Universiteit Eindhoven
Department of Computer Science
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
Tel: +31 40 247 4449, Fax: +31 40 246 8508
m.r.v.chaudron@tue.nl

Abstract. The aim of this paper is to give some recommendations on the design of software composition languages and mechanisms. To this end, we first propose a conceptual model for software composition. Using this model we explain how existing mechanisms can be understood as software composition mechanisms of different levels of abstraction. Based on the analysis of the strengths and weaknesses of several software composition mechanisms, we give some recommendations for their future design.

1 Introduction

The vision of building software systems through the composition of pre-built software building blocks ('components') was introduced in the 1960's (o.a. by McIlroy [4]). Until about 1990 research had focussed on methods and principles for designing software components in such a manner that they could be easily connected together. Since about a decade ago, research in the areas of coordination languages (e.g. [6]) and software architecture [9] has pointed out that composition of software components should be addressed as an issue in its own right.

In this paper, we first propose an abstract model for software composition. We then use this model to explain how existing mechanisms can be understood as software composition mechanisms of different levels of abstraction. We analyse strengths and weaknesses of these mechanisms and give recommendations on the future design of composition languages and mechanisms.

2 A Model for Software Composition

The ideal of component-based software engineering is to be able to build systems by assembling components that may be obtained from different parties. Maintenance of such a system should support the removal and addition of components – ideally while the system is in operation, but without seriously affecting the quality of service of the system. This rises the question on how component can be made to interoperate.

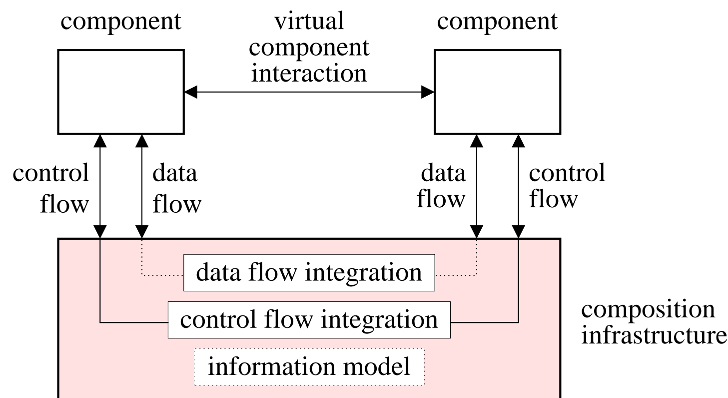

Fig. 1. Abstract Component Model

Different companies are promoting their own component models. However, none of these is likely to become a de-facto standard and imposing global component-standards is not a realistic solution. Instead, flexible composition mechanisms are needed that can operate in open, heterogeneous environments. We propose an abstract model for software composition illustrated in Figure 1.

This model considers systems as a collection of components (in the widest sense of the word) and composition mechanisms. Components perform computations and the composition mechanism manages the interaction between components.

This model identifies the following ingredients of a composition mechanism: data-flow, control-flow and an information model. A composition mechanism should provide policies for integrating control flow and data flow. Alternatively, a composition mechanism may provide a means for user-defined integration policies. We elaborate on the elements of the model in the next sections.

2.1 Control flow

Control flow deals with the way in which activities within and amongst components are controlled.

There are several strategies for dealing with control flow in a composition model. In passing control between components either a fixed strategy can be used or a composition language can be used to define a range of (user-designed) control passing strategies.

An example of a fixed control strategy is the (remote) procedure call, (R)PC. This defines a flow of control from a calling component to a serving component. After the serving component has performed its service, it returns the flow of control to the calling component.

Exception handling is an issue for which users may find it useful to define various control strategies. For example, one strategy first tries to move control from a calling component

to a serving component. When the serving component manages to compute a result, then a response returns this data and the control to the calling component. Alternatively, if the serving component cannot serve the request, it reacts with an error message. In response to this, several reactions are possible. For instance, the exception mechanism continues by passing control to some component appointed for handling exceptions.

Other examples of issues that are appealing for user-defined strategies are:

– A voting mechanism: a request is forwarded to three components, the responses of the servers are collected and the majority is returned to the requesting component.
– In a distributed setting a strategy may be useful that forwards a request to a number of servers and returns to the requesting component the first answer that it receives from any one of the servers and ignores any subsequent responses.

The latter control strategies are examples of policies of a larger granularity. For the effective development of large software systems, it is essential that we find a way of defining and composing such higher-order abstractions. Scripting languages [5] are a technology that explore this direction.

2.2 Data flow

Data flow is concerned with the geometry and dynamics of data flow between components. More often than not, data flow and control flow are closely tied. In data-flow systems, data is leading; in other cases control-flow is leading. Consider, for example, the (R)PC mechanism: data needed for a service is handed over through a request, simultaneously with passing the control flow to the server. The resulting data is returned simultaneously with passing back the control flow. However, tying data flow to control flow introduces an unnecessary coupling. Decoupling control- and data-flow opens up a spectrum of composition policies.

An example of a class of applications where data-flow is a prime concern are streaming application. These are common in the domains of audio/video, telephony, and signal processing systems. Here, a system can be considered as consisting of a number of processes that consume, process, and produce streams of data, respectively. In principle, these processes may operate at their own speeds. Consequently, they may produce and consume data at different speeds. A typical requirement on the composition of such processes is to achieve a certain quality of service in the transmission of the data-stream.

A control-oriented solution to composition would be to synchronize the operating speeds of the components. Alternatively, a data-flow oriented solution could consist of managing the buffering of data. This illustrates that composition through control flow and composition through data flow can be alternatives, but only if data-flow and control-flow can be dealt with separately.

Another example of a data-flow related strategy is caching. This occurs in many contexts, ranging from CPU's up to distributed databases. Often caching strategies are built into

these systems. However, a data flow language may be used to enable users to define their own caching strategy. For instance, it may employ the knowledge of frequency of use of data to bias the placement of data. Alternatively, a user-defined data-flow strategy may take care of replication and consistency policies.

The above examples illustrate that many control flow and data flow structures are possible. This provides challenges for composition mechanisms in two ways. Firstly, for designing new systems, designers should be able to use high-level notions that consist of a collection of control and/or data transfers. Secondly, although it is highly desirable for component-based software engineering to have a framework that is based on the integral design of a component model and a composition model, it is unlikely that such a unified model will emerge. Heterogeneity is a fact of life; hence for integrating existing software, composition languages must provide the means for joining together a highly diverse set of control flow and data flow policies.

2.3 Information model

The information model can be sub-divided into a syntactical and a semantical aspect. Firstly, there has to be some format agreed upon for the representation of data. A common solution to this is through intermediate data description languages. Secondly, there is the issue of the meaning of data used in a system. This deals with the semantics issues. The way of describing the meaning of data independent from a particular application is currently topic of investigation in the database and hypermedia community under the flag of *ontologies*.

For use in open systems, extensible ways for defining representation formats and meaning are required. Languages like XML are addressing these needs.

3 Software Composition at Different Levels of Abstraction

In this section we illustrate how different well-known mechanisms can be understood in terms of the model for software composition presented in the previous section. This illustrates that the model is applicable at several levels of abstraction. For each of these levels of abstraction, we discuss some of the specific needs of composition languages. We conclude the section by a summary of recommendations on the design of composition mechanisms.

3.1 Programming Languages

The purpose of programming languages is the composition of atomic computations into programs. However, in all programming paradigms composition mechanism are subservient to some computational paradigm.

In imperative programming languages, assignment statements are the means for defining the smallest units of computation. These statements can be composed into larger computations by control-oriented constructs such as the ';' (sequential), 'while' (repetition) and

’if-then(-else)’ (conditional) constructs. Here, passing of control is not managed by the atomic computations. Instead, for assignments, control flow is an implicit side-effect, while the sole purpose of the other constructs is manipulation of a program counter. Data flow is implicit through the common access to shared memory: an assignment statement can use a result of a preceding assignment by referring to the variables used.

On the information model we note that there are implicit assumptions about the matching of data-representation (type systems provide compile-time checking – otherwise it is the programmers responsibility to match data-formats).

It can be argued that procedures are the next level of abstraction above instructions. The way of passing control is request-response (typically built into the execution mechanism) and the way of passing data is through parameters. Again, since procedures are used within the scope of a single language, the data representation is implicitly assumed to be similar. Matching semantics of data is calling and serving procedure is the programmer’s responsibility.

3.2 Operating Systems

Operating systems support the composition of programs. They can be seen as a composition infrastructure that provides means for adding and removing programs. At this level of abstraction, programs correspond to the components of our model. Also, operating systems enable interaction between programs. Interaction may take place through messaging or sharing of files (both are data-flow mechanisms) and synchronisation mechanisms (control-flow).

A specific composition mechanism of UNIX operating systems is the pipe-and-filter mechanism (due to M.D. McIlroy [8]). This mechanism assumes a common data representation (ASCII-files), which is also used as the means to integrate data-flow between programs. The pipe and filter mechanism assumes no semantics compatibility between programs. Hence, robustness against nonsensical input has to be built into the programs themselves.

In a pipeline, control is passed from ’left to right’. In principle, there are two strategies for implementing this: a data-driven approach and a control-driven approach. In the data-driven approach, applications would become active when data arrives. Hence, multiple stages of the pipeline could execute concurrently. In the control-driven approach, control is, for example, initially given to the first (’leftmost’) program. When this program finishes, control is passed to its successor and so forth until all programs have terminated.

The UNIX ’language’ for defining pipes is an example of a scripting language. It has the following highly desirable properties:

– Composition is realized through an exogenous composition mechanism (the operating system) rather than through a mechanism built into the components themselves (endogenous). This enables a high degree of decoupling between components.

- Composition is non-intrusive; i.e. no changes have to be made to the internals of the program – not in its source code (design-time), nor in its control-flow or data-flow (run-time).

3.3 Middleware

Middleware provides services that enable interoperation between components that may reside on different locations in a network and are potentially written in different programming languages. Examples are CORBA, RMI and DCOM [7].

Currently most middleware is based on the request-response interaction style. However, other styles such as publish-subscribe are gaining popularity. Also, the latest CORBA standard provides a set of interaction styles. It would be interesting to push this another step further and develop composition languages that allow user-defined interaction styles and system-wide program composition. Using these, it should for instance be possible to define pipe-and-filter structures that are distributed across a network.

One aspect that distinguishes middleware from the other examples in this section is the effort that is paid to semantic interoperability in specific application domains. The CORBA standard provides so-called 'vertical standards' which define domain specific information models and services.

Many middleware systems aim at enabling composition of components written in different programming languages. The mechanism introduced for crossing language borders is the 'interface definition language' (IDL). Such IDL's essentially are the 'language interface' to an underlying composition mechanism.

3.4 Internet's World Wide Web

In this section we look at the World Wide Web as a composition mechanism. We motivate its correspondence with our model (refer to Figure 1) as follows. Consider applications, which could be just 'passive' html-pages or applications with a web-interface (possibly based on a complex back-office) on a node of the internet as components and the network infrastructure (including all software needed to keep that running) of internet as a composition mechanism.

The success of the WWW is partially due to the fact that it has well-defined rules for the extension and scaling of the infrastructure (the internet). These rules actually define the composition mechanism at a meta-level: they describe how it can evolve. This requirement of scalability of a composition mechanism is generally not relevant at the level of programming languages, but it is an essential requirement for many distributed systems. An additional challenge is in the combination of openness and scalability (which allows third-party scalability).

It is interesting to observe that the internet caters easily for the addition and removal of components – while the internet as a whole continues operation. Partially this is due to the fact that web-applications cannot make operation-critical assumptions about other

web-applications. This enforces the avoidance of dependencies between web-applications. While this design constraint is readily accepted for web-applications, full evasion of dependencies between components is not believed realistic at the programming level. It would be interesting to capture the essential properties of the internet as composition mechanism and impose those on a programming paradigm.

In contrast to common opinion, the composition mechanism WWW is not best viewed as a client-server/request-response system, but as a system based on a data space/blackboard as illustrated in the following. The collection of all available web pages forms a space of "content". Components can freely add content to the space. When content is added to the space, this is not directed towards a particular destination (anonymous, undirected). Content can be searched in an associative manner through search engines. A possible problem with the data space style is that only an owner is allowed to remove its data from the space.

To summarize this section, we list our recommendation for the design of composition languages and mechanisms:

- Composition should be exogenous to components; i.e. the mechanisms for composition should not be built into components themselves.
- Composition should be non-intrusive; internals of components should not have to be modified in order to be composed.
- Composition mechanisms should provide separate mechanisms for dealing with data-flow and control-flow. However, it should be possible to relate data- and control-flow.
- Composition languages should provide means for building higher-level/larger granularity composition abstractions.
- For application to existing software, composition mechanisms should provide support for bridging multiple control- and data-flow styles.
- In particular application domains, composition mechanisms are subject to general quality requirements such as timeliness, reliability, extensibility, scalability etc.

4 Directions for Future Research: Style-less Components

Components that use different interaction styles are very difficult to compose into a single system. Architectural mismatches [3] between components are cue to the fact that the interaction-styles are built into the components. The reusability of components can be increased by devising a component model where components are 'style less' and the interaction style(s) between components is fully determined by the composition mechanism(s) used to connect components together.

Clearly this requires that components are designed in a constrained manner. However, the stakeholders of a system may be willing to sacrifice freedom of design against increased (de)composability. A possible approach to this is presented in Figure 2.

The functionality of components should be modeled by some mechanism that does not have a bias towards a particular style of interaction. A model that provides the best match

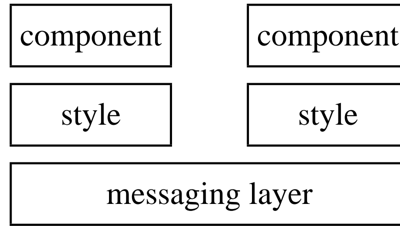

Fig. 2. Decomposition Model

for this requirement is the Gamma model [2] of chaotically executing multiset rewrite rules. Gamma programs can be encapsulated by wrappers which impose a particular interaction style. We propose to use the PICCOLA language in the style proposed in [1] for this.

In order to apply these ideas in distributed systems, we propose to use a messaging layer because this is the form of moving data around that has the least bias towards particular interaction styles. Other – more complex – styles can be built on top of it.

5 Concluding Remarks

Until recently, software composition has either been ignored or has been subservient to mechanisms for specifying computations. Recently, software composition has been recognized as a 'first-class' citizen.

Software composition mechanisms arise at different levels of abstraction such as instruction-, procedure-, program-, and system-level. The requirements for composition languages and mechanism at different levels of abstraction and in different application domains may differ. However, at any level, a composition language should provide means for bridging different policies for control flow and data flow as well as between non-uniform information models.

We recommend that a composition language should provide separate means for defining strategies for transfer of control and transfer of data because this increases the (de)composability of systems. This can be further improved by separating the dependencies of components through separate interfaces for interaction with peer-components and for interaction with the composition infrastructure, respectively.

Composition languages are the linguistic interface to composition mechanisms. For open, distributed systems such mechanisms may be of higher algorithmic complexity than the binding mechanisms that are currently focus of research on composition languages. In open, distributed systems, composition mechanisms are subject to quality requirements such as scalability and timeliness.

For increased composability components should not have a hard-coded interaction style. In addition, they should be robust against nonsensical input.

References

1. Franz Achermann, Stefan Kneubühl, and Oscar Nierstrasz. Scripting Coordination Styles. In António Porto and Gruia-Catalin Roman, editors, *Coordination Languages and Models*, LNCS 1906. Springer, September 2000.
2. Jean-Pierre Banatre and Daniel Le Metayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, January 1993.
3. David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 12(6):17–26, November 1995.
4. M.D. McIlroy. Mass Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering*. NATO Science Committee, January 1969.
5. John K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, 31(3):23–30, March 1998.
6. George A. Papadopoulos and Farhad Arbab. Coordination Models and Languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*. Academic Press, August 1998.
7. F. Plasil and M. Stal. An Architectural View of Distributed Objects and Components in CORBA, Java RMI, and COM/DCOM. In *Software – Concepts and Tools*. Springer, 1998.
8. Dennis M. Ritchie. The Evolution of the Unix Time-sharing System. *AT&T Bell Laboratories Technical Journal*, 63(8):1595–1605, October 1984.
9. Mary Shaw. Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In D.A. Lamb, editor, *Studies of Software Design*, LNCS 1078, pages 17–32. Springer, 1996.

©2002 Swinburne University of Technology
ISBN 0 85590 785 1