Use of OWL for describing Stream Processing Components to enable Automatic Composition

Zhen Liu, Anand Ranganathan, and Anton Riabov

IBM T.J. Watson Research Center
{zhenl,arangana,riabov}@us.ibm.com

Abstract. Stream Processing Applications analyze large volumes of streaming data in real-time. These applications, consist of data sources, which produce raw streams, and processing elements, which operate on the streams to produce new derived streams. In this paper, we describe an OWL-based model for describing these stream processing components. The streams produced by data sources are described as RDF graphs, consisting of OWL ABox assertions. The input requirements and output capabilities of PEs are described using RDF graph patterns. The main use of this model is to allow automatic composition of applications that produce certain desired results. Our model allows us to answer two key questions in automatic composition: can a certain stream be given as input to a processing element, and what are the new streams produced as a result. Based on this model, we outline a planning approach to automatic application composition.

1 Introduction

Stream processing systems are needed in situations where the source data is too voluminous to be stored before being analyzed. Such data, collected at high rate, must be processed on the fly by stream processing applications that are deployed response to user queries. The data may be unstructured, semi-structured or structured, and may be in different formats including text, video, audio and images. Examples of such systems include multimedia processing and delivery networks, stream-processing systems, sensor networks and other information analysis, mining and retrieval systems.

Many systems model stream processing applications as processing graphs of components, that are deployed in (possibly distributed/networked) computer systems, and that can extract meaningful information from streaming data. In our work, we use a specific stream processing system called System S [1], which provides a scalable distributed runtime for stream processing of structured or unstructured data. There are two kinds of components in a System S processing graph : Data Sources and Processing Elements (PE). Data sources produce raw streaming data to be analyzed. PEs are reusable software components that can take in one or more input streams, process them and produce one or more new, derived output streams. Such a component-based programming model has various advantages, including reusability and scalability.

A key challenge in stream processing systems lies in the construction of the processing graphs that can satisfy user queries. With large numbers of disparate data sources and processing elements to choose from, we cannot expect the end-user to craft these graphs manually. The set of PEs and data sources can also change dynamically as new sources are discovered or new PEs are developed. Different end-users express widely varying queries, requiring a large number of different graphs to be constructed. Since there is a large number of possible graphs for a given number of data sources and PEs, it is not feasible to pre-construct all the graphs, manually, to satisfy the wide variety of end-user queries.

Hence, for purposes of usability and scalability, the system must compose the processing graphs on behalf of the end-user automatically, and on-the-fly, whenever a query is submitted. For automatic composition, we need rich descriptions of the different components and the kinds of data they take as input and produce as output. In this paper, we propose an expressive model for describing these software components that is based on OWL. The streams produced by data sources are described as RDF graphs, consisting of OWL ABox assertions. The input requirements and output capabilities of PEs are described using RDF graph patterns.

A key element of our semantic model is in laying out the conditions for connecting a stream to a PE and determining the semantics of the output stream produced by the PE. We use Description Logic reasoning to determine if a stream matches the input requirements of a PE. The use of reasoning helps increase the power of matching. Our model also includes the notion of semantic propagation, i.e. the semantics of the output stream produced by the PE depends on the semantics of the input stream.

There is an important difference between our model and other existing component models. Many other component description models, both semantic and syntactic (like WSDL, OWL-S, SAWSDL, Java interfaces, CORBA IDL, etc.), describe the inputs and outputs of components in terms of datatypes or classes (or concepts in an ontology in the case of semantic models). Our model describes inputs and outputs in terms of semantic graphs based on instances (or individuals). Our instance-based approach allows associating constraints on the input and output data based on both the classes they belong to and their relationship to other instances. Such constraints are more difficult to express in class-based representations and often require the creation of a large number of additional classes corresponding to different combinations of constraints. As a result, our model allows associating rich semantic information about components, which aids the automatic composition of processing graphs.

There is another key difference between our model and semantic web service models (like OWL-S and WSML). OWL-S and WSML define preconditions and effects on the state of the world for a service. WSML also defines preconditions and postconditions on the information space of a service. Our model, however, defines rich constraints on the input and output streams for a component. This model is particularly suited for describing stream processing components, whose primary job is to process data, and which hence need rich descriptions of the data that they process.

The main purpose of this model is to enable automatic composition of processing graphs given a user query, which is also expressed as an RDF graph pattern. We are investigating various approaches for automatic composition including a planning and a rule-based approach. We have developed a planner that can automatically build applications given a user query. The planner uses reasoning based on Description Logic Programs (DLP) [2]. One of the features of the planner is the use of a two-phase approach, where offline reasoning is performed on component descriptions, and plans are built online for different user queries using the cached results of the offline reasoning.

The key contribution of our paper is the description of an OWL-based model for describing stream processing components and for determining if a data stream can be connected as input to a component. We have experimented with this model and described several stream processing components in our system, System S [1]. In Section 2, we describe stream processing graphs. In Sections 3 and 4, we describe the stream, source and PE models. In Section 6, we briefly describe our planning approach. Section 7 has related work and conclusions.

2 Stream Processing Applications

Stream Processing Applications in System S are modeled as Processing Graphs, that describe the flow of data from the sources and through a number of PEs to finally produce some desired end result. In our work, the processing graphs are described as DAGs (Directed Acyclic Graphs).

The running example that we will use in the paper is based on a system that provides real time traffic information and vehicle routing services based on analysis of real-time data obtained from various sensors, web pages and other data sources. Let us assume that a user has given a continuous query for traffic congestion levels for a particular roadway intersection, say Broadway and 42'nd St. in New York City. A data flow that is constructed for such an query may use raw data from different sources. It may use video from a camera at the intersection by extracting images from the video stream and examining them for alignment to visual patterns of congestion at an intersection (the upper thread in Figure 1). In order to improve the accuracy, it may also get audio data from a sound sensor at the intersection and compare it with known congestion audio patterns. The end result is achieved by combining feeds from the two analytic chains. In the following sections, we will describe how these components are described.

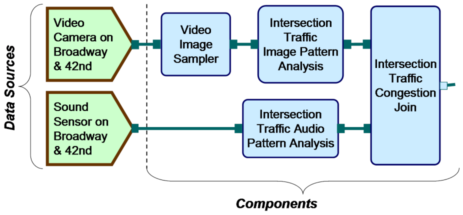
Data Sources
Video Camera on Broadway & 42nd
Sound Sensor on Broadway & 42nd
Video Image Sampler
Intersection Traffic Image Pattern Analysis
Intersection Traffic Audio Pattern Analysis
Intersection Traffic Congestion Join
Components

Fig. 1. Example Processing Graph

3 The Stream Model and Description of Data Sources

A stream carries zero or more data objects, that we call SDOs (Stream Data Objects). Each stream is associated with a semantic description, which describes the data elements present in a typical (or exemplar) SDO on the stream, and any constraints that are satisfied by the data in terms of a set of OWL facts.

Formally, a stream is of the form $S(E, R)$ where

- E is the set of data elements in a typical SDO, that are represented as OWL individuals or literals.
- R is an RDF graph that describes the semantics of the data elements in the message. The RDF graph consists of a set of OWL facts (ABox assertions).

One challenge with such a description is that different SDOs carried by the stream may have different values of the data elements. Hence, we provide a layer of abstraction over the actual values and define the data elements contained in an exemplar SDO as exemplar individuals or exemplar literals. An exemplar individual is represented in OWL as belonging to a special concept called Exemplar. An exemplar literal is of a user defined type called xs:exemplar, which is derived from xs:string.

Exemplar individuals and literals act as existentially quantified variables. In a particular SDO, they may be substituted by a value that belongs to the set of non-exemplar individuals (i.e. an OWL individual not belonging to the concept Exemplar) or non-exemplar literals (i.e. a literal that is not of type xs:exemplar). In this paper, we represent all exemplar individuals and literals with a preceding "__". In order to allow exemplar literals to be value of datatype properties, we model all OWL datatype properties as having a range which is the union of the original datatype and xs:exemplar.

For example, consider the stream produced by the video camera source in Figure 2. Every SDO on this stream has two data elements: a video segment and a time interval. These elements are described as exemplar individuals, __VideoSegment_1 and __TimeInterval_1. The semantic constraints obeyed by these data elements are described as an RDF graph, consisting of OWL ABox assertions.

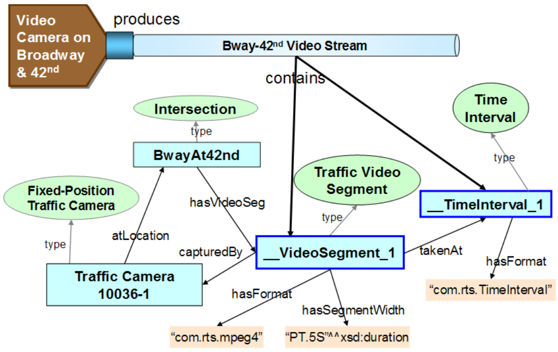

Fig. 2. Data Source semantic description

The power of this semantic description is that it explicitly describes all the semantic relationships between the different data elements in a typical SDO as well as between the elements and other in the domain (such as BwayAt42nd). It describes the streams in terms of individual-based graph patterns. This is in contrast to class-based descriptions that are commonly used in various other models (like OWL-S). The individual-based descriptions allow associating rich semantics to the web service, by specifying complex inter-relationships between different instances (for example, the relations between __VideoSegment_1, __TimeInterval_1, TrafficCamera10036-1 and

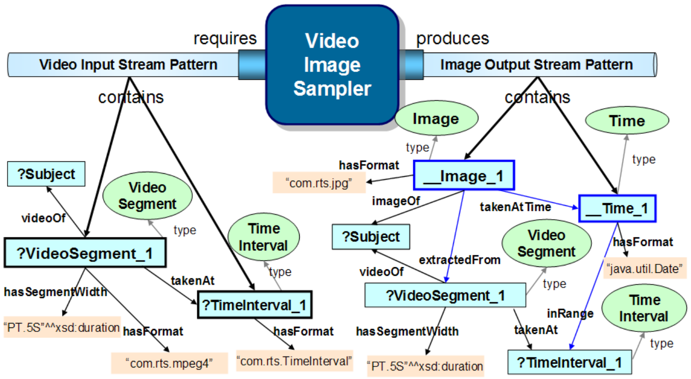

Fig. 3. PE semantic description

BwayAt42nd. Such relationships are more difficult to capture using class-based descriptions without having to create a large number of new classes for different combinations of the relationship constraints. Also, class-based descriptions cannot capture cyclic relationships between individuals (e.g. when two individuals are related to another anonymous individual via different property paths). Such cyclic relationships can be captured in individual-based descriptions.

The semantic descriptions are based on one or domain ontologies in OWL that define various concepts, properties and instances (i.e. both TBox and some parts of the ABox) in a domain of interest. A domain ontology in this scenario may define concepts like Intersection and TrafficVideoSegment, properties like capturedBy and hasVideoSeg, and individuals like BwayAt42nd.

4 Model of PEs

PEs are described by the kinds of streams they require as input and the kinds of streams they produce as output. A PE takes m input graph patterns, processes (or transforms) them in some fashion and produces n output graph patterns. Our model provides a blackbox description of the PE; it only describes the inputs and outputs, and doesn't model the internal state of the PE.

For example, consider the VideoImageSampler in Figure 3, which has one input and one output. Any input stream connected to this PE must carry SDOs that contain two data elements: a video segment (?VideoSegment_1) and a time interval (?TimeInterval_1). The component analyzes this input SDO and produces as output an SDO containing two new objects: an image (__Image_1) that it extracts from the video segment, and a time (__Time_1) for the image, which lies within the input time interval. There are other constraints associated with these data elements in the input and output SDO, such as(?VideoSegment_1 takenAt ?TimeInterval_1), and (?VideoSegment_1 hasSegmentWidth PT.5S^^xsd:duration). Namespaces of terms are not shown in the figure.

We now describe the component model formally. Some of the elements of this model are adapted from SPARQL, a standard RDF query language.

Let U be the set of all URIs and RDF_L the set of all RDF Literals. The set of RDF Terms, RDF_T, is $U \cup RDF_L$.

A ***variable*** is a member of the set V where V is infinite and disjoint from RDF_T. A variable is represented with a preceding "?" .

A ***triple pattern*** is a member of the set $(RDF_T \cup V) \times U \times (RDF_T \cup V)$.

A ***graph pattern*** is a set of triple patterns.

An ***input stream pattern*** describes the properties of SDOs in an input stream required by a PE. It is of the form $ISP(VS, GP)$:

- VS is a set of variables representing the data elements that must be contained in the exemplar SDO. $VS \in 2^V$.
- GP is a graph pattern that describes the semantics of the data elements in the SDO.

In an output SDO, a PE may create new objects that did not appear in the input SDO. In the output stream pattern description, these new objects are represented as *exemplars*.

Let E represent the set of all exemplars. An ***output stream pattern*** is of the form $OSP(OS, GP)$:

- OS is a set of variables and exemplars, that represent the data elements that are contained in the exemplar output SDO. $OS \in 2^{V \cup E}$.
- GP is an graph pattern that describes the semantics of the data elements in the output message.

The output stream description has a combination of variables and exemplars. Variables represent those entities that were carried forward from the input stream description; and exemplars represent those entities that were created by the PE in the output stream description.

A ***PE*** is described as taking a set of input stream patterns and transforming into a set of output stream patterns. It is of the form $PE(\langle ISP \rangle, \langle OSP \rangle)$ where

- $\langle ISP \rangle$ is a set of input stream patterns that describe the input requirements of the component. The different message patterns may overlap; i.e. the graph patterns they contain may share common nodes and edges. The overlap helps describe dependencies between different input stream patterns.
- $\langle OSP \rangle$ is a set of output stream patterns, that describe the outputs of the component. Again, the different stream patterns may overlap amongst themselves as well as with the input message patterns.
- The set of variables in $\langle OSP \rangle$ is a subset of the set of variables that are described in $\langle ISP \rangle$. This helps ensure that no free variables exist in the output description.

Note that the actual messages need not be in the form of RDF graphs. Depending on the actual middleware and communication mechanism, these messages may be in different formats (such as XML messages in the case of web services; serialized objects in the case of CORBA and Jini; or various streaming audio, video and image formats in the case of multimedia networks). In our system, each message is formatted as a collection of serialized Java objects. For example, the component description states that the format of ?VideoSegment_1 should be the Java class (com.egs.mpeg4), which represents a byte array containing the video segment.

5 Matching streams to the input requirements of PEs

The semantic description of a PE gives a general, application-independent, description of the kinds of streams it takes in and the kinds of streams it produces. In a given application, the PE is going to be given a set of input streams, as a result of which it produces certain output streams.

One of the key parts of any automatic composition process in our system is to determine if a given set of streams can be given as input to a PE. We define the notion of a match between a stream and an input stream pattern that represents the component's input requirement. We define the match, in terms of a pattern solution, which expresses a substitution of the variables in the input message pattern.

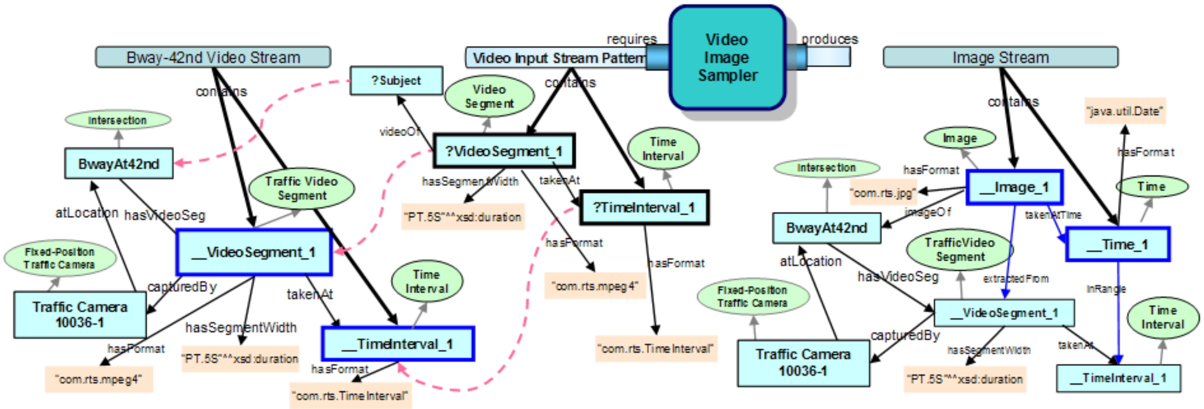

Fig. 4. Example stream that has been matched to the input message pattern of the PE. The corresponding output stream is also shown. Inferences performed to obtain match are not shown.

Pattern Solution. A pattern solution is a substitution function ($\theta : V \rightarrow RDF_T$) from the set of variables in a graph pattern to the set of RDF terms. For example, some of the mappings defined in a possible definition of θ for the example graph pattern include : θ(?VideoSegment_1) = __VideoSegment_1, θ(?Subject) = BWayAt42nd, etc.

The result of replacing a variable, v is represented by $\theta(v)$. The result of replacing all the variables in a graph pattern, GP, is written as $\theta(GP)$.

Condition for match. Consider an input stream-pattern $P(VS, GP)$, and a stream, $S(E, R)$. We define that P is matched by S, based on an ontology, O, if and only if there exists a pattern solution, θ, defined on all the variables in GP, such that following conditions hold:

- $\theta(VS) \subseteq E$, i.e. the stream contains at least those data elements that the pattern states it must contain.
- $R \cup O \models_E \theta(GP)$ where O is the common ontology, and $\models_E$ is an entailment relation defined between RDF graphs. In our system, we consider entailment

based on OWL-DLP; though, in general the entailment may be based on RDF, OWL-Lite, OWL-DL or other logics.

We represent this match as $S \bowtie_{\theta} P$ to state that stream S matches the input stream pattern, P with a pattern solution, θ. One way of looking at the above definition is that the stream should have at least as much semantic information as described in the pattern. Figure 4 shows how the BWay-42nd VideoStream might match the VideoInputStreamPattern. The dashed arrows show the variable substitutions. In order to make the match, some DLP reasoning based on subclass and inverse property relationships must be done. For example, the triple (__VideoSegment_1 videoOf BwayAt42nd) is inferred, since videoOf is declared to be an inverse property of hasVideoSeg. Also, the triple (__VideoSegment_1 type VideoSegment) is inferred, since TrafficVideoSegment is declared to be a subclass of VideoSegment. Once the inferences are done, it is clear to see that the graph on the right is a subgraph of the graph on the left; hence a match is obtained.

In a more general case, for a component that has m input stream requirements $(P_1 \ldots P_m)$, we need to give it m input streams $(S_1 \ldots S_m)$ to it, such that $S_i \bowtie_{\theta} P_i$, for $i = 1 \ldots m$ and for some substitution function θ that is common across all streams.

When a set of input streams are given to a PE, the component generates output streams. The actual description of the output streams is generated by combining the descriptions of the input messages with the output message patterns of the component. This combination is defined in terms of a graph transformation operation [3]. This operation captures the notion that some of the semantics of the input streams are propagated to the output streams. This is accomplished by substituting the variables in the output stream pattern by RDF graphs extracted from the input stream. An example is shown in Fig 4, where the output stream of VideoImageSampler is generated based on the stream that is given as input to it. Note how the some of the semantics of the input stream (such as the traffic camera and intersection information) are propagated to the output stream. We call this property *semantic propagation*, where the semantics of the output streams depend on the semantics of the input streams.

6 Automatic Composition

The model of data sources and PEs lends itself to automatic composition through a variety of approaches, such as planning and rules. We have developed a planner that builds an application given a user query. Due to lack of space, we shall only briefly outline the features of the planner and not go into details.

We represent a query to a stream processing system as an input stream pattern. This stream pattern describes the kind of messages (data objects in the message and the semantics of the data objects) that the user is interested in. This stream pattern becomes a goal for our planner. It needs to construct a processing graph that produces a stream which satisfy the stream pattern. The syntax of the query is similar to SPARQL (where SELECT is replaced by PRODUCE to emphasize the continuous query aspect of stream processing systems). An example continuous query for real-time traffic congestion levels at the Broadway-42'nd St intersection is

```
PRODUCE ?congestionLevel, ?time
WHERE (?congestionLevel rdf:type CongestionLevel) , (?time rdf:type Time),
      (?congestionLevel ofLocation BwayAt42nd) , (?congestionLevel atTime ?time)
```

Our planner works by checking if a set of streams can be connected to a PE, and if so, it generates new streams corresponding to the outputs of the component. It performs this process recursively and keeps generating new streams until it produces a stream that matches the goal, or until no new unique streams can be produced, or a certain limit on the size of the plan has been crossed.

One of the key design decisions of the planner is to split DL reasoning and plan-building into separate phases so as to avoid calling a reasoner during planning and thus improve performance, while potentially sacrificing completeness. In the first phase, which occurs offline, it translates the descriptions of components into a language called SPPL (Stream Processing Planning Language) [4]. SPPL is a variant of PDDL and models the state of the world as a set of streams and interprets different predicates only in the context of a stream. During the translation process, the generator also does reasoning based on OWL-DLP (Description Logic Programs) on the output descriptions to generate additional inferred facts about the outputs. The actual reasoner used in this phase is Minerva [5]. The SPPL descriptions of different components are persisted and re-used for multiple queries. The second phase is triggered whenever a query is submitted to the system. During this phase, the generator translates the query into an SPPL planning goal. The SPPL Planner produces a plan by recursively connecting components to one another [4].

A feature of our planning process is that DLP reasoning is performed only once for a component, in an offline manner. During actual plan generation, the planner does not do any reasoning. It only does subgraph matching. This allows the matching process to be faster than if reasoning was performed during matching. The actual reasoning is performed on the DLP fragment of OWL. DLP lies in the intersection of Description Logic and Horn Logic Programs (like Datalog). Inference on the ABox in DLP can be performed using a set of logic rules. This allows us to take a certain assertion and enumerate all possible assertions that can be inferred from this assertion and an ontology using the rules. The ability to enumerate all inferences is a key reason for the choice of DLP for reasoning. Since we cannot directly perform inferences on variables, we convert them into OWL individuals that belong to a special concept called Variable. Using this process, a graph pattern can be converted into an OWL/RDF graph for the purposes of reasoning, and additional facts about variables can be inferred.

Note that the plan building is performed once for a query. No planning or DL reasoning is performed at runtime for each individual stream record, which may arrive at a very high rate.

7 Related Work, Conclusion and Experiences

There is a huge body of work in the area of web service composition (e.g. [6], [7],[8],[9],[10]). Many of these approaches are based on OWL-S, or its predecessor

DAML-S. The main novelty of our approach is in modeling the preconditions and effects of stream processing components in terms of the properties of input and output streams, expressed as individual-based graph patterns. This is in contrast to models like OWL-S and DAML-S which model inputs and outputs as concepts in an ontology. Our composition problem also differs from web service composition, since it requires the production of a stream that satisfies the users query rather than achieving a change in the state of the world.

In this paper, we have presented a model for describing stream processing components based on OWL that allows specifying rich constraints on inputs and outputs. We have experimented with this model in our stream processing system, and have described a number of data sources and PEs in this manner. So far, we have restricted ourselves to the DLP fragment of OWL, as a constraint imposed by our planning process. However, we are investigating other composition approaches that would allow the use of more expressive DL logics.

In our model, we have introduced a novel modeling technique in the use of exemplars to help describe the properties of a class of individuals (i.e. the properties of all SDOs in a stream). This technique allows using ABox assertions to describe the class, rather than TBox axioms. This approach is more natural in a variety of situations, especially when the individuals are related to other individuals in the domain of interest (such as a traffic camera or an intersection).

There are a number of areas of future work, especially focused on making the descriptions of data sources and PEs easy to write for a large number of developers in a collaborative manner. This requires the use of an ontology engineering framework that supports such collaboration. Such a framework is a pre-requisite for automatic composition.

References

1. Jain, N. *et al*: Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In: SIGMOD'06. (June 2006)
2. Grosof, B., Horrocks, I., Volz, R., Decker, S.: Description logic programs: combining logic programs with description logic. In: WWW'03. 48–57
3. Baresi, L., Heckel, R.: Tutorial introduction to graph transformation: A software engineering perspective. In: 1st Int. Conference on Graph Transformation. (2002)
4. Riabov, A., Liu, Z.: Planning for stream processing systems. In: AAAI'05
5. Zhou, J., Ma, L., Liu, Q., Zhang, L., Yu, Y., Pan, Y.: Minerva: A scalable OWL ontology storage and inference system. In: 1st Asian Semantic Web Symp. (2004)
6. Sirin, E., Parsia, B.: Planning for Semantic Web Services. In: Semantic Web Services Workshop at 3rd ISWC
7. Narayanan, S., McIlraith, S.: Simulation, verification and automated composition of web services. In: WWW. (2002)
8. Sheshagiri, M., desJardins, M., Finin, T.: A planner for composing services described in DAML-S. In: Web Services and Agent-based Engineering - AAMAS'03
9. Traverso, P., Pistore, M.: Automated composition of semantic web services into executable processes. In: ISWC. (2004)
10. Lécué, F., Léger, A.: A formal model for semantic web service composition. In: ISWC. (2006)