Distributed Components Composition Using .NET

José Luis Pastrana
University of Málaga
Despacho 3.2.51 ETSI Informática. Campus Teatinos s/n
Spain (29071), Málaga
pastrana@lcc.uma.es

Ernesto Pimentel
University of Málaga
Despacho 3.2.10 ETSI Informática. Campus Teatinos s/n
Spain (29071), Málaga
ernesto@lcc.uma.es

Miguel Katrib
University of La Havana
F.Matemática y Computación.
San Lázaro y L, Vedado,
La Habana, Cuba
mkm@matcom.uh.cu

ABSTRACT
Software systems grow each day in size and complexity. In an effort to manage increasing complexity and to maximize the reuse of code, the software engineering community has, in recent years, put considerable effort into the design and development of component-based software methodologies and tools. Inspired by the "Design by Contract" metaphor proposed by Meyer in [MEY00], this paper presents a simple, useful and elegant methodology for component composition, coordination and dynamic adaptation based on semantic web techniques. Our proposal is based on connectors enriched with contracts, making software architecture more explicit at implementation level. Components and connectors in our proposal will be Web Services implemented in C# using .NET platform.

Keywords
Components, Composition, Adaptation, Contracts, Web Services, Semantic Web.

1. INTRODUCTION
Most software technologies support component-based design. Generally this support exists only in the sense that components are syntactic modules (black boxes showing only their interfaces): systems can be readily constructed from components that are designed and implemented by other parties. Components can be compiled separately, and they can be composed at compiling time or at run time.

One problem with current component based software engineering practice is that component interfaces are generally just a syntactic definition and not behavioral modules. Therefore, normal situations such as software systems developed by teams, the replacement of individual parts of a software application, and the component software market, require the definition of the behavior, interaction and coordination among components. However, such information cannot be extracted from the signature description of the component (the so-called interface).

In addition, there are many problems to be considered when designing and implementing components for open systems where the life of a component is usually shorter than the life of the application and some components could be inserted, modified and dropped during the lifetime of the application. Furthermore these components may be developed by different teams, technologies, platforms, etc. Therefore, such problems should be solved if we want to have a component market where developers only want to put components to work in their applications.

This way, a component designer only has to keep in mind what the component does (its functionality) but he/she knows neither the users nor the application where it will be used. So, it is suitable to have a clear separation between the functional aspects of the components and other requirements such as synchronization, coordination, persistence, replication, distribution, real time, etc., specific to the application scenario where the components will be used and known by the whole system designer.

Allen and Garlan, for modeling software architectures, distinguish between implementation relationships and interaction relationships of software modules or components: "Whereas the implementation relationship is concerned with how a
Component achieves its computation, the interaction relationship is used to understand how that computation is combined with others in the overall system” [AG94]. They propose a formal model for software design that makes the interaction relationships between components explicit by using the abstraction of a connector. Describing software architectures in terms of interaction relationships among components brings us closer to a compositional view, and hence a more flexible or open view of an application. Thus, our work proposes the connectors as a way of viewing an application’s architecture as a composition of independent components.

Connectors support better flexibility, since each component could engage in a number of different agreements, thereby increasing the potential reuse of individual components. Separating connectors from components also promotes reuse and refinement of typical interaction relationships, allowing the refinement of connectors and the construction of complex connectors out of simpler ones. But interaction relationships are rarely captured by programming language constructs as method calls. In this sense, traditional object-oriented languages provide little support for explicit representation of software architecture.

Type hierarchies are the only design elements explicitly visible at implementation level - but they represent inheritance relationships, and do not reflect an application’s architecture. In contrast, interaction relationships, such as coordination and synchronization of a group of objects collaborating to achieve a task, manifest themselves as patterns of message exchanges.

Such patterns of communication have a logical and conceptual identity at design level but this identity is lost when we move from design to implementation because the information about such collaborations is spread out among the participating objects. The loss of this information makes the resulting application opaque with respect to its architecture, difficult to understand, to reuse and to reenginee.

The first step towards more explicit application architecture at code level is to enable the location of information about interactions inside the application’s code. In this paper we propose the following solution: enriching component interfaces with contracts. These contracts are formed by logical asserts and the behavior we want when the asserts are not satisfied. Besides this methodological proposal we have developed a tool for generating automatically the connectors. This idea represents the interaction relationships between components. The main contribution of this paper is to provide connectors at implementation level. Therefore connectors will be runtime entities that not only describe, but actually control inter-component communication and behavior against unexpected situations.

2. CONNECTORS
We have talked about open system, components, etc. Firstly, it is necessary to fix some definitions that will be used throughout this paper:

A system will be considered as a set of tools and mechanisms allowing users to create and interconnect software components as well as a set of services to help components running on it. Component interfaces present a syntactic definition of the services (operations) offered by the server, as well as the types and parameters needed. And, components are binary software units (executable by a machine) that can be inserted into a system and put to work [SZY 00].

Component-based Software Engineering is concerned with the development of systems from reusable parts (components), the development of components, and also system maintenance and improvement by means of component replacement or customization. Building systems from components and building components for different systems requires established methodologies and processes not only in relation to development/maintenance phases, but also in relation to the entire component and system life-cycle including organizational, marketing, legal, and other aspects.

The current proposal presents a real implementation of the [AGA96] software architecture model which has been improved with contracts and dynamical adaptation using semantic web techniques.

The main idea of the approach presented lies in the separation of the roles. This way, the component designer will be the person taking care of the component functionality and the system designer will be the person in charge of making the system work properly. The system designer (a client component) takes a set of components that have a well-known functionality and he/she builds the system interconnecting those components by means of connectors where he/she is able to express the semantics of the system in terms of a set of contracts.

Nowadays, Web Services are an important emerging technology for software component development and integration. [W3C01] defines a Web service as a
A software system designed to support interoperable machine to machine interaction over the internet. It has an interface described in a machine-processable format (e.g., WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. WSDL describes the static interface of a Web Service. It defines the protocol and the message features of end points. The goal of Web Services is to automate and to make business process collaborations easier both inside and outside enterprise boundaries. Useful business applications of Web services in business to business, business to consumer and enterprise application integration environments will require the ability to carry out complex and distributed Web service integrations and the ability to describe the relationships between the constituent low-level services.

Web Services are a growing technology based on standards, a fact which has motivated our decision to study the development of connectors. However, WSDL describes only the "static interface" of a Web Service, then several questions arise when using and reusing components in applications: How can you trust a component? What if the component behaves unexpectedly, either because it is faulty or simply because you misused it? We need to determine whether we can use a component within a certain context. We would like a specification that tells us what the component does without entering into the details of how it does it. Furthermore, such a specification should provide parameters in order that a kind of contract between the component and its users, meaning that the WSDL would be enriched.

Components are mainly for reuse. One of the essential tasks in the component based software development process is finding the right component providing the functionality and interface expected by component consumers. Once the right components are identified, a connector is used to adapt the component behavior without modifying the components themselves. A connector is an external module that sits between the component consumer and the component server and mediates the mismatch between them. Hence, the connector mechanism does not alter any internal part of the component.

Therefore, a connector is a component for managing, connecting and adapting the behavior of one (or more) server component. A client component wanting to get some services from a server component has two possibilities: it can either use a server using a given behavior only using a known connector, or it is possible to obtain a different behavior by developing a new connector. A connector tests the behavior of a server and schedules which requests will be served, when they will be served and which server can attend the reclaimed service (in the case of a connector having more than one server attached).

All the constraints set for the server are established as contracts (preconditions, postconditions and invariant) and represented by logical expression. The connector tests the invariant and preconditions for each service before calling it and tests postconditions and the invariant when the server returns. The connector can also execute some defined actions when the contract fails. This way, a client is able to set how he/she wants the server to behave and he/she is also able to set what should be done when the their requirements fail. Our proposal starts from server features definition (its interface) which is extended to define server location, requirements and behaviors. Connectors have the following features:

- Connectors are defined by clients.
- Connectors use contracts to express the non-functional properties expected by the client.
- Connectors are Web Services that will be automatically generated. The tool COMPOSITOR (COMPOnent connectorS Ipso facto generaTOR) takes the extended component interface as the starting point for connector generation.
- Connectors allow clients to work with one or more different servers having the same behavior.
- Connectors are able to schedule and balance the computation charge, when a connector manages more than one server supporting fault-tolerance when a server goes down.
- Connectors allow clients to monopolize a server (or several servers) using the predefined predicates Available() or Available(name) as part of a precondition.
- Connectors are able to be enriched statically and dynamically with an ontology of the server domain in order to adapt
requests on the domain context using semantic web techniques.

3. CONNECTOR FEATURES

We have developed a graphical tool named COMPOSITOR to make the connector definition easier starting from the remote component. COMPOSITOR reads the WSDL of the remote component (its location as an url is needed) and it presents you the interface of the remote component in a familiar syntax (C#-style). Then the user may define his/her own non-functional requirements via contract and the behavior to be executed when contracts fail. Because a connector is a Web Service implemented in C# and ASP.NET, preconditions, postconditions and invariants will be valid boolean expressions written in C#. The behaviors associated to each contract are C# sentences. Both, contracts and behaviors can use a number of predefined predicated and functions like Available, Fail, Retry, Suspend, timeout, etc. In addition, external sentences can be used to define other components (or connectors) you are going to use in the context of your application allowing the use of external components as part of the contracts.

We will use an example throughout this point in order to obtain a better illustration of the features of the connectors. Let us imagine we want to develop a client application (for example a web application) for an auction system. We have the Figure 1 description of the AuctionHouse component that manages all the auction processes.

![AuctionHouse Interface](image)

We show below the syntax and the semantics of these predefined functions and predicates available for the connector designer.

- External Components (external at): They express external dependencies of the component. This allows us to ensure that all the external components required (managed as remote references) are ready. External references are active throughout the whole life of the connector. Syntax:

  ```
  external < type > < name > at < location > :
  ```

  Example: (not used in the chess example)

  ```
  external PayPal pp at "http://.../PayPal.asmx";
  ```

- Preconditions (require): They set up the conditions that the client wishes to be fulfilled before the service is carried out. Syntax and Example:

  ```
  require: < boolean expression >
  ```

  Example: require: (me == GetHighBidder(ItemId))

- Postconditions (ensure): They set up the conditions that the client wishes to be fulfilled after the service is carried out. The predefined variable _result could be used in order to express properties about the returning value of a service. Syntax:

  ```
  ensure: < boolean expression >
  ```

  Example: ensure: (abs(_result)==1)

- Invariant (invariant): It sets up the properties that should be completed in order to consider the component is in a consistent state. In this example, the number of the players connected cannot be greater than two (0 means no one is playing, 1 means there is a player waiting for an opponent and 2 means the game has begun).

  Syntax and Example:

  ```
  invariant: < boolean expression >
  ```

  Example: invariant: (Currency>0)

- Behavior Statements (on failure): They allow setting up the behavior of the component in the case where any of its contracts were not satisfied. The default behavior is an exception, but any C# sentence and the predefined primitives (Suspend, Retry and Fail) can be written.

  ```
  on failure: < boolean expression >
  ```

  Example: on failure: (abs(_result)==1)

- Behavior Statements (on failure): They allow setting up the behavior of the component in the case where any of its contracts were not satisfied. The default behavior is an exception, but any C# sentence and the predefined primitives (Suspend, Retry and Fail) can be written.
is activated. To avoid the loss of information concerning a change in the state of the server component produced by the non desired and non recommended access to the server component directly instead of using the connector, there is also a background thread that periodically reviews suspended services when there has been no activity for some time. For example, two people can not bid for the same item at the same time. Example:

```java
void Bid(int ItemId, int BiderId, double Amount)
require: (Available())
on failure: Suspend;
Retry retries the service. It is typically useful when some "repairing" action can be done before retrying. It is also useful for debugging. For example, the amount bided has to be greater then previous one. So we could retry a service adapting the situation to avoid errors. Example:

```java
public void Bid(int ItemId, int BiderId, double Amount)
require: (Amount>GetItemInfo(ItemId).Currency)
on failure: Amount = GetItemInfo(ItemId).Currency + 0.50;
Retry;
```

`Fail` raises an exception to the client. So, it is the client who must manage that exception. For example, an exception is raised when somebody tries to buy one item and he/she is not the higher bidder. Example:

```java
void Buy(int ItemId, int BiderId)
require: (BiderId == GetHighBidder(ItemId))
on failure: Fail;
```

4. DYNAMIC ADAPTATION

Component-Based Software Engineering (CBSE) focuses on building large software systems by integrating existing software components. The old notion of developing a system by writing code has been complemented by assembling existing components. The main aim of CBSE is to enhance the flexibility and maintainability of systems. Some of these systems are critical, and their maintenance should be ongoing. Such systems must be adapted on the fly. In this section, we focus on the dynamic adaptation problem (dynamic means the ability to change an application at run-time).

Adapting a component-based application means adapting one or more of its components, and in general, adapting a component at run-time means disconnecting it from the application and connecting a new version of this component [KET02]. A dynamic adaptation may be performed for multiple reasons classified into four categories: corrective, adaptive, extending or perfective.

A corrective adaptations removes the faulty behavior of a running component by replacing it with a new version that provides exactly the same, but correct, functionality. An adaptive adaptation adapts the application in response to changes affecting its environment (OS, hardware components, etc). The extending adaptations extend the application by adding new components to provide the desired new functionalities. And the perfective adaptations aim to improve the application even when it runs correctly. For example, replacing a component for a new one with optimized implementation.

Independently of the reason for the adaptation, several types of adaptation may be identified: the architecture adaptations affect the structure of the application. This can be performed by adding new components, removing existing components or modifying their interconnections.

An implementation adaptation is motivated by performance, correction reasons or environmental changes not considered when the component was first implemented. Therefore, the interfaces exposed by the component must be maintained the same.

An interface adaptation modifies the list of services provided by the component. In component-based software, that can be performed by adding
(removing) some services in (from) the list of services supported by the component.

A location adaptation corresponds to the migration of components from one site to another, to balance load for example. That does not affect the application’s architecture, however, the communication between the moved component and other components should be adapted according to the new location.

Multiple requirements must be satisfied to accurately perform an adaptation. The most important requirements to keep in mind are consistency, performance and degree of automation.

The consistency concerns the adaptation operation which is instigated by the control application (the application used to adapt the running application) has high priority. However, it does not mean granting all rights to the control application to do anything at any time. The adaptation operation must preserve the application consistency.

The performance is related to the duration of the adaptation and the number of components affected. Even though the adaptation operation is seldom carried out during the running application life cycle, it should be effective, and its duration should be as minimal as possible. Also, the number of components affected by the adaptation operation should be minimal.

The degree of automation is also important. It represents the ability of an application to adapt itself; this is possible because during run-time, the application has all the information and capabilities necessary to carry out such an operation.

As we have mentioned before, adapting a component-based application means in general, adapting a component at run-time which implies disconnecting it from the application and connecting a new version of this component. Therefore, the use of connectors has to provide a way of changing the component you are connected to as well as the connector you are using at run time.

Changing the connector you are using (maybe you want new contracts or behavior) is not a problem. The connector is represented by its url and this is stored and managed as an string. So, the client application only has to change the url of the connector to connect to another and this can be done either at compile time or at run time.

The problem arises when we want to change the component we are connected to. On the one hand we have to inform the connector that the server component has changed, but on the other hand the new server can be slightly (or very) different from the previous one and their interfaces may be different. Therefore, all the problems found have to be solved as automatically as possible at run time by the connector.

The way to switch to another server component is very easy. You only have to set the new server component using the predefined primitive void nowConnectTo(string url);

There is no problem when the new component has the same interface as the old one. However, the problem arises when the interfaces are not equals but the services they offer are (or could be) equivalent. Three different kinds of conflict will be identified:

- **Name Conflict**: it happens when the same service has a different name in the new server but its parameters are the same.
- **Parameters Conflict**: it happens when the same service needs different parameters though the information provided is the same (maybe different types, distribution, etc).
- **Composition Conflict**: this is the hardest problem and it happens when there is no a direct equivalence among services. However, old services can be seen as the composition of two (or more) new services.

The objective of the Semantic Web is to provide languages to express the content of Web pages and to make accessible to agents and computer programs the information that those pages contain. More precisely, the Semantic Web is based on a set of languages such as RDF and OWL that can be used to markup the content of Web pages. The second element of the Semantic Web is a set of ontologies, which provide a conceptual model to interpret the information provided.

The Semantic Web has the potential to provide the Web services infrastructure with the semantic information that it needs. It could provide formal languages and ontologies to reason about service descriptions, message content, business rules and relations between these ontologies. In this way, the Semantic Web and Web services are synergistic: the Semantic Web transforms the Web into a repository of computer readable data, while Web services provide the tools for the automatic use of that data.

One approach solving these levels of conflicts is to define ontologies depending on the application domain. In the context of knowledge sharing, the
term ontology means a specification of a conceptualization. That is, ontology is a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents. This definition is consistent with the usage of ontology as a set-of-concept-definition, but more general. And it is certainly a different sense of the word than its use in philosophy. What is important is what ontology is for. We design ontologies so that we can share knowledge with and among these components.

Ontologies have been used in the context of translation. The idea of using ontology to solve the adaptation problem is born from the fact that the only knowledge we have to make that adaptation is the server interface: service names, parameters, etc. The grammar used in natural language is wide and complex; however, programming languages have a well defined grammar. In our particular case, we only have to keep in mind the invocation grammar which is as shown in Figure 2

![Invocation Ontology](image)

Looking at the features found in OWL Lite we find the Equality and Inequality properties. Those properties allow us to set which concepts are equivalent to other ones. We will use in the current proposal: equivalentClass, equivalentProperty and sameAs.

This way we can set that one service can be replaced by other if the are the same (sameAs) or they are properties that are equivalent (equivalentProperty). In addition, two types or two components can be replaced when they are the same (sameAs) or they belong to an equivalent class (equivalentClass).

5. RELATED WORKS

The proposal presented is based on the idea of developing a system by the connection of pre-existent components distributed all over the internet (the functional properties of the system), their coordination, synchronization and adaptation using contracts and the behavior defined when those contracts fail (the non-functional properties of the system).

On this idea of separating functional and non-functional properties of a system or an application, the Aspect Oriented Programming (AOP) uses new languages or extensions of existent languages to describe the different aspects in one application. That works well in sequential systems. But the problem arises when defining aspects over distributed components where the weaver cannot weave functional and non-functional codes. One solution is presented in the Dynamic Wrappers [PAW02] proposal. It is based on defining before, after, and around advices for the remote objects and wrapping middleware objects (stub/skeletons). However, it does not allow us to have different behaviors for the same remote component and cannot suspend a call.

On the other hand, the work carried out by Cristina Lopes [LOP98] in the field of the COOL language could resembles our work, because she uses a coordinator with some functions similar to our connectors. The Coordinated Roles [MUR99] work improves on Lope’s model because it allows us to generate, change and modify the coordination at run-time. However, the difference can be found in where the coordinator is and who sets the rules of the coordination. Lope’s coordinator is tied (a thread) to the server which sets the rules of the coordination and synchronization. However, a server can be accessed by many different connectors at the same time and the connectors are defined by the clients. It is the client who determines the server behavior, the coordination and synchronization rules.

Another interesting proposal is the Polyphonic C# [BEN03]. Polyphonic C# extends the C# programming language with new asynchronous concurrency abstractions, based on the join calculus. The language presents a simple and powerful model of concurrency which is applicable both to multithreaded applications running on a single machine and to the orchestration of asynchronous, event-based applications communicating over a wide area network. However, Polyphonic C# is event oriented, not service oriented and the join calculus used to coordinate is done and defined on the server side, for example:

```csharp
public String get()& private async contains(String s)
```

Finally, Web Services Choreographies [W3C01] such as BPEL4WS or WSCI, are message interchanging models that tell us how components interact in a system, but tell us nothing about properties of the system and what to do when something goes wrong.
6. CONCLUSIONS AND FUTURE WORKS
As far as we know, there are many proposals for component synchronization, coordination, adaptation, etc., but none of them manage all the problems at the same time. So, we present a methodology and tool for developing distributed component systems giving solutions to the most of the problems that can be found such as synchronization, coordination, adaptation, etc.

Connectors are implemented as Web Services. This is very important because Web Services are based on international standards. Therefore, connectors can be used for any application or platform (not just Microsoft or .NET).

This proposal allows a client to set server requirements and behaviors when contracts fail, in contrast to the traditional way where the server is the one who sets what it requires and what happens when these needs are not satisfied.

As future work, we want to improve our tool for allowing remote server components using CORBA, TCP, etc. This way, COMPOSITOR increases the interoperability skill in a system because a connector is not just a wrapper, it can coordinate and manage behaviors. Currently, we assume all the remote components are Web Services, but it is only necessary to generate a different proxy for a different remote server type.

As well, we are going to continue studying the semantic web techniques to improve a better adaptation and studying the inference mechanisms used in the different proposals in order to adopt the one we think it will better for our model or developing our own inference to find the server, service or type equivalent to the failing one.

7. REFERENCES