Search Strategies for Automatic Web Service Composition[1]

Nikola Milanovic, Miroslaw Malek
Humboldt University Berlin
{milanovi,malek}@informatik.hu-berlin.de

ABSTRACT:

We investigate architectural properties required for supporting automatic service composition. First, composable service architecture will be described, based on modeling Web services as abstract machines supported by formally defined composition operators. Based on the proposed infrastructure we give several options for achieving automatic service composition by treating it as a search problem. Namely, basic heuristic, probabilistic, learning-based, decomposition and bidirectional automatic composition mechanisms will be presented and compared. Finally, we discuss the impact and outlook for automatic composition.

KEY WORDS:
Web services, automatic composition

INTRODUCTION

Service Oriented Architectures (SOA) and Web Services 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) (Papazoglu, 2003). Apart from that, WS architecture stack is mainly empty, meaning unstandardized.

There are many additional WS-frameworks and specifications aspiring to become standards (e.g., 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 (Vinoski, 2004). Our goal is identification of key SOA elements and constraints required to support service composition, verification of composition correctness and automatic composition. In this paper, we briefly present our previous work on architectural concepts and requirements, and focus on the problem of automatic service composition. Although remaining part of the paper is based 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.

The need for automatic service composition is justified by the ubiquity of the Internet which is forcing enterprises to abandon their heritage business models and legacy systems, and organize themselves into virtual enterprises (Heuvel, 2003) . On demand creation of virtual enterprises can shorten delivery times, increase product quality, deliver personalized services, decrease transaction costs, and accommodate short-term cooperating relationships, which can be as brief as a single business transaction. This paradigm requires a shift from tightly coupled business

[1] This paper presents an extension of the paper Architectural Support for Automatic Service Composition, published by the same authors in Proceedings of the IEEE International Conference on Services Computing (SCC 2005), Orlando, Florida, USA, 2005, 133-140.

components to more flexible and loosely coupled ones (Webber, 2003) that now dynamically interact with each other through automatic composition in ways that were not predefined and/or predicted in deployment time. The two major attributes required for such environment are *extensibility* and *adaptivity*. It is clear that in open environment like this, where services dynamically interact with each other on demand, being able to ensure correctness (dependability, security, timeliness) plays a crucial role. Web service architecture is considered a solution that can support extensibility and adaptivity required for dynamic composition (Yang, 2000).
The rest of the paper is organized as follows: first, our previous work in the area of the composable service architecture and modeling services as abstract machines will be described, that will serve as an environment in which automatic composition will be performed. Then, automatic service composition will be defined as a search problem and relevant properties of the problem will be examined. The state space and equality of abstract machines will be defined before proceeding with the following automatic composition mechanisms: basic heuristic, probabilistic, learning-based, backwards (decomposition) and hybrid. Finally, comparison with related approaches will be given, followed by conclusion and future work.

COMPOSABLE SERVICE ARCHITECTURE

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 (Milanovic, 2004a). At this level, it is discussed how to orchestrate or choreograph services in different execution patterns using solutions like BPEL (Curbera, 2003; Andrews, 2004) and BPELJ (Blow, 2004), Web component (Yang, 2002; Yang, 2004), semantic Web and OWL-S (Ankolekar, 2002; McIlraith, 2002; Narayanan, 2002), Petri nets (Hamadi, 2003; Zhang, 2004), and finite state machines (Berardi, 2003; Fu, 2002; Bultan 2003). General problem of "industrial" approaches is lack of formal verification mechanisms, while more "academic" approaches are not easily applied in real-word production and enterprise frameworks and some face scalability problems. The issue that has been rarely addressed at all is modeling of non-functional properties, although it has received some attention lately (Zhang 2005).
Modeling service composability at the architectural level is in its embryonic stage and has some roots in architecture description languages (ADLs) (Medvidovic, 2000), 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 (Schmidt, 2001) that new mission-critical and service-oriented applications require additional properties, namely *trust* and *dependability* analysis.
Most of the composition approaches 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 our previous work on one possible architectural solution supporting these requirements is described in (Milanovic, 2004; Milanovic, 2005a; Milanovic, 2005b). In this section we will present only the basic idea. The architectural concept is shown on Figure 1.

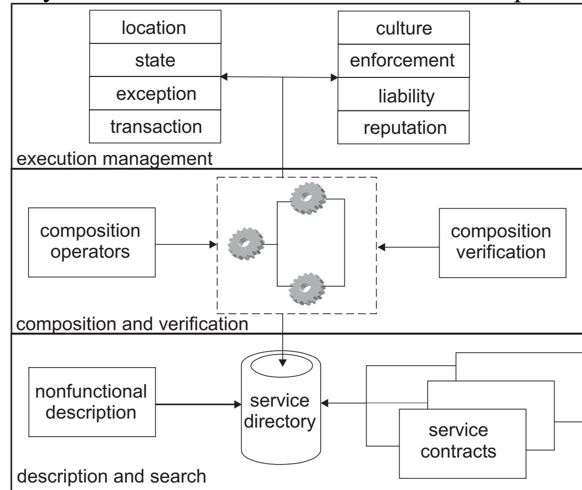

Figure 1: Composable Service Architecture

Extending service description is based on Design by Contract paradigm (Meyer, 1992). 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 (Abrial, 1996). 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. An algorithm was developed that transfers XML contract representation into abstract machine notation. The abstract machine structure is:

```
MACHINE M(X,x)
CONSTRAINTS C
CONSTANTS c
SETS S; T={a,b}
PROPERTIES P
```

```
VARIABLES v
INVARIANT I
ASSERTIONS J
INITIALIZATION U
OPERATIONS
  u1 <- O1(w1) = PRE Q1 THEN V1 END
  …
  un <- On(wn) = PRE Qn THEN Vn END
END
```

This is a parameterized abstract machine having free dimensions X (set) and x (scalar). CONSTRAINTS describes conditions on machine parameters. SETS contains finite or named sets that the machine can use, 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 pre-conditions PRE and post-conditions THEN.
Operation body of an abstract machine modifies a machine state. For expressing formally how such modification takes place, logical predicates relating the values of state variables just before the operation is invoked to the values just after the operation completes are used. This method is called substitution.
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* ($\|_P$)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* ($\propto_P$) executes a service iteratively until an exit condition is met.

New services are constructed by applying composition 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 operations if pre-conditions hold.
- *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).

A composition example is shown on Figure 2. Three services are composed, one that accepts a credit application from a client and returns it’s credit rating, and two banking services that accept credit rating and offer a loan. Banking services are executed asynchronously and the better offer is then chosen.

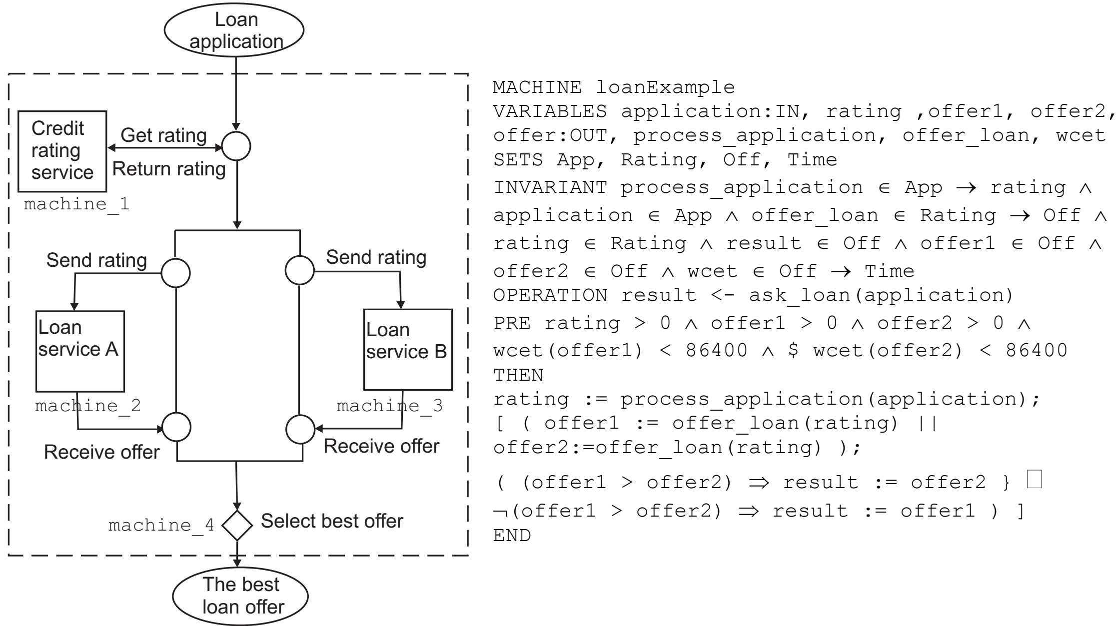

Figure 2: Loan composition example

One possibility to express this composition formally is:

$$\texttt{machine_1}\ \nabla\ (\texttt{machine_2}\ \|_{\texttt{offer1>offer2}}\ \texttt{machine_3})$$

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: $o: E \times E \times ... E \rightarrow E$. Not all composed services are, however, valid members of the architecture. Therefore we introduce a function *correct*: $E \times E \times ... E \times O \rightarrow \{true, false\}$. 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):

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

In order to further support deployment 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 (Gray, 1993; Mikalsen 2002; Tartanoglu, 2003). 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 (undo). 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

heterogeneous 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 (Vogels, 2003), 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 (Foster, 2004). 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 (Czajkowski, 2004).

EQUALITY OF ABSTRACT MACHINES

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 $e(e_1,...,e_n,o_1,...,o_m)$ where $e_1,...,e_n \in E$ and $o_1,...,o_m \in O$, 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.
However, information whether two machines are equivalent is not particularly useful on its own. In the process of automatic composition it is more important to know how two machines differ, and to be able to quantify their difference. Therefore we introduce metrics for calculating distance between two abstract machines. Distance is a number of dimensions and substitutions they differ in. Lexical differences are not taken into account, that is, it is allowed to rename clauses of one machine. The difference between machines m_1 and m_2 is thus given by:

$$\delta(m_1, m_2) = \frac{1}{2\alpha}\sum_{i=1}^{\alpha}\delta_d(d_{1i}, d_{2i}) + \frac{1}{2\beta}\sum_{j=1}^{\beta}\delta_s(s_{1j}, s_{2j})$$

where $|m|_d$ is the number of machine dimensions (state variables, machine formal parameters and constants), $|m|_s$ is the number of substitutions that make operation body (post-conditions), $\alpha = max(|m|_{1d},|m|_{2d})$, $\beta = max(|m|_{1s},|m|_{2s})$ and δ_d and δ_s calculate number of differing dimensions and substitutions:

$$\delta_d(d_1, d_2) = \begin{cases} 0, d_1 = d_2 \\ 1, d_1 \neq d_2 \end{cases}$$

$$\delta_s(s_1, s_2) = \begin{cases} 0, s_1 = s_2 \\ 1, s_1 \neq s_2 \end{cases}$$

Two dimensions are equivalent if and only if their types and directions (in case of state variables) are equal. Dimensions are therefore compared using their types and directions, not their names. In that respect, variables `input` and `in` are equivalent:

```
MACHINE A
SETS InputSet
VARIABLES input:IN
OPERATION doSomething PRE input ∈ InputSet
…
END
```

```
MACHINE B
SETS InputSet
VARIABLES in:IN
OPERATION doElse PRE in ∈ InputSet
…
END
```

Although they have different names, these variables have the same direction (IN) and their type-checking evaluates to the same set: *type*(input)=InputSet and *type*(in)=InputSet. Constants and machine formal parameters also constitute dimensions and their properties and constraints are implicitly taken into account in the process of type checking the same way that invariants and pre-conditions are used to type state variables.
Two substitutions are equivalent if and only if they perform the same substitution on equivalent state variables in the equivalent order. Consequently, two operations are equivalent if and only if all their substitutions are equivalent. The following two machines have equivalent operations:

```
MACHINE A
SETS InputSet, OutputSet
VARIABLES input:IN, output:OUT
INVARIANT output ∊ OutputSet
OPERATION output <- doSomething(input)
PRE input ∊ InputSet THEN
input := input + 1 □ input := input -1;
output := input END
END
```

```
MACHINE B
SETS InputSet, OutputSet
VARIABLES int:IN, out:OUT
INVARIANT out ∊ OutputSet
OPERATION out <- doSomethingElse(in)
PRE in ∊ InputSet THEN
in := in - 1 □ in := in +1;
out := in END
END
```

For multiple generalized substitutions and choice substitution, the order is irrelevant. That is the reason why operations of machines A and B are equivalent, even if the order of substitutions under choice clearly differs. On the other hand, if operation of machine B is defined as:

```
OPERATION out <- doSomethingElse(in)
PRE in ∊ InputSet THEN
output := in; in := in -1 □ in := in + 1 END
END
```

Operations are not equivalent anymore since they differ in the order of two substitutions. Generally speaking, judging difference of substitutions is far more difficult when compared to difference of dimensions. It is possible to develop a more precise measure of substitution difference by taking into account the actual substitution type and creating a function that is not purely binary, but offers a finer measurement of substitution equality. Therefore, function δ_s is modified as follows:

$$\delta_s(s_1, s_2) = \frac{1}{2} weight(s_1, s_2) + \frac{1}{2\gamma} \sum_{k=1}^{\chi} \delta_d(d_{1k}, d_{2k})$$

The second part of this function simply calculates the number of dimensions that two substitutions s_1 and s_2 differ in, where $\gamma = max(|s|_{1d}, |s|_{2d})$ and $|s|_d$ is the number of dimensions of substitution s. The first part is the weighted function that describes a semantic difference between substitution types. The function *weight* is given in Table 1.

	S;*T*	*S*‖*T*	PRE	CHOICE	IF	ELSE	ANY	WHILE
S;*T*	0							
S‖*T*	0.6	ι						
PRE	1	1	κ					
CHOICE	1	1	1	ι				
IF	1	1	0.5	1	κ			
ELSE	1	1	0.75	1	0.5	κ		
ANY	1	1	0.95	0.9	0.9	0.95	κ	
WHILE	0.9	0.95	0.9	1	0.9	0.95	0.95	κ

Table 1: Weight of substitutions

Value κ is used to compare two exact substitutions that may differ in their predicates.It makes sense to give two exact preconditions score 0 (equality), and score 0.5 otherwise, since two preconditions differ less than pre-condition and while substitution. The remaining difference in predicates will be calculated by the second part of the function δ_d. Therefore, κ is defined:

$$\kappa = \begin{cases} 0, \text{predicates equal} \\ 0.5, \text{otherwise} \end{cases}$$

Similarly, when comparing two multiple or choice substitutions, the value of 1 could be assigned if they differ, but again additional measure is introduced to soften this criteria by comparing number of operators (|| or □) in which they differ. Therefore, ι is introduced:

$$\iota = 1 - \frac{\min(|s_1|_{op}, |s_2|_{op})}{\max(|s_1|_{op}, |s_2|_{op})}$$

where $|s|_{op}$ is the number of operations in a substitution s. Values in the weight table are not fixed, nor do they have any constraints imposed upon them. Each value is inductively developed by observing behavior of different substitutions.

PROBLEM PROPERTIES AND SEARCH STRATEGIES

The problem of automatic service composition is essentially a search problem. To be able to formulate strategies for automatic composition of abstract machines, certain elements need to be defined for designing adequate search methods:

- *State space* containing all possible configurations of the objects upon which a search is performed. State space comprises atomic services and all correct composition of thereof.
- *Starting state*, which is one or more states from state space that describe possible situations (configurations) from which a search can start. These states are also called initial states. Starting/initial states are atomic services.
- *Goal state* is one or more states that can be accepted as a solution of a search. End state is a target service.
- Finally, a set of *rules* that describe the actions that are available for transforming initial state towards goal state. In our case rules are obviously composition patterns.

We examine some properties of this problem, based on which we decide on search strategies (Rich, 1983). The problem of automatic service composition is decomposable, under the assumption that target service (machine) is correct. A problem is decomposable if it can be transformed into a set of independent smaller or easier subproblems. Typical example of decomposable problem is symbolic integration. In that sense service composition can be treated as decomposable, but the practical applicability of decomposability is somewhat limited. For example, request for a service that makes flight and hotel room reservation can be decomposed into two subproblems: hotel reservation and flight reservation. Such decomposition, however, is not always obvious and/or easy to identify. Decomposition, as a divide-and-conquer methodology, will be investigated in more detail in Section about decomposition of abstract machines.

The problem universe is predictable since application of rules has a certain outcome. Indeed, predictability of composition properties was one of the main reasons for introducing abstract machines and composition patterns. It is always known what will be the exact result of applying a certain rule (operator) to the current state (composition). In other words, every time we make a move in the state space, we know precisely the following, resulting state. That means that we can plan an entire sequence of moves in advance. However, this is true only if trust is assumed. As already discussed, we compose service contracts trusting them to be correct and accurate representation of relevant properties of underlying implementation.

The rules application is recoverable. That means that we can go back if a certain search path is misleading, but we will need to backtrack to a certain point since rule application cannot just be ignored: a part of a solution will have to be "undone" or "uncomposed". For example, if we compose one hotel reservation service with one flight reservation service and then find out that flight reservation service does not support transactions causing the entire solution not being able to execute in one transaction, we may decide to drop the particular flight reservation service and try to locate another one. However, once another candidate has been located, we can not just compose it on top of previous solution. We must first backtrack to the point in state space where the previous flight reservation has been composed. That means that adequate control structure must be introduced to enable backtracking. The simplest way to do this is to use a stack to record rule (composition patterns) applications.
Goal solution is absolute, assuming equality of machines is defined. Once a satisfactory solution has been found, the search can be stopped. That means that we do not need to search further and compare multiple solutions, since only an equivalent solution can be found. This is only true, however, if an absolute solution can be found. Otherwise, suboptimal solution can be negotiated. For example, if a travel reservation system cannot locate both hotel and flight for a given price, it can offer the next best (although more expensive) solution to the user.
Rules are consistent, assuming that composition patterns are defined and proved. It means that it is allowed to use only the patterns that were previously defined (sequence, choice, parallel and loop composition) to move through the state space. Under this assumption, problem is consistent. This is only true if no additional knowledge is being used for reaching the goal. Otherwise, as the following sections will show, special attention must be paid that additional knowledge is also consistent.
We aim for a solution where no intermediate interaction with the end (human) user will be required. In case that two machine entities are communicating and trying to compose new service, such interaction is also not necessary. Human interaction can be required for two reasons: to provide additional input during the search process or to provide additional reassurance and justification of the solution to the user. If a solution cannot be found, a solitary way of solving automatic composition problem can be transformed into a conversational one, where end user is offered sub-optimal solution on one or more criteria, and has to accept this solution explicitly via some sort of interface. This should not, however, be the basic mode of operation.
Finally, since the objective of automatic composition is to find a path through a state space connecting starting state and goal state, there are two directions in which the search can proceed: moving from starting state towards goal state (forward search) or moving from goal state towards starting state (backward search). When deciding which strategy to use, number of start and goal states is usually taken into account. We would like to move from the smaller set of states to the larger (thus easier to find) set of states and in direction of the lower branching factor. The branching factor is the average number of states that can be reached directly from a single state. In both respects backward search seems to be more appropriate, since branching factor can be only equal or less when compared with forward search and number of goal states is certainly smaller. However, applying composition patterns in the *reverse* order to decompose an abstract machine is not trivial. Therefore we first investigate and develop forward search mechanisms, then introduce a way to decompose abstract state machines and finally propose a hybrid bidirectional solution.

MODELING STATE SPACE

Before proceeding to search methodologies, we must define how state space elements and traversals will be modeled. Basically, there are two choices: to model state space as a tree or as a graph (with option to switch between the two).

One form of a tree that can be used to represent state space transitions is a syntax tree (Aho, 1987). In a syntax tree, inner nodes represent composition patterns and leaf nodes represent services (abstract machines). Examples of some syntax trees for services *A*, *B* and *C* and composition patterns ∇ (sequence) and || (parallel) are given on Figure 3.

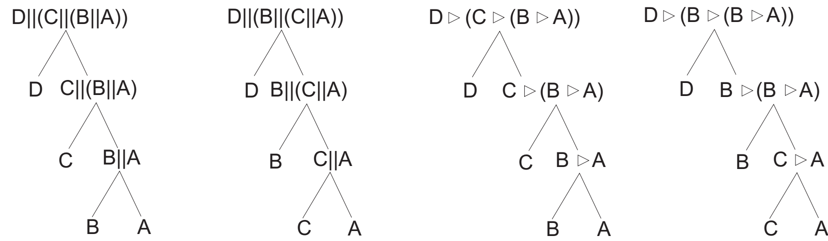

Figure 3: Syntax tree

The benefit of using syntax tree to model state space is that a syntax tree can be converted to deterministic/nondeterministic finite automaton. Such an automaton can be queried whether a given composition can be found in a syntax tree. The problem is, however, that this would require that entire (or at least a significant part of) state space is already expanded in a syntax tree. Syntax trees are not well balanced and addition of new nodes and elimination of duplicate ones are not trivial or cheap operations. An alternative to tree is to represent a state space as a graph.
The graph form we use is similar to AND-OR graphs which consist of OR edges and AND arcs, where one AND arc can point to any number of successor nodes. It is used primarily to represent problems that can be decomposed into smaller problems connected by AND arcs that must all be solved in order for the original problem to be satisfied. We use a similar idea to allow multiple arced edges to connect nodes that are being composed using given composition pattern. Instead of AND operation, arc can represent any composition pattern (Figure 4).

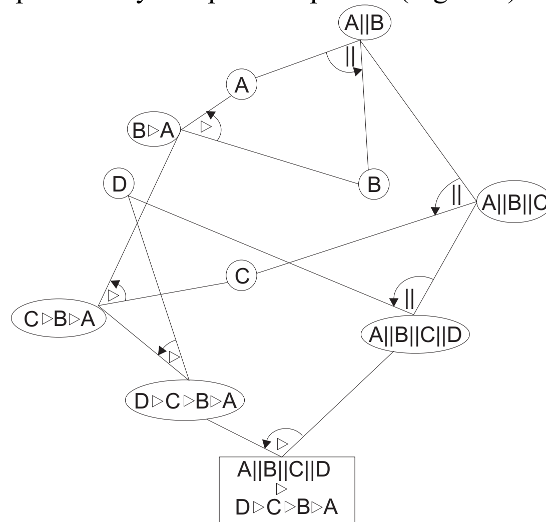

Figure 4: Composition graph

The benefit of using graph representation is that new nodes are added easily, as can bee seen with composition resulting in (*A*||*B*||*C*||*D*) ∇ (*D* ∇ *C* ∇ *B* ∇ *A*). It is also possible to define operand order using right-hand rule. Naturally, right-hand rule serves only to eliminate possible ambiguities in graphic representation and does not carry any other deeper meaning, since this is not a geometric graph. Naturally, there is an option to switch to a search forest. For every atomic service a tree is created with the atomic service as its root by taking all graph nodes in which that atomic service is the left-most operand, removing all nodes representing right-most operands, and

containing operators and right-most operands implicitly in the graph arcs, thus producing a search forest shown on Figure 5. We will use both representations interchangeably.
The issue remains how to construct and move through such composition graph/forest in order to find desired composition. In the next sections we present few strategies, starting with basic heuristic search.

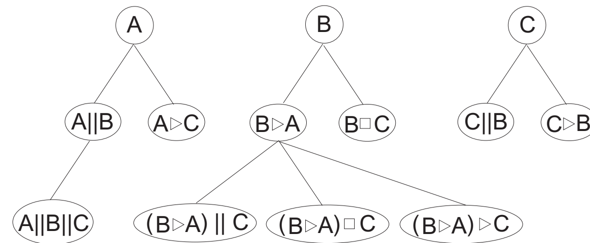

Figure 5: Part of a search forest

BASIC 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.
We first investigate two well-known "weak" methods that require no additional heuristics: depth-first search and breadth-first search (West, 1996) 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 the number of states.
- 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 try to develop appropriate heuristic function. A heuristic function maps from problem state description to measure of desirability (usually quantified as number). That means that for each element of state space it gives quantitative measure how close that state is to a solution. The purpose of a heuristic function is to guide the search process in the most promising (profitable) direction. It does so by suggesting which path to follow through the state space when more than one is available. The more accurate heuristic function is (the more accurately it evaluates the merit of each state), the faster and more direct will be the whole search process. Two well known heuristic approaches that fit our problem description are A* (Hart, 1968; Hart, 1972) and AO* (Martelli, 1973; Martelli, 1978; Nilsson 1980} search algorithms. We use them as basis for developing a range of heuristic search approaches for automatic composition of abstract machines.
Before proceeding to the description of the search algorithm, several elements will be introduced. Since state space is infinite, a measure of *futility* is introduced in order to cut search paths that

will never lead to the result. A value F will be a measure for the futility of a given search path. Different measures for futility can be accepted: it can be a value of heuristic function that shows that a distance to the goal is too big to be realistically reached, or a number of current solution's dimension which can be too large to fit into the goal machine. In any case, F must be such that it can guarantee that abandoning any search path will not result in a solution being missed, that is, that subsequent composition will not change the value of F so it becomes favorable again.

During the execution of the algorithm, three lists are maintained:

- `OPEN` contains nodes that have been generated and heuristic function has been applied to them, but they have not yet been expanded, (their successors have not yet been generated).
- `CLOSED` contains nodes that have already been examined and expanded, and have not crossed futility value.
- `LIMIT` contains expanded nodes that have crossed futility value.

For every `CURRENT` node that is being expanded a heuristic function f' is given with:

$$f'(\texttt{CURRENT}) = \delta\,(\texttt{CURRENT}, \texttt{GOAL})$$

where `GOAL` is the node representing composition target (search goal), and δ is the distance function given in Section 'Equality of Abstract Machines' . It will be used to pick the nodes that are closest to the goal node (with the smallest value of δ) to be expanded first.

Apart from using futility F and heuristic function δ to guide heuristic search, it is also important to detect and cut off equivalent search paths as early as possible. For example, it is obvious that compositions $A||B$ and $B||A$ are equivalent, yet they can appear more than once in search forest. All subsequent compositions based on these two nodes would be also equivalent, therefore one of the nodes can be safely removed. In order to deal with this issue, several rules are introduced that enable detection and handling of such cases:

1. Every abstract machine is a term.
2. If A and B are terms, than $A \nabla B$, $A \parallel B$, A □ B, $A \parallel_P B$, $A \propto_P B$ are also terms.
3. Every abstract machine is equivalent to itself.
4. Two terms $A_1 \bullet B_1$ and $B_2 \bullet A_2$ are equivalent if:
   - operator $\bullet$ is commutative and,
   - A_1 is equivalent to A_2 and B_1 is equivalent to B_2, or
   - A_1 is equivalent to B_2 and B_1 is equivalent to A_2.
5. Two terms $A_1 \bullet (B_1 * C_1)$ and $(A_2 \bullet B_2) * C_2$ are equivalent if:
   - operators $\bullet$ and $*$ are associative, and
   - A_1 is equivalent to A_2, B_1 is equivalent to B_2 and C_1 is equivalent to C_2.
6. Two terms $A_1 \bullet (B_1 * C_1)$ and $(A_2 \bullet B_2) * (A_2 \bullet C_2)$ are equivalent if:
   - operator $\bullet$ is distributive with respect to $*$ and,
   - A_1 is equivalent to A_2, B_1 is equivalent to B_2 and C_1 is equivalent to C_2

Finally, for every node function g is defined that is used for algorithm termination, if the value of g exceeds the threshold value F. Function g is accumulated during the forest traversal, and for every `NEW` node is given by:

$$g(\texttt{NEW}) = g(\texttt{CURRENT}) + \delta(\texttt{NEW},\texttt{GOAL})$$

The algorithm executes in the following steps:

1. `OPEN` contains the atomic services only. `CLOSED` and `LIMIT` are empty. Value of function g for every atomic service is 0. F is given an initial value.
2. Until the goal is reached or `OPEN` and `CLOSED` are empty, the following steps are repeated:
   - The node with the smallest value of δ is chosen from `OPEN`, assigned identifier `CURRENT` and removed from `OPEN`.
   - If `CURRENT` is equivalent to `GOAL`, `CURRENT` is returned and search is ended.

- Otherwise, successors of CURRENT are generated. For each NODE in OPEN, CLOSED and {CURRENT} the following is performed:
  a) Node NEW is generated from CURRENT, operator and NODE.
  b) If NEW is a solution, return NEW and exit.
  c) g(NEW) = g(CURRENT) + δ(NEW, GOAL) is calculated.
  d) If there is no equivalent node to NEW in OPEN, CLOSED or LIMIT, g(NEW) is compared to F. If g(NEW) > F, NEW is put to LIMIT, otherwise to OPEN.
- If no new nodes are added to OPEN, that is, all successors of CURRENT have g value larger than F, CURRENT is put to LIMIT, otherwise to CLOSED.
- If OPEN is empty, exchange OPEN and CLOSED.

At the start of the algorithm, list OPEN contains all atomic services, and all other lists are empty. F is assigned an initial value. The search will go on until a goal abstract machine is reached, or all subsequent expansions cross the limit F. For each node in *OPEN*, function δ is calculated and stored. The node with the smallest value of δ, that is, the node that is closest to the goal machine is chosen for composition. If more nodes have the same value of δ, choice is performed randomly. The chosen node is then composed with all nodes from OPEN and CLOSED using all operators. This is the expansion phase, where node successors are generated. The current node that is being expanded is always the leftmost operand. If any of the generated nodes is equivalent to the goal machine, search is ended. For each new node function g is calculated and compared to futility value F. In case $g > F$ the node is too far away from the goal to be considered further and the cut off is performed by storing the node in LIMIT. Each new node is also compared to all generated nodes that are stored in OPEN, CLOSED or LIMIT using equality rules. This step ensures that equivalent paths are detected and cut off. Finally, if OPEN is empty and no solution has been found yet, nodes that have not been considered (CLOSED) are moved to OPEN and expanded. A complete step-by-step example of the algorithm execution can be found in (Lenk, 2005).

At the end, complexity of the proposed solution will be examined. We first observe the complexity of one algorithm cycle (search for candidate nodes with the smallest δ and their expansion) as a function of the number of nodes n in the OPEN list. The complexity is proportional to:

$$n \cdot (n + n \cdot n) = n^3 + n^2$$

Therefore, complexity of one algorithm cycle is $O(n^3)$, where n is the current number of the nodes in the OPEN list. This number, however, changes with every algorithm cycle, therefore a more important question is how this number changes in every algorithm cycle and how many cycles does the algorithm require in order to find a solution. Let m be the number of atomic services, o the number of composition operators and s the number of algorithm cycles. The number of generated nodes in each algorithm cycle is then given by:

$$m \cdot (o + 1)^s$$

Clearly, it follows that algorithm has an exponential complexity. However, s is the function of F, meaning that exponential growth of generated nodes is the upper complexity bound, or the worst case complexity, when all generated nodes fall below the futility value and heuristic function return equal values for all nodes. Analytical dependency between s and F is very difficult to express in the general case as shown in (Gashing, 1979) for the A* algorithm and therefore is outside the scope of this paper. Instead, we will discuss the possible ways to build value F in order to reduce the complexity.

One simple way to determine the value for F is to calculate the number of generated state variables for each node. If this number is greater than the number of state variables for the goal node, the current node is moved to the LIMIT list, as subsequent compositions can only increase the number of state variables. Another way is to introduce execution time as futility value,

limiting search temporally. This clearly makes sense in some applications where time limit can be naturally determined (e.g., bank transaction involving currency conversion, credit card verification and payment should not take longer than 2 minutes). Finally, the number of generated nodes can also terminate the search, as storage capacity is not infinite, and all generated nodes must be stored, even those that have crossed the futility value (`LIMIT`) or have already been expanded (`CLOSED`). This is required because of the equivalence comparison when new nodes are generated in order to detect equivalent and redundant paths.
However, the best way to guarantee efficient and fast heuristic algorithm is to have quality heuristic function that will guide the search in the most promising direction. Various additions to the heuristic function presented here will try to improve this result by using more efficient heuristic functions. They will always have more favorable average execution complexity than this approach, as they will fall back to the basic heuristic search in case the solution has not been found using advanced heuristics.

PROBABILISTIC AUTOMATIC COMPOSITION

Heuristic function from the previous section uses distance between abstract machines as a measurement which branch is most promising to follow. In this section we augment that heuristic with the idea of probabilistic search. The additional heuristic is represented as weighted directed graph with vertices representing services and edges representing composition patterns, as shown on Figure 6. 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 equal to one, all possible interactions for a given service are known (which is very unlikely). Otherwise, there is a possibility that unknown services can cooperate with a given one, with probability of $1-\sum_{i=1}^{k} w_i$, where w_i is weight of an outgoing edge and k is number of outgoing edges.

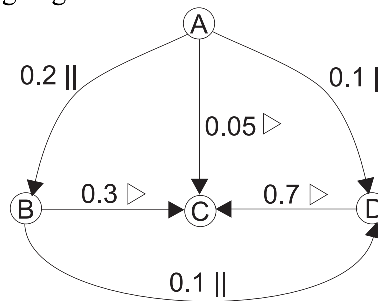

Figure 6: Cooperation graph

The probability of a branch of length k is calculated by multiplying probabilities of constituent edges:

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

For example, if we are looking for sequential compositions ending with service C from Figure 6, there are four possible branches: $A \nabla C$, $A \parallel B \nabla C$, $A \parallel B \parallel D \nabla C$ and $A \parallel D \nabla C$, where $\parallel$ and ∇ denote parallel and sequential composition respectively. Probabilities of identified branches are 0.05, 0.06, 0.014 and 0.07, therefore the path $A \parallel D \nabla C$ 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:

$$P(AB) = P(A \mid B)P(B)$$

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 $A_1,...,A_n$ are events, the following holds:

$$P(A_1 A_2 ... A_n) = P(A_1)P(A_2 \mid A_1)P(A_3 \mid A_1 A_2)...P(A_n \mid A_1 A_2 ... A_{n-1})$$

Using these results we create additional conditional probabilities in cooperation graph that describe causality effect of choosing previous nodes (Figure 7). Let us assume that $P(D \nabla C \mid A)=0.1$ and $P(D \nabla C \mid B)=0.6$, that is, we now distinguish between cases $D \nabla C$ when we arrive to D from A and from B. Now $P(A \parallel D \nabla C) = P(A \parallel D)P(D \nabla C \mid A) = 0.01$ and $P(A \parallel B \parallel D \nabla C) = P(A \parallel B)P(B \parallel D)P(D \nabla C|B) = 0.012$. The most favorable path now changes to $A \parallel B \nabla C$. 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 graph with independent probabilities.

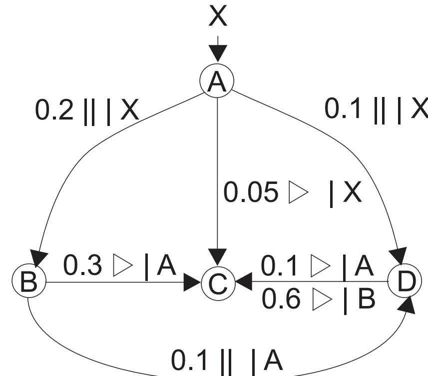

Figure 7: Causal cooperation graph

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. Fixed probabilities however are not very realistic. Therefore we try to make the approach 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. For all composition patterns two tables are maintained: *compositions* and *probabilities*. In *compositions* rows and columns represent services and entries number of successful compositions. Table *probabilities* has the same structure and at the beginning is populated with initial probabilities. After assigning initial probabilities $P_{init}(N_i,N_j,op)$, *compositions* table entry (N_i,N_j,op) is incremented when N_i and N_j are composed using pattern op. Total number of compositions for each pair (n) is also maintained. After each composition, current probability is calculated $P_{current}(N_i,N_j,op) = k(N_i,N_j,op)/n$, where k is *compositions* table entry for (N_i,N_j,op). This probability is not automatically stored in the table *probabilities*. If any value in row N_i becomes such that $|P_{init}(N_i,N_j,op) - P_{current}(N_i,N_j,op)| > \varepsilon$, probabilities of entire row are recalculated and stored in *probabilities*: $P_{new}(N_i,N_j,op) = P_{current}(N_i,N_j,op)$. To prevent fast (and non-realistic) oscillations, comparison of $P_{current}$ and values from *probabilities* can be done periodically and not every time *compositions* is updated. Also,

table *compositions* can be reset, with current probabilities accepted as new initial probabilities. Value of ε can also change if necessary.
The usability of this approach depends almost entirely on the way initial probabilities are assigned. Initial probabilities are important for two reasons: they determine starting conditions under which services compete, and can also be used when resetting *compositions* table, if for some reason one does not want to use current probabilities. If initial probabilities are not realistic, convergence to optimal balance can take a lot of time and render the whole approach unusable. Therefore a method for assigning initial probabilities is proposed, using service classification.
Figure 8 shows layered service classification, similar to OSI layered network model, comprising physical, network and application service classes. Classes can talk to neighboring classes only, e.g., service of class physical can talk to network class only, network can talk to both physical and application, while application can interact with network class. This does not mean that actual interaction among physical class and application class is not possible, but only that in the process of distributing initial probabilities such interactions are not taken into account. There is a special subclass of all three classes called agent. Agents represent generic properties of each class and can interact with each other across any class. For example, memory agent of the physical class interacts with convert agent of the application class with probability 0.3 (Figure 8).

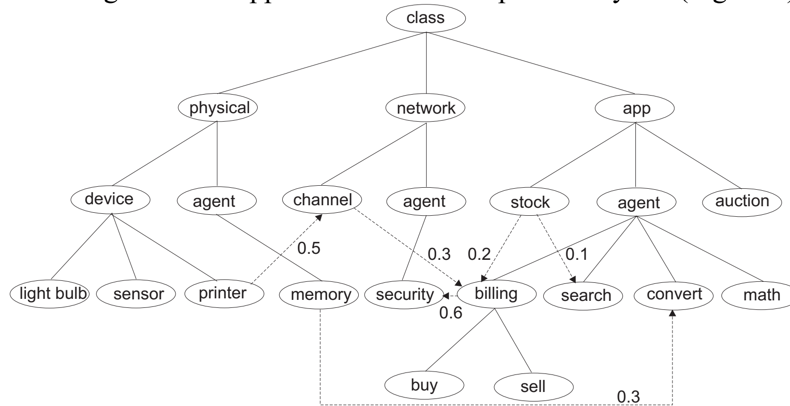

Figure 8: Service classification and initial probabilities

Initial probabilities can be assigned directly in the classification, or as a set of rules. Rules have the syntax (*source_class*, *destination_class*, *probability*). Rule is applicable to all subclasses of a given class. A subclass can redefine one or more rules, and the new rule is then further applicable to itself and its subclasses. Cooperation graphs define fine-grained interaction between services as they determine probabilities that particular service instances will cooperate using particular composition patterns. Classification defines coarse-grained interaction between service classes (not services) that can be applied to all services belonging to a given class.

AUTOMATIC COMPOSITION BY LEARNING

The method of adaptive conditional probabilities works good when starting probabilities have favorable approximation and the weighted graph quickly converges towards optimal. When neither of these conditions are met, e.g., when nature of the requests changes frequently, basic heuristic approach achieves better results. Another possible approach for relatively stable environments (one where requests can be at least typed) is learning-based composition.
Let T be a set of target abstract machines and S a set of all possible solutions (compositions). In a learning-based automatic composition a system is presented a target abstract machine $t \in T$ and then demonstrated a solution $s \in S$ (possible composition), which is afterwards persisted in a directory. A system is then presented with a set of all possible target machines $\{t_1,...,t_n\} \subseteq T$ such

that $\forall\ m \in \{t_1,...,t_n\} \mid \delta(t,m) = 1$ and demonstrated a set of solutions $\{s_1,...,s_n\} \subseteq S$. 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 $\{s_1,...,s_n\}$ x $\{s_1,...,s_n\}$ 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 is found, use basic heuristic search, not considering branches already visited.

The key premise 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 introduced in the previous section, 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. An element can be substituted with either an element or the same class or element from any of its subclass. For example, member of network agent class can be replaced by another member of network agent class or by a member of security class.

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 gives best results in more controlled environments, e.g., inside an enterprise, since precise classification and ability to determine whether two services can be substituted is required.

DECOMPOSITION OF ABSTRACT MACHINES

Decomposition of abstract machines is backwards search methodology. We begin from the goal state (target abstract machine) and iteratively try to decompose it into simpler machines connected with composition patterns until we reach a starting state consisting of atomic (available) machines. The inverse path taken from goal state to starting states is the required composition.

Decomposition should clearly be the preferred way of doing automatic composition, since we are moving from the known goal state (match of user's requirements) to the smaller (compared to all possible compositions) and thus easier to find set of starting states (atomic abstract machines). The approach to decomposition is based on transforming target machine substitutions to postfix form. The decomposition algorithm has three main phases:

1. Convert operation body to postfix representation.
   a. Generate relevant clauses for all variables copied to the output thus creating corresponding abstract machine.
   b. Verify correctness of every variable (machine) copied to the output.
   c. If a copied machine exists in directory, mark it as finished and put in finished list.

2. Scan finished postfix string and determine possible compositions.
3. Check composition to determine whether all elements are in the finished list.

We now describe the first phase in more details. Postfix conversion is performed on target machine operation body. During conversion, operator priorities are evaluated using Table 2. Result of conversion are variables and operators. Variables are machine state variables; operators are either composition patterns or generalized substitutions. In the process of postfix conversion we gradually decompose abstract machine operation body by extracting its sub elements (state variables) and composition patterns that build the goal abstract machine. The process of postfix conversion consists of the following steps:

1. Let the final `END` of the target abstract machine be a terminating symbol.
2. Terminating symbol is pushed onto the stack.
3. Variables are always copied to the output.
4. Left parenthesis is always pushed onto the stack.
5. When a right parenthesis is encountered, the symbol at the top of the stack is popped off the stack and copied to the output. This is repeated until top of the stack is left parenthesis. Then both parenthesis are discarded.
6. If an operator has a higher priority that the operator at the top of the stack, it is pushed onto the stack and stack pointer is incremented.
7. If the priority of the operator is lower or equal to the operator on top of the stack, one element of the stack is popped to output. The stack pointer is not decremented. Instead the current operator is compared with the new top of the stack.
8. When the end symbol is reached, the stack is popped to the output until terminating symbol is also reached. Then the conversion terminates.

priority	operator
3	$\propto_P$
2	∇
1	$\mid\mid$, $\mid\mid_P$
0	□

Tables 2: Composition operator priority

Since we operate on operation body only, we need to generate other abstract machine clauses when a variable is copied to the output. For each state variable, target machine clauses are scanned. If a state variable appears in a given target machine clause, that clause is copied to the output and joined to the corresponding variable. Therefore, if the current variable being scanned is `ticketPrice` and there exists a pre-condition `PRE ticketPrice > 0` in the target machine, then this pre-condition is associated with the variable at the output.

Finally, based on a postfix string and the content of the finished list, adequate composition is created. This step is best explained using an example. Suppose that we want to build a composite service that takes a loan application from the client, determines its credit rating, applies for a loan with two banks and then chooses the better (higher) loan offer. We start by specifying target abstract machine (service), that we already used in section Composable Service Architecture:

```
MACHINE loanExample
VARIABLES application:IN, rating ,offer1, offer2, offer:OUT, process_application,
offer_loan, wcet
SETS App, Rating, Off, Time
INVARIANT process_application ∈ App → rating ∧ application ∈ App ∧ offer_loan ∈ Rating →
Off ∧ rating ∈ Rating ∧ result ∈ Off ∧ offer1 ∈ Off ∧ offer2 ∈ Off ∧ wcet ∈ Off → Time
OPERATION result <- ask_loan(application)
PRE rating > 0 ∧ offer1 > 0 ∧ offer2 > 0 ∧ wcet(offer1) < 86400 ∧ wcet(offer2) < 86400
THEN rating := process_application(application);
[ ( offer1 := offer_loan(rating) || offer2:=offer_loan(rating) );
```

```
( (offer1 > offer2) ⇒ result := offer2 □
¬ (offer1 > offer2) ⇒ result := offer1 ) ]
END
```

Machine accepts variable `application` representing loan application. Application is processed and as a result variable `rating` is produced representing applicant credit rating. Two loan offers, `offer1` and `offer2` are then generated in parallel by sending credit rating to two banks. After both loan offers are ready (service will wait up to 24 hours which is specified in the pre-condition), they are compared and the better (higher) one is chosen. The process of postfix conversion produces the following string (line breaks are added for clarification only):

```
rating process_appplication(application) :=
offer1 offer\_loan(rating) := offer2 offer\_loan(rating) := ||
offer1 offer2 > result offer2 := ⇒
offer1 offer2 > ¬ result offer1 ⇒ □ ; ;
```

The postfix string is then iteratively scanned from left to right, in an attempt to extract possible constituent abstract machines and composition patterns connecting them. Variables are scanned until a first operator is reached, which is then applied to the variables. All variables scanned in a single pass are assigned to a single abstract machine. If the obtained abstract machine exists in the finished list, the machine construction is finished and the scan continues with the new machine. Otherwise new variables/operators are being added to the same machine. The process continues until the end of the postfix string is reached (Figure 9).

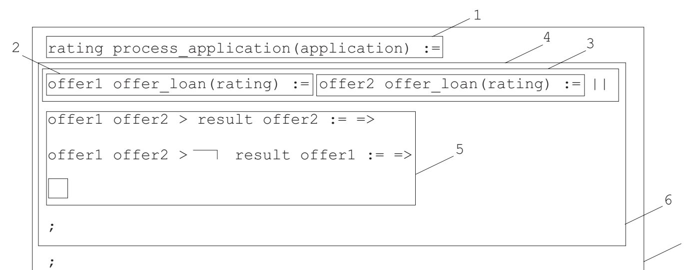

Figure 9: Postfix string scan

In step one, variables `rating` and `process_application(application)` are connected with assignment operator. Suppose that the following abstract machine is in the finished list, that is, it exists in a directory:

```
MACHINE machine_1
VARIABLES application:IN, rating:OUT, process_application
SETS App, Rating
INVARIANT process_application ∊ App → Rating ∧ application ∊ App ∧ rating ∊ Rating
OPERATION rating <- op_1 (application)
PRE THEN rating := process_application(application)
END
```

Note that clause `PRE rating` > 0 from the original machine is not included, since `rating` is the output parameter for which pre-conditions cannot be defined. Since the previous part of the string exists in a directory, in steps two and three, variables `offer` and `offer_loan (rating)`, as well as `offer2` and `offer_loan(rating)` are connected using assignment operator. Suppose that the following two machines are also in the finished list:

```
MACHINE machine_2
VARIABLES rating:IN, offer1: OUT, offer_loan
SETS Rating, Off
INVARIANT rating > 0 ∧ offer1 ∊ Off ∧ offer_loan ∊ Rating → Off
```

```
OPERATION offer1 <- op_2(rating)
PRE rating > 0
THEN offer1 := offer_loan(rating)
END

MACHINE machine_3
VARIABLES rating:IN, offer2: OUT, offer_loan
SETS Rating, Off
INVARIANT rating > 0 ∧ offer2 ∈ Off ∧ offer_loan ∈ Rating → Off
OPERATION offer1 <- op_3(rating)
PRE rating > 0
THEN offer2 := offer_loan(rating)
END
```

The postfix string is scanned further. The next symbol in step four is parallel composition operator that is applied to `machine_2` and `machine_3`. In step five, the first scanned expression yields the following machine:

```
MACHINE machine_4
VARIABLES offer1:IN, offer2:IN, result:OUT, wcet
SETS Off, Time
INVARIANT offer1 ∈ Off ∧ offer2 ∈ Off ∧ result ∈ Off ∧ wcet ∈ Off → Time
OPERATION result <- op_4(offer1, offer2)
PRE offer1 > 0 ∧ offer2 > 0 ∧ wcet(offer1) > 86400 ∧ wcet(offer2) > 86400
THEN (offer1 > offer2) ⇒ result := offer2
END
```

Suppose, however, that this machine does not exist in the finished list. This is the key point in the decomposition algorithm where a decision has to be made whether to proceed further with postfix string or to go back. If a decision to go back is taken, the previous compositions have to be decomposed in a backtracking process. We would effectively try to incorporate `machine_4` in `machine_3` or `machine_2`. The other solution is to proceed forward and to try to incorporate the next part of the postfix string in `machine_4`. The approach we adopt is that we move one time in each direction until a service that exists is reached. Therefore, in the continuation of step five, we modify `machine_4` by scanning postfix string until we reach the next operator:

```
MACHINE machine_4
VARIABLES offer1:IN, offer2:IN, result:OUT, wcet
SETS Off, Time
INVARIANT offer1 ∈ Off ∧ offer2 ∈ Off ∧ result ∈ Off ∧ wcet ∈ Off → Time
OPERATION result <- op_4(offer1, offer2)
PRE offer1 > 0 ∧ offer2 > 0 ∧ wcet(offer1) > 86400 ∧ wcet(offer2) > 86400
THEN (offer1 > offer2) ⇒ result := offer2 □
¬ (offer1 > offer2) ⇒ result := offer1
END
```

Supposing that this machine exists, we move to step six in which `machine_4` is connected to parallel composition of `machine_3` and `machine_2` using sequence operator. Finally, in step seven `machine_1` is sequentially composed to the already built composition. The result is:

$$\texttt{machine_1}\ \nabla\ (\texttt{machine_2}\ \|\ \texttt{machine_3})\ \nabla\ \texttt{machine_4}$$

In case that `machine_4` did not exist in a directory, decomposition algorithm would try to backtrack and incorporate all scanned but unassigned variables into the last valid construction, that is, into parallel composition of `machine_2` and `machine_3`. The result of this step is:

```
MACHINE machine_5
variables rating:IN, offer1, offer2, result:OUT, offer_loan, wcet
SETS Rating, Off, Time
INVARIANT offer_loan ∈ Rating → Off ∧ rating ∈ Rating ∧ offer1 ∈ Off ∧
offer2 ∈ Off ∧ result ∈ Off
OPERATION result <- op_5(rating)
```

```
PRE rating > 0 ∧ offer1 >0 ∧ wcet(offer1) < 86400 ∧ offer2 > 0 ∧
wcet(offer2) < 86400
THEN [ ( offer1 := offer_loan(rating) || offer2:=offer_loan(rating) );
( (offer1 > offer2) ⇒ result := offer2 
¬ (offer1 > offer2) ⇒ result := offer1 ) ]
END
```

Now, if this machine exists in a directory, decomposition result would be:

$$\texttt{machine_1} \nabla \texttt{machine_5}$$

If not, a process of postfix conversion is applied again, this time to machine_5, which finally yields that:

$$\texttt{machine_5} \equiv \texttt{machine_2} \parallel_{offer1>offer2} \texttt{machine_3}$$

and therefore final decomposition is:

$$\texttt{machine_1} \nabla (\texttt{machine_2} \parallel_{offer1>offer2} \texttt{machine_3})$$

The last example outlines two major problems of decomposition, namely:

- Incorrect abstract machines: what is the behavior of the algorithm when abstract machine scanned in a postfix string is not correct?
- Algorithm missing decomposition path: what happens when abstract machine scanned in a postfix string does not exist in finished (list) directory, and moving forward/backward does not produce an existing machine?

In both cases decomposition algorithm can not guarantee that a solution will be found. This is the consequence of a very simple control strategy that we adopted for both cases. When either a machine is incorrect or does not exist in a directory, first a forward scan is performed, and in case that also does not result in a valid/existing machine, solution is backtracked in a backward scan. Instead of providing more complex control strategies that could achieve higher hit probability, we develop a hybrid bidirectional search mechanism.

HYBRID BIDIRECTIONAL AUTOMATIC COMPOSITION

The last automatic composition mechanism that will be described is hybrid bidirectional search. The basic idea is to use advantages of both forward and backward search in order to eliminate two problems: long execution time of heuristic search and missing solution path in decomposition. Bidirectional search simultaneously performs forward and backward search thus creating two search paths. The search is over when (if) the two paths meet, and the solution is constructed by merging them. Forward search can be performed using any of the methodologies presented so far: basic heuristic search, probabilistic search, or search by learning. Backward search is performed by decomposing target abstract machines. There are three ways to perform bidirectional search: no special control strategy, depth specification, means-ends analysis.

When no special control strategy is employed to steer direct bidirectional search, steps in both direction are made sequentially. After each step, current states are compared. If the current state of backward search is subset of the current state of forward search, the algorithm terminates. This approach is simple to implement as it does not require any additional control protocol. However it suffers from one problem: search paths may miss each other, completely or partially (Figure 10).

In the worst case of successful execution, this kind of search will perform complete forward search, and also spend additional overhead for unsuccessful backward search. That means that it will last longer that forward search only. Therefore, in the second approach, depth of both directions can be specified as an input parameter. For example, backward search can be allowed to progress for only one level, thus making it easier to satisfy several "smaller" target machines with a subsequent forward search. Similarly, forward search can be allowed to progress to a certain level thus decreasing the number of goal states for subsequent decomposition. In this case, bidirectional search is not performed concurrently in both directions. Rather, first one direction is

explored to a certain level, in order to make the search in the opposite direction easier and/or more effective. Bidirectional search (without special control strategy and with both depths set to one) for the loan application example from the previous section is shown on Figure 11.

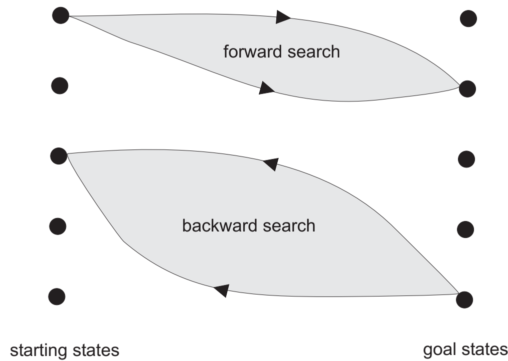

Figure 10: Bidirectional search problem

There are still two open issues: which direction to favor (forward or backward) and up to which depth to limit the auxiliary search? We answer both questions by using means-ends analysis (Ernst, 1969). It detects differences between current and goal state and tries to make a move in a state space that will reduce this difference. We modify this method to allow for a decision in which direction to move and for how many steps. The algorithm proceeds like this:

- Until the current state of backward search is not a subset of the current state of the forward search, or difference table offers no more options, do the following:
    1. Calculate the difference between two current states.
    2. Use a distance function and a difference table to determine whether to execute a forward or a backward move
    3. Update current states accordingly
- If goal is achieved, the algorithm terminates successfully, otherwise it fails.
-

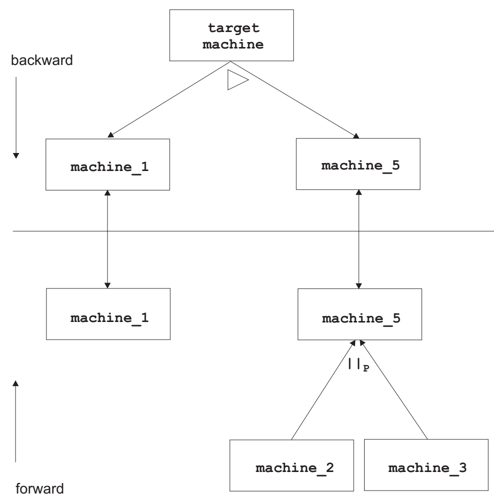

Figure 11: Bidirectional search example

During search process, pre-conditions and post-conditions are evaluated in order to determine whether forward of backward search should be performed. In forward search, post-conditions of available services are matched against pre-conditions of target services. In backward search, pre-conditions of target services are matched to post-conditions of available services. Matching is performed using a difference table that describes which operation (direction) is appropriate to

reduce the difference between the current and the goal state. Suppose we want to print a Word document and have only a printer that can print Postscript with embedded fonts. Available are also the following services: converter from Word to PDF, converter from PDF to Postscript and service that can embedd fonts to PDF. Abstract machines describing these services are:

```
MACHINE PrintPS
SETS Type={Word,PDF,PS}, Paper
VARIABLES Doc, print, fonts, type
INVARIANT print ∈ Doc → Paper ∧ fonts ∈ Doc → {Embedded,NotEmbedded} ∧
type ∈ Doc → Type
OPERATION printPS(Doc)
PRE type(Doc) = PostScript ∧ fonts(Doc)=Embedded
THEN result:= print(Doc)
END

MACHINE Fonts2PDF
SETS Type={Word,PDF,PS}
VARIABLES type, Doc, Fonts
INVARIANT type ∈ Doc → Type ∧ fonts ∈ Doc → {Embedded,NotEmbedded}
OPERATION fonts2PDF(Doc, Fonts)
PRE type(Doc) = PDF
THEN type(Doc) = PDF ∧ fonts(Doc) = Embedded
END

MACHINE PDF2PS
SETS Type={Word,PDF,PS}
VARIABLES Doc, type
INVARIANT type ∈ Doc → Type
OPERATION pdf2PS(Doc)
PRE type(Doc) = PDF
THEN type(Doc) = PS
END

MACHINE Word2PDF
SETS Type={Word,PDF,PS}
VARIABLES Doc, type
INVARIANT type ∈ Doc → Type
OPERATION word2PDF(Doc)
PRE type(Doc) = Word
THEN type(Doc) = PDF
END
```

Difference table is shown on Table 3. Sometimes there may be more than one operator that can reduce a given difference, but also one operator may be able to reduce more than one difference.

	word2pdf	pdf2ps	fonts2pdf	printps
print				X
convert	X	X	X	
embedd			X	

Table 3: Difference table

Assuming that target machine is given below, the search proceeds like on Figure 12:

```
MACHINE print
SETS Type={Word,PDF,PS}, Paper
INVARIANT type  ∈ Doc → Type ∧ print ∈ Doc → Paper
VARIABLES Doc, type, print, result
PRE type(Doc) = Word
THEN result := print(Doc)
END
```

In the first step, backward search is performed and from difference table operation `printPS` is selected. However, service `PrintPS` is not equivalent to the goal state, since it accepts only Postscript documents with embedded fonts. Further decomposition yields no result, because there is no way to either transfer Word file directly to Postscript or to embed fonts into a Postscript document. Therefore, at this point a difference is determined and forward search is attempted. The difference is type of input parameters and whether fonts are embedded within the document. Difference table suggests using operation `fonts2pdf` to reduce font difference. By examining pre-conditions we see that this operation can be performed for PDF documents only. That means

that this difference cannot be reduced at this point. Difference table suggests three operations for reducing type differences: `word2pdf`, `pdf2ps` and again `fonts2pdf`. Pre-conditions determine that only `word2pdf` operation can reduce the difference. At this point we have a PDF document without embedded fonts. Now operation `fonts2pdf` can be applied thus eliminating one difference (font embedding). After that `pdf2ps` is applied to reduce the last remaining difference (document type). The two searches meet and the algorithm terminates.

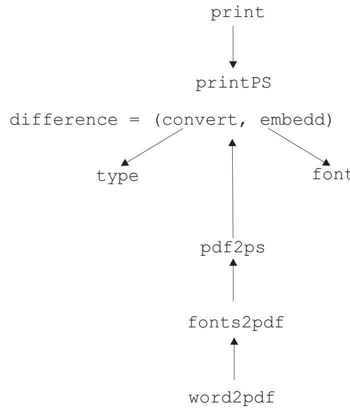

Figure 12: Means-ends bidirectional search

RELATED APPROACHES

In this section a discussion of the existing related approaches to automatic service composition is presented. Two main strategies will be described, namely constraint satisfaction and planning, together with comparison with the methods we have proposed.

In (Aggrawal, 2004; Verma, 2004} a method is presented 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. Competition is based on the idea of automated service discovery (Cardoso, 2003; Paolucci, 2002}. Automated discovery is performed using user defined requirements and produces set of candidate services. After discovery candidate services are selected on the basis of process and business constraints.

The main stages of creating dynamic process is development, annotation, discovery, composition and execution. Different semantics can be used: data, functional and quality of service. The part that is most relevant for automatic composition deals with design of abstract processes. It involves the following steps (Figure 13): creation of desired flow using control flow constructs provided by BPEL4WS, annotating BPEL flow using templates that express service properties, and specifying constraints that will be used for optimization.

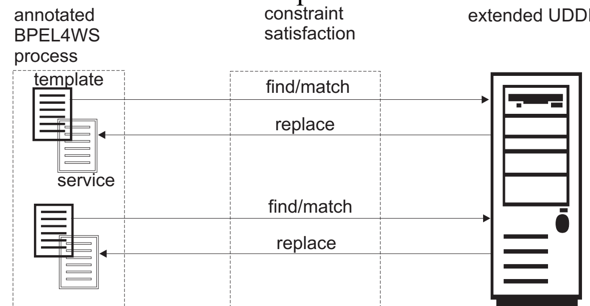

Figure 13: Constraint satisfaction-based automatic composition

BPEL annotation is performed using different ontologies. Partner services are represented as annotated abstract services. A search on extended UDDI is performed, and for each template a set of matching services is identified. In the process of optimization, constraints are evaluated and candidates are eliminated. Constraint satisfaction can be performed upon service dependencies, querying and cost estimation or process constraints. After candidate services have been filtered and identified, BPEL process is translated into executable form by adding physical addresses of selected partner services to BPEL deployment descriptor and sent to BPEL server for execution.
In (Ghandeharizadeh, 2003) 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 plans that integrate Web service. Besides planning, Proteus offers plan execution and monitoring. The system behavior is very similar to constraint satisfaction. Service description is annotated with additional expressions using WS-Inspection (Ballinger, 2001). The annotated plan is submitted to a search engine that tries to find adequate services at run-time and substitute them in the plan. An integration plan is generated that binds identified services into the requested plan and this plan is then executed.
In (Zeng, 2004) another method similar to constraint satisfaction is presented. Composition is modeled using statecharts over which execution paths are constructed. Statechart comprises generic elements called tasks. Execution path is any complete path of tasks from starting state to ending state. Any subset of a set of all possible execution paths for which quality of service properties are evaluated is a potential execution plan. The problem is how to identify the execution plan that corresponds to the user-expressed requirements. If there are n tasks (states) and m candidate Web services that are identified for each task, the number of all execution plans is m^n which makes this method of automatic composition impractical. Therefore, integer programming (Karloff, 1991) is used for selecting an optimal execution plan without the need to generate all possible plans.
Inputs to planning are a set of variables, objective function and set of constraints. Set of variables describe quality of service properties of each task that is being considered. Constraints are used-defined limitation on price, execution duration, execution price, reputation, success rate and availability. Objective function compares current execution plan to the constraints. Both objective function and constraints are linear. In the process of composition value of objective function is maximized or minimized by adjusting the values of variables while enforcing the constraints. The output is the maximum (or minimum) value of the objective function from which the values of the variables can be extracted for this maximum (minimum). The set of variables determines which candidate service instances actually populate tasks during physical execution.
Finally, in (Sirin, 2003) 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).
Automatic composition has two main components: composer and inference engine. The inference engine is essentially a directory that has the capability to find matching services that best fit specified abstract processes. It is designed as a knowledge base using Prolog. The composer is the interactive part of the system. It enables user to create a workflow of services and it also presents all available choices to the user at every step. That means that despite knowledge base, composition has to be performed (partly) manually. At every step, functional and non-functional properties of participating services are matched, and some candidates are rejected. This process can also be assisted by a human operator. After a desired composition has been found, that is, after all abstract processes from OWL-S process model have been substituted by real services from directory, the entire composite process is stored in a directory from which it can be invoked.
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 composition described in a generic manner 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, semantic web-based automatic composition and integer programming belong to the first approach. They all 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. It does not prejudice composition process by requiring that certain services should be composed in such-and-such manner. Identification of composition patterns and candidate services is left to the automatic composition algorithm based on the target abstract machine. The problem specification domain is in that way elegantly decoupled from the solution specification domain. By specifying target abstract machine, we do not think about *how* to solve a problem, but *what* and under which *conditions* we need to achieve. We see this as clear advantage compared to other automatic composition concepts that require that entire solution should be premeditated in advance.

CONCLUSION AND FUTURE WORK

Development of a unified WS-Architecture is an enormous task, and as such cannot be carried out by a single person or institution. Our intended contribution is to offer an architectural approach that can be used for achieving a certain degree of automatization of service composition. We presented foundations of such architecture and then considered several mechanisms for automatic composition: from basic heuristic 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. In non-favorable conditions however, they degrade to 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 basic heuristic approach is universally applicable, but face some performance issues. We expect that by combining proposed methods a viable architectural solution for automatic service composition will be devised. We also presented 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. We are currently investigating use cases in which average performance of presented algorithms can be determined, using implementation of the composition and verification server presented in (Milanovic, 2005a).

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. 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.

ACKNOWLEGMENT

The authors would like to acknowledge Maren Lenk, who helped in developing basic heuristic search as part of her master thesis.

REFERENCES

Abrial, J.R. (1996), *The B Book*, Cambridge University Press.

Aggarwal, R., Verma, K., Miller, J., Milnor, W. (2004), Constraint Driven Web Service Composition in METEOR-S, *Proceedings of the IEEE Service Computing Conference*, Shangai, China, 23-30.

Aho, A.V., Sethi, R., Ullman, J.D. (1987), *Compilers, Principles and Tools*, Addison Wesley.

Andrews, T., et al. (2004), Business Process Execution Language for Web Services, *www.ibm.com/developerworks/library/ws-bpel,* accessed in April 2005.

Ankolekar A., et al. (2002), DAML-S: Web Service Description for the Semantic Web, *Proceedings of the International Semantic Web Conference*, Sardinia, Italy, 348-363.

Ballinger, K., et al. (2001), Web Services Inspection Langauge (WS-Inspection), http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html, accessed in June 2005.

Berardi, D., Calvanese, D., Giuseppe, D.G., Lenzerini, M., Mecella, M. (2003), Automatic composition of e-services that export their behavior, *Proceedings of 1st International. Conference on Service Oriented Computing*, Trento, Italy, 43-58.

Blow, M., et al. (2004), BPELJ: BPEL for Java, *www-106.ibm.com/developerworks/webservices/library/ws-bpelj/*, accessed in May 2005.

Bultan, T., Fu, X., Hull, R., Su, J. (2003), A New Approach to Design and Analysis of E-Service Composition, *Proceedings of the 12th International World Wide Web Conference*, Budapest, Hungary, 403-410.

Cardoso, J., Seth, A.P. (2003), Semantic E-Workflow Composition, *Journal of Intelligent Information Systems*, 21(3), 191-225.

Curbera, F., Khalaf, R., Mukhi, N., Tai, S., Weerawarana, S. (2003). The Next Step in Web Services, *Communications of the ACM*, 46(10), 29-34.

Czajkowski, K., et al. (2004), The WS-Resource Framework, *http://www.globus.org/wsrf/specs/ws-wsrf.pdf*, accessed in May 2005.

Ernst, G.W., Newell, A. (1969), GPS: *A case study in Generality and Problem Solving*, Academic Press, New York.

Foster, I., et al. (2004), Modelling Stateful Resources with Web Services, *http://www.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf,* accessed June 2005.

Fu, X., Bultan, T., Su, J. (2002), Formal Verification of E-Services And Workflows, *Proceedings of Workshop on Web Services, e-Business and the Semantic Web; Foundations, Models, Architecture, Engineering and Applications*, Toronto, Canada, 188-202.

Gasching, J. (1979), Performance Measurement and Analysis of Certain Search Algorithms, *PhD Dissertation, Carnegie-Mellon University*.

Ghandeharizadeh, S., et al. (2003), Proteus: A System for Dynamically Composing and Intelligently Executing Web Services, *Proceedings of 2003 International Conference on Web Services,* Las Vegas, USA, 17-21.

Gray, J., Reuter, A. (1993), *Transaction processing: concepts and techniques*, Morgan Kaufmann.

Hamadi, R., Benatallah, B. (2003), A Petri Net-based model for Web Service Composition, *Proceedings of the 14th Australasian conference on database technologies*, Adelaide, Australia, 191-200.

Hart, P.E., Nilsson, N.J., Raphael, B. (1972), Correction to 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths', *SIGART Newsletter,* 37.

Hart, P.E., Nilsson, N.J., Raphael, B. (1968), A Formal Basis for the Heuristic Determination of Minimum Cost Paths, *IEEE Transactions on SSC,* 4(2), 100-107.

Heuvel, W.J. van den, Maamar, Z. (2003), Moving toward a framework to compose intelligent Web services*, Communications of the ACM*, 46(10), 103-109.

Karloff, H. (1991), *Linear Programming*, Birkhauser.

Lenk, M. (2005), Heuristic Composition of Abstract Machines, *Master Thesis, Humboldt University Berlin*.

Martelli, A., Montanari, U. (1973), Additive And/Or Graphs, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence,* Stanford, USA, 1-11.

Martelli, A., Montanari, U. (1978), Optimization Decision Trees Through Heuristically Guided Search, *Communication of the ACM,* 21(12), 1025-1039.

McIlraith, S., Son, T.C. (2002), Adapting Golog for Composition of Semantic Web Services, *Proceedings of the International Conference on the Principles of Knowledge Representation and Reasoning*, Toulouse, France, 482-496.

Medvidovic, N., Taylor, R.N. (2000), A classification and comparison framework for software architecture description languages, *IEEE Transactions on Software Engineering,* 26(1), 70-93.

Meyer, B. (1992), Applying Design by Contract, *IEEE Computer*, 25(10), 40-51.

Mikalsen, T., Sai, S., Rouvellou, I. (2002), Transactional Attitudes: Reliable Composition of Autonomous Web Services, *Proceedings of the Workshop of Dependable Middleware-based Systems*, Washington D.C, USA, 44-53.

Milanovic, N., Malek, M. (2004a), Current Solutions for Web service Composition, *IEEE Internet Computing,* 8(6), 51-59.

Milanovic, N., Malek, M. (2004b), Extracting Functional and Non-functional Contracts from Java Classes and Enterprise Java Beans, *Proceedings of the Workshop on Architecting Dependable Systems,* Florence, Italy, 282-286.

Milanovic, N., (2005a), Contract-based Web service Composition Framework with Correctness Guarantees, *Proceedings of the 2nd International Service Availability Symposium,* Berlin, Germany, 46-59.

Milanovic, N., Malek, M. (2005b), Architectural Support for Automatic Service Composition, *Proceedings of the International Service Computing Conference*, Orlando, US, 133-140.

Narayanan, S., McIlraith, S. (2002), Simulation, Verification and Automated Composition of Web Services, *Proceedings of the 11th International World Wide Web Conference*, Honolulu, Hawaii, USA, 77-88.

Nilsson, N.J. (1980), *Principles of Artificial Intelligence,* Tioga, Palo Alto.

Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K. (2002), Importing the Semantic Web in UDDI, *Proceedings of Web Services, E-Business and Semantic Web Workshops*, Toronto, Canada, 225-236.

Papazoglou, M.P., Georgakopoulos, D. (2003), Service Oriented Computing, *Communications of the ACM*, 46(10), 25-28.

Rich, E. (1983), *Artifical Intelligence,* McGraw-Hill.

Schmidt, H., Poernomo, I., Reussner, R. (2001), Trust-by-Contract: Modelling, analysing and predicting behavior of software architectures, *Journal of Integrated Design and Process Science*, 5(3), 25-51.

Sirin, E., Hendler, J., Parsia, B. (2003), Semi-automatic Composition of Web Services using Semantic Descriptions, *Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS*, Angers, France, 17-24.

Tartanoglu F., Issarny, V., Romanovsky, A., Levy, N. (2003), Coordinated Forward Error Recovery for Composite Web Services, *Proceedings of the 22nd International Symposium on Reliable Dependable Systems,* Florence, Italy, 167-176.

Verma, K., Sivashanmugam, K., Sheth, A., Patil, A., Oundhakar, S., Miller, J. (2004), METEOR-S WSDI: A Scalable Infrastructure of Registries for Semantic Publication and Discovery of Web Services, *Journal of Information Technology and Management*, 17-39.

Vinoski S. (2004), WS-Nonexistent Standards, *IEEE Internet Computing,* 8(6), 25-28.

Vogels, W. (2003), Web Services are not Distributed Objects: Common Misconceptions about the Fundamentals of Web Service Technology, *IEEE Internet Computing,* November/December, 59-66.

Webber, J., Parastatidis S. (2003), Demystifying Service-Oriented Architectures, *Web Services Journal*, 3(11), *http://webservices.sys-con.com/read/39908.htm*, accessed in May 2005.

West, D.B. (1996), *Introduction to Graph Theory,* Prentice Hall.

Yang, J., Papazoglou, M.P. (2000), Interoperation support for electronic business, *Communications of the ACM*, 43(6), 39-47.

Yang, J. (2003), Web Service Componentization, *Communications of the ACM*, 46(10), 35-40.

Yang, J., Papazoglou, M.P. (2002), Web Component: A Substrate for Web Service Reuse and Composition, *Proceedings of the 14th Conference. on Advanced Information Systems and Engineering*, Toronto, Canada, 21-36.

Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalaganam, J., Chang, H. (2004), QoS-Aware Middleware for Web Services Composition, *IEEE Transactions on Software Engineering*, 30(5), 311-327.

Zhang, J., Chang, C.K., Chung, J.Y., Kim, S.W. (2004), S-Net: A Petri-net Based Specification Model for Web Services, *Proceedings of IEEE International Conference on Web Services,* San Diego, CA, USA, 420-427.

Zhang, J. (2005), Trustworthy Web Services: Actions for Now, *IEEE IT Professional*, 7(1), 32-36.

ABOUT THE AUTHORS

Nikola Milanovic is is a research fellow and a PhD candidate at the Institute for Informatics, Humboldt University, Berlin. His research interests include component- and service-based environments, service composition, ubiquitous computing, ad-hoc networks, and wireless communication. Milanovic received a Dipl. Ing. In electrical engineering from the University of Belgrade. Contact him at milanovi@informatik.hu-berlin.de

Miroslaw Malek is a professor and chair of computer architecture and communication at Humboldt University, Berlin. His research focuses on high-performance responsive computing, including parallel architectures, real-time systems, networks and fault tolerance. Malek received a PhD in computer science from the Technical University of Wroclaw, Poland. Contact him at malek@informatik.hu-berlin.de