$\mathcal{ESC}$: A Tool for Automatic Composition of *e*-Services based on Logics of Programs

Daniela Berardi[1], Diego Calvanese[2], Giuseppe De Giacomo[1]
Maurizio Lenzerini[1], and Massimo Mecella[2]

[1] Università di Roma "La Sapienza"
Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
Via Salaria 113, 00198 Roma, Italy
{berardi,degiacomo,lenzerini,mecella}@dis.uniroma1.it
[2] Libera Università di Bolzano/Bozen
Facoltà di Scienze e Tecnologie Informatiche
Piazza Domenicani 3, 39100 Bolzano/Bozen, Italy
calvanese@inf.unibz.it

Abstract. In this paper we discuss an effective technique for automatic service composition, and we present the prototype that implements it. In particular, we characterize the behavior of a service in terms of a finite state machine. In this setting we discuss a technique based on satisfiability in a variant of Propositional Dynamic Logic that solves the automatic composition problem. Specifically, given *(i)* a client specification of his *desired service*, i.e., the service he would like to interact with, and *(ii)* a set of available services, our technique synthesizes the orchestration schema of a composite service that uses only the available services and fully realizes the client specification. The developed system is an open source software tool, called $\mathcal{ESC}$ (E-Service Composer), that implements our composition technique starting from services each described in terms of a WSDL specification and a behavioral description expressed in any language that can capture finite state machines.

1 Introduction

One of the basic aspects of the *Service Oriented Computing*, and of the Extended Service Oriented Architecture proposed by [18], is the composition of services. Basically, service composition addresses the situation when a client request cannot be satisfied by any (single) available service, but a *composite* service, obtained by combining "parts of" available *component* services, might be used [17, 10, 6].

Service composition involves two different issues. The first, referred to as *composition synthesis* is concerned with synthesizing such a new composite service, thus producing a specification of how to coordinate the component services to obtain the composite service. Such a specification can be obtained either *automatically*, i.e., using a tool that implements a composition algorithm, or *manually* by a human, possibly with the help of CASE-like . In what follows, we will refer to such a specification of the composite service as *orchestration schema*, according to [1]. The second issue, referred to as *orchestration*, is concerned with coordinating, during the composite service execution, the various component services according to the orchestration schema previously synthesized, and also monitoring control and data flow among the involved services, in

order to guarantee the correct execution of the composite service. Such activities are performed by the *orchestration engine* [1].

It has been argued [18, 1], that in order to be able to automatically synthesize a composite service starting from available ones, the available services should provide rich service descriptions, consisting of *(i)* interface, *(ii)* capabilities, *(iii)* behavior, and *(iv)* quality. In particular, the service interface description publishes the service signature[3], while the service capability description states the conceptual purpose and expected results of the service. The (expected) behavior of a service during its execution is described by its service behavior description. Finally, the Quality of Service (QoS) description publishes important functional and non-functional service quality attributes [4].

Several works in the service literature (refer to [16] for a survey) address the problem of service composition in a framework where services are represented in terms of their (static) interface. The aim of this work is twofold: first, we discuss an effective technique for automatic service composition, when services are characterized in terms of their behavior, and then we present the prototype design and development of an open source software tool implementing our composition technique, namely $\mathcal{ESC}$ (E-Service Composer)[5].

In [7, 6] we have devised a framework where services export their behavior as finite state machines, and in [6] we have developed an algorithm that, given *(i)* a client specification of his *desired service*, i.e., the service he would like to interact with, and *(ii)* a set of available services, synthesizes the orchestration schema of a composite service that uses only the available services and fully realizes the client specification. We have also studied the computational complexity of our algorithm: it runs in exponential time with respect to the size of the input state machines. Observe that, it is easy to come up with examples in which the orchestration schema is exponential in the size of the component services. However, practical experimentation conducted over some real cases with the prototype, shows that, given the complexity of the behavior of real services, the tool can effectively builds a composite service.

Although some papers have already been published that discuss either behavioral models of services ([16]), or propose algorithms for computing composition (e.g., [17, 10, 19]), to the best of our knowledge, our research is the first one tackling *simultaneously* the following issues: *(i)* presenting a formal framework where the problem of service composition is precisely characterized, *(ii)* providing techniques for automatically computing service composition in the case of services represented as finite state machines and, *(iii)* implementing our composition technique into an effective software tool.

The rest of the paper is organized as follows. In Section 2 we discuss our framework for services that export their behavior. In Section 3 we present our technique for automatic service composition. In Section 4 we describe our tool. Finally, in Section 5 we draw conclusions and discuss future work.

---

[3] E.g., as a WSDL file.

[4] E.g., service metering and cost, performance metrics (e.g., response time), security attributes, (transactional) integrity, reliability, scalability, availability, etc.

[5] cf. the PARIDE (Process-based frAmeworK for composItion and orchestration of Dynamic E-services) Open Source Project: `http://sourceforge.net/projects/paride/` that is the general framework in which we intend to release the various prototypes produced by our research.

2 General Framework

A service is a software artifact that interacts with its client and possibly other services in order to perform a specified task. A client can be either a human or a software application. When executed, a service performs a given task by executing certain actions *in coordination* with the client.

We characterize the exported behavior of a service by means of an *execution tree*. The nodes of such a tree represent the sequence of actions that have been performed so far by the service, while the successor nodes represent the actions that can be performed next at the current point of the computation. Observe that in such an execution tree, for each node we can have at most one successor node for each action. The root represents the initial state of the computation performed by the service, when no action have been executed yet. We label the nodes that correspond to completed execution of the service as "final", with the intended meaning that in these nodes the service can (legally) terminate.

We concentrate on services whose behavior can be represented using a *finite number of states*. We do not consider any specific representation formalism for representing such states (such as action languages, situation calculus, state-charts, etc.). Instead, we use directly deterministic finite state machines (i.e., deterministic and finite labeled transition systems) [6].

The alphabet of the FSM (i.e., of the symbol labeling transitions) is formed by the actions that the service can execute. Such actions are the abstractions of the effective input/output messages and operations offered by the service. As an example, consider a service that allows for searching and listening to mp3 files; in particular, the client may choose to search for a song by specifying either its author(s) or its title (action `search_by_author` and `search_by_title`, respectively). Then the client selects and listens to a song (action `listen`). Finally, the client chooses whether to perform those actions again. The WSDL interface of this service and the finite state machine describing its behavior are reported in Figure 1[7].

To represent the set of services available to a client, we introduce the notion of *community* $\mathcal{C}$ of services, which is a (finite) set of services that share a common (finite) set of actions Σ, also called the *alphabet* of the community. Hence, to join a community, a service needs to export its behavior in terms of the alphabet of the community. From a more practical point of view, a community can be seen as the set of all services whose descriptions are stored in a repository. We assume that all such service descriptions have been produced on the basis of a common and agreed upon reference alphabet/semantics.

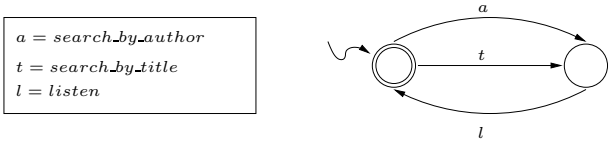
Given a service A_i, the execution tree $T(A_i)$ *generated* by A_i is the execution tree containing one node for each sequence of actions obtained by following (in any possible way) the transitions of A_i, and annotating as final those nodes corresponding to the traversal of final states.

[6] FSMs can capture an interesting class of services, that are able to carry on rather complex interactions with their clients, performing useful tasks. Moreover, several papers in the service literature adopt FSMs as the basic model of exported behavior of services [16, 1]. Also, FSMs constitute the core of statecharts, which are one of the main components of UML and are becoming a widely used formalism for specifying the dynamic behavior of entities.

[7] Final nodes are represented by two concentric circles.

```
<definitions ...
   xmlns:y="http://new.thiswebservice.namespace"
   targetNamespace="http://new.thiswebservice.namespace">

   <!-- Types -->
   <types>
      <element name="ListOfSong_Type">
         <complexType>
            <sequence>
               <element minOccurs="1"
                        maxOccurs="unbound"
                        name="SongTitle"
                        type="xs:string"/>
            </sequence>
         </complexType>
      </element>
   </types>

   <!-- Messages -->
   <message name="search_by_title_request">
      <part name="containedInTitle" type="xs:string"/>
   </message>
   <message name="search_by_title_response">
      <part name="matchingSongs" xsi:type="ListOfSong_Type"/>
   </message>
   <message name="search_by_author_request">
      <part name="authorName" type="xs:string"/>
   </message>
   <message name="search_by_author_response">
      <part name="matchingSongs" xsi:type="ListOfSong_Type"/>
   </message>
   <message name="listen_request">
      <part name="selectedSong" type="xs:string"/>
   </message>
   <message name="listen_response">
      <part name="MP3fileURL" type="xs:string"/>
   </message>

   <!-- Service and Operations -->
   <portType name="MP3ServiceType">
      <operation name="search_by_title">
         <input message="y:search_by_title_request"/>
         <output message="y:search_by_title_response"/>
      </operation>
      <operation name="search_by_author">
         <input message="y:search_by_author_request"/>
         <output message="y:search_by_author_response"/>
      </operation>
      <operation name="listen">
         <input message="y:listen_request"/>
         <output message="y:listen_response"/>
      </operation>
   </portType>

</definitions>
```

(a) WSDL



(b) FSM

Fig. 1. The MP3 service

When a client requests a certain service from a service community, there may be no service in the community that can deliver it directly. However, it may be possible to suitably orchestrate (i.e., coordinate the execution of) the services of the community so as to provide the client with his desired service. In other words, there may be an orchestration that coordinates the services in the community, and that realizes the client desired service.

Formally, let the community $\mathcal{C}$ be formed by n services $A_1, \ldots, A_n$. An orchestration schema O of the services in $\mathcal{C}$ can be formalized as an *orchestration tree* $T(O)$:

- The root ε of the tree represents the fact that no action has been executed yet.
- Each node x in the orchestration tree $T(O)$ represents the history up to now, i.e., the sequence of actions as orchestrated so far.
- For every action a belonging to the alphabet Σ of the community and $I \in [1..n]$ [8] ($1, \ldots, n$ stand for the services $A_1, \ldots, A_n$, respectively), $T(O)$ contains at most one successor node $x{\cdot}(a, I)$.
- Some nodes of the orchestration tree are annotated as *final*: when a node is final, and only then, the orchestration can be legally stopped.
- We call a pair $(x, x{\cdot}(a, I))$ an *edge* of the tree. Each edge $(x, x{\cdot}(a, I))$ of $T(O)$ is labeled by a pair (a, I), where a is the orchestrated action, $I \in [1..n]$ denotes the nonempty set of services in $\mathcal{C}$ that execute the action.
  As an example, the label $(a, \{1, 3\})$ means that the action a requested by the client is executed by, more precisely delegated to, the services A_1 and A_3.

Given an orchestration tree $T(O)$ and a path p in $T(O)$ starting from the root, we call the *projection* of p on a service A_i the path obtained from p by removing each edge whose label (a, I) is such that $i \notin \{I\}$, and collapsing start and end node of each removed edge.

We say that an orchestration O is *coherent* with a community $\mathcal{C}$ if for each path p in $T(O)$ from the root to a node x and for each service A_i of $\mathcal{C}$, the projection of p on A_i is a path in the execution tree $T(A_i)$ from the root to some node y, and moreover, if x is final in $T(O)$, then y is final in $T(A_i)$.

In our framework, we define *client specification* a specification of the orchestration tree according to the client desired service. Of the orchestration tree, the client *only* specifies the actions he would like to be executed by the desired service. The client specification can be *realized* by an orchestration tree only if it is possible to find a suitable labeling for each action with a non empty set I of (identifiers of) services that can execute it. In this work, we consider specifications that can be expressed using a finite number of states, i.e., as FSMs.

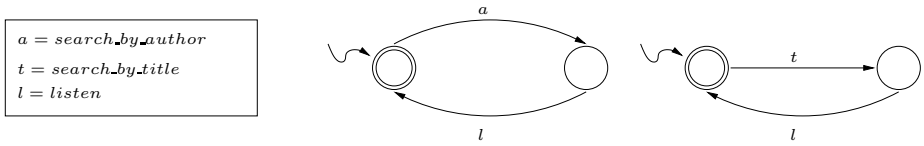
Given a community $\mathcal{C}$ of services, and a client specification $\mathcal{A}_0$, the problem of *composition existence* is the problem of checking whether there exists an orchestration schema that is coherent with $\mathcal{C}$ and that realizes $\mathcal{A}_0$. The problem of *composition synthesis* is the problem of synthesizing an orchestration schema that is coherent with $\mathcal{C}$ and that realizes $\mathcal{A}_0$.

Since we are considering services that have a finite number of states, we would like also to have an orchestration schema that can be represented with a finite number of

[8] We use $[i..j]$ to denote the set $\{i, \ldots, j\}$.

states, i.e., as a Mealy FSM (MFSM), in which the output alphabet is used to denote which services execute which action.

As an example, consider the case in which the service community is constituted by two services, A_1 and A_2, whose behaviors/FSMs are shown in Figure 2. A_1 allows for searching for a song by specifying its author(s) and for listening to the song selected by the client; then, it allows for executing these actions again. A_2 behaves like A_1, but it allows for retrieving a song by specifying its title.



(a) FSM for A_1 (b) FSM for A_2

Fig. 2. Services in the community.

If the client specification is the FSM shown in Figure 1(b), then a composition exists, and its orchestration schema is the Mealy FSM shown in Figure 3, in which all the actions requested by the client are delegated to services of the community. In particular, the execution of `search_by_author` action and its subsequent `listen` action are delegated to A_1, and the execution of `search_by_title` action and its subsequent `listen` action to A_2.

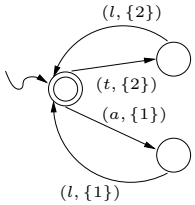


Fig. 3. Composition of A_0 wrt A_1 and A_2

3 Automatic Service Composition

In the framework presented in the previous section, we are interested in knowing whether: *(i)* it is always possible to check the existence of a composition; *(ii)* if a composition exists, there exists an orchestration schema which is a finite state machine, i.e., a *finite state composition*; *(iii)* if a finite state composition exists, how to compute it. Our approach is based on reformulating the problem of service composition in terms of satisfiability of a suitable formula of Deterministic Propositional Dynamic Logic

(DPDL [14]), a well-known logic of programs developed to verify properties of program schemas. DPDL enjoys three properties of particular interest: *(i)* the *tree model property*, which says that every model of a formula can be unwound to a (possibly infinite) tree-shaped model; *(ii)* the *small model property*, which says that every satisfiable formula admits a finite model whose size is at most exponential in the size of the formula itself; *(iii)* the EXPTIME-completeness of satisfiability in DPDL.

We represent the FSMs of both the client specification A_0 and the services $A_1, \ldots, A_n$ of community $\mathcal{C}$, as a suitable DPDL formula Φ. Intuitively, for each service $A_i, i = 0 \ldots n$, involved in the composition, Φ encodes *(i)* its current state, and in particular whether A_i is in a final state, and *(ii)* the transitions that A_i can and cannot perform, and in particular which component service(s) performed a transition. Additionally, Φ captures the following constraints: *(i)* initially all services are in their initial state, *(ii)* at each step at least one of the component FSM has moved, *(iii)* when the desired service is in a final state also all component services must be in a final state.

The following results hold [6, 5]:

1. From the tree model property, the DPDL formula Φ is satisfiable if and only if there exists a composition of A_0 wrt $A_1, \ldots, A_n$.
2. From the small model property, if there exists a composition of A_0 wrt $A_1, \ldots, A_n$, then there exists one which is a MFSM of size which is at most exponential in the size of the schemas of A_0, $A_1, \ldots, A_n$.
3. From the EXPTIME-completeness of satisfiability in DPDL and from point 1 above, checking the existence of a service composition can be done in EXPTIME.

As an example, we can encode in a DPDL formula ϕ both the client specification shown in Figure 1(b) and the services in the community of Figure 2. Then we can use a DPDL tableaux algorithm to verify the satisfiability of ϕ. Such an algorithm returns a model that corresponds to the composition shown in Figure 3 (cf. [5]).

4 The Service Composition Tool $\mathcal{ESC}$

In this section we discuss the prototype tool $\mathcal{ESC}$ that we developed to compute automatic service composition in our framework.

Figure 4 shows the high level architecture for $\mathcal{ESC}$. We assume to have a repository of services, where each service is specified in terms of both its static interface, through a WSDL document, and its behavioral description, which can be expressed in any language that allows to express a finite state machine (e.g., Web Service Conversation Language [12], Web Service Transition Language [8], BPEL4WS [2], etc.). The repository implements the community of services and can be seen, therefore, as an advanced version of UDDI. The client specifies his desired service in terms of a WSDL document and of its behavioral description, again expressed using one of the language mentioned before[9]. Both the services in the repository and the client desired service are then abstracted into the corresponding FMS (`Abstraction Module`). The `Synthesis Engine` is the core module of $\mathcal{ESC}$. It takes in input such FSMs, processes them according to our composition technique and produces in output the

[9] Of course, we assume that the behavioral description of both the client specification and the services in the repository are expressed in the same language.

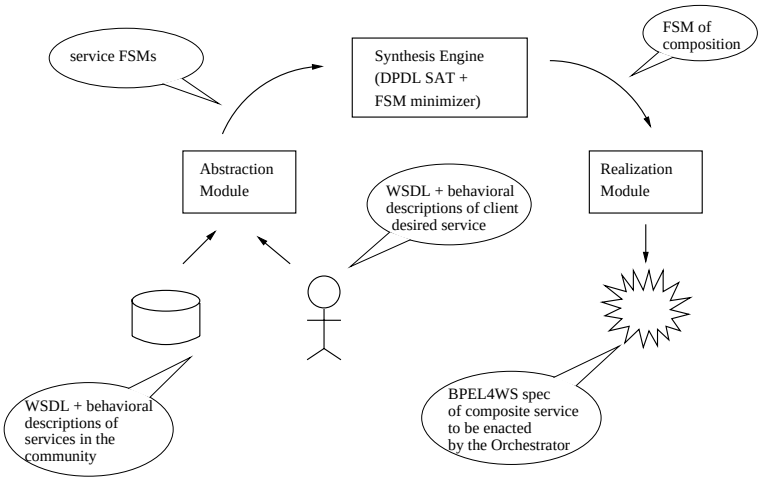


Fig. 4. The Service Composition Architecture

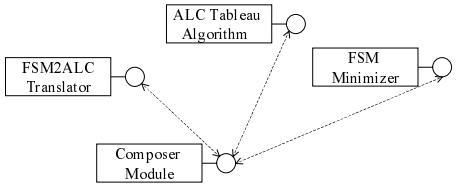


Fig. 5. Sub-modules of the `Synthesis Engine`.

MFSM of the composite service, where each action is annotated with (the identifier of) the component service(s) that executes it. Finally, such abstract version of the composite service is realized into a BPEL4WS specification (`Realization Module`), that can be executed by an orchestration engine, i.e., a software module that suitably coordinates the execution of the component *e*-Services participating to the composition.

The implementation of the `Abstraction Module` depends on which language is used to represent the behavioral description of services. For example we could use Web Service Transition Language, which can be translated into FSMs [8]. Or we could use a BPEL4WS specification, which can be translated into an extended version of guarded automata [11], and in turn to FSMs.

In the next subsections we will explain in detail the implementation of the `Synthesis Engine` and of the `Realization Module`.

4.1 Implementation of the Synthesis Engine Module

From a practical point of view, in order to actually build a finite state composition, we resort to Description Logics (DLs [3]), because of the well known correspondence be-

tween Propositional Dynamic Logic formulas (which DPDL belongs to) and DL knowledge bases. Tableaux algorithms for DLs have been widely studied in the literature, therefore, one can use current highly optimized DL-based systems [15, 13] to check the *existence* of service compositions. However, such the state-of-the-art DL reasoning systems cannot be used to *build* a finite state composition because they do not return a model. Therefore, we developed our $\mathcal{ESC}$ that, implementing a tableau algorithm for DL, builds a model (of the DL knowledge base that encodes the specific composition problem) which is a finite state composition. For our purpose the well-known $\mathcal{ALC}$ [3], equipped with the ability of expressing axioms, suffices.

The various functionalities of the `Synthesis Engine` are implemented into three Java sub-modules, as shown in Figure 5.

- The `FSM2ALC Translator` module takes in input the FSMs produced by the `Abstraction Module`, and translates them into an $\mathcal{ALC}$ knowledge base, following the encoding presented in [4].
- The $\mathcal{ALC}$ `Tableau Algorithm` module implements the standard tableau algorithm for $\mathcal{ALC}$ (cf., e.g., [9]): it verifies if the composition exists and if this is the case, it returns a model, which is a finite state machine.
- The `Minimizer` module minimizes the model, since it may contain states which are unreachable or unnecessary. Classical minimization techniques can be used, in particular, we implemented the *Implication Chart Method* [20]. The minimized FSM is then converted into a Mealy FSM, where each action is annotated with the service in the repository that executes it.

Since these three modules are in effect independent, they are wrapped into an additional module, the `Composer`, which also provides the user interface.

4.2 Implementation of the Realization Module

The technique for realizing an executable BPEL4WS file (i.e., an executable orchestration schema) starting from the automatically synthesized MFSM is as follows:

- The BPEL4WS file is built visiting the graph of the MFSM in depth, starting from the initial state and applying the previous rules, so that the nesting on `pick` and `sequence` operations reproduces the automata behavior. In Figure 6 it is shown the pseudo-code [10] of the whole BPEL4WS file obtained by the MFSM of Figure 3.
- All the transitions originating from the same state are collected in a `<pick>` operation, having as many `<onMessage>` clauses as transitions originating from the state.
- Each transition in the MFSM corresponds to a BPEL4WS pattern consisting of *(i)* an `<onMessage>` operation (in order to wait for the input from the client of the composite service), *(ii)* followed by an invocation to the effective service (i.e., the deployed service that executes the operation), and then *(iii)* a final operation for returning the result to the client. Of course both before invoking the effective

[10] For sake of simplicity, we omit all BPEL4WS details and provide an intuitive, yet complete skeleton of the BPEL4WS file.

```
<process>
    <pick>
        <onMessage="t">
            <sequence>
                <copy>...</copy>
                <invoke operation="t" on service A2 />
                <copy>...</copy>
                <reply ... />
                <pick>
                    <onMessage="l">
                        <sequence>
                            <copy>...</copy>
                            <invoke operation="l" on service A2 />
                            <copy>...</copy>
                            <reply ... />
                        </sequence>
                    </onMessage>
                </pick>
            </sequence>
        </onMessage>
        <onMessage="a">
            <sequence>
                <copy>...</copy>
                <invoke operation="a" on service A1 />
                <copy>...</copy>
                <reply ... />
                <pick>
                    <onMessage="l">
                        <sequence>
                            <copy>...</copy>
                            <invoke operation="l" on service A1 />
                            <copy>...</copy>
                            <reply ... />
                        </sequence>
                    </onMessage>
                </pick>
            </sequence>
        </onMessage>
    </pick>
<process>
```

Fig. 6. BPEL4WS pseudo-code for the MFSM shown in Figure 3

service and before returning the result, messages should be copied forth and back between the composite and the effective service. As an example, Figure 7 shows the BPEL4WS code corresponding to the MSFM transition for the `listen` operation relative to the MFSM of Figure 3.

5 Final Remarks and Future Work

In this paper we have presented a presented $\mathcal{ESC}$, a prototype tool for automatic composition, which starting from a client specification and a set of available services, synthesize a finite state composition.

We are currently extending our framework by allowing some advanced forms of don't care non determinism in the client specification and we are studying automatic composition techniques in this enhanced framework. In the future, we plan to produce a new version of our prototype tool that takes such extensions into account.

Finally, far-reaching future work may be identified along several directions. First of all, it could be interesting to study the situation when the available services export a partial description of their behavior, i.e., they are represented by non deterministic FSMs.

This means that, a large (possibly infinite) number of complete description for services in the community exists that are coherent with each partial description. In such case, the orchestration schema that is to be synthesized should be coherent with all such possible complete descriptions. Therefore, computing composition in such a framework is intuitively much more difficult that in the framework presented here. Also it is interesting to study how to add data in our framework and how this impacts the automatic service composition. In particular, it is worth studying how to introduce data in a way that the problem of automatic service composition, while exponential in the size of the service description, remains polynomial in the size of the data.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services. Concepts, Architectures and Applications*. Springer-Verlag, 2004.
2. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services (Version 1.1). http://www-106.ibm.com/developerworks/library/ws-bpel/, May 2004.
3. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
4. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. *e*-Service Composition by Description Logic Based Reasoning. In *Proc. of DL 2003*.
5. D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of *e*-services. Technical Report 22-03, 2003.
6. D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of *e*-Services that Export their Behavior. In *Proc. of ICSOC 2003*.
7. D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. A Foundational Vision of *e*-Services. In *Proc. of CAISE-WES 2003*.
8. D. Berardi, F. De Rosa, L. De Santis, and M. Mecella. Finite State Automata as Conceptual Model for e-Services. In *J. of Integrated Design and Process Science*, 2004. To appear.
9. M. Buchheit, F. M. Donini, and A. Schaerf. Decidable Reasoning in Terminological Knowledge Representation systems. *J. of Artificial Intelligence Research*, 1:109–138, 1993.
10. 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 WWW 2003*.
11. X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW 04*.
12. A. K. H. Kuno, M. Lemon and D. Beringer. Conversations + Interfaces = Business Logic. In *Proc. of VLDB-TES 2001*.
13. V. Haarslev and R. Möller. RACER System Description. In *Proc. of IJCAR 2001*.
14. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, 2000.
15. I. Horrocks. The FaCT System. In *Proc. of TABLEAUX 1998*.
16. R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: A Look Behind the Curtain. In *Proc. of PODS 2003*.
17. S. McIlraith, T. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2), 2001.
18. M. Papazoglou and D. Georgakopoulos. Service Oriented Computing (special issue). *Communications of the ACM*, 46(10), October 2003.
19. F. Pistore, M.and Barbon, P. Bertoli, and P. Shaparau, D.and Traverso. Planning and Monitoring Web Service Composition. In *Proc. of ICAPS-P4WGS 2004*.
20. R.H. Katz. *Contemporany Logic Design*. Addison-Wesley, 1993.

```
<?xml version="1.0" encoding="UTF-8"?>
<process ... >

    <partnerLinks>
    <!-- The 'client' role represents the requester of this service. It is used for callback.
         In our case it is the client of the composite service -->
            <partnerLink name="client"
                         partnerLinkType="tns:Transition"
                         myRole="MP3ServiceTypeProvider"
                         partnerRole="MP3ServiceTypeRequester"/>
            <partnerLink name="service"
                         partnerLinkType="nws:MP3CompositeService"
                         myRole="MP3ServiceTypeRequester"
                         partnerRole="MP3ServiceTypeProvider"/>
    </partnerLinks>

    <variables>
    <!-- Reference to the message passed as input during initiation -->
        <variable name="input" messageType="tns:listen_request"/>
        <!-- Reference to the message that will be sent back to the
             requestor during callback -->
               <variable name="output" messageType="tns:listen_response"/>
               <variable name="request" messageType="nws:listen_request"/>
               <variable name="response" messageType="nws:listen_response"/>
        </variables>

        <pick>
            <onMessage partnerLink="client"
                       portType="tns:MP3ServiceType"
                       operation="listen"
                       variable="input">
                       <sequence>
                            <assign>
                                 <copy>
                                     <from variable="input" part="selectedSong"/>
                                     <to variable="request" part="selectedSong"/>
                                 </copy>
                            </assign>
                            <invoke partnerLink="service"
                                    portType="nws:MP3ServiceType"
                                    operation="listen"
                                    inputVariable="request"
                                    outputVariable="response"/>
                            <assign>
                                 <copy>
                                     <from variable="response" part="MP3FileURL"/>
                                     <to variable="output" part="MP3FileURL"/>
                                 </copy>
                            </assign>
                            <reply name="replyOutput"
                                   partnerLink="client"
                                   portType="tns:MP3ServiceType"
                                   operation="listen"
                                   variable="output"/>
                            <!-- Other operations here for describing the next transitions -->
                       </sequence>
            </onMessage>
            <onMessage>
            <!-- Other sequences here for describing the other possible transitions originating
                 from the same state -->
            </onMessage>
        </pick>
</process>
```

Fig. 7. BPEL4WS code for the `listen` transition of the MFSM shown in Figure 3