

Towards Adaptive and Composable Services

Nikola Milanovic, Vladimir Stantchev
Jan Richling, Miroslaw Malek

Humboldt University, Berlin

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

July 1, 2003

Abstract

In complex distributed environments, it is important to achieve high degree of machine-to-machine interaction, not only on a functional, but on a semantic level. Therefore, we propose the NOMADS (Networks of Mobile Adaptive Dependable Systems) Republic, a vision of the future community open to all interested hardware and software entities, and based on adaptive and, if necessary, composable services. One needs very little to become citizen of this Republic: one needs to know how to communicate and to discover, provide and/or use NOMADS services, in an infrastructure that guarantees three basic properties: mobility, adaptivity and dependability.

We concentrate on NOMADS services which could potentially provide an unprecedented level of highly adaptive and semantic collaboration. The idea is to hide the complexity of underlying infrastructure from the users, showing only the services they can invoke. This approach promises seamless interoperation. We provide a way to discover not only what services can collaborate with each other on the functional basis, but rather on the semantic basis, meaning which service compositions can produce useful results. This way we are able to offer new services not originally programmed.

1 Introduction

The two important breakthroughs in computer science and industry were object oriented paradigm and the Internet. When merged, they produced component-based software and Web services. The area of service models is where the research is heavily concentrated today. In a way, services are taking us back to the realm of client-server computing. However, old client-server model cannot respond to requirements of distributed computing we face today. The major requirements of (web) services are dependability, composability and reuse, which we will address in this paper. The remainder of the paper is structured as follows. In Section 2 we present state-of-the-art in service development as we

describe Web services and JINI architecture, their advantages and drawbacks. In Section 3 we introduce the concept of service contract, and in Sections 4, 5 and 6 we show how we can achieve composition, decomposition and adaptivity of services using contracts. In Section 7 we propose several ways to ensure dependability and fault-tolerance, Section 8 deals with service reuse, and Section 9 concludes with pointers for the future work.

2 Service Architectures

The service paradigm tries to hide the complexity of today's distributed hardware/software environment behind the notion of service providers and service requestors. Using service discovery and description methods, services try to achieve interoperability in a relatively transparent manner [3]. Different architectures approach the problem from different standpoints and with different goals.

Web services are pieces of code available for invoking over a network. What distinguishes them from traditional remote procedure calls is the support for interoperability, discovery and description. Web services are platform-independent, since all messages between service requester and service provider are transferred in XML format using SOAP protocol over (for example) HTTP. Web services can be discovered via UDDI (Universal Description, Discovery and Integration), which specifies a registry for dynamically locating and advertising Web services. Specification of service is done using WSDL (Web Services Description Language) which defines service's abstract interface describing messages that can be exchanged with a service, and binding information that contains details specific to the protocol used [13].

JINI is Java-based technology enabling hardware and software entities to participate in service environment [10, 11]. JINI provides means for service description and discovery via service directories (*lookup* services). The general idea is to federate computers and other computing devices into what user perceives as a single system. Communication between services is established using remote method invocation (RMI). This requires that all participants (service requestors, as well as service providers) have Java virtual machine capability. Service providers publish service descriptions and service objects (proxies) to lookup services. Lookup services send service objects to interested clients, which then invoke methods on them.

However, mentioned service architectures still lack fault-tolerance mechanism. Then, support for collaboration between different services is just developing [2]. Service reuse is poorly addressed. These are the key requirements that must be supported before either architecture can assume its position of major open distributed environment for the future.

3 Contracts for Services

Our service model consists of service providers and service requesters. Service providers offer methods that can be invoked by service requesters. Service requesters can be either arbitrary clients, or other service providers. Service provider can register its methods with service directory, or can provide services directly to interested parties, peer-to-peer. Service discovery and/or registration is done by publishing a *contract*.

We propose service contracts as a way to ensure dependability, composability and reuse. Each service method has a contract. Contracts should be thought of as extended WSDL service descriptions, that encompass not only extensive functional description, but also non-functional properties of a service. Composition and decomposition of services is done with respect to contracts.

We define several classes of contracts. **Type** contracts define arguments that service method requires, and values that it returns. **Semantic** contracts define preconditions, postconditions and invariants for each method invocation. Preconditions determine what a client must provide, while postconditions determine what a service will guarantee. Invariants are rules that must hold after each method invocation. Precondition and postcondition contracts can be nested within other contracts. **Performance** contracts define performance attributes of a given service provider. They depend on the platform service is executing on, and can vary with time. **Dependability** contracts encompass a large class of contracts that define dependability attributes of a service provider. They include properties such as availability, means of ensuring reliability and fault-tolerance (replication, checkpointing, transactions), security, etc. **Event** contracts are used for exposing events to which other services can subscribe and depend on. They will receive notification when an event occurs. **Rendering** contracts are needed for front-end services only, when communicating with the front-end user. Since human users will use a variety of devices to interact with services, this will enable front-end services to render the response appropriately.

Our intention is to introduce *trusted services*. A trusted service is a service that has a contract as specified above, and behaves according to the attributes stated in it. Every service does not need to declare all contract classes, since some classes are mandatory and some are optional. For example, service must declare a type contract because functional composition requires it, but does not have to declare rendering contract if it does not interact with end-users.

There are several possibilities for contract implementation. One is to implement the contract manually in a separate XML file, and to parse it when needed. That is in accordance with current trends in Web service implementation (WSDL). However, this approach suffers from one major problem: synchronization of contracts and service implementation. Whenever implementation changes, contract must be manually updated. Then, contract can get separated from service implementation, or get mixed with another contracts. That is the reason why we should consider another approach, which is to embed the contract into service implementation, like in Eiffel programming language [8]. This approach has two benefits: whenever service implementation changes, contract

is automatically updated; and we could try to develop a formal method for verification and proving correctness with respect to the contract [1]. This would solve many security and safety issues, because we would be able to not only to expect, but guarantee that service will behave according to the contract.

4 Service Composition

The goal of service composition is to determine which service providers can collaborate, and to derive new methods and related behavior out of existing ones. Service composition is done with respect to contracts, or parts of contracts.

In order to express this formally we use the concept of a system architecture presented in [12]. This concepts introduces an architecture A as a set of elements E and a set of composition operators O with $\circ \in O \wedge \circ : E \times E \rightarrow E$ and

$$A : E \times E \times O \rightarrow \{true, false\}$$

The function $A(d, e, \circ)$, $d, e \in E, \circ \in O$ is *true* if the architecture A allows the composition of the elements d, e using the composition operator \circ and *false* otherwise. Regarding to that definition, an architecture describes how and which elements can be composed to obtain new elements.

If we map this approach to services, we can define service architecture A_s . Every service provider is an element of that architecture. Let E_s be the set of all elements (service provides), and O_s the set of all composition operators describing how services can be composed in that architecture. Then, service architecture is:

$$A_s : E_s \times E_s \times O_s \rightarrow \{true, false\}$$

When computing function A_s , we evaluate or *negotiate* elements of contracts. Every contract element is called a *property*. We first perform functional evaluation, which is checking for functional compatibility between services, typically by evaluating type and semantic contracts. If that returns *true*, we derive new functional contracts.

After that we derive non-functional parts of contracts. To achieve all this we define several classes of properties similar to the distinction between different qualities in [12]. If a property is to remain unchanged after the composition, we call it *invariant* property and it belongs to *invariant class*. If a property still exists after the composition, but its value is changed, we call it *bounded* and it belongs to *bounded class*. If a property does no longer exist after the composition, we call it *vanishing* and it belongs to *vanishing class*. If a property can generate new property, we call it *emerging* and it belongs to *emerging class*. At the end, if a property vanishes, but together with another property can generate emergent property, we call it *transferred* and it belongs to *transferred class*.

Let P be the set of all properties for a given contract, C the set of all property classes, and O_s the set of all composition operators. We define a relation *propertyClass*:

$$propertyClass : P \times O_s \rightarrow C$$

that for each property defines a class to which it belongs under each composition operator. Using this relation we evaluate elements of contracts when composing.

Suppose we have two service providers, $S_A \in E_s$ and $S_B \in E_s$, and each of them exposes one method with contracts $A(p_{A1}, \dots, p_{An})$ and $B(p_{B1}, \dots, p_{Bn})$. We want to compose them using composition operator $\circ \in O_s$. We compute $A_s(A, B, \circ)$, and if it returns *true* we can derive a new method with new contract. Properties of the new contract are computed using relation *propertyClass* for composition operator \circ and for every combination of properties from two existing contracts. That way we get new contract $C(p_{C1}, \dots, p_{Cn})$.

Figure 1 shows composition of a print service and a camera service. The print service has method *print*, which exposes a *print contract*, while the camera service has method *getImage*, which exposes a *getImage contract*. Camera can produce an image, while printer can consume it and print it, which is resolved during evaluation of two contracts. New contract is computed, *printImage*, and new service is created with new method, which enables direct printing of image from the camera representing an emerging property.

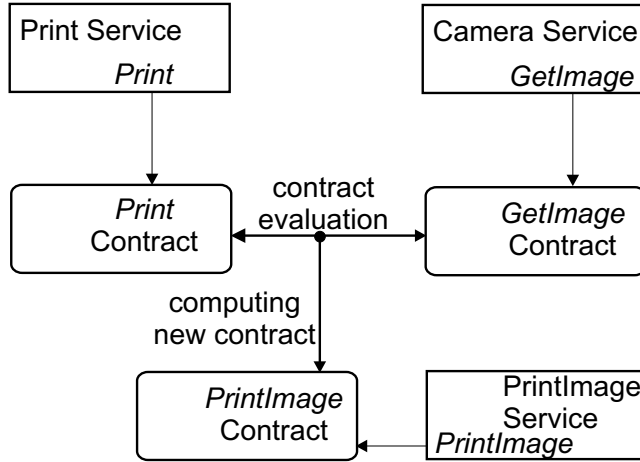


Figure 1: Derivation of a new service contract and composition of a new service.

To further facilitate computation of function A_s and generation of new contract, we introduce other mechanisms, such as creating a *hierarchy of service classes*, *collaboration graphs*, *learning* and *dictionaries*.

Every service provider is derived from a root *Service* class, which provides no additional information about the service whatsoever. Then we define subclasses of this root class (formally they can be seen as subsets of E_s), such as *Device*, *CommunicationLink*, *Client*, *Agent*, etc. If a service provider is declared as a member of some of these classes, we have more knowledge about it and about semantics of possible compositions. We know that composing device with a communication link is different from composing two communication links, and we use this knowledge when applying same composition operators on members of different classes. Class hierarchy can be provisionally deep, for example, *Agent* class can be further subclassed into action such as *Buy*, *Sell*, or *Find*. Related to this approach are collaboration graphs and learning. Once function A_s is computed, composition participants can cache the result in a collaboration graph that defines classes and instances they can collaborate with. Next time this function does not need to be computed if composition partner is already in the graph. This way service learns about its environment and can share collaboration graph or parts of it with other interested services in order to spread knowledge about possible compositions.

The other mechanism is to create a dictionary. Dictionary is a form of distributed database that resides on service directories. When publishing a contract to service directory, service provider can add entries to dictionary that can help when negotiating contracts, that is, computing A_s . For example, type details can be stored in dictionary. If print service from Figure 1 declares it accepts a *document*, and camera service does not understand what a document is, then it can check it in the dictionary. Printer service can add entry $\{(Document), (JPEG, TXT, PDF, PS)\}$ to the dictionary.

5 Service Decomposition

Many times there is a need to decompose specification of one 'big' task into a sequence or a graph of 'smaller' ones. In service environment we call it *requesting a custom service* or a *service decomposition*. The idea is to publish a contract and then find a composition of services that match that contract. The process is actually reversed composition of contracts. Here a final or target contract is given, and we need to find existing contracts and composition operators that together give the required service, specified by target contract. We need to ensure that chosen contracts and operators are composable by evaluating A_s on them and then to show that resulting properties match required ones. Formally, if we want to decompose contract $A(p_{A1}, \dots, p_{Aq})$, we identify *candidate contracts* and *candidate operators*. Suppose we identified contracts $B(p_{B1}, \dots, p_{Bn})$ and $C(p_{C1}, \dots, p_{Cm})$ as candidate contracts, and operator \circ as candidate operator. Then we need to prove that the following holds:

$$A_s(B, C, \circ) = true$$

$$A = B \circ C$$

Proving both parts is relatively easy, however, the major obstacle is finding adequate candidate contracts and operators. Collaboration graphs can be of great help here. Sometimes, candidate contracts may match functionally, but non-functional properties do not hold. That means that specified task can be accomplished, but without required non-functional attributes, such as execution time, price, or security. There is also a problem with *conflicting requirements*. For example, a contract can specify security and safety as desired properties. However, if we satisfy the safety requirement by introducing replication, we end with a breach of security requirement, since we are compromising the system by introducing another potentially unsecure copy. Therefore, we need to find another, non-conflicting way to satisfy both properties.

Figure 2 shows an example of a service decomposition. Suppose we have a service architecture with primitive mathematical operations, such as addition, subtraction, multiplication, division, power and square root. Let us say we want to find roots of the quadratic equation using this architecture. We know that for equation $Ax^2 + Bx + C = 0$, we can find roots using coefficients A , B and C . The formula for finding roots is our target contract here. Furthermore, we want to have the results in no more than 100 ms. This is a non-functional property we want to achieve. When we identify candidate contracts and operators, we show that when composed they produce required values (match target contract), and we create a graph and evaluate non-functional properties, in this case, worst case execution time. However, in order for this to be possible, the required non-functional property must be the *safety property* of the architecture. That means that every element must have that property. Indeed, it is impossible to give estimation on WCET if at least one service in the graph does not have that property.

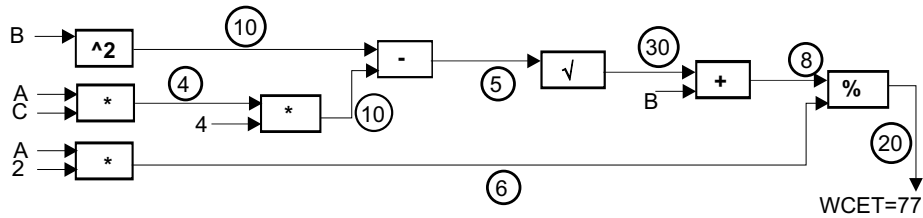


Figure 2: Service decomposition with respect to a non-functional property (Worst Case Execution Time).

6 Service Adaptivity

Adaptivity question was already partly addressed in previous sections, namely by showing composition and decomposition of contracts. However, there are two other situations where adaptation should be emphasized: *converter services (implicit composition)* and *rendering contracts*.

Implicit service composition can be done in cases when a client tries to invoke a service method, but fails due to some incompatibility (functional or non-functional). Composition can be done by service directory, in order to achieve desired form of compatibility. We see this as an adaptation and introduce converter services, that are used by service directories to achieve interoperation of mutually incompatible services. Following the previous examples, on Figure 3 we show how a service directory intercepts a message from a client to a printer service.

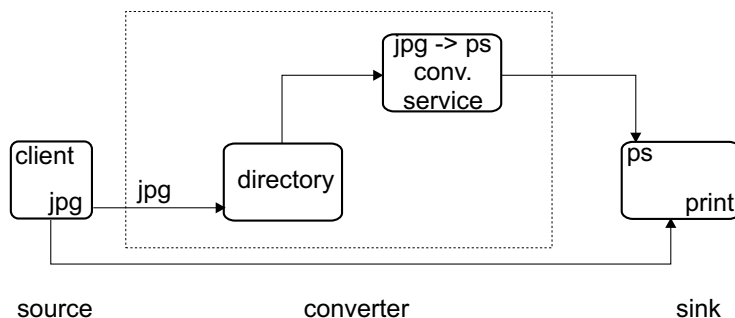


Figure 3: Converter service adapting client's request.

In this example we have a client wishing to print an image using a printer service. However, client can supply image in jpg format only, while printer can print only postscript. This is a clear case of functional incompatibility between the client and the service provider. The directory service can try to find some available compositions that would eliminate this incompatibility, that is, to convert an image from jpg to ps format. It intercepts the client request, locates appropriate converter service, composes converter and printer service, thus obtaining printer service that can print both jpg and ps formats. This action is entirely transparent to the client, which thinks it is communicating with print service only, and is unaware of the mediating composition.

Certainly, we are aware of the potential security implications this interception and mediation can introduce.

Since clients are accessing services using various devices, rendering contracts allow for user interface adaptation. They are bidirectional, in a way that they allow for service to present its data to user according to preferences and functionality of a client device, but also define means for getting input from a user. That means that during interaction both service provider and client must expose rendering contracts, which are negotiated and possibly composed with other services during execution to provide optimal interface.

7 Contracts and Dependability

Dependability is generally insured using either *forward* or *backward error recovery*. Backward error recovery means returning system to a state before an error occurred. It is usually implemented using a transaction mechanism. Forward error recovery tries to translate system into any stable, or correct, state and it can be implemented by exception handling mechanism.

Transactions are extremely successful in enforcing dependability in closed distributed systems. However, they are not equally adequate for service environments, because each service may implement transaction mechanism in a different, sometimes non-negotiable manner. Resource locking until transaction terminates is also not appropriate for services where many clients may issue simultaneous requests. This issue is resolved by introducing *split (open nested) model* [9]. Here every transaction splits into a number of concurrent subtransactions which can commit independently of each other. That way we have no overhead due to resource locking, but we must introduce compensating operations for all methods, because if one subtransaction aborts, others must compensate their actions. That means that dependability contract must define compensating action for every method. If a contract does not specify compensating operations, corresponding service will not be allowed to participate in split transactions. Split transactions and compensations are shown on Figure 4.

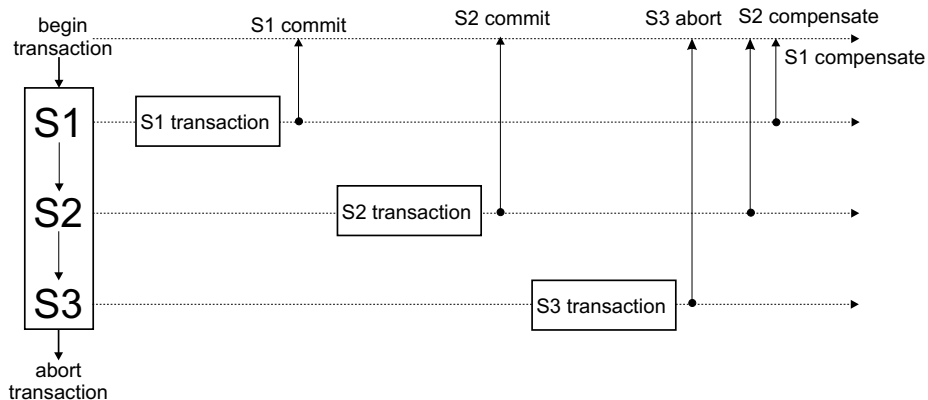


Figure 4: Backward error recovery using split transactions.

Forward error recovery can use exception handling. When exception occurs inside one service, it is handled locally and service is transferred into some correct state. However, other involved services remain unchanged. Even if they are notified about the exception, they may not be able to handle it. So the question is how to ensure integrity when exception occurs inside one service? One solution is rollback, but then it comes down to transactions and compensating operations. We introduce *aggregate exceptions* that are handled by all participating services concurrently. They are wrappers for internal service exceptions

which may not be understandable to other services. Wrapper exceptions enable uniform exception handling. In a case where wrapper exception cannot be handled by all interacting services, it is propagated to the client, and execution is stopped.

8 Service Reuse and Correctness

If we look at services as components, we see a great opportunity for reuse. Services could be the building blocks of future distributed architecture, the way components are today for software development. However, there is always the problem of trust. Can we trust that services offered by others will not compromise design or implementation in any way? Can we guarantee performance, quality of service or security if we reuse services?

One solution is to use contracts that specify all the relevant attributes for reuse, as we already showed. Use of contracts encourages reuse, because it lowers the probability of failures due to inappropriate or erroneous reuse. But can we trust contracts? What if a malicious service declares one thing in its contract and then performs something else? What we need is a way to prove correctness of a service implementation with respect to declared contract. That way we could prevent situations of inconsistent contract and implementation. Much work has been done in this area, especially in the area of *trusted components* [6], and that results can be used here. However, it is important to note that it is naive to expect that trusted initiative will result in a tool that will be able to prove correctness of *any* program. Proving correctness makes sense only in a certain context, such as proving properties of a program with respect to the contract it declares [7].

9 Conclusion

We have introduced basic concepts for the service composability and adaptivity. They are based on *contracts* which can be either generated manually and stored separately for each service provider, or embedded into service implementation. We showed several classes of contracts and how services can be composed and decomposed with respect to the contracts. Then, we showed what is needed to achieve adaptivity, dependability and reuse.

This effort is a part of the larger project, called NOMADS (Networks of Mobile Adaptive Dependable Systems) [4, 5]. The goal is to provide infrastructure for ubiquitous computing that has three basic properties: mobility, adaptivity and dependability. The key element is to ensure machine-to-machine interaction in a growing community of participating systems. We do it by introducing services, and hiding underlying software and hardware complexity behind contracts. Therefore, contracts must be expressive and diverse enough to cover many different systems and requirements, but also strict and formalized because it is the only way to achieve uniform composition and decomposition.

The further research should concentrate on developing various aspects of proposed service model based on contract evaluation.

References

- [1] J.R. Abrial: *The B Book*. Cambridge University Press, 1996.
- [2] D. Florescu, A. Grunhagen, D. Kossman: An XML language for Web service specification and composition. *Proceedings of the WWW'02 Conference*, 2002.
- [3] I. Foster, C. Kesselman, J.M. Nick, S. Tuecke: The Physiology of the Grid. <http://www.globus.org/research/papers/ogsa.pdf>, 2003.
- [4] M. Malek: Towards Dependable Networks of Mobile Arbitrary Devices - Diagnosis and Scalability. *Future Directions in Distributed Computing*, vol. 2548, *Lecture Notes in Computer Science*, A. Schiper, A. Shvartsman, H. Weatherspoon, and B.Y. Zhao, Eds. Springer 2003, 191-196.
- [5] M.Malek: The NOMADS Republic. *Proceedings of SSGRR2003s*, L'Aquila, Italy, 2003.
- [6] B. Meyer: The Grand Challenge of Trusted Components. *25th International Conference on Software Engineering*, Portland, Oregon, 2003, pp. 660-667.
- [7] B. Meyer: A Framework for Proving Contract-Equipped Classes. *Abstract State Machines 2003, Advances in Theory and Practice, 10th International Workshop*, Taormina (Italy), March 3-7, 2003, eds. E. Brger, A. Gargantini, E. Riccobene, Springer-Verlag, 2003, pp.108-125.
- [8] B. Meyer: *Eiffel: The Language*. Prentice Hall International, 1992.
- [9] T. Mikalsen, S. Tai, I. Rouvellou: Transactional attitudes: Reliable composition of autonomous Web services. *DSN 2002, Workshop on Dependable Middleware-based Systems (WDMS 2002)*, 2002.
- [10] Sun Microsystems: JINI Architecture Specification. <http://www.sun.com/software/jini/specs/jini2.0.pdf>, 2003.
- [11] Sun Microsystems: JINI Technology Core Platform Specification. <http://www.sun.com/software/jini/specs/core2.0.pdf>, 2003.
- [12] M. Werner, J. Richling: Komponierbarkeit nichtfunktionaler Eigenschaften - Versuch einer Definition. *GI Fachtagung Betriebssysteme*, Berlin, 2002.
- [13] W3C: Web Services Description Language (WSDL) Version 1.2: Core Language. <http://www.w3.org/2002/ws/desc/>, 2003.