# Coordinating Web Services Using Channel Based Communication

Theophilos Lemniotes and George A. Papadopoulos

*Department of Computer Science*
*University of Cyprus*
*75 Kallipoleos Str, P.O.B. 537*
*CY-1678 Nicosia, Cyprus*
*E-mail: {theo,george)@cs.ucy.ac.cy*

Farhad Arbab

*Department of Interactive Systems*
*CWI*
*Kruislaan 413, 1098 SJ Amsterdam*
*The Netherlands*
*E-mail: farhad@cwi.nl*

## Abstract

*In this work we investigate the use of a new concept in component communication during the coordination of Web Services, expressed by the channel based coordinating communication system called Reo. The role of Reo is to construct and manage connectors, the latter being patterns of connected channel communicators. The communication and coordination of components lying over a distributed address space has been dealt so far with stream or datagram connections created and controlled by the participating calculation and coordination components. Web Services can take advantage of the Reo channel system that separates communication from computation concerns using components that have independent sink and source ports that can be attached to Web Services components, thus overcoming the problem of compatibility in distributed systems. The flow of information is entirely regulated by the channel interconnections.*

## 1. Introduction

Coordination is important for Web Services: such systems combine services that are located on different sites and this combination implies the need for coordination of their activities in order to regulate the flow of information and guarantee the reliability of the shared information. The aim of this study is to integrate a number of languages and protocols in order to facilitate the transportation of the functionality of the *Reo* system ([2]) across the Internet. *Reo* is a channel based coordination system. The dynamic connectors of channels that the functionality of this system offers are used in the management of the communication of distant components. The primitives of the *Reo* coordination language offer a great variety of synchronous and asynchronous channels with respect to access rights, mutability, reliability, grouping of connecting nodes, and execution model ([1]). The objects are all distributed across the participating machines and a web service invocation is achieved with SOAP-XML messages over TCP/IP connections that at the bottom level implement these channels. The architecture described is similar to that of an XML based multiple heterogeneous system ([4]).

## 2. *Reo* and Coordination Among Components

In many network architectures, each process in the network is either client or a server. This channel-based system is an application of synchronization and communication (through an adequate channel system) rather than a combination of proxies of services to build a web component. This is how the basic architecture of the system should look like.

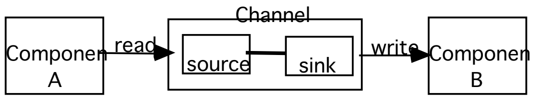

**Fig. 1:** A *Reo* Channel

By definition every channel in *Reo* represents a connector ([2]). More complex connectors are made out of simpler ones. The `create` operation creates channels with specified *channel end unique identification variables* (cev) that either can be source or sink end. Every cev is said to coincide on a node N that it may be connected, with the `connect` operation, to one (among many of) component instance(s). The operation `disconnect` applies to a channel end likewise, while `read` suspends the instance that performs this operation waiting for a value that can much a data pattern *p* that is expected to be read into a variable. The operation `take` is a destructive variant of `read` and the channel loses the value that is read. Similarly, the `write` operation suspends the operation of the calling instance until its value is written to the source channel end port. Moreover, there is a `wait` operation that suspends the operation of the executing component if some predefined *nconds* conditions parameters become true. The node operation `join` merges two nodes (with their connectivity) to one, and `split` produces a new node N' and splits the set of channel ends that coincide on a node N between the two nodes N and N' according to the set of the specified edges. The node operation `hide` hides a specified node N from participation in the future node-modifying operations. Further node operations are `forget` that changes cev (or connected to it node N) so it no longer refers to the channel end it designates and `move` that relocates `cev` or `N` to a new location given as `loc.`

The ***coordination*** of web components should include primitives for *channel communication* like `create, read, write`, etc. ([1]). The construction of and access to a web component is performed with XML function transportation ([8]). The basic idea is that a web service should be able to receive and transmit its operation related information to a sink requestor component via a *Reo* channel. Moreover, a third attached component would be able to influence the flow of data between the original two components with its own absorption of data ([2]). The components that execute the *Reo* operations (`read write` and `take`) work with constructed components that implement a communication pattern independently, i.e. the server does not have to know about the requestor. Of course this pattern can change with the execution of certain commands like create a channel, connect to a channel and join two channel nodes (these operations help in the channel management) at run time so the structure of the formed connectors is not static. These operations can be classified as the management construction primitives while the former implement the data message passing primitives. This functionality is very useful because it separates the communication components (created by web channel services components) from the coordination concerns (incited by the web services). This in turn makes our system much more flexible with the ability for dynamic change of the communication pattern while running, as well as for dynamic change of the coordinating components. The overall architecture for such a system involves three types of components:

- The actual web services and client components.
- The channel services components.
- The "Channel-end Register" node components.

The latter plays the role of a service register of a web service whose main task is to offer channel end references (according to a certain algorithm). So the management of this channeling system is actually undertaken by node processes which act like brokers. The former is a candidate for using the channel-end registry and to do so it has to initially create a channel locally (with a primitive `_create` operation). The returned channel-ends IDs are stored with two respective node instances. The referred nodes are instantiated by the node-level `create` operation.

The actual channel-service components are descriptions of types of methods like `create` which implement this channel service. These descriptions have to be expressed in WSDL in order to be transmitted over the Internet via SOAP. The description should include definitions of the operations performed by the channel service, the required messages, the data types in use and the chosen communication protocols. The purpose of the WSDL is to describe these services in XML form over the Internet. Channel services represented by node brokers exchange WSDL files to restore connectivity and perform operations. SOAP comes in once a channel service is to be invoked. There would be no need for this intermediate conversion if the system should consider only interaction between (say) Java programs with RMI calls. The XML-based Web Services enable the definition of objects written in any language into language neutral types, and vice versa.

## 3. XML declarations for the *Reo* operations

Below is the declaration of the XML code for the description of the basic *Reo* operations as described in section 2. The purpose of encapsulating these primitives in XML code is to help in the implementation of the system through the web services. The coding begins with declaring the URLs for the xml namespaces that are going to be used:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
   name="Reo"
   targetNamespace="http://www.yourcompany.com/Reo.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.yourcompany.com/Reo.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsd1="http://www.yourcompany.com/Reo.xsd1">
```

The `<types>` element defines the data types that are used by the web service. For maximum platform neutrality, WSDL uses XML schema syntax to define data types. In this example the `simpleType` is used for the basic operations of *Reo*.

```
<types> <xsd:schema
   targetNamespace="http://www.yourcompany.com/Reo.xsd1"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:SOAPENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.yourcompany.com/Reo.wsdl"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://www.yourcompany.com/Reo.xsd1">
  xsd:complexType name="ArrayOfString">
     <xsd:complexContent>
     <xsd:restriction base="SOAP-ENC:Array">
     xsd:sequence>
  <xsd:element maxOccurs="unbounded" minOccurs="0"
name="item" type="xsd:string"/>
  </xsd:sequence>
  xsd:attribute ref="SOAP-ENC:arrayType"
wsdl:arrayType="xsd:string[]"/>
  </xsd:restriction> </xsd:complexContent>
  </xsd:complexType>
  xsd:simpleType name="OperationType">
     <xsd:restriction base="xsd:ENTITIES">
     <xsd:enumeration value="create"/>
     <xsd:enumeration value="connect"/>
     <xsd:enumeration value="read"/>
     <xsd:enumeration value="write"/>
     </xsd:restriction> </xsd:simpleType>
```

```
    </xsd:schema>
</types>
```

The **<message>** element defines the data elements of an operation. Each message can consist of one or more parts. The parts can be compared to the parameters of a function call in a traditional programming language. They normally have to do with requests and responses.

```
<message name="ReadResponse">
   <part name="dataMessage" type="xsd:string"/>
</message>
<message name="ConnectRequest">
   <part name="endPoint" type="xsd:string"/>
   <part name="endPoint" type="xsd:string"/>
</message>
<message name="ReadRequest">
   <part name="endPoint" type="xsd:ID"/>
</message>
<message name="CreateReaquest">
   <part name="channelType" type="xsd:string"/>
</message>
<message name="ConnectResponse"> </message>
<message name="WriteResponse"> </message>
<message name="CreateResponse">
   <part name="endPoint1" type="xsd:string"/>
   <part name="endPoint2" type="xsd:string"/>
</message>
<message name="WriteRequest">
   <part name="endWrite" type="xsd:ID"/>
   <part name="dataMessage" type="xsd:string"/>
</message>
```

The **<portType>** element is the most important WSDL element. It defines a web service, the operations that can be performed, and the messages that are involved (declared above). The <portType> element can be compared to a function library (or a module or class) in a traditional programming language.

```
<portType name="ReoPortType">
   <operation name="Create">
   <input message="tns:CreateReaquest"/>
   <output message="tns:CreateResponse"/>
</operation>
<operation name="Connect">
   <input message="tns:ConnectRequest"/>
   <output message="tns:ConnectResponse"/>
</operation>
<operation name="Read">
   <input message="tns:ReadRequest"/>
   <output message="tns:ReadResponse"/>
</operation>
<operation name="Write">
   <input message="tns:WriteRequest"/>
   <output message="tns:WriteResponse"/>
</operation>
</portType>
```

The **<binding>** element defines the message format and protocol details for each port and its operations.

```
<binding name="ReoBinding" type="tns:ReoPortType">
<soap:binding style="rpc"
traport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="Create">
<soap:operation
soapAction="capeconnect:Reo:ReoPortType#Create"/>
<input> <soap:body use="literal"/> </input>
<output> <soap:body use="literal"/> </output>
</operation>
<operation name="Connect"> <soap:operation
soapAction="capeconnect:Reo:ReoPortType#Connect"/>
<input> <soap:body use="literal"/> </input>
<output> <soap:body use="literal"/> </output>
</operation>
<operation name="Read">
<soap:operation
soapAction="capeconnect:Reo:ReoPortType#Read"/>
<input> <soap:body use="literal"/> </input>
<output> <soap:body use="literal"/> </output>
</operation>
<operation name="Write">
<soap:operation
soapAction="capeconnect:Reo:ReoPortType#Write"/>
<input> <soap:body use="literal"/> </input>
<output> <soap:body use="literal"/> </output>
</operation>
</binding>
```

Finally, the **<service>** element is a collection of related ports. The name attribute provides a unique name among all services defined within the enclosing WSDL document.

```
<service name="Reo">
<port binding="tns:ReoBinding" name="ReoPort">
<soap:address location="http://localhost:8000/ccx/Reo"/>
</port> </service> </definitions>
```

## 4. The Usage of the *Reo* Operations

The above construction of XML function structures can be used in the definition of new service composition specification tools. The characteristics of these tools are:

- The use of the *Reo* system by all the participating members (installed on every machine locally).
- The sharing of information regarding the use of the *Reo* primitives. The exchange of messages is done at the web service level and concerns the channel end structures and references, because the ends of a channel must internally know each other to keep the identity of the channel and control communication.
- If the type of a channel is asynchronous it must also have a reference to the buffer that implements every channel.
- An interface reference from a component to a channel end restricts the actions of the component to only the predefined operations on the channel.

- The predefined operations are `create`, `connect`, `disconnect`, `forget`, `read`, `write`, `move`, `join`, `split`. These can be classified into the following categories:
  1. The `create` operation creates a new channel with a specified channel type.
  2. The `connect`, `disconnect` operations connect and disconnect respectively a specified node to the calling component instance.
  3. The `forget` operation changes the specified channel end id and the `move` one changes the channel end to the new node.
  4. The `read`, `write` and `take` operations perform a read, a write, or a destructive read from/to a specified variable to/from the connected specified node.
  5. The `join` operation is a node merging operation producing a new node from two other specified nodes. The `split` operation creates a new node and splits the attached channel ends between the new and the old one according to a specified list.

The type of the chosen channels plays an important role in the outcome of the mode of execution of the node operations. The synch channels offer synchronous unbuffered transmission and the FIFO channels offer asynchronous unbounded buffered transmission. Buffers can generally be used as sequencers and Synch/SynchDrain channels as flow regulators. The mapping of the messages is achieved by the references to the buffers and/or the channel ends.

## 5. A Case Study Involving Web Services

Using the definitions of the previous chapter the web resources can be encapsulated in distributed objects, and the web can be transformed from a collection of clients and servers (serving web pages) into an object space of distributed objects ([7]). The requests between such objects should be carried out via a web browser, as shown in Fig. 2.

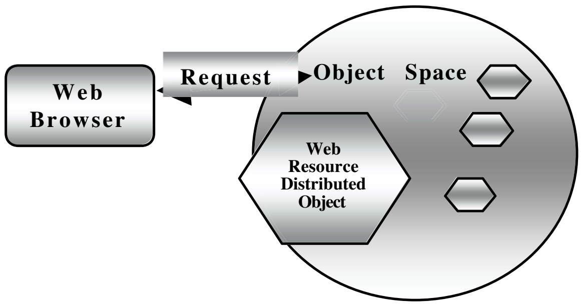

**Fig. 2**: Distributed objects in an object space

The web browser communicates with another through a local representative that each Distributed Shared Objects (DSO) has as shown in Fig. 3. The details of this architecture will be explained later on.

### 5.1 Using Channels in a Web Component Case Study

Channels offer the necessary subtle connectivity required in the implementation of such diversity and complexity as the Distributed Object Space over the Internet. Moreover the operations on *Reo* channels can be used as a language for the

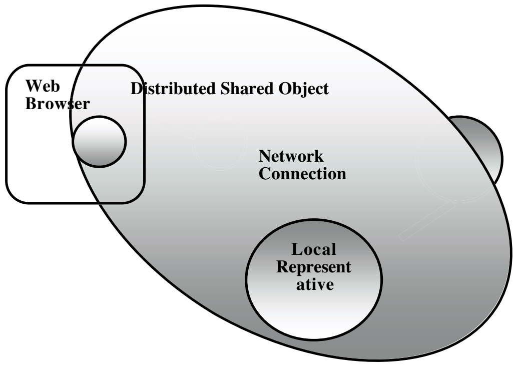

**Fig. 3**: A distributed shared object

coordination of concurrent services or as a connector constructing language for the binding of the component spaces (connectors) in a component based system.

The use of WSDL representation of component classes and its corresponding XML expression should be regarded as the means for the construction of a message that will incite a channel operation at a remote (hosting) side. In this case it could relate to a class or method associated with the use of *Reo* channels that exist on this remote site. Each information source should have a channel where requests can be issued. Clients to a channel end have to be aware of their cev ID (reference). The XML message contains the code for some *Reo* operations with the corresponding cev references. These messages cause one or more remote executions in the sites hosting the *Reo* services. In the present web component case study we deal with the formation of DSOs which reside in different web sites.

Here the web resources can be encapsulated in distributed objects and the web is transformed from a collection of clients and servers (serving web pages) into an object space of distributed objects ([7]). In our implementation this is achieved via the requests between such objects that are carried out via a web browser.

The web browser communicates with another through a Local Representative that each DSO has as shown in Fig. 4. The details of this architecture will be explained below.

### 5.2 The Binding of a Candidate Object

To communicate with a DSO, clients must bind to the object. This causes a new *Local Representative* (LR) of the DSO to be created at the client's address space, effectively connecting that address space to the rest of the DSO. To achieve this, the binding process has two main phases:

- Find where a host side of the DSO is, and
- Initialise a Local Representative.

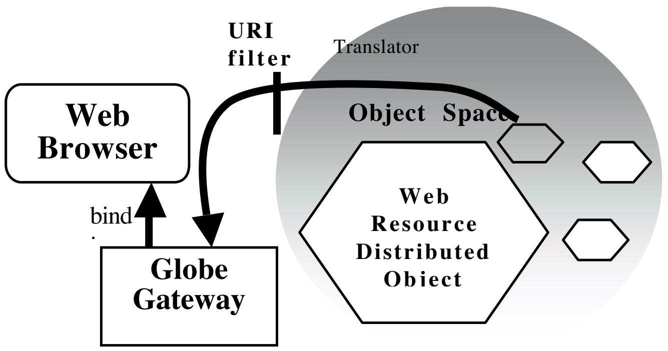

**Fig. 4**: The route for binding

The shared objects of this system are all hosted under a common class (Globe) name. A proxy called *the translator* accepts requests from the *DSO browser* that the client uses. A filter can sort out Globe related URLs. Such names are forwarded to a special Globe gateway, which performs the binding to the object and the callings of the appropriate methods. The aim of every client request and binding is to obtain its own LR in its own address space, effectively connecting that address space to the rest of the DSO

The name given by the client is passed to the name service (NS). At this point there is an interprocess communication between the translator site and the Globe Gateway. For the mobile channel system to be operable we consider that a channel has already been created on every site with a Local Representative. The binding process then, proceeds along the following steps:

1. The first stage of the clients binding process begins by sending the name ID of a DSO to the name service which maps names to location transparent object handles (OH).
2. Then the name service returns an object handle.
3. The object handle is passed to the location service.
4. The location service retrieves a contact address.
5. A contact address represents a contact point of the DSO. Contact addresses identify a LR that should be loaded into the client's address space. The contact address contains an implementation handler. This is sent to the implementation repository, which in turn returns a class archive.
6. The class archive is in turn used by the class loader for extracting the implementation code in order to create the actual LR, so that the client's address is connected to the rest of the DSO.

The overall view of information exchange is shown in fig. 5.

## 6. The Advantages of the DSO Communication Through *Reo* Channels

The communication between the DSO address space and a potential member is done with the use of channels, one from each object of a particular address space. Every member

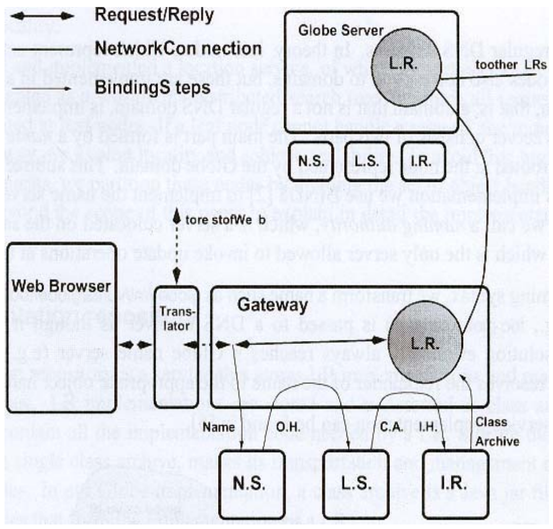

**Fig. 5**: The overall view of the binding process

address possesses the ends of two channels One of them is ready to receive applications from potential new members. The XML-`connect` predicate is used by every applicant to supply the cev ID to a DSO site. This connects the specified channel end cev to the component instance that contains the active entity which performs this operation. The XML message sends the cev ID to the site that 'owns' this channel. The execution of `connect` at a site results in the 'attachment' of the calling procedure to the called channel end. The *Reo* commands are executed locally, i.e. all channels are created at member sites or candidate member sites. For example to have the *binding* operation performed with the use of channels, both the *web browser* and the *translator* have to own at least one channel each, one for requesting and one for replying respectively. The sites are remote and the *web browser* component sends its cev ID to the *translator* component along with a `connect` XML message. The channel-end level primitives are indicated by the underscore.

```
ChannelEnd(timelimit,translator_loc)
```

The translator performs a `connect` operation with the received cev ID and connects to the browser's channel end.

```
_connect(timelimit,cev_wb) //primitive channel operation
```

From then on the client's browser will receive information from its created channel. Likewise, the browser can perform the operation connect with the cev ID of the translator and so get a request through the client's channel, e.g. the translator performs:

```
ChannelEnd(timelimit,webBrowser_loc)
```

And the browser responds with:

```
_connect(timelimit,cev_trans)
```

The translator belongs to the object space where the channels are already created and connected. So the following read and write operations have to be performed. The gateway has a channel for receiving (read) information for binding while the components name service, location service and implementation repository have a special channel that reads requests from the gateway. The channel of the gateway moves from the name service to the location service and from there to the implementation repository in order to perform the binding.

```
_connect(timelimit,cev_nameService)// to NS site
UseChannel_write(timelimit,var)
_write(timelimit, cev_nameService)
_connect(timelimit,cev_locService)// to LS site
UseChannel_write(timelimit,var)
_write(timelimit,cev_locService)
_connect(timelimit,cev_implemService)// to IR site
UseChannel_write(timelimit,var)
_write(timelimit, cev_implemService)
```

At every binding stage, the NS, LS, and IR components respond with performing a write operation on the channel that the gateway owns.

```
_connect(timelimit,cev_gateway)// to Gateway
UseChannel_write(timelimit,var)
_write(timelimit, cev_gateway)
```

Once the binding is completed the `join` operation is executed by the web browser site's local representative and the local representative of the gateway in order to `Join` the node that contains the rest of the object space channels.

```
ChannelEnd(timelimit,translator_loc)// to translator site
_connect(timelimit,cev_gateIR)
join(node_webBrowser, node_translator)
```

## 7. Conclusions and Related Work

In this paper we investigate the use of a channel operation system in coordinating web services. The new-formed system is process oriented in the sense that the processes participating in the construction of a distributed object perform the channel operations.

The flow of data does not decide the execution of a read or a write. On the other hand the relocation of a channel does not influence the reliability of the data carried. The outcome from this is a configurable net of components that connect/disconnect to and from nodes at run time.

At the node level nodes with their attached channel ends can join and split between them. This allows the dynamic reconfiguration of connections at real time and the redirection of flow of data between the member objects.

The set of operations on channels introduced in this work essentially provides a solution for the composition of a communication pattern among components through the XML encapsulation of the Reo primitives. The major advantage is that a web server is enough for the passage of the execution messages. The construction and management of the communication concerns of the created channels can take advantage of the fast evolving web communication protocols.

The use of channel composition in the coordination and synthesis of component-based systems (web-based or otherwise) is a promising approach. This paper extends the work presented in [3] where channel based coordination is applied to commercial software development environments and [6] where the case of real-time software composition (and coordination) is being examined. Finally, the survey on coordination models and languages ([5]) presents a *tour de force* of this field.

## References

[1] F. Arbab, F. S. de Boer, J. G. Scholten and M. M. Bonsangue, "MoCha: A Middleware Based on Mobile Channels", *26th International Computer Software and Applications Conference (COMPSAC 2002)*, Oxford, England, 26-29 August, 2002, IEEE Computer Society Press, pp. 667-673.

[2] F. Arbab and F. Mavaddat, "Coordination through Channel Composition", *5th International Conference on Coordination Models, Languages and Applications, (Coordination 2002)*, York, UK, 8-11 April, 2002, LNCS 2315, Springer Verlag, pp. 22-39.

[3] A. Chimaris and G. A. Papadopoulos, 'Control-Driven Coordination Based Assembling of Components', *Twenty sixth Annual International Computer Software and Applications Conference (COMPSAC 2002)*, Oxford, England, 26-29 August, 2002, IEEE Press, pp. 572-577.

[4] G. Gardarin, F. Sha and T. D. Ngoc. "XML-based components for Federating Multiple Heterogeneous Data Sources", *18th International Conference on Conceptual Modeling, (ER '99)*, Paris, France, 15-18 November, 1999, pp. 506-519.

[5] G. A. Papadopoulos and F. Arbab, 'Coordination Models and Languages', *Advances in Computers*, Marvin V. Zelkowitz (ed), Academic Press, Vol. 46, August, 1998, pp. 329-400.

[6] G. A. Papadopoulos and F. Arbab, 'Coordination of Systems With Real-Time Properties in Manifold', *Twentieth Annual International Computer Software and Applications Conference (COMPSAC'96)*, Seoul, Korea, 19-23 August, 1996, IEEE Press, pp. 50-55.

[7] M. van Steen, F. J. Hauck and A. S. Tanenbaum, "A Scalable Location Service for Distributed Objects", *Second Annual Conference of the Advanced School for Computing and Imaging (ASCI'96)*, Lommel, Belgium, 5-7 June, 1996, pp. 180-185.

[8] S. Szykman, J. Senfaut and R. Sriram. "The Use of XML for Describing Functions and Taxonomies in Computer-based Design", *1999 ASME Design Engineering Technical Conferences (19th Computers and Information in Engineering Conference)*, Las Vegas, NV, 12-15 September, 1999, Paper No. DETC99/CIE-9025.