DYNAMIC COMPOSITION OF COMPONENTS USING WEBCODS

G. Succi, W. Pedrycz, and R. Wong

Abstract

The ubiquitous presence of the Internet creates new opportunities for component distribution. Infrastructures for dynamic, web-based composition of software components appears to be greatly needed. WebCods targets this need: it is a web-based system that supports dynamic component composition over the web. This article discusses the component composition aspects of WebCods. Specific attention is devoted to the model of composition, the interconnection between components, and the implementation strategy. An example is presented.

Key Words

Dynamic component composition, component brokerage, architecture description languages

1. Introduction

The popularity of the Internet has altered the way producers deploy their software systems to customers. Customers can now directly download the required software from the web and execute the code in their local environment. The idea of using the Internet for matching the offer and the demand for software goods has been effectively described with the concept of the virtual agora [1].

A virtual agora is like the Greek Agora (agora means "square") where people meet, discuss, and exchange products. In a virtual agora, providers offer their products to suppliers using the web as a means. This model can be extended to "rent" software components to users. Such mechanism looks particularly effective, especially for new firms providing software components [2]. Clearly, a model like the virtual agora requires a suitable infrastructure.

WebCods (Web COTS-On-Demand System) is a system that provides a framework to support component composition over the web. WebCods identifies three major roles for component supply, identification, and composition: providers, customers, and a broker. Providers develop components and offer their components to customers through the mediation of a broker. The customer can use the components as they are, or compose them together either with a composition language or a graphical component editor.

WebCods raises several issues, including the payment schema, how to support the negotiation process, the structure of the dynamic, and web-based composition. This work focuses on the last topic. Components in WebCods can be dynamically loaded and composed to form new components. The composition mechanism includes datastreams, events notifications, files, and sockets as proposed by Deline [3]. The current version of WebCods supports Java-based components.

The article is organized as follows. Section 2 reviews existing proposals on component composition. Section 3 introduces WebCods and presents its component composition model. Section 4 discusses the implementation strategy for dynamic component composition. Section 5 contains an example application. Section 6 draws some conclusions and briefly outlines the direction of future research.

2. State of the Art

The research interest in component composition started more than 25 years ago. Since then, several Architectural Description Languages (ADLs) have been proposed to specify requirements for components and connectors, such as MIL [4], Rapide [5], UniCon [6], Wright [7], and ACME [8].

The Module Interface Language (MIL) is the first ADL used to describe the interconnectivity among software components [4]. A complete MIL description consists of a system tree made up of components. The interaction between components is specified by the availability and accessibility of resources. Conic, an extension of MIL, has been used to specify components in distributed environments [9]. The runtime environment supports dynamic configuration of the system. Components include ports that interact with the environment based on message passing. The interaction between components is constrained by interfaces defined in the environment.

Rapide supports modelling and simulation of the dynamic behaviour of component-based systems [5]. Components are defined using interfaces, connections, and constraints. The specifications of connections are embodied in components, and connecting mechanisms cannot be
Table 1
Comparisons of Supported Features of Component Composition in ADLs

<table>
<thead>
<tr>
<th>ADL</th>
<th>Connectors as 1st Class Entity</th>
<th>Supports Typed Components</th>
<th>Supports Typed Connectors</th>
<th>Supports Dynamic Composition</th>
<th>Works with Executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIL</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Conic</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Rapide</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UniCon</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Wright</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ACME</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ARMANI</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

changed after deployment. The language also limits the communication between components to event propagation. When communication links are instantiated, a specific implementation is required for each connection.

UniCon considers connectors as first-class entities [6-8]. It supports a predefined set of connectors with defined semantics: pipe, file I/O, procedure calls, data accesses, and remote procedure call. It specifies software architectures in terms of rules that define the admissible connections between connectors and components. The core components are typed, and a type-checking-like mechanism is used to ensure the validity of the connection. Components in UniCon can be source code, object-code components, or executable components.

Wright is an extension of UniCon that uses formal specification to specify protocols of interaction between components [9]. The protocols are defined by communicating sequential processes [10]. Since connectors and components are specified with formal specifications, properties of connections can be checked automatically for consistency and compatibility.

ACME is an Architecture Description Interchange Language used to provide intermediate representations for ADLs [11]. The core architectural entities supported are components, connectors, systems, ports, roles, representations, and re-visions. Compositions of entities are interpreted into identified architectural templates. These templates support the abstraction of information from the style and are used to interchange the architecture description between ADLs.

ARMANI provides a design vocabulary, design rules, and architecture style to capture details for architectural design [12]. It supports definition of styles in terms of component types, connector types, and architectural constraints.

Table 1 summarizes the comparisons of supported features between different ADLs. The table shows that UniCon is the only ADL that supports dynamic composition of executable using the predefined connectors. For this reason, UniCon has been selected as the reference for this work.

There are other approaches to perform component composition. Among these, it is important to mention RI SOLVE, which aims at reducing the gap between form specifications and actual implementations [13, 14]. Components are defined as abstract components (specifications) and concrete components (implementations). The abstract component defines the structural and functional properties of the encapsulated component, and the concrete component contains implementation-related information and the actual implementation of the component. The composition of the components is based on the interface specified by the abstract components.

Tools exist to support the development of a run-time system based on an ADL description. MIL, Conic, and UniCon have translators that generate skeleton code for the components. Other component composition tools are also available for static composition, such as Ciao [15], Larch [16], and WorkBench [13].

Espresso is a component composition model that allow users to compose in a tool-based environment [17]. The model requires each Espresso component to be a JavaBean to take advantage of the supported composition schema based on method invocation.

Compositional component composition framework supports the composition of binary components [18]. Components are loaded at start-up, and the framework does not support dynamic clustering of existing components to form new components. The interfaces and properties of components are specified in a separate system configuration file. All in this composition model, components are connected and via method invocation.

Code composition methods based on classes' interfaces also exist, such as IDL, Microsoft COM, COBRA, and E-Speak [19].

3. The WebCODS Component Composition Model

WebCODS is a system to provide brokerage of software components on the web. At runtime, components can be loaded and unloaded, and new components can be defined as composition of existing components. Each component in WebCODS is characterized by a connectivity descri-
code that start the applications. The description containing unresolved components is sent to the client machine with the Zip or Jar file of the classes to be used. On the client machine, the unresolved components are then resolved with the actual instances of the target objects available from the Zip/Jar files.

Description of the component is associated to components as a set of suitable objects; they are instances of classes corresponding to each connection type. In this way, we can use reflection to analyze typing details. This forms the "reference component." The reference component contains information about (1) the implementation of the component and (2) the types of ports. It is the actual entity that keeps the references to the running objects either directly or indirectly in a set of unresolved components. The information stored in the reference component can be automatically reverted to the UniCon-based description.

The reference component is used with both primitive and composite components. When it refers to a primitive component, the concrete implementation stored in the object refers to the start-up Java class. When the typed component represents a composite component, the component stores references to a collection of other reference components.

4.3 Connectors

Connectors are used to link components together. They require adaptations when different components are involved. These adaptations include customization of a connector's parameters and generation of component-specific glue-code to handle the connections. Consider the previous example in Fig. 3. Port ζ of A and port ζ of B are connected with a nonbuffered stream connector. Component A uses the method setOutputStream and B uses setInputStream to set up the reference to the pipe established by the connector.

```java
public class MyConnector extends NonBufferedConnector {
    ComponentB objectB;
    ComponentA objectA;

    InputStream in;
    OutputStream out;

    public MyConnector(ComponentA objectA, ComponentB objectB) {
        // Set up reference to the Pipe initialized in the connector
        objectA.setOutputStream(out);
        objectB.setInputStream(in);
    }
}
```

Figure 4. Possible Java code for the connection of A and B in Fig. 2.

Fig. 4 details a piece of Java code that could implement the required connection. However, this approach is not usable because the name of the actual procedures to be called, `setOutputStream` and `setInputStream`, are known only at run-time.

![Figure 5. Template used by the nonbuffered connector.](image)

Reflection is used for dynamic instantiation of classes and invocation of methods in arbitrary objects. Therefore, connectors can be rewritten using the reflection property and the template-method design pattern [20]. Each type of connector has a template to handle the connection. Fig. 5 illustrates the proposed solution for the nonbuffered connector.

An alternate approach could be based on dynamic generation and compilation of components. This approach has been investigated and rejected, as it would result in higher design and computational overhead.

5. Example of Application

In this section, we detail a scenario using the system: a service provider has two component offers to the WebCDS broker: a C/C++ Preprocessor and a C Language Software Metrics Extractor. The preprocessor is the front end of a C-based language compiler. It processes compiler directives and outputs preprocessed C/C++ code in a format suitable for subsequent analysis. The Software Metrics Extractor extracts metrics from source code and displays them. The UniCon-based descriptions of these two components are shown in Fig. 6. Both components are primitive.

The Preprocessor starts with the class Preprocessor. The preprocessed code is output to an output stream through an "out" pipe port -PreprocOutput. The reference to the output stream is set by the method setOutputStream. The Software Metrics Extractor starts with the class CParser. The component accepts source code input from an input stream through an "in" pipe port -CodeInput. The reference to the stream is set by the method setInputStream. For simplicity, we omit the description of the port of the input file for the Preprocessor and of the output file for the Parser.

![Figure 6. UniCon descriptions for C/C++ Preprocessor and CPParser in WebCDS.](image)

Clients in WebCDS can assemble a C Software Metrics Extraction System by downloading the Preprocessor and the C Software Metrics Extractor, and by linking them...
together. Fig. 7 shows the resulting graphical component editor panel when the client has downloaded the primitive components and linked them together with a buffered stream connector with a buffer size of 1,024 bytes.

Figure 7. The panel of the graphical component editor used to link components.

A C Software Metrics Extraction System can be built by linking the preprocessor and the software metrics extractor. The new composite component can be bundled by the provider. The formal description of the newly bundled component is generated automatically by WebCods (Fig. 8). Such a description could be used to invoke the new component, which can be made available to other clients through WebCods.

![Component Description](image)

Figure 8. UniCon description for CmetricsExtractionSystem.

6. Integration of Third-Party Tools

Existing Java applications can be converted into WebCods components. The conversion requires a wrapper class to:
1. Characterize the interface of the component
2. Provide interaction points for connectors to communicate to the component
3. Specify the execution sequence of the component

For example, a Java application, dxf2j3d, is a Java freeware utility that converts an object drawn out of 3D Polylines in AutoCad in .dxf file into a ready-to-compile Java3D java file [21]. The application contains four classes bundled into a jar file. The command line used to execute the application is:

```
java dxf2j3d <filename.dxf> -<option>
```

The interface of the component contains an input port to read file and an output port to output file. The input port provides the name of the file that is going to be processed by the utility. The input file name supplied from the connector is saved in the wrapper class and used when the component is executed. After execution, the component outputs the result in the file. The final step is the mapping of the output file to the file specified in the output file connector.

The command line execution of Java application can be simulated in the environment by invoking the main method. The arguments required for execution are supplied at the time of execution. The wrapper class required for dxf2j3d utility is shown in Fig. 9.

```java
public class DXF2J3DConverter extends Thread {
    String outputFileName, inputFileName;
    public void setOutputFileName(String name) {
        outputFileName = name;
    }
    public void setInputFileName(String name) {
        inputFileName = name;
    }
    String[] arguments;
    public DXF2J3DConverter(String[] args) { arguments = args; }
    public DXF2J3DConverter createInstance(String[] args) {
        return new DXF2J3DConverter(args);
    }
    public void run() {
        //Create reordered arguments
        String[] newArguments = messageArguments;
        //Execute the application
        DXF2J3D.main(newArguments);
        //Mapping the current output with the connector
        mappingOutputFile(outputFileName);
    }
}
```

Figure 9. Wrapper class for dxf2j3d utility.

The wrapper class provides two methods for File connectors to set up input and output file names. The names are stored in the class as instance variables and used when the component executes.

The passing of runtime parameters to the component uses the createObjectWithParameters method. The method in this case does not actually create any object instances of dxf2j3d; it stores the parameters to be used when the component instantiates.

After the wrapper class is developed, it is compiled and bundled into a jar file with all other necessary executables. The component provider can create the corresponding description for the newly formed primitive component and market it as a component in the broker.

In the composition environment, the user downloads the DXF2J3DConverter, a file editor with a file output port and a viewer with a file input port. After downloading the components, the user performs component composition by linking them together with two file connectors. The file connectors are specified with the names of files being input into and output from DXF2J3DConverter. Fig. 10 shows the use of file connectors to composed application with downloaded components.

The DXF2J3DConverter can be executed with parameters supplied at runtime. Fig. 10(b) shows that the converter is instantiated with the -nb option that turns off the use of bubble sort in the CAD layers. After all the parameters are set, the application is ready to be executed.
Figure 10. (a) Composition and (b) execution of the downloaded components in the WebCODS environment.

7. Conclusion

This article describes a system called WebCODS to support dynamic composition, retrieving, and transmission of software components. The key areas supported by WebCODS environment are:

1. Dynamic component loading and unloading to providers, clients, and the component broker
2. Transferability of WebCODS components via the network with existing or proposed protocols
3. Instantiation and execution of the component upon arrival in the clients’ environments
4. Support of dynamic composition with other components

The current system focuses on precompiled Java components and UniCon-based descriptions. As in UniCon, the system provides a predefined set of connectors for the composer to link components together. The composition occurs dynamically, taking advantage of Java reflection: it is well suited for use with COTS and web-based software distribution schemes.

The next steps in this research include the extension of WebCODS to support a language-independent ACME-base component specification language; a description in ACME can be translated into a description in most other ADLs. In addition, further kinds of connectors will be supported, such as procedure calls and shared variables. Because the rules for composition are based on type-checking, more extensive composition rules are also needed to handle consistency.

Acknowledgements

This research has been partly supported by the Canadian Natural Sciences and Engineering Research Council, the Government of Alberta, the University of Alberta, the University of Calgary, and the Alberta Software Engineering Research Consortium. The authors also thank Eric Liu and Jason Yip for reviewing early drafts of this work.

References


Appendix

The syntax in BNF format of the modified UniCon language follows.
Component := COMPONENT ComponentName
   (PortList)*
   [Implementation]
END COMPONENT

PortList :=
   PORT PortName MODE aMode IS Type
   MEMBER(StringWithMemberName)
END PORT

Implementation :=
   IMPLEMENTATION
   [INSTANCE InstanceName = ComponentName]*
   [Connect] *
END IMPLEMENTATION

Connect :=
   CONNECT PortName TO PortName (*, PortName)*
   TYPE Type IS ConnectionMethod
   [PARAM attribute]

Mode :=
   IN | OUT | INOUT

Type :=
   EVENT | FILE | PIPE | SOCKET | DATAGRAMSOCKET
   | BUFFEREDCONNECTOR | NONBUFFEREDCONNECTOR
   | SOCKETCONNECTOR | DATAGRAMSOCKETCONNECTOR

ComponentName, InstanceName, PortName are the usual C-like identifiers. StringWithMemberName is a string with the name of the method to use in it.

Biographies

Giancarlo Succi, Ph.D., P.Eng., is Professor at the faculty of Computer Science of the Free University of Bozen, where he directs the Center for Applied Software Engineering. The target of his current research involves software engineering over the Internet, that is, how it is possible to organize geographically distributed teams to develop and sell products using the Internet as core medium; object-oriented business modelling using UML and activity-based costing and management; theoretical and applied investigation of the effects of integrating a reuse policy and an object-oriented approach inside a software development process, taking into account the analysis of the domain, programming languages, process maturity, productivity, quality, long-term returns of investment, effects on the market structure, and relevant accounting techniques; and determining the influence of compatibility on the reuse of coarse-grained software components and product. Dr. Succi is a registered professional engineer in Alberta and in Italy, and official consultant of the court of Genova, Italy. He is a consultant for several North American and European firms and for the European Commission, and serves as reviewer and member of the editorial board of international scientific journals and magazines. Dr. Succi is the author of more than 100 papers published in journals, proceedings of conferences, and books. He has also authored a book on software reuse and is the editor of a book on extreme programming and one on logic programming.

Witold Pedrycz, Ph.D., P.Eng., is Professor, Canada Research Chair, IEEE Fellow, and Department Chair at the Department of Electrical and Computer Engineering of the University of Alberta. He is actively pursuing research in many areas, including computational intelligence (CI) software metrics, dynamic objects composition, knowledge discovery and data mining, and knowledge-based systems. His research is supported by several grants from NSERC, the Province of Alberta, the Canadian Space Agency, and industry. He has published more than 100 conference papers and more than 200 journal articles as well as several research monographs and textbooks. He is also a co-Editor-in-Chief of the Handbook of Fuzzy Computation (Oxford University Press). Dr. Pedrycz has served on many program committees of international conferences and has delivered numerous invited talks and tutorials. He has been serving on the editorial boards of journals on fuzzy set theory and neurocomputing, including IEEE Transactions on Fuzzy Systems and IEEE Transactions on Neural Networks. He has been actively engaged in the process of engineering education, by serving as Associate Head of Computer Engineering at his previous position at the University of Manitoba, and by participating in several accreditation visits to various Canadian universities. Dr. Pedrycz contributes to the community of professional engineers as well as to the IEEE.

Raymond Wong received his B.Sc. in electrical engineering with a computer engineering minor at the University of Calgary in 1998 and his M.Sc. in electronics and computer engineering at the University of Alberta in 2001. He is now with PSynch in Calgary, AB. His research interests are in dynamic component composition and web-based applications.