

Requirements for an Architectural Composition Language[†]

Johann Oberleitner and Thomas Gschwind

Technische Universität Wien
Institut für Informationssysteme
Abteilung für Verteilte Systeme
Argentinierstraße 8/E1841
A-1040 Wien, Austria
<joe,tom@infosys.tuwien.ac.at>
Phone: +43-1-58801-58400

Abstract. ADLs are used in Software Engineering to describe the architecture of an application. This description can be used by different tools to support architectural analysis. Composition Languages on the other hand focus on the construction of executable applications out of components. Unlike ADLs a composition language allows the construction of new components and frameworks from smaller entities. By combining these two languages into an Architectural Composition Language (ACL), however, it is possible to combine their advantages. In this paper we present the requirements for ACLs and the design of our Architectural Composition Language. A combination of both approaches allows the explicit description of an architecture while still permitting the construction of executable applications and new components from fine-grained existing components.

1 Introduction

Architectural Description Languages (ADLs) are used for various purposes in the design stages of a project. ADLs are used to describe the architecture of an application, and thereafter check if the architecture fulfills the constraints specified by the software engineers. Most ADLs, however, provide only a static definition of an architecture or cannot be executed directly. Additionally, they cannot be used for the composition of new components. Although ADLs can be used for the generation of the outline of an application's implementation they focus on the description of the application's architecture. After the application's outline has been generated application developers have to fill in the missing parts.

Composition languages [11] require the ability to define new components and component frameworks reusable within the language. Component-based software engineering relies heavily on previously built components and demands a powerful language to describe the application's architecture in a way that cannot only be executed but analyzed as well. Since such a composition language is the synthesis of ADLs [10] and composition languages [11] we call languages that fulfill the above requirements *Architectural Composition Languages* (ACL).

In this paper we present the requirements for ACLs:

- Constructs for describing applications built out of COTS components based on standard component models such as JavaBeans, EJB, CORBA and COM+ in one application.

[†] This work was supported in part by an IBM University Partnership Award from IBM Research Division, Zurich Research Laboratories and the European Union as part of the EASYCOMP project (IST-1999-14191).

- An explicit view of the architecture by separating the definitions of the COTS components available for an application, the connectors, and the application configurations into distinct parts that allow constraint checking on the architecture. On this basis it is possible to check whether the architecture fulfills the required constraints.
- Constructs for the definition of new component frameworks, new components and new connector types that can be instantiated as connectors and components.
- In contrast to ADLs, an application modeled with an ACL has to be executable.

This paper is structured as follows. In section 2 we present related work and the terminology used throughout this paper. In section 3 we define the requirements for an ACL. Section 4 mentions the research issues we will focus on in the future and we draw our conclusions in section 5.

2 Background

Our work is closely related to two classes of research. The first is research based on traditional architectural description languages. *Architectural Description Languages (ADLs)* are languages that can be used to model and describe software architectures. An architectural description consists of several building blocks: components, connectors, and architectural configurations. *Components* are the units of computation. *Connectors* are used to model interactions among components and implement constraints on these interactions. *Architectural Configurations* describe an architectural structure (topology) between components and connectors [3]. According to [10] an ADL has to model components, connectors, and their configurations explicitly. Tools can make use of the information stored within the ADL to analyze if the architectural structure enables proper communication between the components and if the desired behavior can be achieved with the deployed connectors [14].

Various architectural description languages are in use today [3, 4, 9, 10, 14] that have their focus on different issues. For instance, *Darwin* is solely concerned with the structural aspects of an architecture [9], *Wright* focuses on architectural behavior and provides a notion for handling with reconfiguration [3], and ACME is an architecture description interchange language [4]. UniCon [14] supports rate monotonic analysis important in the area of real-time scheduling. An ADL that comes close to our requirements is Rapide. Rapide, however, is targeted at the execution of prototypes and assumes that interfaces are defined prior to their implementation [8].

The second class of research is concerned with composition languages. Requirements for composition languages are stated in [11]. All existing composition languages we have found refer to some or all of these requirements. Among the existing composition languages PICCOLA [1, 2], implements most of these requirements and can be adapted or extended to various application environments.

The goal of *Composition Languages* is the description of applications consisting of already existing components. Code written in a composition language is executable in the sense that it can be interpreted at run-time or that it can be compiled into an executable representation form. Most composition languages allow the definition of new composition elements that can be used within the same composition configuration or within other configurations that reference the former.

Although according to [11] composition languages can make an applications architecture explicit, none of the existing composition languages allows to express an architecture in a

way similar to ADLs. Unlike ADLs, composition languages do not clearly separate the description of components, connectors, and composition configurations. Hence, the analysis of an architecture described with a composition languages is hardly possible since connectors and connections are not described explicitly. A clear discrimination, however, if a particular language is an ADL or is a composition language might not always be easy. For instance, the composition language Piccola is able to describe a system's architecture as well [1] but for an architectural description as defined in [10] the description is still too low-level.

Traditional scripting languages such as Tcl/Tk, JavaScript or the languages based on the Microsoft Windows Scripting Host also fulfill the requirements of a composition language. The Bean Markup Language [16] allows the description of Java programs consisting of JavaBeans written in XML. The package ships with a player that executes the scripts. None of these languages, however, allows to describe an application's architecture.

Workflow Description Languages such as WSFL [7] or XLANG [15] are used to describe workflows based on web services. These languages address the orchestration of web services, and provide direct mappings of the underlying business workflow to an application built from web services. Since web services always interact in a peer-to-peer manner [7], the constructs for connecting web services are restricted to this form of composition. In addition it is not possible to define new connector types in contrast to what we have stated as a requirement for an ACL. Workflow languages are good for the composition of web services but are not expressive enough when accessing arbitrary software components. WSFL provides constructs for binding web service activities to implementations available as executable programs or Java classes. However, to use these constructs it is necessary to provide code that converts between the data types used by the web services and the data types used within the implementations.

3 Requirements for Architectural Composition Languages

An Architectural Composition Language (ACL) has to be designed to fulfill the requirements of an architectural description language [10] and those of a composition language [11]. This allows for the execution of the application on the basis of this language while maintaining the application's architectural view. Hence, software developers only have to take care of a single system description while maintaining the flexibility of both approaches. The requirements that have to be met by an ACL are:

1. *Architecture*. It must resemble constructs from ADLs to allow the explicit description of an application's architecture [4]. Tools can exploit this description to reason about capacity, throughput, consistency and component compatibility as mentioned in [14]. To support this reasoning facilities the descriptions of components and connectors have to include semantic information required for reasoning tools.
2. *Extensibility*. It must be possible to define new components, new connection styles and new component frameworks. These new elements must be indistinguishable from those styles already built into the language. [11]
3. *Executability*. It must be possible to execute an application described within an architectural composition. Additionally, it may be possible to compile an ACL description to generate a binary format of the application. Executability means that not only code can be generated that has to be filled with functionality by developers but that applications can be described completely with language scripts.

4. *Openness*. The language has to be language-independent and support components of different component models. Additionally, the language itself should be open for extensions.

On the basis of these requirements, an ACL has to explicitly model component definitions, connector definitions, and composition configurations. Component definitions describe the set of available components, connector definitions describe how the semantics for an interconnection between a set of components looks like, and the composition configuration defines the relation between the components and connectors. Hence an ACL can be seen as an ADL according to [10] that provides the functionality of an *explicit* and *implementation-independent* ADL.

Architectural styles and frameworks can be defined using framework-templates that can either be components, connectors, or both. These frameworks are generalizations of composition configurations.

3.1 Components

Components are the major units of composition. An ACL should distinguish between components and instances of components in a similar way as OOP distinguishes between classes and objects. Component definitions are used to describe the components required by the composition.

Each component definition specifies a *component name* that identifies the component within the ACL. The component definition consists of information about the component's underlying component model and the component's class within this model. Additionally, it may specify the features provided and the constraints required by the component if they cannot be derived automatically from the component. Other characteristics of the component can be specified too, such as information required for reasoning about the whole architectures.

Although the class name of the underlying component could be used to identify the component our approach has several advantages. It avoids implementation dependent component identifiers within the composition configuration and provides a clear separation of the components' composition specification, therefore increasing the readability and portability of an ACL specification.

Depending on the component model, it may also be possible to pass *instantiation parameters* to the component. While in the case of JavaBeans, for instance, the class name is sufficient, distributed component models, however, such as Enterprise JavaBeans require more parameters such as the name server to be used. In case that references to several services provided with typical CORBA ORBs are required, a description for a CORBA object can have about ten instantiation parameters.

3.2 Connectors

Connector definitions declare the connector styles used by a component configuration and the properties they exhibit. They are used to model the different interaction patterns and rules that govern the interaction between components. Additionally, they define the components that may participate in a connection and the roles they may play.

Connector definitions define a *connector name* for the connection style. On the basis of this name, a connector can be referenced and instantiated to form a connection within a component configuration. In our composition language a connector is designed as a set of instances of

connections that have the same interaction pattern but possibly different components assigned to the interaction roles provided by the connection. Connectors have several elements that are necessary to define their functionality sufficiently within the language.

Connectors are the low-level connection elements of ACLs and define the semantics of the different connection styles. To be able to generate an application out of a component configuration the interpreter of the ACL has to provide support for the different connection styles. Since we allow the definition of new connector styles, it must be possible to extend the ACL parser as well. This can be achieved easily, however, by providing a shared library or class file representing the according connection style. As an implementation hint, connectors may include a reference to an implementation.

Connectors are used to build abstract interconnections between components not known until instantiated within concrete compositions. We use *roles* to denote how the composition will bind instances of components or other instances of connection end-points to connectors. It is the responsibility of the implementation of the language that bindings are compatible with their roles. Roles do not consist solely of the components that are used within an interconnection but can be more fine-grained and specify particular features such as events or methods.

3.3 Composition Configuration

A composition configuration defines the instances of components and the connections that participate in the configuration. Connections are instantiated by specifying the connector and assigning components or features of components to the connector's roles.

Hence, an ACL must provide structures for the instantiation and naming of component instances. Component names have to be unique such that they can be referred to by the connection specifications. Connection specifications denote the connector that has to be used for the interaction between a given set of components and the roles of the components participating in the connection. As with component instances connections have names to reference them.

3.4 Frameworks

One of the goals of ACLs is to be able to define composition frameworks where components and connectors can remain unspecified and be bound at a later — possibly execution — time. A framework can be reused within composition configurations or within other framework definitions.

Dependent on their application, frameworks can be used for different purposes. They can be used as a component, as a connector, and as a hybrid framework combining the former two. As shown in figure 1 a framework is similar to a composition configuration except that some parts of the composition are left unspecified and that we need to be able to export some of the features provided by its constituent components to the framework boundary.

It is necessary to specify the feature and component roles that need to be supplied when the framework is instantiated. Since frameworks can be seen as generics these roles can be seen as the type parameters to be bound when instantiated. Since a framework can be used as a component or a connector within compositions all provided features have to describe how they build the functionality with the use of existing components, connectors and frameworks.

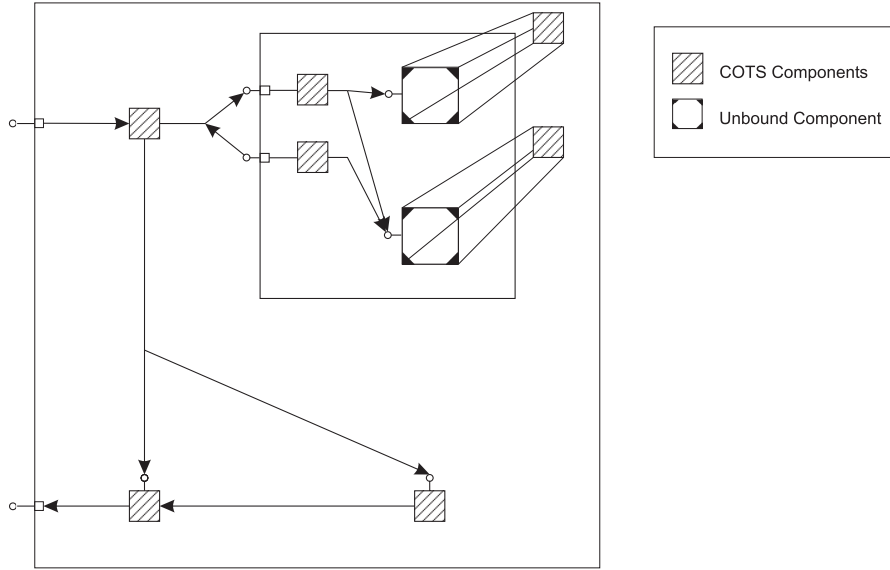


Fig. 1. Framework

Hence, it is necessary to specify the features of the components that should be available at the framework boundary.

A component framework can only be instantiated if all of the framework's roles are bound and if the assigned roles fulfill the requirements of the framework's underlying composition.

3.5 Component-Related Extensions

The previous sections have described the core requirements for an ACL. Other requirements that might have to be considered by an ACL are the initial states of the components used within a composition configuration and the possibility to allow the substitution of components to be substituted with different but equivalent ones.

4 Discussion

In this position paper we have motivated for the need of an ACL and defined their requirements. Currently, we are in the process of implementing *ACL/1*, our own simple ACL. *ACL/1* will serve as our testbed to experiment with ACLs and the functionalities that should be provided beyond those presented in this paper.

Our initial results of this language show that ACLs can be used to describe the architecture of an application in a language independent way. Additionally, due to the decoupling of components and their implementation it should be possible to substitute components unavailable on a particular system with other similar components during application start-up. Slightly incompatible components could be easily adapted using type-based adaptation [5] or dynamic component extension [6].

Currently, *ACL/1* is based on XML. While this simplifies the implementation of the language due to the wide availability of XML parsers the resulting language is not as compact as

could be possible. While this does not matter, if the architecture is defined using a visual design tool such as the Component Workbench [12], it will find little acceptance by programmers that prefer to write such code themselves.

In **ACL/1** we use Java classes that implement the low-level connectors necessary for the inter-connection semantics such as the invocation of a method if an event occurs. Although only a small number of connectors are required to build various frameworks with **ACL/1**, this might be cumbersome if it were the only way to integrate new semantics such as parameter transformations. A straight-forward solution to this problem would be to embed Java source code into the connector description that will be compiled when the application is executed the first time. An alternative would be the integration or fusion with another composition language that provides powerful mechanisms for glue code description such as **PICCOLA** [2]. Design tools, however, can soften this problem by generating the connector's implementation when necessary.

We also plan to write a parser of **ACL/1** architectures for the .NET framework. We have already realized a mapping for component and component features from one component model to another one [5]. With **ACL/1**, the uniform component model of the Component Workbench [13] and this mapping feature we can execute applications that were targeted for the Java platform on a .NET platform using similar but different components.

Since ACLs combine a composition language with an architectural description language it is natural to make analysis on the architecture of an application. Since our language implements some of the elements **ACME** [4] provides it should be possible to transform the ACL architectural part to an ADL and use the tools already available for this ADL.

Since the architecture is described explicitly the implementation of a tool for determining numerical boundaries on distributed interaction execution times becomes much easier than the analysis of interaction code somewhere inside the glue code. Such values are helpful to select among various distributed component technologies and application server implementations and application server platforms.

Finally, we want to investigate existing architectural description languages and composition languages whether they fulfill the requirements of ACLs. So far, however, we have not found such an ADL or CL that can be classified as an ACL.

5 Conclusions

In this work we have presented the requirements for an Architectural Composition Language, a hybrid language that combines aspects of Architectural Description Languages and of Composition Languages. The idea of ACLs is the adoption of the concise and explicit view of the architecture from ADLs while it implements the composition and execution facilities of composition languages. While traditional ADLs do not allow the execution of an application the existing composition languages do not support architectural analysis. ACL joins both approaches and therefore combines their advantages.

ACLs allow parts of a component configuration to be left unspecified. We call such configurations component frameworks that can be used like components. They can be reused within other component frameworks or architectures allowing an ACL to be used for differently sized components.

Since ACLs provide an architectural view of an application similar to that of an ADL, it should be possible to translate the architecture defined with an ACL into an according

architecture defined with one of the existing ADLs. This allows to reuse existing tools available for these ADLs.

References

1. Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola — a Small Composition Language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Computing — A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.
2. Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts — A tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.
3. Robert J. Allen, Remi Douence, and David Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Fundamental Approaches to Software Engineering*, LNCS 1382. Springer-Verlag, April 1998.
4. David Garlan, Robert T. Monroe, and David Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, November 1997.
5. Thomas Gschwind. *Adaptation and Composition Techniques for Component-Based Software Engineering*. PhD thesis, Technische Universität Wien, February 2002.
6. Thomas Gschwind and Johann Oberleitner. Dynamic Component Extension to Support Cross-Platform Development. Technical Report TUV-1841-2002-19, Technische Universität Wien, March 2001.
7. Frank Leymann. *Web Service Flow Language (WSFL 1.0)*. IBM Software Group, may 2001. <http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>.
8. David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
9. Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 3–14, October 1996.
10. Nenad Medvidovic and Richard N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In Mehdi Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 60–76. Springer-Verlag, 1997.
11. Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a Composition Language. In *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer-Verlag, 1995.
12. Johann Oberleitner. The Component Workbench: A Flexible Component Composition Environment. Master's thesis, Technische Universität Wien, October 2001.
13. Johann Oberleitner and Thomas Gschwind. Composing Distributed Components with the Component Workbench. Technical Report TUV-1841-02-17, Technische Universität Wien, January 2002. Accepted for publication in the Proceedings of the 3rd Software Engineering and Middleware Workshop.
14. Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.
15. Satish Thatte. *XLANG: Web Services For Business Process Design*. Microsoft Corporation, 2001. http://www.gotdotnet.com/team/xml_wsspecs/xlangc/default.htm.
16. Sanjiva Weerawarana, Francisco Curbera, Matthew J. Duftler, David A. Epstein, and Joseph Kesselman. Bean Markup Language: A Composition Language for JavaBeans Components. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technology Systems (COOTS 2001)*, pages 173–187. USENIX, January 2001.