

Transforming Application Compositions with XSLTs³

Johann Oberleitner¹ and Thomas Gschwind²

*Distributed Systems Group, Institut für Informationssysteme, Technische Universität Wien,
Argentinierstraße 8/E1841, A-1040 Wien, Austria*

Abstract

Architectural Description Languages (ADLs) allow developers to describe the architecture of a software system but provide only limited support for their execution. In this paper, we present ACL/1, an XML based Application Composition Language, that describes the architectural view of a software system while at the same time enabling the execution of the software application. As we will show, our approach allows us to use XSLTs to transform the architectural view of the application into source code and it may be used to refactor the underlying software application.

1 Introduction

Architectural Description Languages (ADLs) are used to describe and model many different aspects during the design phase of a software project. ADLs are used to describe the architecture of an application, and allow to check if the system architecture fulfills constraints imposed by the system requirements [11].

Most ADLs, however, provide only a static definition of an architecture and cannot be executed directly. Although some ADLs support the generation of prototypes [9], their support for the implementation of an application is limited. This is because they focus mostly on architectural properties useful in the early stages of a new system. Additionally, they cannot be used for the construction of new components.

Another kind of language are composition languages [12]. Composition languages require the ability to define new components and component frameworks that extend the language automatically.

¹ Email: joe@infosys.tuwien.ac.at

² Email: tom@infosys.tuwien.ac.at

³ We gratefully acknowledge the financial support provided by the European Union as part of the EASYCOMP project (IST-1999-14151) and an IBM University Partnership Award from IBM Research Division, Zurich Research Laboratories.

Component-based software engineering relies heavily on previously built components and demands a language to describe the application's architecture in a way that cannot only be executed but may be analyzed as well. To solve this problem we have developed a language that allows the construction of applications where the architecture remains explicit. We achieved this by combining the concepts of ADLs [11] with those of composition languages [12]. Such a language is called *Application Composition Language (ACL)*.⁴

The requirements for such an ACL [13] are:

- Constructs for describing applications built out of COTS components based on standard component models such as JavaBeans, EJB, CORBA and COM+ in one application.
- An explicit view of the architecture by separating the definitions of the COTS components available, the connectors, and the application configurations in a way that allows us to check the constraints of an architecture.
- Constructs for the definition of new components, new connector types and new component frameworks.
- The application modeled with an ACL has to be executable.

This paper focuses on the implementation of ACL/1 our own ACL, an XML based language, that allows the construction of component-based applications. This allows us to use an Extended Stylesheet Language Transformation (XSLT) [3] to apply refactorings and maintenance tasks as well as for the generation of standalone executables and application documentation. The advantage of this approach is that the transformations (i.e., for generating Java Applets or standalone Java programs) are written independently of an application configuration, thus allowing them to be used in combination with any given application configuration.

The paper is structured as follows. In section 2 we present the terminology used throughout this paper. In section 3 we define the constructs realized for our ACL. Section 4 explains how XSLT can be used to apply refactorings and patches and how application descriptions are transformed into real applications. Section 5 mentions the research issues we will address in the future. Section 6 presents work related to ours and we draw our conclusions in section 7.

2 Terminology

Before we can discuss application composition with ACLs as well as the code generation and transformation using ACL/1 in combination with XSLTs, it is necessary to define the terminology we are using in the context of this paper. This is necessary since some of the terminology is used with a slightly different meaning within the software engineering research community.

⁴ Although we have introduced ACLs as Architectural Composition Languages in [13], we came to the conclusion that Application Composition Language better describes the purpose of this type of languages.

A *component* is a piece of software that conforms to a component model [5] such as the JavaBeans component model or COM+. A *component model* defines the basic architecture of a component, specifying the structure of its interfaces and the mechanisms it uses to interact with its environment. The functionality a component provides is defined by the interfaces it implements. Other pieces of software interact with the component only through the use of these interfaces.

A component has several different *features* that are available through its interfaces. Features are typically events which are emitted by the component, properties that may be changed, or simply the methods that can be used to interact with the component.

A *connector* defines how two components interact with each other [11]. Within an ACL components are always connected using connectors. Hence, components cannot be connected directly with each other. Connectors specify the components to be connected as well as the features the components have to provide. The advantage of this approach is that within a given composition a component can be more easily replaced with another component since it only has to provide the features required by the connectors connecting to it. If the features are available in a slightly different form, feature-mappings as presented in [8] can be used. Additionally, a connector may be responsible to change the data transferred between the components.

A *component framework* is the composition of several components using connectors. A component framework does not require the use of concrete components. Instead it allows developers to use placeholders for specific components within the composition. When a component framework is instantiated, however, it is necessary to specify the components that should be used for the individual placeholders. To a certain degree, a component framework may be compared to C++ templates or to Java Generics used on an architectural level.

3 Application Composition

We have implemented several language elements into ACL/1 to provide application construction. Some of these constructs have their origin in ADLs [11] responsible for describing application architecture. Other constructs we have defined enable ACL/1 to execute the composed application.

This allows for the execution of the application on the basis of ACL/1 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.

ACL/1 uses three different types of descriptions: component descriptions, connector descriptions and configurations. Component descriptions describe the components available for building an application. Connector descriptions comprise the glue between the components. These two describe static elements and can be reused for many other applications. Configurations, the third type, concrete realizations of an application which are specific to one particular application.

Architectural styles and frameworks can be defined using framework-templates that can either be components, connectors, or both. Since we currently do not support frameworks they are not covered by the descriptions in this paper.

3.1 Components

Components are the major units of composition. Our language distinguishes between component classes and instances of components. In the following component classes are denoted as components while component instances are denoted as instances.

Components are described in component definition files. These components can be implemented by different component models.

Figure 1 shows an excerpt of a component definition file that contains descriptions for the JButton JavaBean component and the Microsoft Internet Explorer browser COM component.

Each component definition is initiated by the component tag. The `xsi:type` attribute specifies the XML schema type of this component description. Besides allowing the validation of component files this `xsi:type` allows ACL/1 to uniquely identify the component's component model. Inside the component tag is a name element that uniquely describes a component within ACL. Optionally a description can be provided. Builder tools can exploit these descriptions to provide human readable information about the individual components.

The remaining part of the component definition consists of information about the components' underlying component models and hence is component model dependent. We have specified definitions for JavaBeans, Enterprise JavaBeans, Microsoft COM and CORBA distributed objects. In case of JavaBeans and COM components only the component's Java class name or a COM programmable name are required while other component models might require additional data such as naming information and information about event channels being used.

```
<component xsi:type="JavaBeans">
  <name>JButton</name>
  <description>A Java Swing JButton</description>
  <class>javax.swing.JButton</class>
</component>

<component xsi:type="COM">
  <name>IE</name>
  <description>The MS IE browser</description>
  <progName>Shell.Explorer.1</progName>
</component>
```

Fig. 1. Example for Component Description

Although the class name of the underlying component could be used to identify the component our approach has several advantages. It avoids implementation de-

pendent 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.

3.2 Connectors

Connector definitions declare the connection styles supported by ACL/1 and the properties they exhibit. They are used to model different interaction patterns and rules among components. Figure 2 shows how a connector definition that connects an event to a method call looks like.

As for components 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. A connector may be instantiated several times where each instance may use a different component as communication endpoint.

Other elements beside the name can be specified. The connector implementation has to evaluate these elements to provide the correct connector semantics. In figure 2 the source role contains an instance and an event parameter and the target role contains an instance and a method qualifier. These elements have to be specified in the configuration element when the connector is instantiated.

```
<connector xsi:type="binary">
  <name>EventToMethodCall</name>
  <role>
    <name>Source</name>
    <param name="instance" type="component"/>
    <param name="event" type="event-qualifier"/>
  </role>
  <role>
    <name>Target</name>
    <param name="instance" type="component"/>
    <param name="method" type="method-qualifier"/>
  </role>
</connector>
```

Fig. 2. Example for Connector Description

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 Application Configuration

An application 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 language constructs for the instantiation and naming of component instances. Instance identifiers have to be unique such that they can be referred to by the connection specifications. Figure 3 shows an application configuration that contains the two components defined above and that uses the formerly described connector to connect the action event of the JButton instance to the Navigate method of the browser instance.

Connection specifications denote the connector that has to be used for the interaction between a given instance. As can be seen in figure 3 the parameters defined in the connector definition have to be bound to actual values.

```
<configuration app-name="TestApp">
  <instance id="sourceComp">
    <name>JButton</name>
    <attributes>
      <entry key="bounds" value="..." />
    </attributes>
  </instance>
  <instance id="browser">
    <name>IE</name>
    <attributes>
      <entry key="bounds" value="..." />
    </attributes>
  </instance>
  <connection id="EventToMethodCall">
    <bind-role name="source" >
      JButton.action
    </bind-role>
    <role>
      <name>Source</name>
      <bind-to name="instance">sourceComp</bind-to>
      <bind-to name="event">action</bind-to>
    </role>
    <role>
      <name>Target</name>
      <bind-to name="instance">browser</bind-to>
      <bind-to name="method">Navigate</bind-to>
    </role>
  </connection>
</configuration>
```

Fig. 3. Example for Configuration Example

4 ACL/1 Transformations

This section shows how an application description using the constructs of the previous section can be transformed using XSL transformation processors. Before we explain the transformations that can be applied to ACL/1 we give a short overview on XSLT.

The Extensible Stylesheet Language (XSL) provides constructs for transforming XML documents and a vocabulary for formatting XML documents. In the context of our ACL only the transformation part [3] is used.

XSL Transformations can be applied by any W3C compliant transformation processor. Many transformation processors provide a proprietary API that can be used to enhance the transformation processor itself. We restricted our work to the portable XSLT file format to remain compatible with most XSLT processors.

Transformation rules are defined by `<xsl:template>` elements. The `match` attribute of these elements is responsible for distinguishing between different transformation templates. The transformation processor parses the source XML file and builds a document model. The processor traverses this document model and at each node it checks if the selection criterion of one template's `match` attribute is satisfied by the currently processed node. This selection criterion can be any XPath expression [4]. XPath not only supports the specification of element names, but provides for element values, existence and non-existence of attributes, and more complex expressions that look up ancestors and descendants in the XML document model. Different expressions can be joined with simple boolean operators. Once a template rule has been selected, its content part is further processed by the XSLT processor. If the content part of a template rule consists of further XSL specific elements, these elements are processed by the XSLT engine. All other elements are put into the target document. This target document can be an arbitrary text file although XSLT provides special instructions for generating XML files.

Since an automatic selection of transformation rules would be too static XSLT provides additional constructs for user-defined rule selection. Hence, the automatically selected order of processing can be overruled. One construct initiated by `<xsl:apply-templates>` forces the transformation processor to apply templates specified by the `apply-templates`'s `select` attribute to the current node. The predefined `<xsl:for-each>` element allows to build a simple loop expression to process nodes that have the same `match` criterion. XSLT provides also conditional processing based on XPath expressions by its `<xsl:if>` and by its `<xsl:choose>` and `<xsl:when>` constructs, both comparable to Java's `if` and `switch` instructions. In addition to rule selection of templates by the processor, XSLT supports the `<xsl:call-template>` instruction that calls a template similar to a subroutine call in conventional programming languages.

4.1 Executable Generation

We have implemented an interpreter for ACL that is included in the Component Workbench our composition tool [14].

To allow users the execution of ACL outside CWB standalone executables have to be created. XSLT allows us to transform a whole application described by ACL into a set of Java source files. These files can then be compiled and used without CWB provided all the components used by the application are installed on the target system. Figure 4 shows a simplified example stylesheet that shows how source code can be generated from ACL/1 code. Figure 5 shows the output after the XSL transformation has been applied.

Since the system representation in ACL is platform and language independent it is possible to create platform and language independent binaries provided all components are available on the target platform. Since this is only seldomly the case, we have implemented a mechanism that allows the substitution of missing components with other but similar components. These components need to be adapted either with our component mapping technique [8] or using type-based adaptation [7].

4.2 Application Refactoring

ACL/1 is based on XML. Hence, we can use XSLTs to apply refactorings. We have defined a set of XSL files that describe transformations on an ACL application.

Since different component instances in an ACL application are connected by connector instances it is reasonable to instrument these connection points to modify application behavior. One apparent modification is the instrumentation of the ACL files to enable the validation of user input. We have built a stylesheet that searches for specific connectors. When such a connector instance is found it is replaced by two connector instances and an intermediate validation component. The other ends of these two connector instances connect to the component instances of the original ACL file. Similar to aspect weaving in AOP it is not necessary to integrate this input validation in the ACL file. Furthermore, XSLTs allow us to use additional XML files where concrete input data boundaries can be specified. This data boundaries can be included into the transformed ACL file as input parameters to the validation component.

5 Future Work

In traditional programming languages building system documentation out of source code is easy. Tools such as JavaDoc or DoxyGen automatically generate documentation out of source files. No such tools, however, are currently available for ADLs. Using XSLTs, however, to also generate structured system architecture documentation out of the application description should be straight-forward. The documentation process can be supported by the developer by including additional description statements in the ACL file. Using XSLTs these documentation statements can be collected in a way similar to JavaDoc documentation. Additionally, the generation

```

<xsl:stylesheet>
<xsl:template match="configuration">
  public class <xsl:value-of
                select="@app-name"/> {
    <xsl:apply-templates select="instances"
                        mode="declaration"/>

    public void main(String[] args) {
        createInstance();
        layoutComponents();
    }
    public void createInstance() {
        <xsl:apply-templates select="instances"
                            mode="create-instance"/>
    }
  }
</xsl:template>

<xsl:template match="instance"
                mode="declaration">
  Object <xsl:value-of select="@id"/>;
</xsl:template>

<xsl:template match="instance"
                mode="create-instance">
  <xsl:value-of select="@id"/> =
    new <xsl:value-of select="@class"/>();
</xsl:template>
...
</xsl:stylesheet>

```

Fig. 4. Generation result

of formatted documents in many different formats such as PDF or the creation of architectural diagrams of the system should be also possible.

We also plan to write an XSLT to generate code for Microsoft's .NET framework out of ACL/1 architectures.

Our current implementation of ACL/1 uses Java classes that implement the low-level connectors necessary for the inter-connection of the components 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. Before we can attack this issue, however, it is necessary to gain more experience with our approach to verify whether this in fact is a problem.

```

public class TestApp {

    Object sourceComp;
    Object browser;

    public void main(String[] args) {
        createInstance();
        layoutComponents();
    }
    public void createInstance() {
        sourceComp = new JButton();
        browser = new IE();
    }
    ...
}

```

Fig. 5. Executable Generation (simplified)

6 Related Work

Various architectural description languages are in use today that have their focus on different issues. For instance, *Darwin* is solely concerned with the structural aspects of an architecture [10], *Wright* focuses on architectural behavior and provides a notion for handling with reconfiguration [2], and ACME is an architecture description interchange language [6]. UniCon [15] 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 [9].

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 [12] composition languages can make an application's architecture explicit, none of the existing composition languages allows to express an architecture in a way similar to ACLs. Unlike ACLs, 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 ACL 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 [11] the description is still too low-level.

7 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.

XSLT provides a convenient way to apply transformations on application description files. We have built transformations to generate source code for standalone Java programs as well as Java Applets out of ACL description files. Subsequent compilation leads to standalone executables and applets that do neither require XML files nor our ACL/1 interpreter. The transformations, we have written are generic enough to be used in combination with different ACL descriptions. Since the generation process can be extended to other programming languages and target systems, we plan to provide such transformations with future versions of ACL/1.

Different XSLTs have been used to implement simple refactorings. These XSLTs transform existing ACL/1 files into new ACL/1 files. Separation of concerns can be realized by applying such transformations.

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] 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.
- [3] James Clark. *XSL Transformation (XSLT) Version 1.0*. W3C, November 1999. <http://www.w3.org/TR/xslt>.
- [4] James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0*. W3C, November 1999. <http://www.w3.org/TR/xpath>.
- [5] Linda G. DeMichiel, L. Ümit Yal cinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification, Version 2.0*. Sun Microsystems, April 2001. Proposed Final Draft 2.
- [6] David Garlan, Robert T. Monroe, and David Wile. ACME: An architecture description interchange language. In *Proceedings of CASCON'97*, November 1997.
- [7] Thomas Gschwind. Type Based Adaptation: An adaptation approach for dynamic distributed systems. In *Proceedings of the 3rd International Workshop on Software*

- Engineering and Middleware*, volume 2596 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003. To appear.
- [8] Thomas Gschwind and Johann Oberleitner. Dynamic component extension to support cross-platform development. Technical Report TUV-1841-2002-19, Technische Universität Wien, March 2001.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a Composition Language. In *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 147–161. Springer-Verlag, 1995.
- [13] Johann Oberleitner and Thomas Gschwind. Requirements for an architectural composition language. Technical Report TUV-1841-02-20, Technische Universität Wien, June 2002. Presented at the 2nd International Workshop on Composition Languages.
- [14] Johann Oberleitner and Thomas Gschwind. Composing distributed components with the Component Workbench. In *Proceedings of the 3rd International Workshop on Software Engineering and Middleware*, volume 2596 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003. To appear.
- [15] 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.