

Architectural Support for Automatic Service Composition

Nikola Milanovic and Miroslaw Malek

Humboldt University, Berlin

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

Abstract

We consider architectural properties required for supporting automatic service composition. After defining composable service architecture, we proceed to examine the role of trust and reputation systems in such environment. Based on the proposed infrastructure we give several options for achieving automatic service composition, under the assumption that previously defined requirements are architecturally supported. Finally we discuss the impact and outlook of automatic composition.

Keywords: automatic service composition, architecture, Web services, search, trust, correctness

1. Introduction

Service Oriented Architectures (SOA) and Web Services (WS) are present in the mainstream scientific and industrial focus for many years. SOA promised advances in enterprise integration, B2B interactions and novel ways to process business workflows. However, industry is still using SOA mainly *inside* an enterprise as a helper for integration of different systems. Native WS capabilities are standardized: communication (SOAP), description (WSDL) and discovery (UDDI) [24]. Apart from that, WS architecture stack is mainly empty, meaning unstandardized.

There are many additional WS-frameworks and specifications aspiring to become standards (WS-Addressing, WS-Transactions, or WS-Coordination). What is not clear, however, is how they can or will cooperate with one another. Each solution targets a specific problem not taking into account other requirements. What is currently missing is a unification effort towards WS-Architecture [33]. One possible definition of a software architecture is [11]: A software architecture is defined by a configuration of architectural elements - components, connectors, and data - constrained in their relationships in order to achieve a desired set of architectural properties.

Our goal is identification of key SOA elements and constraints required to support service composition, verifica-

tion of composition correctness and automatic composition. Although remaining part of the paper focuses on Web services as the most prominent SOA available today, proposed methods are not limited to solving WS-specific issues only, since they offer architectural approach for designing SOA supporting automatic service composition property.

2. Related Work

Web service composition, as well as component composition in general, can be observed at two levels: component (service) and architectural level. Our survey of the composition proposals at the component level can be found in [21]. At this level, it is discussed how to orchestrate or choreograph services in different execution patterns using solutions like BPEL [6, 10] and BPELJ [1], Web component [35, 36], semantic Web and OWL-S [7, 16, 26], Petri nets [15], and finite state machines [4, 5, 12]. General problem of "industrial" approaches is lack of formal verification mechanisms, while more "academic" approaches are not easily applied in real-world production and enterprise frameworks and some face scalability problems. The issue that has been rarely addressed at all is modeling of non-functional properties.

Modeling service composability at the architectural level is in its embryonic stage and has some roots in architecture description languages (ADLs) [25], which are used to specify high-level compositional view of a software application. ADL focuses on software generation out of deployed components and offers state-transition semantics for analysis and verification of application specification. However, it has been noted [28] that new mission-critical and service-oriented applications require additional properties, namely *trust* and *dependability* analysis.

3. Composable Service Architecture

Most of the composition approaches presented are concerned with application level - how to facilitate construction of complex applications from available Web services. We form architectural foundation that enables not only creation

of new applications by the means of traditional programming techniques, but also adds methods for verification of composition correctness and automatic composition. The basic elements of *composable service architecture* are:

- Extended functional and non-functional (QoS) service description
- Enhanced search capabilities compared to UDDI
- Formal methods for composition and verification of composition correctness
- Distributed transaction management
- Distributed exception handling
- State management

Detailed description of one possible architectural solution supporting these requirements is described in [22, 23, 20]. Here we present only the basic idea.

Extending service description is based on Design by Contract paradigm [17]. We introduce an XML-based Contract Definition Language that enables specification of relevant functional and non-functional properties. Contract does not describe service implementation (how a service operates), but only semantics of its execution (what and under which conditions it provides). The main contract elements are pre-conditions, post-conditions and invariants. Pre-conditions are associated with Web methods and describe what a service expects from its clients. Post-conditions describe what a service will deliver in case pre-conditions are met. Invariants describe static service properties. Contract addresses other issues than connectivity (solved by Web Service Description Language - WSDL), such as service non-functional properties like dependability, security, real-time, location, price.

Searching is closely coupled with extended service description. UDDI directories limit searching for services belonging to organizations. Organizations can be searched by name, description (keywords) and classification. Only when an organization has been found, can its services be further searched. It is also possible to search using WSDL document names. These limiting search possibilities present an obstacle for industrial exploitation of Web services. Introducing extended service description enables richer search options, e.g., a possible query would be to find all services in the 1 km radius of the user's current location that accept postscript documents and print them in color with 1200 dpi resolution, free of charge if a user can supply a security credential of certain type. Besides being a clear advantage compared to UDDI when searching for single services, the ability to perform such complex queries is very important when searching for adequate composition partners.

Based on the contract description a formal composition framework is provided. One possibility is to model service contracts as abstract machines [2]. An abstract machine is characterized by statics (state variables) and dynamics (state functions). Web methods are represented as functions that change the service state. All functions are equipped with formally defined pre-conditions, post-conditions and invariants. Therefore, service has dual representation: XML contract (transport over a network) and abstract machine (formal reasoning). Service composition is then performed by merging abstract machines using five basic composition patterns (operators):

- *Sequence* executes two or more services in a sequential order.
- *Choice* executes two or more services in parallel and then non-deterministically chooses output of one and only one of them.
- *Parallel with communication* executes two or more services concurrently, then performs a logical operation on their outputs, and based on the operation result chooses one of the outputs.
- *Parallel without communication* executes two or more services concurrently without any communication and synchronization between them.
- *Loop* executes a service iteratively until an exit condition is met.

New services are constructed by applying different patterns to existing services. After a new composed service has been constructed, its correctness must be checked. The process of correctness verification takes the following steps:

- *Type checking* ensures that all types are correctly defined and that there is no infinite set inclusion in the resulting abstract machine.
- *Invariant preservation* ensures that composed invariant is preserved by all new operations.
- *Correct termination* ensures that all operations will terminate correctly (establish their post-conditions or abort), and that all operations are feasible (will establish exactly one or none, but not any post-conditions).

The elements introduced allow us to define a composable service architecture. It is defined as tuple $A(E, O)$ where E is a set of initial (atomic) services, and O is a set of composition operators. An operator $o \in O$ is a function that maps two (or more) services to a new service: $o : E \times E \times \dots E \rightarrow E$. Not all composed services are, however, valid members of the architecture. Therefore we introduce a function $correct : E \times E \times \dots E \times O \rightarrow \{true, false\}$.

Function $correct(e_1, e_2, ..e_n, o)$, where $e_1...e_n \in E$ and $o \in O$, is *true* if the composition of elements $e_1...e_n$ using composition operator o is correct, and returns *false* otherwise. The function $correct$ is calculated for the composite element $e(e_1, ..., e_n, o)$ in the following way ($check(e)$ is type checking, $proof(e)$ is invariant preservation, $trm(e)$ is correct termination and $fis(e)$ is feasibility):

$$correct(e) \iff check(e) \wedge proof(e) \wedge trm(e) \wedge fis(e)$$

A composable service architecture supports composability with respect to a correctness property. The traditional approach to composability works in the domain of elements being composed. We transpose this into the domain of system architecture which must guarantee *safety* property. The safety property must be possessed by all elements of the architecture. An architecture is then composable with respect to a safety property if and only if it allows the composition of elements having this property. That way we shift the focus from design of systems to design of architecture. Once a composable architecture has been defined, systems can be composed out of valid elements with safety guarantees.

In order to further support deployment and exploitation of composed services, issues related to cooperative execution of components in different containers must be solved, namely transaction, exception and state management. We implemented a distributed transaction management scheme known as *split* or *open nested* transaction model [14, 19, 30]. In this model, one transaction can be split into a number of subtransactions, that can commit independently. However, if one subtransaction aborts, others that have already committed must compensate. Therefore, for each Web method that can be involved in a transaction, a service must provide a compensate method. This model is well-suited for service architectures where it is expensive to lock resources for the duration of the whole transaction, and where transactions can take very long time to finish.

BPEL has an excellent solution for exception handling which is essentially distributed `try...catch` implemented within *scopes*. We augment it with generic exception wrappers enabling heterogenous components (throwing platform-specific exceptions) to be included in exception handling chains.

Although Web services are inherently stateless, many of them allow for the manipulation of the state, such as persisting data into databases, file systems, or coordinating dependent messages. There is ongoing debate in the community whether Web services should or should not support state management. One view is that Web services are not another Object Request Broker architecture, and therefore should have no notion of state [34], while the other view is that state management plays the critical role in distributed computing and as such must be addressed at the architectural level [8]. The former point may be true at the fundamental

level of Web services (discovery, description, invocation), but our position is that for the purpose of complex service interactions the latter view is correct. A solution for state management that we adopt is WS-Resource initiative [9].

4. Trust and Reputation Systems

Before proceeding to automatic composition, we investigate the notion of *trust* in a composable service architecture. Trust can be modeled at various levels: single services, composition patterns and composite services and reputation systems.

Solving trust at the level of a single service is relatively simple: when a service is deployed to a directory, its contract is verified for correctness. Publication of incorrect services is forbidden. After verification is passed, it is assumed that a service is correct and subsequent verification is performed only when its contract changes. Therefore, all published services are treated as correct themselves.

Dealing with composed services is a bit more complicated. One way is to perform verification after each composition. However, there is another way. Suppose we have two (or more) abstract machines M_1 and M_2 representing Web services and having *distinct* state variables x_1 and x_2 and invariants I_1 and I_2 . Let $P_1|S_1$ and $P_2|S_2$ be two methods of these services, where $P|S$ denotes substitution (operation body) S performed under pre-condition P (for details see [20]). Let us suppose that both services are correct. That means that, apart from other things, the following holds:

$$\forall x_1 \cdot (I_1 \wedge P_1 \Rightarrow [S_1]I_1)$$

$$\forall x_2 \cdot (I_2 \wedge P_2 \Rightarrow [S_2]I_2)$$

$[S]I$ denotes substitution S that preserves the invariant I .

Since we assumed that x_1 and x_2 are independent:

$$\forall (x_1, x_2) \cdot (I_1 \wedge I_2 \wedge P_1 \wedge P_2 \Rightarrow [S_1]I_1 \wedge [S_2]I_2)$$

It can be shown [2] that following holds, if substitutions S and T work on distinct (independent) state variables:

$$[S]P \wedge [T]Q \Rightarrow [S||T](P \wedge Q)$$

where $||$ is multiple simple substitution, or essentially a parallel composition pattern. Since we assumed that x_1 and x_2 are independent:

$$\forall (x_1, x_2) \cdot (I_1 \wedge I_2 \wedge P_1 \wedge P_2 \Rightarrow [S_1||S_2](I_1 \wedge I_2))$$

This gives us an important result: operation specified as $(P_1 \wedge P_2)|(S_1||S_2)$ preserves the invariant $I_1 \wedge I_2$. This suggests a mechanism by which we could *compose* correct abstract machine out M_1 and M_2 : the new machine invariant and pre-condition is calculated as conjunction of composed

machines' invariants and pre-conditions, while substitution (operation) is achieved by multiple substitution. It can also be shown that any combination of operations specified with other substitution (composition) constructs will also preserve $I_1 \wedge I_2$. In this case, verification of the composed machine is not necessary. It is enough that we know if starting machines are correct and operators *safe*, that is, proved to reestablish the invariant assuming correctness of starting services. Why is this important? Instead of proving complex invariant of the form $I_1 \wedge I_2 \dots \wedge I_n$ it is enough to prove that starting services are correct. If the services participating in the composition do not operate on independent variables (e.g., sequential composition), the above claim does not hold and composite invariant must be proved.

This all works assuming that contract is a faithful representation of underlying service. It should be noted, however, that we implicitly assumed that every service behaves strictly according to its contract and have proved contract correctness accordingly, but not actual service implementation correctness. This is, unfortunately, not entirely realistic assumption. The real question is how far can we trust a service contract? There are at least two cases where contract does not represent underlying service accurately:

- Mistake/inexperience of service deployer causing publication of correct but misleading (inaccurate) contract.
- Intentional and malicious forging of contract causing publication of malicious content under false contract.

Although some steps have been taken towards proving correctness of implementation with respect to formal specification [18], both cases are still virtually untraceable and undetectable. Therefore, in order for above discussion to be complete, a reputation system is required to maintain robustness and decrease probability of false/misleading/erroneous content being advertised through otherwise correctly formed service contract. The task of a reputation system is to keep track of submitted requests and subsequent responses and to rank performed operations according to several criteria:

- Did operation functionally succeed? Did return parameters match contract?
- Were there exceptions thrown in any scope up to the scope of the caller?
- Did operation terminate (were there deadlocks)?
- Is operation feasible (does it produce uniform results)?

This ranking is maintained for single as well as for composite services. When a single service ranks low, it is an indication that either its contract is malformed, or that it has implementation error. When a composite service ranks low,

reasons can be either on the side of service deployer (same as for single services), or on the side of service composer and consumer (specifying correct composition, but one that does not match problem specification).

5. Automatic Composition

The true expected value of service oriented architectures lies in business to business (B2B) interactions, where many services belonging to different organizations cooperate in solving complex tasks. The proposed framework addresses one important need of such B2B interactions: verification of composition correctness. However, another important need is automatic composition of services.

The problem of automatic service composition can be defined as follows: given the sets of available services E and composition operators O , and target (goal) service $t \notin E$, find the composition $e(e_1, \dots, e_n, o_1, \dots, o_m)$ where $e_1, \dots, e_n \in E$ and $o_1, \dots, o_m \in O$, such that $correct(e) = true$, and $e \equiv t$, where $t \notin E$. In other words, the task of automatic composition for a given target (goal) service t is to find adequate composition based on available services and operators that will produce a correct service e *equivalent* to t . We treat machine equivalency as syntax equivalence only. Therefore two machines are equivalent if and only if after renaming machine clauses (state variables and operation names) we obtain two identical machines.

Now we have defined a state space (atomic services and all correct composition of thereof), starting state (atomic services), goal state (target service) and rules (composition operators). We examine some properties of this problem, based on which we decide on search strategies [27]. The problem of automatic service composition is decomposable, assuming that target service (machine) is correct. The problem universe is predictable, if trust is assumed. That means that we always know what will be the result of applying a certain rule (operator) to the current state (composition). The rules application is recoverable, meaning that we can go back if a certain search path is misleading, but we will need to backtrack since rule application cannot just be ignored: a part of a solution will have to be "undone" or "uncomposed". Goal solution is absolute, assuming equality of machines is defined. That means that we do not need to compare multiple solutions. This is only true, however, if an absolute solution can be found. Otherwise, suboptimal solution can be negotiated. Rules are consistent, assuming that composition operators are well defined and proved. Finally, we aim for a solution where no interaction with the end user will be required.

Based on these observations, we formulate the following search strategies: heuristic search of state space, probabilistic automatic composition, automatic composition by learning.

5.1. Heuristic search

It should be obvious that brute force search, where all possible combinations of services and composition operators are explored until a composition matching the target is found, is unrealistic because combinatorial explosion renders it non-practical. It is not so much a problem of number of services, as of number of composition operators. The fact that operators can be n-ary and that same service can appear in a composition more than once further complicates any kind of non-heuristic search.

Since the state space is already well defined, the main problem is how to define a heuristic function, that is, how to measure how "close" one abstract machine is to another. It is naturally assumed that from this point further only correct compositions are taken into account (as defined by function *correct*), therefore abandoning all search paths that are incorrect with that respect.

We first implement two well-known "weak" methods that require no additional heuristics: depth-first search and breadth-first search in which we systematically generate correct compositions starting from available services and operators. Both algorithms deteriorate rapidly with the expansion of state space, former with increasing number of services and latter with increasing number of composition operators. Therefore, some additional knowledge of the problem domain is necessary. Introducing a simple heuristic for abandoning a certain branch:

- If more states are generated that the target machine has, abandon the branch since further application of composition operators can only increase or leave the number of states unchanged
- If composition operator is commutative and equivalent branch has been explored, abandon the branch.
- If composition operator is associative, and one association has been explored, ignore the branch with other association.
- If composition operator is distributive, and either distributed or condensed formula has already been applied, ignore the branch with the other one.

This heuristics improves the performance, but has problems if many composition operators are introduced, especially if they are *not* commutative, associative or distributive. Therefore, we introduce function $\delta(s_1, s_2)$ that defines a distance between two services s_1 and s_2 , and apply it as a heuristic for AO* search. δ evaluates between 0 and 1, where 0 denotes machine equality, while 1 denotes that we are comparing an empty machine. Metrics used can be simple number of dimensions (variables) that machines differ in, but more sophisticated schemes can be used where certain differences would be weighted more than the others. In each step

of AO* algorithm, function f' is calculated for all generated nodes:

$$f' = g(\text{start}, \text{current}) + \delta(\text{current}, \text{goal})$$

Function g estimates cost of getting from initial state to the current state. It can be a simple count of composition operators applied (if we want to minimize operator usage), but can also be more sophisticated by preferring certain operators to another or using contract-specific information (e.g., service price). By minimizing value of f' in each step of AO* algorithm, we are exploring branches that are more probable and lead faster to the solution.

AO* provides good results, especially when g has good approximation. Otherwise (e.g., if g is badly approximated or equal to zero) its usefulness degrades.

5.2. Probabilistic search

Another possibility is to use probabilistic search in order to reduce the number of branches explored. State space is represented as weighted directed graph with vertices representing services, and edges representing composition operators (Figure 1). Each edge is assigned a probability that a service from which an edge is originating will cooperate with a service in which the edge is ending. The sum of all outgoing weights for any node must be less or equal to one. If it is equal to one all possible interactions for a given service are known. Otherwise, we leave a possibility that other service(s) may cooperate with a given one, with probability of $1 - \sum_{i=1}^k w_i$, where w_i is weight of an outgoing edge.

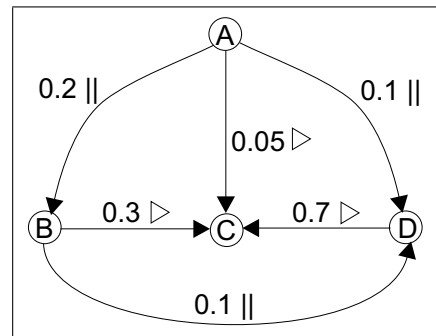


Figure 1. Cooperation Graph

The probability of a branch with length k is calculated by multiplying probabilities of edges building it:

$$P(B_k) = \prod_{i=1}^k w_i$$

For example, if we are looking for sequential compositions ending with service C from Figure 1, there are four

possible branches: $A \triangleright C$, $A||B \triangleright C$, $A||B||D \triangleright C$ and $A||D \triangleright C$, where $||$ and \triangleright denote parallel and sequential composition respectively. Probabilities of identified branches are 0.05, 0.06, 0.014 and 0.07, therefore a path $A||D \triangleright C$ is chosen.

This assumes, however, that events of choosing next cooperating service are independent. In our example there is no difference whether we arrived at node D from A or B : node C will be subsequently picked with 0.7 probability. Therefore we introduce causality by adding conditional probabilities. Assume A and B are two events, then:

$$P(AB) = P(A|B)P(B)$$

where $P(A|B)$ denotes probability of event A under the assumption that event B took place, while $P(AB)$ is the probability that both events occurred. Furthermore, if A_1, \dots, A_n are events, the following holds:

$$P(A_1A_2\dots A_n) = P(A_1)P(A_2|A_1)P(A_3|A_1A_2)\dots P(A_n|A_1A_2\dots A_{n-1})$$

Using these results we create additional conditional probabilities in cooperation graph that describe causality effect of choosing previous nodes. Let us assume that $P(D \triangleright C|A) = 0.1$ and $P(D \triangleright C|B) = 0.6$, that is, we now distinguish between cases $D \triangleright C$ when we arrive to D from A and from B . Now $P(A||D \triangleright C) = P(A||D)P(D \triangleright C|A) = 0.01$ and $P(A||B||D \triangleright C) = P(A||B)P(B||D)P(D \triangleright C|B) = 0.012$. The most favorable path now changes to $A||B \triangleright C$. Using formula for n-conditional events we could go deeper into the cooperation graph. However adding even this one level of causality provides a significant improvement compared to the graph with independent probabilities.

By creating cooperation (probability) graphs, we exploit implicit human knowledge of a state space properties, by assigning higher probabilities to combinations that are more likely to work out together. For example, a stock ticker service is more likely to cooperate with a stock trading or a printing service than with a book searching service, although all combinations are functionally possible. The usability of this approach depends almost entirely on the way initial probabilities are assigned. Therefore we try to make it more flexible by allowing probabilities to change over time. In an adaptive process, probabilities of branches (compositions) that are used more frequently are increased and vice versa. This change is made for all edges in a branch, while assuring that sum of all edges originating from any node in a branch does not exceed 1.

5.3. Automatic Composition by Learning

The method of adaptive conditional probabilities works good when either starting probabilities have favorable ap-

proximation or the weighted graph quickly converges towards optimal. When neither of these conditions are met, e.g., when nature of the requests changes frequently over time, AO* approach achieves better results. Another possible approach for relatively stable environments (one where requests can be at least classified or typed) is learning-based composition.

In a learning-based automatic composition a system is presented a target abstract machine t and then demonstrated a solution s (possible composition), which is afterwards persisted in a directory. A system is then presented with a set of all possible target machines $\{t_1, \dots, t_n\}$ such that $\forall m \in \{t_1, \dots, t_n\} | \delta(t, m) = 1$ and demonstrated a set of solutions $\{s_1, \dots, s_n\}$. This completes a 1-distance training. If a system is then presented with a 2-distance target machine d ($\delta(t, d) = 2$), the solution is obtained as follows:

1. Create all combinations of 1-distance solutions $\{s_1, \dots, s_n\} \times \{s_1, \dots, s_n\}$ and see if they match d .
2. If any matches d exit, else start substitution.
 - (a) For each element of newly generated solution set, consult service hierarchy and substitute one service at a time in each composition.
 - (b) Revalidate solution.
 - (c) If new solution matches d exit, else continue with substitution until all elements in all solutions have been substituted.
3. No solution has been found, proceed to AO* search, not considering branches already visited.

This approach can be extended to n-distance targets, by iteratively including $(k + 1)$ -distance solutions, where $k = 1, 2, \dots, p$, and p is the depth to which we want to perform training. The key idea is that by combining 1-distance solutions, 2-distance targets can be reached quickly. Obviously this cannot be proved, therefore a process of additional substitution is introduced. It is based on service hierarchy that is developed from classification information provided in a service contract. Classification is hierarchical, and determines what services are taken into consideration for substitution. If 1-distance solutions themselves cannot provide solution, substitution will try to locate services of similar capabilities and replace some of them. Service of similar capabilities is at the same hierarchy (classification) level as a service being substituted, or below it.

A simple scenario of this idea would work like this. Suppose we have a printer that can print only postscript. A system is taught to solve all printing requests by sending them to the printer. This works only if the document being printed is in the appropriate format. Therefore, a system is taught how to convert other formats: there is a class of converter

services that supply different types of conversions. Suppose further that a system is taught how to convert jpg file to ps by invoking appropriate converter service. If a system now receives a request to print a pdf file, it will look into all 1-distance compositions offering printing and find how to print jpg files. Then it will try to substitute a converter service with another service from the same class, or to substitute a printer service. Either way it will end up with a) printer that can print pdf, or b) converter service that can turn pdf into ps.

This approach will work only in relatively closed environments, e.g., inside an enterprise, since precise classification and ability to determine whether two services can be substituted is required.

6. Comparison and Industrial Perspective

In this section a discussion of the existing related approaches to automatic service composition is presented. In [3, 32] a method is proposed that reduces service composition to a constraint satisfaction problem. The main entity is an abstract process which contains abstract services. An abstract service is a placeholder for a set of physical (real) services that match the abstract service template, effectively competing for its place. Message exchange logic is specified in BPEL4WS. Competition is based on the idea of automated service discovery. After discovery candidate services are selected on the basis of process and business constraints.

In [13] a system called Proteus is presented that uses planning techniques for dynamic composition and execution of Web services. The main feature of Proteus is dynamic composition of integration plans. The annotated plan is submitted to a search engine that tries to find adequate services at run-time and substitute them in the plan.

Finally, in [29] an approach to automatic composition based on semantic web is presented. It is based on OWL-S ontologies. More specifically, OWL-S process model is used to develop a desired composition by creating a composite process comprising choreographed atomic processes. After composite OWL-S process is created, a search is performed in order to find the best matching services that can replace atomic processes (abstract service placeholders).

From the solution we proposed and from the related solutions presented in this section, it can be seen that there are two fundamentally different ways to handle automatic service composition:

- To start with the pre-defined generic composition and to perform 1-1 search to replace every generic element of a composition with a real service.
- To describe a set of goals and try to achieve them by building the whole process from scratch.

Clearly, constraint satisfaction, planning and semantic web-based automatic composition belong to the first approach. They provide methodology to describe pre-designed service choreography (empty composition skeleton) with placeholders that are to be filled with real services. They thus reduce the problem of automatic composition to the problem of finding adequate replacement for every abstract element of the pre-defined composition. We feel that service-oriented application developer should not think in terms of pre-defined compositions, but in terms of the problem that is to be solved. Our approach therefore does not require that composition should be pre-defined. Target abstract machine specifies properties of the problem (goal) itself, and not the way to achieve it.

Introduction of composable service architecture can impact both business and infrastructure (application development) layers. Business advantages and impact of moving towards an economy where services are associated on-demand in short running transactions have been discussed to some extent [31]. At the application development layer, profiling of three roles can be expected: architecture deployer, application developed and user. Architecture deployer defines composability rules, sets up service directory and deploys atomic services. Application developer acts upon deployed infrastructure in an attempt to build value added functionalities on demand using composition patterns. Finally, users invoke atomic or composite services from a directory. This profiling can make a dramatic change in the ways applications are developed and consumed. Potential impact lies in increased code dependability (through verification of pre-deployed services), easier application management and automated end-user support.

7. Conclusion

Development of a unified WS-Architecture is an enormous task, and as such can not be carried out by a single person or institution. Our intended contribution is to point to one architectural approach that can be used for achieving a certain degree of automatization of service composition. We presented foundations of such an architecture and then considered several mechanisms for automatic composition: from heuristic AO* search that guarantees solution but can take a long time to find one, to probabilistic composition and learning, which in some cases converge to a solution very quickly, but in non-favorable conditions degrade to AO* and slower methods, claiming additional overhead. Probabilistic composition and learning are more suited to limited use in a controlled environments where nature and frequency of service requests is known or can be predicted, while heuristic approaches are universally applicable, however face serious performance issues. We expect that by combining proposed approaches a viable architectural so-

lution for automatic service composition will be devised. We are developing additional hybrid mechanisms, such as backwards search, where state space is traversed from the goal (target service) to the starting state (available services) by means of algebraic decomposition of abstract machines, or a bidirectional search which performs forward and backward search simultaneously.

References

- [1] BPEL for Java technology. <http://www-106.ibm.com/developerworks/webservices/library/ws-bpelj/>, 2004.
- [2] J.R. Abrial. *The B Book*. Cambridge University Press, 1996.
- [3] R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint Driven Web Service Composition in METEOR-S. In *Proceedings of the IEEE SCC*, 2004.
- [4] D. Berardi, D. Calvanese, D. G. Giuseppe, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Proc. of the 1st Int. Conf. on Service Oriented Computing (ICSOC 2003)*, 2003.
- [5] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In *Proceedings of WWW2003*, 2003.
- [6] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana. The Next Step in Web Services. *Communications of the ACM*, October 2003.
- [7] A. Ankolekar et al. DAML-S: Web Service Description for the Semantic Web. In *Proceedings of the International Semantic Web Conference (ISWC)*, 2002.
- [8] I. Foster et al. Modeling stateful resources with web services. <http://www.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf>, 2004.
- [9] K. Czajkowski et al. The WS-Resource Framework. <http://www.globus.org/wsrfspecs/ws-wsrf.pdf>, 2004.
- [10] T. Andrews et al. Business Process Execution Language for Web Services. www.ibm.com/developerworks/library/ws-bpel/, 2004.
- [11] R. T. Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. doctoral dissertation, Dept. of Computer Science, Univ. of California, Irvine, 2000.
- [12] 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.
- [13] S. Ghandeharizadeh, C.A. Knoblock, C. Papadopoulos, C. Shahabi, E. Alwagait, J.L. Ambite, M. Cai, C. Chen, P. Pol, R. Schmidt, S. Song, S. Thakkar, and R. Zhou. Proteus: A System for Dynamically Composing and Intelligently Executing Web Services. In *Proceedings of the 2003 International Conference on Web Services (ICWS'03)*, Las Vegas, USA, 2003.
- [14] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, 1993.
- [15] 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.
- [16] S. McIlraith and T.C. Son. Adapting Golog for Composition of Semantic Web Services. In *Proceedings of the International Conference on the Principles of Knowledge Representation and Reasoning (KRR'02)*, 2002.
- [17] B. Meyer. Applying Design by Contract. *IEEE Computer* vol. 25, no. 10, pp. 40-51, October 1992.
- [18] B. Meyer. Towards practical proofs of class correctness. In *ZB 2003: Formal Specification and Development in Z and B, Proceedings of 3rd International Conference, Turku, Finland*, June 2003.
- [19] T. Mikalsen, S. Tai, and I. Rouvellou. Transactional Attitudes: Reliable Composition of Autonomous Web Services. In *DSN 2002, Workshop on Dependable Middleware-based Systems (WDMS)*, 2002.
- [20] N. Milanovic. Contract-based Web Service Composition Framework With Correctness Guarantees. In *Proceedings of the International Service Availability Symposium (ISAS 2005)*, Berlin, Germany, 2005.
- [21] N. Milanovic and M. Malek. Current Solutions for Web Service Composition. *IEEE Internet Computing*, November/December 2004.
- [22] 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*, Florence, Italy, 2004.
- [23] N. Milanovic and M. Malek. Verifying Correctness of Web Services Composition. In *Proceedings of the 11th Infodest*, Budva, Montenegro, 2004.
- [24] M.P.Papazoglou and D. Georgakopoulos. Service Oriented Computing. *Communications of the ACM*, 46, No. 10, pp 25-28, October 2003.
- [25] N. Medvidovic N and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70-93, 2000.
- [26] S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proceedings of the WWW02 Conference*, 2002.
- [27] E. Rich. *Artificial Intelligence*. McGraw-Hill, 1983.
- [28] H. Schmidt, I. Poernomo, and R. Reussner. Trust-by-Contract: Modelling, analysing and predicting behaviour of software architectures. *Transactions of the SDPS*, 5(3):25-51, September 2001.
- [29] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic Composition of Web Services using Semantic Descriptions. In *Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003*, Angers, France, April 2003.
- [30] F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. Coordinated Forward Error Recovery for Composite Web Services. In *Proceeding of the 22nd International Symposium on Reliable Dependable Systems, SRDS 2003*, Florence, Italy, 2003.
- [31] W.J. van den Heuvel and Z. Maamar. Moving toward a framework to compose intelligent web services. *Communications of the ACM*, 46, No. 10, pp 103-109, October 2003.
- [32] K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar, and J. Miller. METEOR-S WSDI: A Scalable Infrastructure of Registries for Semantic Publication and Discovery of Web Services. *Journal of Information Technology and Management*, 2004.
- [33] S. Vinoski. WS-Nonexistent Standards. *IEEE Internet Computing*, pages 25–28, November/December 2004.
- [34] W. Vogels. Web Services are not Distributed Objects: Common Misconceptions about the Fundamentals of Web Service Technology. *IEEE Internet Computing*, November/December 2003.
- [35] J. Yang. Web Service Componentization. *Communications of the ACM*, 46, No. 10, pp 35-40, October 2003.
- [36] 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.