

VERIFYING CORRECTNESS OF WEB SERVICE COMPOSITION

Nikola Milanovic and Miroslaw Malek

Humboldt University, Berlin

{milanovi,malek}@informatik.hu-berlin.de

Abstract

We evaluate current efforts for Web service orchestration and composition. Most approaches reduce composition to programming the flow of partner services, according to the specified goal. We show an alternative approach to Web service composition, where partners are modeled as state machines and composed into a single resulting state machine, which guarantees functional and non-functional properties. The proposed model ensures predictability and verifies correctness of composition.

Keywords: services, contracts, orchestration, composition, correctness, predictability

1 Introduction

Service oriented computing (SOC) is the computing paradigm that uses services as fundamental elements of application development process. Services offer operations that clients (or other services) invoke. They can be rapidly deployed, easily reused, and are platform- and network-independent. To be able to operate in a SOC environment, services must declaratively define their properties in a standard and machine readable format. Based on description, SOC offers native capabilities: discovery, selection and binding. Service composition is introduced on top of native capabilities. It governs the way applications are developed from basic services.

However, most proposals that address service composition tend to cover only the issue of connectivity between partner services. They offer 'composition languages' that enable orchestration of services in different flows of execution. The problem is that these approaches do not address correctness of the flow. There is no formal way to verify properties of achieved composition (such as security or dependability). We will try to introduce a contractual composition model that ensures predictability and correctness, by treating partner services as state machines, and achieves composition by merging them in a new state machine.

2 Related Work

There are several approaches dealing with the problem of service composition. The most prominent among them is BPEL4WS (Business Process Execution Language for Web Services) [3]. BPEL is an XML language that supports process-oriented service composition. BPEL composition

is a process or a flow that interacts with a certain set of Web services in order to achieve given task or goal. The result of BPEL composition is called a process, and participating services are partners. BPEL can be used alone, or with two additional specifications, Web Services Coordination and Web Services Transaction.

The concept of a Web Component [11], treats services as components. Web Component wraps basic or complex services and exposes their functionality in form of a class definition. It can be further reused, extended or specialized.

In the area of semantic web, a lot of effort is focused on the issue of service composition. The most complete framework is DAML-S (Darpa Agent Markup Language Services) [4], which aims to provide automatic Web service discovery, invocation, composition, interoperation and execution monitoring.

However, all approaches that we mentioned do not address the issue of correctness. Recently, other models have appeared trying to introduce formal verification of service composition, such as one proposed by Meredith [8] where services are modeled as mobile processes and their composition is verified using π -calculus. Hamadi and Benatalah propose using Petri Nets for modeling composition [6], while Fu, Bultan and Su use model checking and finite state machines for formal verification of workflows [5]. The problem of π -calculus approach is that it offers typing for order and format of messages. Other properties, like quality of service or timeliness, are not addressed. The Petri Net model does not cover nonfunctional properties, such as transactions, exception handling, etc, while the problem of model checking approach is complexity: one can run out of time and space and still have no idea whether a given service (or composition) is behaving according to the model.

3 Composition vs. Orchestration

Most proposals that address service composition treat issue of connectivity between services, but say very little about correct functioning of services obtained by composition. Granted, BPEL has the elaborate exception handling mechanism. Still, a good error handling does not prevent users from making mistakes and composing incorrect services. Purely connectivity approach, where composition is modeled with message passing is not enough. We must be able to determine properties of a composed service (predictability), and to guarantee that it will behave as intended (correctness).

Web services are based on a notion of message passing and ports. Since all composition models exploit this basic functional aspect of Web services architecture, non-functional properties are not taken into account when building composition models. Yet, we argue that specification of non-functional properties (e.g., quality of service, time, security, dependability, etc.) is essential for correct service composition.

The difference between orchestration and composition is that orchestration includes only functional properties, while composition must account for both functional and nonfunctional properties. Orchestration can only preserve properties, and composition must be able to delete, transfer, create and bind properties. Orchestration is unable to guarantee properties of orchestrated flow. Composition on the other hand must guarantee properties and verify correctness of the composed service. Having that in mind, we now define orchestration and composition:

Def. 1 *Service orchestration is a process of coordinating execution of two or more services, using flow control commands (sequence, parallel, switch) that determine topology of message exchange.*

Def. 2 *Service composition is a process of binding execution of two or more services, such that functional and non-functional properties can be determined and guaranteed, ensuring predictability and correctness.*

Orchestration is a way to 'program' services into different paths of execution using flow control commands (sequence, parallel, condition, loop). Service composition, on the other hand, is a process of building a new service for which we want to guarantee how it will behave with respect to functional properties (what it will deliver) and non-functional properties (how and under which conditions will it deliver). We will present two examples illustrating the need for predictability and correctness.

Suppose we have a trusted and untrusted service, where trust is defined by the service architecture (security, dependability, etc.). What happens when we orchestrate these services in sequence? Is this orchestration trusted, untrusted, or something in between? There is no way to answer this question using orchestrating languages like BPEL, yet it is crucial that we know whether our application is still secure or dependable. And what happens when we orchestrate two trusted services? Do we assume that the orchestration of trusted services will also be trusted? Or that orchestration of untrusted services will always be untrusted? What if we execute them in parallel instead of in sequence? Questions of how the orchestrated service will behave remain unanswered. This example establishes a clear need for predictability of service properties.

Another example is temporal extension of composition. Let us observe a simple handshaking example with two partner services, one wanting to invoke a method of another. However, the client service is expecting to be notified when it can apply (invoke a method), while the

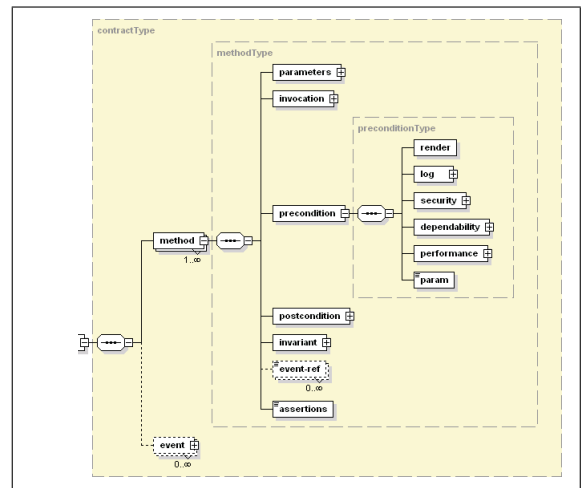


Figure 1. Logical contract structure

provider service is expecting to be notified that the client wishes a processing. In this way, the orchestration will not produce useful or expected results, unless we know in advance about handshaking requirements. If we do not obey expectations of the provider service (preconditions) we will likely end up with invalid orchestration. This demonstrates a need for verifying correctness of composition.

4 Contractual Composition Model

In order to achieve predictability and correctness of service composition, we model partner services as abstract machines. Service composition is performed by merging two abstract machines using composition operators. We introduce composable service architecture and service contracts that make this possible. Service contracts are extended service descriptions (augmenting Web Service Description Language - WSDL) while architecture defines allowed composition operators (ways to compose services).

4.1 Contract Definition Language

The idea to introduce contracts to the area of service oriented computing was partially motivated by Design by Contract paradigm [9]. The contract specifies what a component expects from its clients, and what clients can expect from a component.

A service is perceived by the clients or other services through contract (Figure 1). It contains information about functional and non-functional service properties. Another motivation for introducing contracts is the fact that they are inherent to the way we think and develop software. It has been shown in [2] that contracts describing functional behavior can be extracted from .NET libraries. We are making a similar effort to prove that contracts are implicitly contained inside Enterprise Java Beans [10].

A contract is comprised of the base, method and event part. *Base contract* specifies basic information about the service: name, URI, service description, price, and available methods. *Method contract* describes one service method. It contains information about parameters, invocation, preconditions, postconditions, invariants, declared events and assertions. Parameters can be in, out, or in/out. Invocation determines whether a service is synchronous or asynchronous and defines asynchronous messages and/or callback functions. Precondition declares client obligations. Service is guaranteed to work correctly if and only if client satisfies precondition. Postcondition declares what a service will guarantee, if precondition holds. Invariants are static service properties that must hold before and after every method invocation. *Event contract* is used for exposing events to which other services can subscribe and depend on. They will receive notification when an event occurs. Inside a method contract, there is a list of references to some (or all) of these events.

We developed an XML schema called Contract Definition Language (CDL) that enables description of contract attributes depicted on Figure 1. Since schema is too large to be presented here, we include a brief description (entire schema is available at <http://informatik.hu-berlin.de/~milanovi/cdl.xsd>). The root element is `<contract>`. Nested inside are `<method>` and `<event>` elements. They are complex elements, and only the `method` element will be described further in this paper, since they are very similar. The attributes of the `contract` element are the name of a service (`serviceName`), uniform resource identifier of a service (`serviceURI`), service description (`serviceDescription`), price that client must pay for using a service and `state` attribute that describes how service handles state between client calls.

The `method` element describes one method of the service provider. Nested inside are complex elements specifying parameters, invocation type, preconditions, postconditions, invariants, events and assertions. Elements `precondition`, `postcondition` and `invariant` share the same structure. They are made of conjoined logical clauses that describe conditions for parameters and return values, security, dependability, performance, and logging. As the requirements grow, new contract elements, such as *real-time*, can be added.

What is the difference between CDL and WSDL? WSDL addresses connectivity issue. We assume that WSDL description is already in place, and focus on definition of semantic and non-functional properties. CDL can be seen as a language that augments WSDL by providing options for richer description of non-functional properties. Following is an example of a printing service described using CDL. Note that WSDL part, dealing with connectivity and binding, is left out.

```
<?xml version="1.0" encoding="utf-8"?>
<contract service name="printService"
serviceURI="localhost/services/print"
serviceDescription="Basic printing service"
price="0" state="stateless">
<method name="Print" methodDescription="Prints a document">
<parameters>
```

```
<parameter direction="in">
<name>doc</name>
<parameterType>Document</parameterType>
</parameter>
<parameter direction="in">
<name>resolution</name>
<parameterType>int</parameterType>
</parameter>
<parameter direction="out">
<name>status</name>
<parameterType>int</parameterType>
</parameter>
</parameters>
<precondition>
<param>Document.type=ps</param>
<param>resolution=Printer.res</param>
</precondition>
<postcondition>
<param>Doc.res=resolution</param>
</postcondition>
<performance>
<type>number-of-concurrent-clients</type>
<unit>int</type>
<value>20</value>
</performance>
<dependability>
<transactions model="split">
<transaction-manager>jrun</transaction-manager>
<resource-manager>/print/drv/file.drv</resource-manager>
<compensate-method>printCompensate</compensate-method>
<timeout unit="ms">1000</timeout>
<enlist>required</enlist>
</transactions>
</dependability>
</postcondition>
<invariant>
<param>Documen.pages<=Printer.paper</param>
</invariant>
</method>
<event name="outOfPaper">
<!-- event definition, similar to method -->
</event>
</contract>
```

4.2 Formal Contract Representation

Contracts are persisted in XML format, but the composition requires more formal expression. We cannot just merge two XML text files and produce a composed contract. It must be computed, and its properties verified. We transfer contracts into formal mathematical notation for the purpose of composition. When a composed contract needs to be computed, XML representation is transferred into B-notation [1]. That can be done once, and then formal notation can be stored in a directory, until the contract changes. When formally specified, contract elements (properties) can receive mathematical treatment necessary for calculation of composed contracts. This means that we have dual contract representation: XML and abstract machine (B notation). XML is necessary for network transport and interoperability, while abstract machine is used for calculations of composed contracts.

The B-notation represents an element (object, class, service, component) as an *abstract machine*. It is characterized by *statics* and *dynamics*. Statics corresponds to the definition of the state, while dynamics corresponds to operations. The basic abstract machine is denoted in the following way:

```
MACHINE M(X, x)
CONSTRAINTS C
CONSTANTS ct
SETS S; T={a,b}
PROPERTIES P
VARIABLES v
INVARIANT I
ASSERTIONS J
INITIALIZATION U
OPERATIONS
ul <- O1(w1) = PRE Q1 THEN V1 END
...
un <- On(w1) = PRE Qn THEN Vn END
END
```

This is a parameterized abstract machine having free dimensions X and x . CONSTRAINTS describes conditions on

machine parameters. `SETS` contains finite or named sets that the machine can use (we call them domains, and use them for verification), while `CONSTANTS` describes constants that the machine understands. `PROPERTIES` takes form of conjoined predicates specifying invariants involving constants and sets. `VARIABLES` lists state variables, and `INVARIANT` describes static properties of the machine, that must be preserved before and after each operation. `ASSERTIONS` is deducible from `PROPERTIES` and `INVARIANT`, and exists purely to ease the proving of machine correctness. `INITIALIZATION` initializes state variables. `OPERATIONS` lists operations of an abstract machine, with preconditions (`PRE`) and postconditions (`THEN`).

The algorithm for mapping CDL into abstract machine works on the principle similar to that of JAXB (Java API for XML Binding), where XSD types are converted into mathematical types and vice versa. All CDL tags describing properties are transferred into state variables with respective domains, while methods are transferred into state functions. The result of this transformation for the printing service would be:

```
MACHINE printer
SETS Document,Resolution,Status,Transaction
VARIABLES doc, resolution, status, printer, ncc, trans_model,
trans_manager, compensate, timeout, enlist
INVARIANT doc.pages <= printer.pages
OPERATIONS status <- print(doc)
PRE doc.type = Document.ps AND resolution IN printer.res
THEN doc.res=resolution AND ncc<120 AND trans_model=Transaction.
SPLIT AND trans_manager=Transaction.RUN AND compensate=
printCompensate AND timeout=Transaction.1000 AND enlist=
Trans.REQUIRED END
status <- compensatePrint() END
```

4.3 Composable Service Architecture

It is very difficult to solve the problem of service composability without defining an environment in which the composition takes place. That is the reason why we introduce an abstract composable service architecture. We are also aware that it is impossible to provide a meaningful interface specification of an open component, without considering the context of use of the component in a particular environment [7]. Therefore, we attribute composability to an architecture. The architecture is a set of initial (or atomic) elements and rules how to create new elements from them. More formally, an architecture is a tuple (E, O) , where E is set of initial elements, and O is a set of composition operators. We treat elements as state machines, and call state variables *properties*. We are interested in how they behave during composition.

If a property is to remain unchanged after the composition, it is *invariant*. If a property still exists after the composition, but its value is changed, it is *bounded*. If a property does no longer exist after the composition, it is *vanishing*. If a property can generate new property, it is *emerging*. If a property vanishes, but together with another property can generate emergent property, it is *transferred*.

A service architecture consists of rules how to create new elements. These rules include common types, allowed operations, restrictions, operators, flow constructs and message dependency. Common types define types that we can

use in contracts and be sure that other services will understand them. Allowed operations are mathematical operations that we can use in a given architecture to derive elements of new (composed) contracts. Restrictions reduce possible number of valid compositions by specifying minimum requirements of services participating in composition. In this way we can define properties that are guaranteed by the architecture. Composition operators are used for performing composition. Flow constructs control flow of execution by specifying how to perform functional composition of service methods. They can express conditions (`if`), branches (`switch`), looping (`for`), and sequential and parallel flow. Message dependency allows for defining how input and output messages are created: synthesis (combine output messages into input), decomposition (decompose input messages of composed services to constituents) and arbitrary mapping between input and output messages. Using flow constructs and message dependency we create required invocation topology. Message dependency extends to all properties, not only to parameters. In that way we define provisional mapping between state variables.

Service composition is achieved through contract composition using composition operators. Composition operators express flow constructs and state variable mapping. For each composition operator, a behavior table is defined. The columns of the behavior table are contract elements, and rows are behavior classes. For each entry in the table, one operation is defined determining action to be taken for computing composed contract element. Before composition can take place, each contract element must have a behavior class assigned (invariant, bounded, vanishing, emerging or transferred). It can be done by static or dynamic assignment. With static assignment behavior classes are assigned at development time, and they are part of the service contract. Every XML element in contract gets additional `behavior` attribute, that defines a behavior class for that element. This attribute can be inherited from parent to child nodes. With dynamic assignment behavior classes are assigned at composition time, by the composition operator. Service contract does not define behavior of contract elements. Instead, composition operator assigns behavior classes to them.

Formally, let us define a set of contract elements $E = \{parameters, invocation, semantic, \dots\}$, a set of behavior classes $C = \{invariant, bounded, vanishing, emerging, transferred\}$, and a set of mathematical and logical operations O that service architecture supports. Then, a table entry is defined as $B[i, j] = operation(E_i, C_j)$, where relation *operation* maps pairs of contract elements and assigned behavior classes into mathematical and logical operations: $operation : E \times C \rightarrow O$. In case of static assignment, there are no restrictions on relation *operation*. However, with dynamic assignment the following must hold: $\forall e \in E, \forall c \in C (\exists ! o \in O | behavior(e, c) = o)$. It is not possible to assign two behavior classes to one contract element, since it would lead to ambiguous composition. With static assignment behavior classes are predefined.

4.4 Proof Obligation

When composing a service, new preconditions, postconditions and invariants must hold. Two services can be functionally compatible, but their non-functional contracts can be conflicting. The composition engine must prove that composed contract is correct.

The proof obligation states: assertion must be deducible from properties and invariants, while initialization, and operation body must establish the invariant. Formally we can denote this as (referring to the abstract machine model and clauses introduced in 4.2):

$$\begin{aligned} C \wedge P \wedge I &\Rightarrow J \\ C \wedge P \wedge J &\Rightarrow I \\ C \wedge P \wedge I \wedge J \wedge Q &\Rightarrow [V]I \end{aligned}$$

With $[V]I$ we denote substitution that preserves the invariant. This provides us with a formal model for performing contract composition and verification. Our intention is to facilitate service composition by verifying properties of a service obtained by composition. This enables us to guarantee properties (predictability) and to prohibit composition of invalid services (correctness).

4.5 Composition Example

We will build a composed service that takes loan application from a client, and returns the best loan offer. We have available one credit rating service that takes loan application and returns a credit rating for applicant. Then we have two loan companies that offer loans based on supplied credit rating. Credit rating service is synchronous, while two loan offer services are asynchronous. We want to construct a service flow that composes these services, so that based on a credit rating for a given applicant we can choose the best (lowest interest rate) loan offer. The desired flow is shown on the Figure 2 (BPEL graph). Contracts of credit card rating and loan offer services are shown on Figures 3 and 4, respectively. We use static behavior assignment.

Let us define a behavior table for one composition operator (\times). The table has one additional row `NO_MATCH`, that specifies actions to be taken when there is no appropriate matching contract element in the partner service, e.g., when only one service defines number of concurrent clients.

	exception	operation	ncc
transfer	U	SEQUENCE	+
bound	NOP	NOP	min
NO_MATCH	NOP	NOP	NOP

Suppose that we have deployed the service `creditRating` corresponding to the contract shown on Figure 3, and two services `loanCompanyA` and `loanCompanyB`, corresponding to contract shown on Figure 4. We can write a following construct:

```
offer = creditRating × switch
    (loanCompanyA, loanCompanyB)min(interestRate)
```

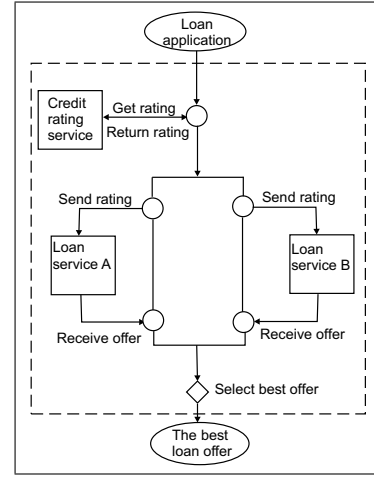


Figure 2. Service flow for loan example

```
MACHINE creditRating
VARIABLES loanApplication(IN):invariant; creditRating(OUT):vanish
INVARIANT loanApplication ∈ Application:invariant
OPERATIONS rating(SYNC) ≜ PRE THEN creditRating = rating (loanApplication):
transfer; wcet=300ms:vanish; ncc=10:bounded
EXCEPTIONS invalidApplication:transfer
```

Figure 3. Contract of credit card rating service

```
MACHINE loanCompany
VARIABLES creditRating(IN):vanish loanOffer(OUT):invariant
interestRate(OUT):invariant
INVARIANT loanOffer ≥ 0
OPERATIONS offer(ASYN) ≜ PRE THEN loanOffer = offer (creditRating) :
transfer; wcet=24h:vanish; ncc=20:bounded
EXCEPTIONS invalidRating:transfer noLoan:invariant
```

Figure 4. Contract of loan company service

We used abbreviated expression `switchp` that takes any number of asynchronous services, executes them in parallel, evaluates for predicate p upon completion and returns one that matches the predicate. In reality, this would be another composition operator. Using composition operator \times we compose this construct with the credit rating service, resulting in the machine shown on Figure 2.

Now we will demonstrate concept of correctness. Since we did not declare any initialization or assertion, we only have to prove that operation body preserves the invariant. Invariant consists of two parts: $\text{loanApplication} \in \text{Application}$ and $\text{loanOffer} \geq 0$. Since operation `bestLoan` does not modify state variable `loanApplication`, the first part of invariant holds. We could also look into machine `creditRating` and find that $\text{loanApplication} \in \text{Application}$ is the invariant. For the second part, we

have to take a look into definition of `offer` operation provided by machine `loanCompany`. Its invariant states that $\text{loanOffer} \geq 0$, therefore, the second part of our invariant is proved. That way we systematically prove that our resulting machine is correct. A contradiction would mean that composition is not valid.

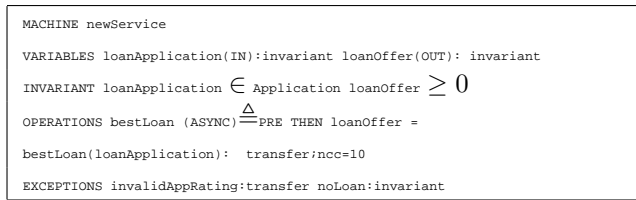


Figure 5. Contract of the composed service

To achieve equivalent composition using composition language such as BPEL, one first has to identify cooperating services and then to manually write flow, invocation and description code of a new service, without knowing anything about properties of a resulting service. We do the same in a single line, using two composition operators, while guaranteeing correctness.

4.6 Implementation

We use Sun's Java Web Services Developer Pack (JWS DP) for implementing the composition engine. It offers useful technologies like Java XML Binding (JAXB) and Java XML Remote Procedure Calls (JAX-RPC) that facilitate composition.

The implementation architecture is shown on Figure 6. Contract Definition Language schema is compiled with binding compiler to produce classes that will contain contracts. Contracts are stored in a directory, and loaded to composition engine when needed by JAXB unmarshaling process. Composition engine converts contracts to abstract machines, computes composed contracts and stores them back at the directory by marshaling their content as XML. Invocation of available services is done by JAX-RPC, according to computed contracts. We use this setup to perform experiments with automatic service composition, and are currently investigating complexity costs of introducing ternary composition operators.

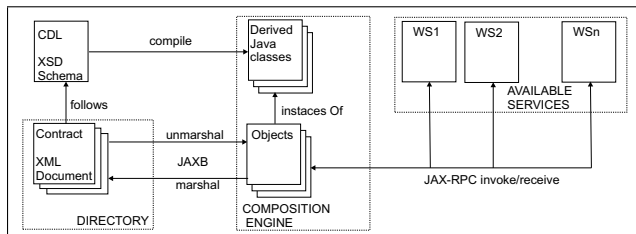


Figure 6. Implementation

5 Conclusion

Service composition is sometimes referred to as 'programming in big'. It is governed by different requirements than component based software development. The main difference is that application developer does not have access to service documentation, source nor binary code, but only to rudimentary functional description (WSDL). Services also execute across firewalls and other trust barriers. For that reason we introduce contracts, extending WSDL with richer set of descriptive options. Composition is then performed by transferring contract description into formal notation (abstract machine) and applying composition operators to compute the resulting machine (service). That way we ensure predictability and correctness of composition.

We are currently working on a decomposition model, where request for a complex service is decomposed and then matched with available services and composition operators, looking for composition that is 'closest' to required specification. We are investigating matching algorithms and metrics for determining distance of the obtained composition to the required service.

References

- [1] J.R. Abrial. *The B Book*. Cambridge University Press, 1996.
- [2] K. Arnout and B. Meyer. Uncovering Hidden Contracts: The .NET Example. *IEEE Computer*, 36, No. 11, pp 48-55, November 2003.
- [3] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana. The Next Step in Web Services. *Communications of the ACM*, October 2003.
- [4] A. Ankolekar et al. DAML-S: Web Service Description for the Semantic Web. In *Proceedings of the International Semantic Web Conference (ISWC)*, 2002.
- [5] X. Fu, T. Bultan, and J. Su. Formal Verification of E-Services and Workflows. In *Proceedings of Workshop on "Web Services, e-Business, and the Semantic Web (WES): Foundations, Models, Architecture, Engineering and Applications"*, 2002.
- [6] R. Hamadi and B. Benatallah. A Petri Net-based model for Web Service Composition. In *Proceedings of the Fourteenth Australasian database conference on Database technologies*, 2003.
- [7] H. Kopetz and N. Suri. On the Limits of the Precise Specification of Component Interfaces. In *Proceedings of the 9th IEEE International Workshop on Object-Oriented Real-time Dependable Systems (WORDS2003F)*, 2003.
- [8] L.G. Meredith and S. Bjorg. Contracts and Types. *Communications of the ACM*, 46, No. 10, pp 41-47, October 2003.
- [9] B. Meyer. Applying Design by Contract. *IEEE Computer vol. 25, no. 10, pp. 40-51*, October 1992.
- [10] N. Milanovic and M. Malek. Extracting Functional and Non-functional Contracts From Java Classes and Enterprise Java Beans. In *Proceedings of the Workshop on Architecting Dependable Systems (WADS 2004)*, Florence, Italy, 2004.
- [11] J. Yang and M. P. Papazoglou. Web Component: A Substrate for Web Service Reuse and Composition. In *Proceedings of 14th Conference on Advanced Information Systems Engineering (CAISE02)*, Toronto, 2002.