Contract-based Web Service Composition Framework with Correctness Guarantees

Nikola Milanovic

Humboldt University, Berlin
milanovi@informatik.hu-berlin.de

Abstract. We present formal and practical foundations for Web service composition framework with composition correctness guarantees. We introduce contractual composition model based on two isomorphic description models: Contract Definition Language (XML) and abstract machines (formal notation). Composition operators (patterns) are used to perform composition which is then formally verified with respect to properties described in service contracts. We also describe Java-based implementation of the system, concentrated around Sun's Java Web Services Development Pack (JWSDP).

Indexed terms: Web services, composition, correctness, contracts

1 Introduction

Web services are emerging as a replacement and/or additional paradigm for the component-based software development. However, Web services aim much further to become not only a new Object Request Broker architecture, but a unifying paradigm for communication among heterogenous groups of software and hardware entities. Web service architecture has three layers: description and basic operations (publication, discovery, selection and binding), composite services (coordination, conformance, monitoring and quality of service) and managed services (certification, rating, liability). Unfortunately, only the bottom layer has been standardized (WSDL, UDDI and SOAP). We are seeking a solution for the second layer dealing with Web service composition. In the next section we discuss the ideas of existing approaches and then present formal foundations and implementation of a contract-based composition framework.

2 Related Work

The composition layer comprises four properties: coordination, conformance, monitoring and quality of service. Coordination determines which services participate in composition and how they exchange messages. Conformance establishes composition correctness, while monitoring basically deals with error and exception handling. Finally, quality of service offers metrics to compare different compositions with respect to nonfunctional properties.

We examined several approaches for Web service composition: Business Process Execution Language for Web Services (BPEL4WS) [4], Semantic Web (OWL-S) [5,10,16], Web component [18], π-calculus [11], Petri Nets [9], model checking [8] and finite state machines [2,3]. Our detailed survey of these solutions and how they relate to the four key composition properties can be found in [14].

The main problem with 'industrial' approaches (BPEL, OWL-S) is the lack of support for verifying composition correctness. Both approaches (and BPEL in particular) offer implementation languages that are simply too expressive for any kind of formal validation. On the other hand, there are other, more formal and abstract approaches (e.g., π-calculus or finite state machines). However, they are often difficult to apply in real-world scenarios, and some of them face serious scalability problems. Our intention is to provide formal and practical foundations for a contract-based composition approach with correctness guarantees.

3 Contracts and Abstract Machines

The concept of design by contract was first introduced in [12] to facilitate component reuse. A contract describes, in a standard way, what a component expects from its clients and what it delivers if those requirements are met. We propose to use contracts as non-functional (QoS) extension of WSDL description.

3.1 Contract Definition Language (CDL)

We assume that messages and port bindings are already defined, separate (WSDL) or as a part of the service contract. The contract itself must provide information that is related to non-functional aspects of the service execution. However it should not include implementation details but semantic information only: what a service will deliver and under which conditions it will execute correctly.

The root section of CDL schema (Figure 1a) comprises `organization`, `types`, `location`, `method` and `event` elements, as well as several basic attributes: uri, name, description, price, state information, version and port. The `organization` element is introduced to maintain backwards compatibility with Universal Description, Discovery and Integration (UDDI) directories. Therefore, every service must belong to an organization that publishes it. For each organization it is possible to define name, description (keywords), classification and primary contact. The `types` element describes complex types that a service supports. It is used when a service accepts or returns non-primitive types (e.g., object of come custom class), and clients should be able to construct/deconstruct them appropriately. The `location` element is introduced to support location-based services. It allows for definition of country, city, street address and GPS coordinates. The `event` element declares all events that a service supports. For each event, its native name is listed, along with a reference and a common environment exception it is mapped into (if available). Finally, the `method` element describes one or more service methods.

Inside the `method` element we can specify information about parameters, persistent resources, invocation, pre-conditions, post-conditions and invariants, events, assertions, classification and method location. For each parameter it is possible to define name, type, restriction and initialization. Furthermore, constants and sets (complex types) that a method understands can be listed. Invocation information is related to component creation and execution (synchronous or asynchronous). Pre-conditions, post-condition and invariants share the same structure (Figure 1b).

Pre-conditions are linked to exported methods and determine obligations of a client. A method is guaranteed to work correctly if and only if a client satisfies pre-condition. Post-condition describes what a method guarantees, if precondition holds. Invariants are properties that must hold before and after invocation of each exported method. They describe general, static properties of a service. The properties that can be described are: rendering, logging, security, dependability (transactions, replication, check-pointing, timeout and exceptions), performance and parameters.

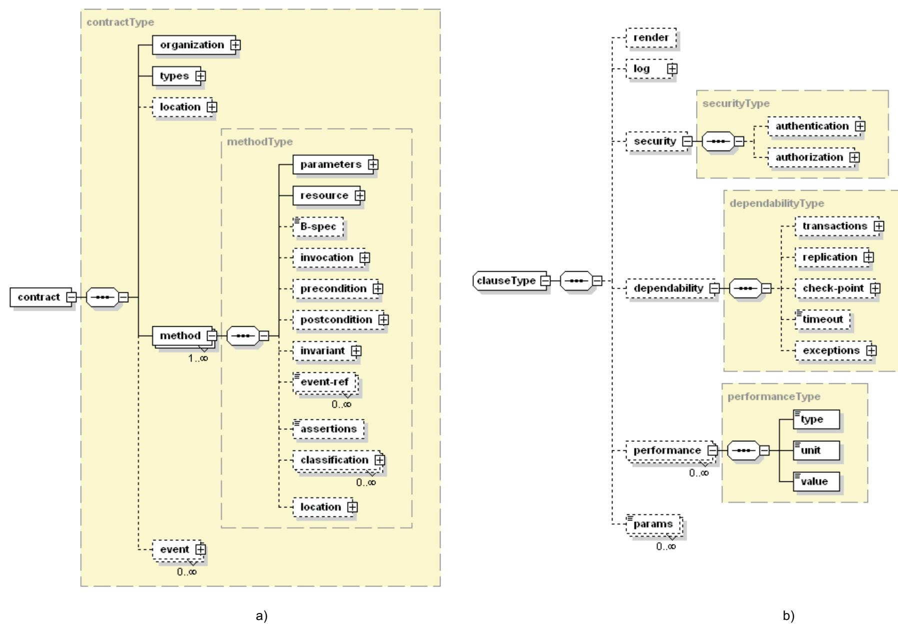

Fig. 1. The root contract structure

We shoved in [15] that it is possible, to some extent, to automate extraction of defined properties from Java classes and Enterprise Java Beans. We also identified the benefits that early contract inclusion has on the typical software lifecycle.

3.2 Modeling Services as Abstract Machines

Our main goal is to support not only reuse, but composition of Web services. CDL syntax offers a richer set of description primitives compared to WSDL, that can be used for specifying relevant non-functional properties. Verification of composition correctness, however, requires a formal approach.

Since contracts, as we presented them so far, are just plain text (XML) files, it would be very difficult, if indeed possible, to judge correctness of their composition. The first problem we would be faced with is actual definition of correctness. What does it mean for a contract to be correct, apart from satisfying XML requirements of being well-defined and well-formed? How can we judge whether two or more contracts are compatible or not conflicting with each other? How to define relations "compatible" and "conflicting"? Finally, how to perform actual composition when working on text files? In order to be able to answer these questions, we introduce a second form for expressing Web service contracts: abstract machine notation (AMN). We need the XML notation in order to transport contracts over a network (interoperability), while AMN serves the purpose of giving contract elements formal mathematical treatment.

Abstract machines are specified using Abstract Machine Notation. Details on AMN can be found in [1]. We give a short overview of AMN principles. An element, which can be a class, a component, or a Web service, is represented as an *abstract machine*. It is characterized by *statics* and *dynamics*. The statics corresponds to the definition of the state, while the dynamics corresponds to the operations:

```
MACHINE M(X,x)
CONSTRAINTS C
CONSTANTS c
SETS S; T={a,b}
PROPERTIES P
COMPLEX Cx
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, we will be using logical predicates relating the values of state variables just before the operation is invoked to the values just after the operation completes. The method we use is called substitution. Let P be a formula, x be a variable and E an expression, then the following denotes the formula obtained by replacing (substituting) all free occurrences of x in P by E:

$$[x := E]P$$

If S is a substitution, and I is a formula, we write that substitution S preserves I in a following way:

$$I \Longrightarrow [S]I$$

This expression says that if the invariant I holds then the substitution S is guaranteed to preserve the same invariant. We now introduce more complex substitutions. If P is a pre-condition and S is the substitution guarded by this pre-condition, then pre-conditioned substitution is $[P|S]R \Longleftrightarrow P \wedge [S]R$. This substitution can also be noted as `PRE P THEN S END`. Often we have to perform simultaneous substitution (multiple simple substitution) $[x, y := E, F]P \Longleftrightarrow [x := E][y := F]P$. It can be expressed also as `x:=E || y:=F`. Bounded choice substitution is used when we have to express a choice between two or more substitutions. It is denoted with $[S \square T]R \Longleftrightarrow [S]R \wedge [T]R$, and can also be expressed with `CHOICE S OR T END`. A substitution can be guarded by a predicate using implication $[P \Longrightarrow S]R \Longleftrightarrow (P \Longrightarrow [S]R)$. It can be denoted `IF P THEN S END`. Combination of bounded choice and guarded substitution is called conditional and is defined `IF P THEN S ELSE T END` $\Longleftrightarrow (P \Longrightarrow S)\square(\neg P \Longrightarrow T)$. A substitution that performs nothing for the target post-condition is empty substitution $[\mathtt{skip}]R \Longleftrightarrow R$. Any combination of previously defined substitutions can be performed simultanously (multiple generalized substitution) using the same notation, e.g., $S||(P|T) = P|(S||T)$, $S||(T \square U) = (S||T)\square(S||U)$, $S||(P \Longrightarrow T) = P \Longrightarrow (S||T)$. A situation where, if a predicate P holds, substitution S is iteratively executed, is denoted with `WHILE P DO S END` and called while substitution.

Mapping from CDL to AMN We have presented two notations for describing Web service contracts. During service exploitation there will be times when we will have to switch between them:

- When composing two services, their CDL descriptions will be transferred into abstract machines to allow for formal treatment of their properties.
- When new service is composed it is constructed by merging abstract machines of the constituent services, thus producing another abstract machine.

In order to make this service available to others and to be able to transport its specification over a network, abstract machine has to be transferred into CDL description.

It can be seen that transformation between CDL and AMN has to be bidirectional. However, since this transformation is linear, once we know how to do it one way, the other way is trivial. The mapping algorithm works as follows:

1. Machine name is constructed from `serviceName` attribute of the `contract` element. All other attributes of the `contract` element, as well as all child elements and attributes of the `organization` and `location` elements are mapped into `CONSTANTS` clause.
2. The `types` element is mapped into `COMPLEX` clause of abstract machine.
3. The `event` element is mapped into `CONSTANTS` clause.
4. For each `method` element, the following is performed:
   (a) State variables are built from properties in `invariant`, `precondition`, `post- condition` elements. To this are added all method parameters.
   (b) All sets defined in the `set` element are added to the `SET` clause.
   (c) Constants from the `constants` element are added to the `CONSTANTS` clause.
   (d) The `INVARIANT` clause is defined in term of conjoined predicates involving state variables, and is mapped directly from the `invariant` element. The `INVARIANT` clause must contain enough conjuncts to allow for the typing of all state variables.
   (e) The `PRE` clause is mapped directly from `precondition` element. State variables designating input parameters must have constraints (or types) defined in this clause.
   (f) Operation body (postcondition, or `THEN` clause) is constructed by conjoining substitutions from the `postcondition` element. All output parameters must have properties (or types) described in this clause.
   (g) All state variables that have `initialization` element defined, are added to the `INITIALIZATION` clause. Additionally, those that are defined as `"INOUT"` are added to the list of machine formal parameters.
   (h) The content of `assertion` element (if exists) is added to the `ASSERTION` clause in form of conjoined predicates.
   (i) The `resource`, `invocation` and `event-ref` elements are of no interest for composition semantics, and are thus not transferred into AMN. They are used for maintaining internal consistency of composition process.

4 Service Composition

We identify five basic patterns (operators) for service composition: sequence, selection, choice, parallel and loop. We show how to construct composite abstract machine clauses for each case and then discuss verification of composition correctness.

4.1 Sequential Composition

The sequence operator executes two (or more) services in an ordered sequence. Sequential composition of services A and B is denoted with:

$$C = A \rhd B$$

Outputs of the left operand (A) become inputs of the right operand (B). Therefore, operation $\rhd$ is not commutative. Figure 2 shows this composition.

The clauses of the resulting abstract machine are calculated in the following way:

- `SETS`, `CONSTANTS`, and `VARIABLES` clauses are concatenated
- `PROPERTIES`, `INVARIANT`, and `ASSERTION` clauses are conjuncted
- `OPERATIONS` clause is constructed by performing substitution of the left operand, then substituting input state variables of the right operand with the output state variables of the left operand, then performing substitution of the right operand, while conjuncting preconditions:

$$\texttt{OPERATION}\ \ output_B \leftarrow \texttt{C}(input_A)$$
$$\texttt{PRE}\ \ P_A \wedge P_B$$
$$\texttt{THEN}\ \ S_A\ ;\ input_B := output_A\ ;\ S_B\ \ \texttt{END}$$

  Here $output_B$ is a set of output state variables of the right operand, $input_A$ is a set of input state variables of the left operand, `C` is the name of a new (composite) operation, $input_B$ is a set of input state variables of the right operand, and $output_A$ is the set of output state variables of the left operand. Naturally, mapping from $input_A$ to $input_B$ has to be provided, unless it is clear that only one mapping is possible.
- `INITIALIZATION` clauses are concatenated, and multiple composed if needed

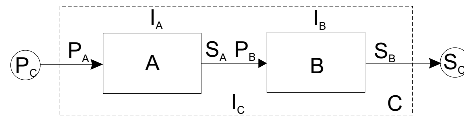

Fig. 2. Sequence Pattern

4.2 Parallel Composition

Parallel composition executes two (or more) services concurrently. Two subtypes of this pattern are allowed: parallel composition with and without communication. In the former case, concurrent services can communicate with each other, for the purpose of synchronization of some state variables. It can be used when a

certain decision has to be reached after parallel processing has been performed, e.g., choosing result of one service and discarding the other. Only operators of the relational algebra are allowed for the state variables upon which the synchronization is performed. Result aggregation of any kind is not allowed, since it would needlessly complicate composition pattern. If data aggregation needs to be performed, additional service should be created and then sequentially composed to the parallel composition. In the latter case (no communication), there is no communication / synchronization between concurrent services. Figure 3a shows parallel composition without communication and Figure 3b shows parallel composition with communication.

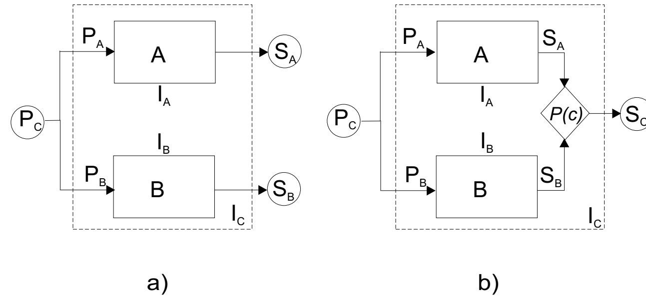

Fig. 3. Parallel Patterns

Parallel composition with communication is denoted with $\|_{P(c)}$, where c are state variables that are being used for synchronization and P is the predicate evaluated upon them, while parallel composition without communication is denoted with $\|$ only:

$$C = A \|_{P(c)} B$$

$$C = A \| B$$

The clauses of the composed abstract machine are constructed in the following way:

- `SETS`, `CONSTANTS`, and `VARIABLES` clauses are concatenated
- `PROPERTIES`, `INVARIANT`, and `ASSERTION` clauses are conjuncted
- `OPERATIONS` clause is constructed differently for composition with and without communication:
  - For parallel composition without communication, pre-conditions are conjuncted and substitutions are performed simultaneously (using multiple general substitution):

    $$\texttt{OPERATION}\ output_C \leftarrow \texttt{C}(input_C)$$
    $$\texttt{PRE}\ \ P_A \wedge P_B$$
    $$\texttt{THEN}\ \ S_A \| S_B\ \ \texttt{END}$$

    Here $output_C = output_A \cup output_B$ and $input_C = input_A \cup input_B$.

- For parallel composition with communication, pre-conditions are conjuncted and substitutions are performed simultaneously. Afterwards, predicate P is evaluated on a subset of state variables c, resulting in choice of output of only one service:

$$\texttt{OPERATION}\ output_C \leftarrow \texttt{C}(input_C)$$
$$\texttt{PRE}\ \ P_A \wedge P_B$$
$$\texttt{THEN}\ \ S_A \parallel S_B$$
$$\texttt{IF}\ P_c\ \texttt{THEN}\ output_C = output_A\ \texttt{ELSE}\ output_C = output_B\ \texttt{END}$$

– `INITIALIZATION` clauses are concatenated, and multiple composed if needed

4.3 Selection Composition

The selection pattern is similar to parallel composition: from the pool of available services, it selects one, without executing the others. Based on the external predicate $C(x)$, this operator selects one candidate service and executes it. Contrary to the parallel composition, selection is performed before execution, thus eliminating concurrent (parallel) execution. This operator can be simulated using parallel communication with communication by supplying $C(x)$ as a synchronization predicate. This is, however, not recommended since it results in an unnecessary loss of resources such as time (for execution and synchronization), network traffic and money (if service invocation or other resource usage is being charged). The composition is shown in Figure 4. This composition pattern is denoted with $\odot_{C(x)}$:

$$C = A \odot_{C(x)} B$$

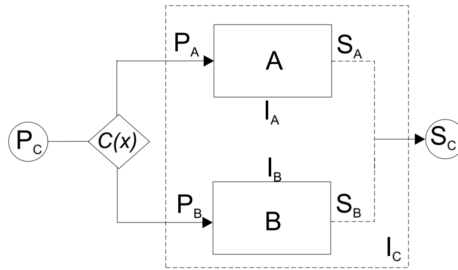

Fig. 4. Selection Pattern

The machine resulting from application of selection is constructed as follows:

– `SETS`, `CONSTANTS`, and `VARIABLES` clauses are concatenated
– `PROPERTIES`, `INVARIANT`, and `ASSERTION` clauses are disjuncted
– `OPERATIONS` clause is constructed by disjuncting pre-conditions, evaluating predicate $C(x)$ and selecting one substitution based on the evaluation result:

$$\texttt{OPERATION}\ output_C \leftarrow \texttt{C}(input_C)$$
$$\texttt{PRE}\ \ P_A \vee P_B$$
$$\texttt{THEN}\ \ \text{IF}\ C(x)$$
$$S_A\,;\ \ output_C = output_A\ \ \texttt{ELSE}$$
$$S_B\,;\ \ output_C = output_B\ \ \texttt{END}$$

– `INITIALIZATION` clauses are concatenated, and multiple composed if needed

4.4 Choice Composition

The choice pattern represents a composition that behaves as either of its constituent services. It is similar to parallel composition pattern with communication, but it is non-deterministic. It is furthermore restricted to compatible services in sense of input parameters and effects, because it is used when it is known in advance that some of the available services can perform the requested operation, without the need to know which one will do so in a particular instance. The most general example is sending the same request to many services and accepting the results from the first one that completes its execution. In general, this operator is used to express that *any* of the listed operands can fulfill the client's request. This composition pattern is denoted with $\Box$ and shown in Figure 5:

$$C = A \Box B$$

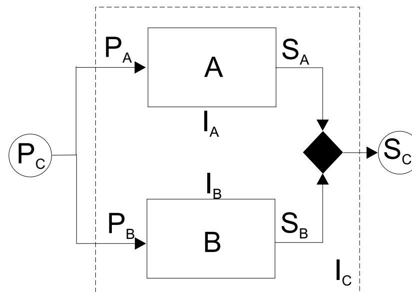

Fig. 5. Choice Pattern

The machine resulting from applying choice pattern is constructed as follows:

– `SETS`, `CONSTANTS`, and `VARIABLES` clauses are concatenated
– `PROPERTIES`, `INVARIANT`, and `ASSERTION` clauses are disjuncted
– `OPERATIONS` clause is constructed by disjoining preconditions and connecting substitutions by bounded choice substitution operator:

$$\texttt{OPERATION}\ output_C \leftarrow \texttt{C}(input_C)$$
$$\texttt{PRE}\ \ P_A \vee P_B$$
$$\texttt{THEN}\ S_A \square S_B\ \texttt{END}$$

Here $output_C = output_A \vee output_B$, which is implied in $S_A \square S_B$.
- INITIALIZATION clauses are concatenated, and multiple composed if needed

4.5 Looping

Looping pattern supports execution of the same service repeatedly, until a certain condition is fulfilled. Based on the condition controlling the loop, unary and binary loop are defined:

$$C = \circlearrowleft_{P(e)} A(e)$$

$$C = W(e) \circlearrowleft_{P(e)} A$$

In both cases, looping is controlled by predicate P evaluated on the variable e. Service is executed until $P(e)$ becomes false. In the unary pattern, e is a state variable of service A, and is changed in every iteration by the execution of A. Therefore, service A controls the loop exit condition. Since this is the loop with the condition on top (exit condition is evaluated prior to execution), variable e must be in the INITIALIZATION clause to enable the first loop iteration. In the binary pattern, there is another service W that controls $P(e)$. In this case service A is not allowed to influence the loop exit condition. Here, service W is executed prior to A and will set value of e, which therefore does not have to be initialized. Unary (a) and binary (b) pattern are shown in Figure 6.

The composite machine is constructed as follows for the unary pattern:

- The clauses SETS, CONSTANTS, VARIABLES, PROPERTIES, INVARIANT, ASSERTION, and INITIALIZATION are kept unchanged. Variable controlling loop exit (e) must appear in the INITIALIZATION clause.
- Operation body is constructed by enclosing original substitution in a WHILE DO block, controlled by $P(e)$:

$$\texttt{OPERATION}\ output_C \leftarrow \texttt{C}(input_C)$$
$$\texttt{PRE}\ P_A$$
$$\texttt{THEN WHILE}\ P(e)\ \ S_A(e)\ \ \texttt{END}$$
$$\texttt{END}$$

Here $ouptut_C = output_A$ and $input_C = input_A$.

For the binary pattern, another service W controls exit variable:

- SETS, CONSTANTS, and VARIABLES clauses are concatenated
- PROPERTIES, INVARIANT, and ASSERTION clauses are conjuncted
- Operation body is constructed by conjoining preconditions, and enclosing both substitutions inside a WHILE DO:

$$
\begin{array}{c}
\texttt{OPERATION}\ output_C \leftarrow \texttt{C}(input_C) \\
\texttt{PRE}\ \ P_A \wedge P_W \\
\texttt{THEN}\ S_W(e) \\
\texttt{WHILE}\ P_e \\
S_A; S_W(e)\ \texttt{END} \\
\texttt{END}
\end{array}
$$

– `INITIALIZATION` clauses are concatenated, and multiple composed if needed

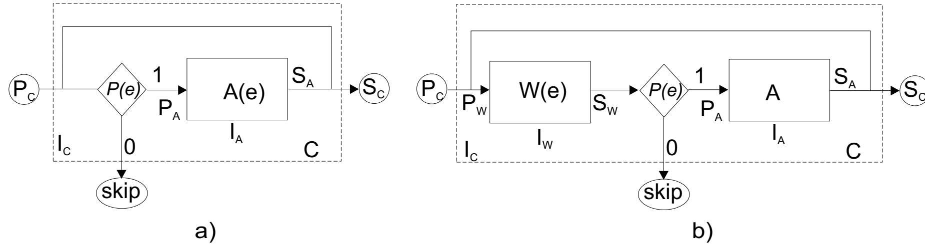

Fig. 6. Loop Pattern

4.6 Correctness Verification

Once an operator has been applied, composition result has to be verified. The whole composition process then proceeds as follows:

– Merging of two (or more) abstract machines using composition operator.
– Type checking of the resulting abstract machine.
– Proving correctness of the resulting abstract machine
– Establishing correct termination of the resulting abstract machine

Type Checking Suppose that we have an expression E and a set s such that $E \in s$. Suppose further that there exists a set t such that $s \subseteq t$. Then it follows that $E \in t$. We can continue by including t in a larger set u, and then E will also belong to u. The purpose of type-checking of the abstract machine is to provide an upper limit for such set containment for all predicates. This upper limit is called a super-set of s and is at the same time the *type* of E. The function `check` is introduced like `ENV` $\vdash$ `check` (P), and for the predicate P means that within the environment (antecedent) `ENV` predicate P type-checks. Referring to the abstract machine from Section 3.2, the type checking consists of the following requirements:

– X, x, S, T, a, b, c, v are all distinct
– Operation names of $O_1 ... O_n$ are all distinct

- S, T, a, b, c, v are not-free in C
- v, X, x are not-free in P
- $X, S, T, a \in T, b \in T \vdash$ `check` $(\forall x(C \Rightarrow \forall c(P \Rightarrow \forall v(I \wedge J \Rightarrow U \wedge O))))$

The last expression means that first universally quantified scalar parameters and their constraints are checked, then universally quantified constants and their properties, then universally quantified variables and their invariant, and finally initialization and operations.

Proof Obligation After type checking has been performed, the resulting machine must be proved correct. The purpose of this is to establish the following:

- Composite initialization must establish composite invariant
- Composite assertion must be deducible from composite properties and invariant
- Composite operation must establish composite invariant

Formally, and again referring to the machine from Section 3.2 we can write it:

$$C \wedge P \Rightarrow [U]I$$

$$C \wedge P \wedge I \Rightarrow J$$

$$C \wedge P \wedge I \wedge J \wedge Q \Rightarrow [V]I$$

Correct Termination After proving machine correct, we have to see whether it will terminate correctly and whether it is feasible. For a given substitution S the construct $trm(S)$ denotes the predicate that holds when substitution S terminates, that is, establishes its post-condition. By requiring that all operations terminate, we ensure elimination of deadlocks. Another construct is defined, $abt(S)$ which denotes aborted substitution, that is, substitution that does not establish anything. Therefore it can be said that $abt(S) \equiv \neg[S]R$ for any predicate R, and accordingly $trm(S) \equiv \neg\ abt(S)$. We define correct termination as $trm(S) \iff [S](x = x)$. Correct termination for substitutions is established in the following way: $trm(P|S) \iff P \wedge trm(S)$, $trm(P \square T) \iff trm(S) \wedge trm(T)$, $trm(P \Rightarrow S) \iff P \Rightarrow trm(S)$.

We also check whether the composite operation is feasible, with respect to the guarded substitution. The feasible operation will establish one, or none post-condition. Non-feasible operation, on the other side, will be able to establish any post-condition. We define feasibility as $fis(S) \iff \neg[S](x \neq x)$. The feasibility of the standard substitutions is calculated in the following way: $fis(P|S) \iff P \Rightarrow fis(S)$, $fis(P \square T) \iff fis(S) \vee fis(T)$, $fis(P \Rightarrow S) \iff P \wedge fis(S)$.

Correct termination for loop operator is outside the scope of this general paper. Suffice to say that we introduce two separate elements in the loop body: invariant and variant (exit condition). We currently limit the evaluation of the exit condition to natural number and observe its monotonicity:

$$trm(\texttt{WHILE P DO S INVARIANT I VARIANT V END}) \iff$$
$$(\forall x \cdot (I \Rightarrow V \in N)) \wedge (\forall x \cdot (I \wedge P \Rightarrow [n := V][S](V < n)))$$

5 Implementation

In previous sections we formed formal foundations needed for developing Web service composition framework. Now we address some implementation issues, namely system organization, communication, service directory and state management. Composition engine comprises four main parts: client application, basic administrative services, database for storing CDL contracts, one or more application servers/containers with deployed services. Service descriptions are stored in a relational database. Client applications access basic functionalities of the engine via Web service middle layer. This middle layer connects to the underlying database, as well as to application servers in which target services are deployed. Clients cannot access database or application servers directly. Therefore, most of the engine's tasks are accomplished in the middle layer. The engine is implemented using Java and Java-related technologies.

5.1 Client and Basic Administrative Services

Client part is realized as Swing application connected to the middle Web service layer that provides administrative functions and operations. Middle layer offers the following operations: publishing new service to directory, modifying and deleting existing service from directory, searching for services, composing new services using existing ones, invoking single or composed services.

In order to achieve these tasks, middle layer communicates with underlying relational database (directory) and application servers hosting target Web services that users want to invoke and/or compose. Since entire application is Web service-based, the communication is realized using Sun's Java Web Services Developer Pack (Sun JWSDP) [13].

We found two technologies provided within JWSDP very useful: Java Architecture for XML Binding (JAXB) and Java API for XML-based Remote Procedure Calls (JAX-RPC). Since CDL schema is very large, encompassing more than 50 complex entities, we need a powerful yet flexible mechanism of translating XML document into Java object representation. JAXB offers a complete solution for transferring XML content into Java object representation and vice versa. JAXB operation is based on three actions: binding XML schema to Java content classes, unmarshalling XML document into content classes and marshalling content classes into XML document. At the beginning CDL schema is compiled with JAXB binding compiler. This action produces a set of Java content classes that reflect the contract structure. The process of unmarshalling takes service contract as input and produces set of instantiated Java content classes populated with data parsed from XML document. During unmarshalling contract is optionally validated with respect to schema. After this step we have in-memory representation of contract. The middle layer uses JAXB to publish and modify service contracts. When a contract is published, Java content classes are persisted in database tables. When a contract needs to be changed, tables are updated, and depending on the origin of update, XML representation is synchronized (via JAXB marshalling).

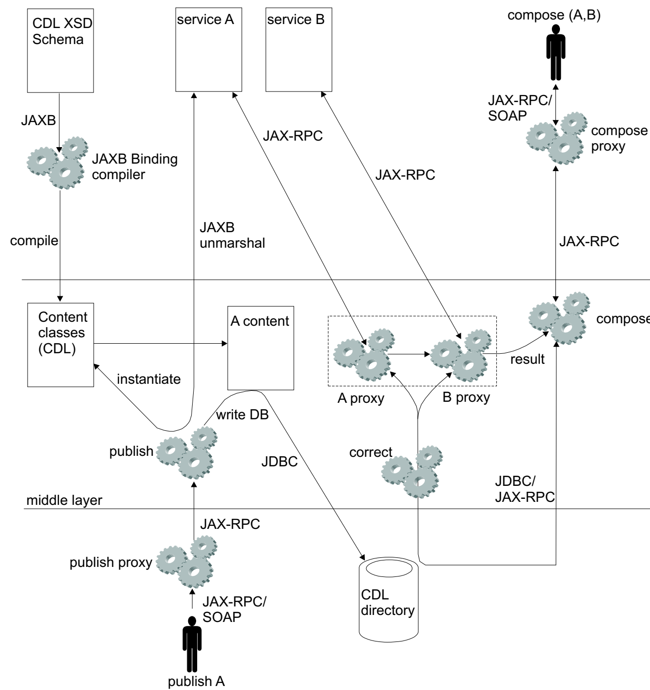

Fig. 7. JWSDP Runtime

JAX-RPC is used for communication with Web services. Since middle layer is also realized as a Web service, clients use JAX-RPC to invoke basic functions of the system, and middle layer uses JAX-RPC to invoke single or composite services. In JAX-RPC a remote procedure call is represented by an XML-based protocol, such as SOAP. Complex SOAP messages and their structure (envelope, encoding rules, conventions for RP calls and responses) are hidden by JAX-RPC API. This API supports development of server side (Web service implementation) and client side (Web service invocation) infrastructure. On the server side, remote procedures (Web methods) are specified by writing Java interface and one or more classes that implement that interface. On the client side, a proxy object is created that represents Web service. All Web methods are invoked on a proxy. Therefore, it is not necessary to generate or parse SOAP messages. The JAX-RPC runtime converts API calls and responses to and from SOAP messages.

The functioning of JAXB and JAX-RPC runtime is shown on Figure 7. It shows two typical use cases: publishing a new service to directory and composition of two services that are already in directory. Prior to any client calls, XSD schema describing Contract Definition Language is compiled with JAXB binding compiler, and content classes are stored in the middle layer. Client publishes

new service by issuing SOAP or JAX-RPC call to the Publish Proxy, which delegates the call to the Publish service in the middle layer. A service that is to be published is located, and its CDL description is unmarshalled into precompiled content classes produced by JAXB compiler. Finally, write to underlying database is performed via JDBC which completes the publish process.

Composition is initiated by sending SOAP/JAX-RPC request to Compose Proxy, and the call is then delegated to Compose service in the middle layer via JAX-RPC. It processes composition request, retrieves partner service information from database using JDBC, verifies composition correctness by calculating function *correct*, and constructs required dynamic proxies that represent partner services using Dynamic Invocation Interface. Each proxy then connects to its implementation and middle layer coordinates message passing in a manner that depends on the composition pattern used. Result is returned to the client via Compose Proxy.

5.2 Service Directory and State Management

Service directory is realized as a relational database. There are several reasons why we use a relational database instead of a native XML database. Current XML databases still do not support W3C XML schema which we use to define CDL. Using native XML database could therefore lead to low data integrity. Furthermore, XML databases use XPath as query language, and it offers no support for grouping, sorting, cross document joins, and data types. Since service directory requires complex queries, this is a very limiting implementation factor. Still another downside is that updating requires retrieving an XML document, modifying it using own API and then returning it to database.

Database was designed to take full advantage of rich descriptive options offered by CDL in order to overcome UDDI limitations. The underlying database schema allows for searching for services directly, using any combination of properties defined in CDL. That means that it is possible to search for services by locations, methods they offer, classifications, and all other properties defined in their pre-conditions, post-conditions and invariants. One example query would be to find all services in the 1 km radius that accept postscript documents and print them in color with 1200 dpi resolution, free of charge if we can supply a security credential of certain type.

Up to now we have been talking about modeling Web services using abstract machines comprising state variables. It is obvious that we have implicitly assumed that some services can maintain their state between calls. However, Web services are stateless and we need to introduce state management mechanism.

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 [17], 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 [6]. Our position is that for the purpose of complex service interactions the latter view is correct. We identify two possible ways to associate a state with a Web service:

- A conversational service implements a series of operations where result of one operation depends on the prior operations of the same or other services. The state is maintained in the logical sequence of messages.
- A service that acts upon one or more persistent resources (database, file), creating, modifying or deleting it based on the messages it sends or receives.

Since conversational state can be implemented using WS-Coordination and WS-Context specifications, we concentrate on the interaction with stateful resources. Furthermore, we consider only relational database as a provider of background persistent resource. Interaction with persistent resource is described within the `resource` element of the CDL. Resource is identified by its name, uri, and resource manager (in our case, relational database driver). For each method acting upon a resource, one of the following actions can be defined: `CREATE`, `READ`, `MODIFY`, `DELETE`. Methods that create resources return resource identifier, while methods that read, modify and delete resources require resource identifier. Finally, resource property defines one or more CDL elements (state variables) that are bound to the underlying resource.

Our efforts in providing state management are compatible with the recent WS-Resource proposal [7], with the main difference being that WS-Resource supports broader range of persistent resources identified using WS-Addressing.

6 Conclusion

If Web services are to become the dominant architecture of future distributed systems, after connectivity is established (standardized) at least two more issues need to be supported at the architectural level: trustworthiness and automatic business to business (B2B) interactions. We try to address both in our proposed framework. We defined trustworthiness not only as security, but as an aggregation of properties (including but not limited to security) that composition process must guarantee. Therefore we adopted *correctness* as a term that best describes a "trustworthy" or a "trusted" composite Web service.

However, composition has still to be performed manually by application developer, albeit much easier and more flexible than in case of the other existing approaches as it now consists only of selecting appropriate services and applying composition operator (pattern). The proposed framework offers possibility of true automatic B2B interactions. Formal treatment of composition process enables use of various search strategies for the purpose of efficient allocation and verification in the process of automatic composition.

References

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

2. D. Berardi, D. Calvanese, D. G. Giuseppe, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Proc. of the 1st Int. Conf. on Service Oriented Computing (ICSOC 2003)*, 2003.
3. T. Bultan, X. Fu, R. Hull, and J. Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In *Proc. of WWW2003*, 2003.
4. F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana. The Next Step in Web Services. *Communications of the ACM*, October 2003.
5. A. Ankolekar et al. DAML-S: Web Service Description for the Semantic Web. In *Proceedings of the International Semantic Web Conference (ISWC)*, 2002.
6. I. Foster et al. Modeling stateful resources with web services. *http://www.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf*, 2004.
7. K. Czajkowski et al. The WS-Resource Framework. *http://www.globus.org/wsrf/specs/ws-wsrf.pdf*, 2004.
8. X. Fu, T. Bultan, and J. Su. Formal Verification of E-Services and Workflows. In *Proceedings of Workshop on "Web Services, e-Business, and the Semantic Web (WES): Foundations, Models, Architecture, Engineering and Applications"*, 2002.
9. R. Hamadi and B. Benatallah. A Petri Net-based model for Web Service Composition. In *Proceedings of the Fourteenth Australasian database conference on Database technologies*, 2003.
10. S. McIlraith and T.C. Son. Adapting Golog for Composition of Semantic Web Services. In *Proceedings of the International Conference on the Principles of Knowledge Representation and Reasoning (KRR'02)*, 2002.
11. L.G. Meredith and S. Bjorg. Contracts and Types. *Communications of the ACM*, 46, No. 10, pp 41-47, October 2003.
12. B. Meyer. Applying Design by Contract. *IEEE Computer 25(10)*, 1992.
13. Sun Microsystems. The Java Web Services Developer Pack. *http://java.sun.com/webservices/downloads/webservicespack.html*, 2004.
14. N. Milanovic and M. Malek. Current Solutions for Web Service Composition. *IEEE Internet Computing*, November/December 2004.
15. N. Milanovic and M. Malek. Extracting Functional and Non-functional Contracts From Java Classes and Enterprise Java Beans. In *Proceedings of the Workshop on Architecting Dependable Systems (WADS 2004)*, Florence, Italy, 2004.
16. S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services . In *Proceedings of the Int. WWW02 Conference*, 2002.
17. W. Vogels. Web Services are not Distributed Objects: Common Misconceptions about the Fundamentals of Web Service Technology. *IEEE Internet Computing*, November/December 2003.
18. J. Yang and M. P. Papazoglou. Web Component: A Substrate for Web Service Reuse and Composition. In *Proceedings of 14th Conference on Advanced Information Systems Engineering (CAiSE02)*, Toronto, 2002.