Specifying and Enforcing High-Level Semantic Obligation Policies

Zhen Liu, Anand Ranganathan, Anton Riabov
IBM T.J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532
{zhenl, arangana, riabov}@us.ibm.com

Abstract

Obligation Policies specify management actions that must be performed when a particular kind of event occurs and certain conditions are satisfied. Large scale distributed systems often produce event streams containing large volumes of low-level events. In many cases, these streams also contain multimedia data (consisting of text, audio or video). Hence, a key challenge is to allow policy writers to specify obligation policies based on high-level events, that may be derived after performing appropriate processing on raw, low-level events. In this paper, we propose a semantic obligation policy specification language called Eagle, which is based on patterns of high-level events, represented as RDF graph patterns. Our policy enforcement architecture uses a compiler that builds a workflow for producing a stream of events, which match the high-level event pattern specified in a policy. This workflow consists of a number of event sources and event processing components, which are described semantically. We present the policy language and enforcement architecture in this paper.

1 Introduction

Obligation Policies define actions that must be performed by some entity when a certain kind of event occurs, provided certain conditions are satisfied. Large scale distributed and pervasive computing systems often produce streams of different kinds of events. Such events include logging and audit messages from different applications and computing nodes, events from different kinds of sensors (such as temperature, humidity and RFID sensors) and events about user activity and system reconfiguration. These systems also often produce streams of unstructured data (such as text and multimedia) from various sources like surveillance video cameras and emergency radio broadcasts. Many of these events are very low-level and high-volume, and often require further processing to extract useful information from them. Event processing techniques include filtration, aggregation, classification, correlation, multimedia data analysis, etc., and they help in producing lower volumes of high-level events. Policies based on these high-level events are easier to specify and maintain than those that are written based on large volumes of low-level events. Such policies also make it easier to specify high-level business rules.

In this paper, we propose a high-level, semantic obligation policy specification language called Eagle, that allows policy writers to specify policies based on semantic event patterns. These event patterns are represented as RDF graph patterns. The policies also specify conditions to be checked and actions to be performed when there is an event in the system that matches the high-level event pattern. The policies use terms defined in domain ontologies, that are expressed in OWL [11]. This semantics-based approach to specifying policies makes the policy language highly expressive, allows the specification of high-level policies and enables all policies in the system to use a common vocabulary, that is defined in domain ontologies.

In order to enforce policies in Eagle, the system needs to produce appropriate high-level events from one or more low-level events. We model the processing of events using a processing graph (or workflow) of components, that are deployed in a distributed system, and that can extract meaningful information from streaming events. A processing graph is a directed acyclic graph (DAG) consisting of Event Sources and Processing Elements (PE) that are interconnected by event streams. Event Sources bring in raw events to be analyzed. PEs are reusable software components that can perform various kinds of operations on events to produce new, derived events. Such a component-based programming model has various advantages including reusability and scalability.

A key challenge lies in the construction of the processing graphs that can produce high-level events specified in a policy. There may be large numbers of disparate event sources and processing elements to choose from, and this set can change dynamically as new sources are added or new PEs are developed. Hence, we cannot expect the policy writer to
craft these processing graphs manually. In fact, it would be preferable to allow policy writers to be unaware of the kinds of low level events in the system and the kinds of event processing that can be done by same system.

Our policy enforcement framework tackles this challenge by automatically composing a processing graph for a given policy. It takes a stream-centric approach by considering all events to be part of streams. The notion of a stream, which is defined as an aggregation of events, is useful for routing certain kinds of events between different components in a processing graph. In our framework, event streams are associated with semantic annotations, expressed as OWL facts and represented using RDF graphs.

Our enforcement framework consists of two parts: a Policy Compiler and a set of Stream Monitors. The Policy Compiler builds processing graphs for producing a stream of high-level events, which match the event patterns specified in the policies. It makes use of descriptions of different event sources and PEs in terms of the semantics of the event streams they consume and produce. The compiler uses AI planning methodologies, reasoning based on Description Logic Programs (DLP) [9] and multi-objective optimization techniques to produce the processing graphs. The Stream Monitors actually enforce the policy. These components monitor the events on the high-level streams, and if certain conditions are satisfied, they perform actions by invoking various services.

We have deployed this policy enforcement framework on top of the System S stream processing system [10]. In this paper, we present the semantic policy model and the framework. We do not consider the problem of detecting policy action conflicts in this work. The paper is organized as follows. In section 2, we present an overview of the system and a sample application. In section 3, we describe the ontologies that form the backbone of our policies. In section 4, we present the Eagle language. In sections 5 and 6, we describe the semantic model of streams and the policy enforcement framework, respectively. In sections 7, 9 and 10 we evaluate our compiler, review related work and conclude.

2 Overview

We illustrate the operation of our system through an example scenario. Consider the customer service department of a fictional multi-national company named EGS (Enterprise Global Services). Events may be generated based on incoming and outgoing service calls, call-transfers, dropped calls, system load and performance, etc. In addition, company policy allows some types of information to be extracted from the corporation’s telecommunications traffic (“your call may be monitored for quality assurance”). Hence VoIP-based calls may be analyzed to determine customer satisfaction, employee courtesy, call quality, etc.
selects optimal graphs according to various quality metrics. The generated processing graphs are finally deployed in the System S Stream Processing Core [10], which is a scalable distributed runtime for data stream processing. In addition, Stream Monitors that enforce the policy are instantiated.

There are a number of advantages in using a semantics-based, stream-centric approach to policy specification and enforcement. These include:

**High expressiveness.** The use of OWL/RDF allows us to represent complex inter-relationships between different entities in an event pattern in a policy.

**Composition Power.** Since policies, as well as event sources and PEs are described using expressive semantic models, this provides more information to the policy compiler for generating plans (or processing graphs) that produce high-level events specified in the policies. In addition, the use of OWL allows the compiler to perform description logic reasoning and also use domain knowledge in ontologies to generate the plans. The notion of streams helps in representing and deploying such processing graphs.

**Formal definition of all terms in ontologies.** All policies and descriptions of event sources and PEs are based on one or more ontologies, which are written in OWL. These ontologies define all terms, in a certain domain of interest, using a formal logic. Thus, events and policies have well-defined and clear semantics (or meaning). This helps reduce the possibility of ambiguity of what a policy means and helps increase interoperability across different parts of a system. While there are many existing information models used for defining policies such as IETF/DMTF [13][7], ontologies allow establishing more formal semantics with terms. In our system, we assume that all components in the system use a common set of domain ontologies. The mapping or integration of heterogeneous ontologies is a separate problem, which is out of our scope in this work.

**3 Ontologies**

Ontologies form the backbone for Eagle policies by providing a formal description of the kinds of salient entities and how they are related. Policies use terms defined in ontologies to describe the high-level event patterns.

OWL ontologies describe **concepts (or classes), properties and individuals (or instances)** relevant to a domain of interest. For the EGS example, we draw on several ontologies that describe domain independent concepts such as Physical Thing, Person and Location, as well as domain-specific concepts like ServiceVolPCall, and Employee. Concepts may be related via subclassOf relationships. A property has a domain and a range. For instance, the domain of atLocation is PhysicalThing and the range is Location. OWL Object Properties (like atLocation) have a range which is a concept, and OWL Datatype properties (like hasName) have a range which is an xsd datatype. Individuals (like EGS and Bob) belong to one or more concepts and are related to one another, or to literal values (like the string “Bob Roberts”), through various properties. Figure 2 shows a portion of the ontologies.

Now, we review some definitions from RDF and OWL, which we shall use when we define policies and the semantic model of event streams, event sources and PEs.

**RDF Term.** Let \( U \) be the set of all URIs. Let \( RDF_L \) be the set of all RDF Literals (which are data values, that may be typed). The set of RDF Terms, \( RDF_T \), is \( U \cup RDF_L \). Note that RDF also defines blank nodes, which we do not include in our model.

**RDF Triple.** An RDF triple contains three components: a subject, a predicate and an object. The subject and predicate are URIs, while the object is an RDF Term. An example of a triple is (Bob employedBy EGS).

**RDF Graph.** An RDF graph is a set of RDF triples. The nodes in an RDF graph include the subjects and objects of triples in the graph. The edges are labeled by the properties.

**OWL Axiom.** An OWL Axiom is a statement in the ontology that gives information about classes and properties. This information includes subclass and subproperty relationships, whether a certain property is transitive, symmetric or functional, or the inverse of another property, restrictions on the values of a property, etc. OWL Axioms may be represented as RDF triples. An example is (Employee subClassOf Person).

**OWL Fact.** An OWL fact is a statement about individuals, in the form of classes that an individual belongs to or relationships between individuals. OWL facts may also be represented as RDF triples. An example is (EGS a Company). The property “a” indicates that the subject, EGS, is an individual of type Company, an OWL concept.

**4 The Eagle Policy Specification Language**

An Eagle policy is defined on a pattern of events. An event pattern describes an equivalence class of events which
can trigger the execution of the policy. Policy actions are performed based on specific events in the system which match the event pattern.

A policy in Eagle consists of an event pattern and a set of condition-action pairs. The event pattern describes the data that must be contained in the event as a set of variables. It also describes additional semantics on this data as a graph-pattern. Conditions are described as a conjunction of expressions, where an expression is either a constraint on a value that appears in a matched event, or a boolean-valued result of invoking a method on a service. Actions are specified as invocations to services with some parameters. Some of the elements of this language are adapted from SPARQL [14], a standard RDF query language. The general syntax of Eagle is shown below.

```
Policy <PolicyName>
OnEvent
  Containing <VariableSet>
  Where <Graph Pattern>
PerformAction <ActionDefinition_1>
  If <ConditionDefinition_1>
...  
PerformAction <ActionDefinition_n>
  If <ConditionDefinition_n>
```

An example of an Eagle policy is shown below. This policy is defined on an event pattern that contains the name of an employee, the email address of his manager and the current courtesy level of the employee in a VOIP call (see Figure 3). The graph pattern is described in N3 [4] format (the “:" in front of some of the terms represents the default namespace). The policy states that if the courtesy level is less than 0.4, then an instant message alert should be sent to the manager; and if the courtesy level is less than 0.2, then the employee should not be allowed to answer any further customer service calls.

```
Policy EmployeeCourtesyPolicy
OnEvent
  Containing ?EmpName, ?MgrEmail, ?CurtLevel
  Where
    ?Employee a :Employee;
    :tookCall ?EmpVoIP_Call;
    :hasName ?EmpName;
    :hasMgr ?Manager;
    :employedBy EGS.
  ?EmpVoIP_Call a :ServiceVoIP_Call;
  :withCurtLevel ?CurtLevel.
  ?Manager a :Manager;
  :hasEmail ?MgrEmail.
PerformAction IMServ.sendAlert(?MgrEmail, ?EmpName)
  If (?CurtLevel < 0.4)
  AND (IMServ.isOnline(?MgrEmail))
PerformAction CallRouteServ.suspendEmp(?EmpName)
  If (?CurtLevel < 0.2)
```

An example of an Eagle policy is shown below. This policy is defined on an event pattern that contains the name of an employee, the email address of his manager and the current courtesy level of the employee in a VOIP call (see Figure 3). The graph pattern is described in N3 [4] format (the “:" in front of some of the terms represents the default namespace). The policy states that if the courtesy level is less than 0.4, then an instant message alert should be sent to the manager; and if the courtesy level is less than 0.2, then the employee should not be allowed to answer any further customer service calls.

```
Policy <PolicyName>
OnEvent
  Containing <VariableSet>
  Where <Graph Pattern>
PerformAction <ActionDefinition_1>
  If <ConditionDefinition_1>
...  
PerformAction <ActionDefinition_n>
  If <ConditionDefinition_n>
```

A variable is a member of the set $V$ where $V$ is infinite and disjoint from $RDF_T$. A triple pattern is an RDF triple, containing a subject, predicate and object, where either the subject or the object is a variable. An example is (?Employee employedBy EGS).

A graph pattern is a set of Triple Patterns. An example graph pattern appears in the Where clause of the sample policy described earlier. An event pattern is a 2-tuple of the form $EP(VS, GP)$ such that
- $VS$ is a set of variables representing the data that must be contained on a matching event. $VS \in 2^V$
- $GP$ is a graph pattern that describes the semantics of the data in the event.

A value constraint is a boolean-valued expression of variables and RDF Terms, e.g., ($?CurtLevel < 0.4$) A boolean method is a remote method invocation on a service that returns either true or false. The method may have parameters, which could be either variables or constants. An example is IMServ.isOnline(?MgrEmail)

A condition is a conjunction of value constraints and boolean methods. An action is a remote method invocation on a service. The method may have parameters, which could be either variables or constants. An example is IMServ.sendAlert(?MgrEmail, ?EmpName)

A policy is a 3-tuple, $(PN, EP, \langle C, A \rangle)$, where
- $PN$ is a string that represents the name of the policy
- $EP$ is an event pattern
- $\langle C, A \rangle$ represents a set of condition-action pairs. All variables used in the definitions of conditions and actions must appear in the variable set of $EP$.

5 Descriptions of Event Streams

Our policy enforcement framework use a stream-centric approach. This allows it to connect different event sources and PEs in a processing graph using streams. Descriptions of event sources and PEs are in terms of streams they consume and produce.
A stream carries zero or more events that satisfy a common set of constraints. These constraints are expressed in the stream metadata, which is described semantically, using OWL facts represented as an RDF graph. The metadata provides rich information about the meaning of the events on the stream, together with its format.

The semantic metadata description of a stream is in terms of the semantics of a typical (or exemplar) event in the stream. It describes the data present in the typical event and any constraints that are satisfied by the data in terms of a graph of OWL facts.

**Exemplar Individuals and Literals.** In order to describe the semantics of a stream of events, we introduce the notion of exemplar individuals and literals. An exemplar individual is a member of the set $E_I$ where $E_I$ is infinite and $E_I \subset U$. In an OWL ontology, it is represented as belonging to a distinguished concept called Exemplar. An exemplar literal is a member of the set $E_L$ where $E_L$ is infinite and $E_L \subset RDF_L$. In an OWL ontology, they have the user-defined type xsd:exemplar.

Exemplar individuals and literals represent existentially quantified variables. In a particular event, they may be substituted by a value that belongs to the set of non-exemplar individuals (i.e., $U - E_I$) or non-exemplar literals (i.e., $RDF_L - E_L$). In this paper, we represent all exemplar individuals and literals with a preceding “__”.

**Stream-Triple.** A stream-triple is an OWL fact where either the subject is an exemplar individual or the object is an exemplar individual or an exemplar literal. An example of a stream-triple is ($x$, a Person) . Different events in a stream may replace $x$ with different values (such as John or Mary). However, any values that $x$ is replaced by must satisfy the condition that it’s type is Person.

**Stream-Graph.** A Stream-Graph is a set of Stream-Triples. An example, in RDF N3 format, is:

```
_:EmployeeVolIP_CallStream_1 a :ServiceVolIP_Call;
a :Exemplar;
ofEmp :__Employee_1;
.withCurtLevel :__EmployeeName_1.
_:Employee_1 a :CustServRep;
a :Exemplar;
.hasName :__EmployeeName_1;
inDept :__Department_1;
.hasMgr :__Manager_1;
.employedBy :EGS.
_:Manager_1 a :Manager;
a :Exemplar;
.hasName :__MgrName_1;
.hasEmail :__MgrEmail_1.
_:Department_1 a :Department;
a :Exemplar;
.hasFormat :com.egs.Dept.
```

**Stream.** The semantic description of a stream identifies the data present in a typical event (on the stream) and any constraints on the data, expressed using a graph of OWL facts. The semantic description of a stream is a 3-tuple of the form $(SN, SD, SG)$ where

- $SN$ is the name of the stream. The stream name is represented as a URI, i.e., $SN \in U$
- $SD$ is the set of data items contained in the typical event in the stream. The data in the typical event is represented as exemplar individuals and exemplar literals. That is, $SD \in 2^{E_I \cup E_L}$
- $SG$ is the stream graph that describes the semantics of the data on the typical event in the stream. The stream graph describes the constraints associated with all the exemplar individuals and exemplar literals that are contained in the stream.

An example of a stream is the EmpCurt_VolP_CallStream (Figure 4), which contains the exemplars __Department_1, __MgrName_1, __MgrEmail_1, __EmployeeName_1 and __EmployeeName_1, partly described by the example stream graph. Each event in the stream contains elements that satisfy all the constraints described on the exemplars.

```
Class EmployeeAvailability{
com.egs.Dept __Department_1;
String __MgrName_1;
String __MgrEmail_1;
floa t __EmployeeName_1; }
```

Also, note that a stream description only contains OWL facts, i.e. assertions about different individuals (exemplar and non-exemplar) and how they are related. It does not define new concepts or properties, or extend the definitions of existing concepts and properties. A stream description only uses concepts and properties defined in the ontologies, such as the one in Figure 1.
Exemplar individuals and literals in a stream-graph act as existentially quantified variables. In a specific event, they are replaced by non-exemplar individuals or literals. For example, a specific event may contain a specific courtesy level, say 0.6, for a specific employee JDoe123. Figure 5 describes the semantics of an example event. The event itself may be represented as a serialized instance of the EmployeeCourteyness class defined earlier.

Figure 5. Example of an event in the stream

5.1 Matching a Stream to an Event-Pattern

Policies are defined in terms of event patterns. Our policy compiler works by generating a processing graph (or workflow) that produces a stream of events which match this event pattern. We now describe what is required for a stream of events to match an event pattern.

In order to define a match, we first define a pattern solution, which expresses a substitution of the variables in an event pattern. We then define the conditions for a match in terms of the existence of an appropriate pattern solution.

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. Variables may also be mapped to exemplar individuals and exemplar literals. For example, some of the mappings defined in a possible definition of \( \theta \) for the graph pattern, in the example policy is: \( \theta(?EmpName) = \_\_EmployeeName_1, \theta(?CurtLevel) = \_\_CourteynessLevel_1, \) 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 event-pattern \( EP(VS, GP) \), and a stream, \( S(SN, SD, SG) \). We define the event pattern, \( EP \) to be matched by the stream, \( S \), based on an ontology, \( O \), if and only if there exists a pattern solution, \( \theta \), defined on all the variables in \( GP \), such that following conditions hold:

- \( \theta(VS) \subseteq SD \), i.e. the stream contains at least those elements that the pattern says it must contain.
- \( SG \cup O \models_E \theta(GP) \) where \( O \) is the common ontology, and \( \models_E \) is an entailment relation defined between RDF graphs. For instance, \( E \) may be RDF, OWL-Lite, OWL-DLP, OWL-DL [2], etc. In other words, we see if it is possible to infer the substituted graph pattern from the stream-graph according to some well defined logical reasoning framework.

We represent this match as \( S \models_E EP \) to state that stream \( S \) matches event pattern, \( EP \) with a pattern solution, \( \theta \). One way of looking at the above definition is that the stream should have at least as much semantic information as described in the policy. Figure 6 shows how the EmpCurto_CallStream might match the EmployeeCourteynessEventPattern. The bold arrows show the variable substitutions. In order to make the match, some DLP reasoning based on subclass and inverse property relationships must be done. The dotted arrows denote the inferred relationships. For example, the tookCall relationship on \(_\_Employee_1\) is inferred, since tookCall is declared to be an inverse property of ofEmp. 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.

6 Policy Enforcement

Policy enforcement consists of two major steps: Policy Compilation and Stream Monitoring. Policy Compilation is the process of producing a processing graph that can generate a stream of events that match the pattern defined in the policy. Stream Monitoring is the process of monitoring the events on the matched stream and performing actions if certain conditions are true.

6.1 Policy Compilation

Figure 7 illustrates the architecture of the policy compiler. The compiler uses semantic descriptions of event sources and PEs to compose a processing graph using planning techniques. The planning is based on SPPL (Stream Processing Planning Language) [15], a specialized language for describing stream-based planning tasks.

There are two phases in this compilation process: pre-reasoning, that takes place off-line, before any policies are compiled, and semantic planning of individual Eagle policies. In the off-line pre-reasoning phase, a part of SPPL planning task called SPPL domain is created, after translating and performing DLP-reasoning on semantic descriptions of PEs and sources. The domain is then persisted and re-used for multiple policies. During the semantic planning phase, the policy is parsed, and OWL ontologies are used to
validate the policy. The event pattern in the policy is translated into a definition of the SPPL problem; i.e., it becomes a planning goal. The planner produces a plan consisting of actions that correspond to PEs and sources. The plan is constructed by recursively connecting components to one another based on their descriptions until the goal stream is produced. This plan is then deployed in the System S stream processing system.

If the number of sources and PEs is large, there may exist multiple alternative processing graphs for the same policy. Our planner uses a number of metrics to compare processing graphs, and returns only the processing graphs that are Pareto optimal, i.e., cannot be improved on any quality dimension without sacrificing in another. The metrics in use include resource utilization, security and privacy risks, and application-specific quality measures. The latter are computed using symbolic computation, under an assumption that PEs are capable of producing streams at fixed quality levels. Examples of such application-specific measures can be output video quality, image resolution, confidence in a forecast, etc. Resource metric is additive across the PEs and sources, and the risk metric is computed according to a soft-constraint-based security model.

6.1.1 Model of Event Sources and PEs

A event source is described as producing a single event stream. An example Event Source is shown in Figure 8. This source produces a stream that contains an exemplar called :VoIP_AudioSegment_1. The semantic graph describes the constraints on :VoIP_AudioSegment_1.

PEs are described in terms of the kinds of streams they require as input and the kinds of streams they produce as output. They are modeled in terms of graph transformations. The basic PE model is that it takes m input graph patterns, processes (or transforms) them in some fashion and produces n output graph patterns. An example PE is shown in Figure 9. This PE requires an input stream which contains ?VoIP_CallAudioSegment_1 that is associated with various constraints. The format of ?VoIP_CallAudioSegment_1 is an RTP packet. The PE extracts the start time and call channel information from the incoming RTP packet and computes the end time from the duration. It then puts this information along with the original ?VoIP_CallAudioSegment_1 on the output stream.

The semantic description of a PE gives a general, application-independent, description of the kinds of streams it requires and the kinds of streams it produces. In a given application (or processing graph), a specific set of input...
VoIP Call reached. The reasoner used by our system is Minerva [17].

sively, in a bottom-up fashion, on the triples in \( SG \) based on definitions in the ontology, \( O \), and generating additional triples about variables and exemplars, until a fix point is reached. The reasoner used by our system is Minerva [17].

As a result of reasoning, more triples, which describe further information about the exemplar individuals and literals defined in the stream graph, may be added to the expanded stream graph. For example, there are rules defined for DLP that allow making inferences based on subclass and subproperty relationships, symmetric, transitive and inverse property definitions, domain and range definitions, value restrictions, etc. These rules allow inferring many facts about a stream that are not contained in the stream description itself. For example, consider the stream produced by the event source in Figure 8. The expanded stream graph includes additional inferences like \( \_\text{VoIP\_Call}\_1 \) is of type \( \text{VolP\_Call} \) (based on subclass relationships).

6.1.3 Semantic Planning for a given Policy

An Eagle policy received by the planner is translated into an SPPL problem. The SPPL model yields a recursive formulation of the planning problem, where policy stream patterns are expressed similarly to PE input requirements, and PE outputs are described similarly to event sources. The planner operates in two phases: a presolve phase and a plan search phase. During the plan search phase the planner performs branch-and-bound forward search by connecting all compatible PEs to streams produced by already added PEs, or available from sources, and then selecting Pareto optimal solutions that match to specified goals. During the presolve phase the planner analyzes the problem structure and complements the search with efficient polynomial time algorithms when possible. Also during presolve analysis the sources that cannot contribute to the goals are eliminated, to help restrict the search only to relevant PEs and sources. After the presolve phase is done, the planner uses backward branch-and-bound search to construct optimal plans [15].

When the planner attempts to connect a stream to a PE as input, it tries to match the expanded stream-graph of the stream, \( SG' \), with the graph pattern, \( GP \), which describes a PE’s input requirement. This matching process is similar to the matching of a stream to an event pattern (describe in Section 5.1). Since reasoning has already been done, the matching reduces to a subgraph-matching problem after some variable substitutions. The planner attempts to find a solution, \( \theta \), such that \( \theta(GP) \) is a sub-graph of \( SG' \), i.e. \( \theta(GP) \subseteq SG' \). If it is able to find such a solution, then the graph-pattern is matched by the stream-graph.

6.2 Stream Monitoring

Once a plan, for a certain policy, is generated and deployed in the System S stream processing system, an additional Stream Monitor component is also instantiated for monitoring the events on the high-level stream and performing the policy actions if the associated conditions are satisfied. This Stream Monitor parses any event that comes on
the stream and extracts its contents. Since in our system, an event is a serialized Java object, it deserializes the object using a class definition of the event obtained from the planning process. Next, it checks the conditions in the policy. For some conditions, it may need to call other services in the system. In our system, these invocations are done using Java RMI and discovery of service references is done using a Naming Service. If a condition evaluates to true, then it performs the associated action by invoking the specified service with the appropriate parameters. Note that some of the parameters used in invoking the service may have been extracted from the event.

### 7 Compiler Performance

Scalability of our approach depends on the ability of the compiler to plan with large numbers of event sources and PEs. Since there are no standard datasets for stream composition, we evaluate compiler performance by measuring planning time on increasingly large randomly generated sets of PEs and sources. Experiments were carried out on a 3GHz Intel Pentium 4 PC with 500 MB memory.

For our experiments, we generated random DAG plans that contained PEs and sources with randomly constructed semantic descriptions, and then evaluated the time it took the compiler to construct these plans. We modeled event sources as PEs with no inputs. The DAGs were generated by distributing the nodes randomly inside a unit square, and creating an arc from each node to any other node that has strictly higher coordinates in both dimensions with probability 0.4. The link may reuse an existing output stream (if one exists) from the PE with probability 0.5, otherwise a new output stream is created. The resulting connected components are then connected to a single output node. Each link is associated with a randomly generated RDF graph from a financial services ontology in OWL that had about 200 concepts, 80 properties and 6000 individuals. The time taken to plan the DAGs are shown in Table 1. The table has columns for the number of streams and PEs in the generated graph, as well as time measurements for the online and offline phases of semantic planning.

The experiments show that there is a noticeable increase in planning time as the size of the problem increases. Our pre-reasoning approach, nevertheless, makes semantic planning practical by improving planner scalability. Although pre-reasoning is time consuming, the results of pre-reasoning can be shared between multiple policy compilations. Therefore, the actual response time of the planning system in practice is close to planning phase time. We can see that even for plan graphs involving 100 PEs, the compiler is able to produce the plan in less than 30 seconds, which is an acceptable performance.

### 8 Discussion and Experiences

The Eagle language allows high-level policies to be written. These policies are not written for a specific event or even a specific stream; they are written in terms of a pattern of event streams. This pattern is then matched by the policy compiler to a specific stream depending on which event sources and processing elements are currently available in the system. This approach allows policies to be independent of the actual raw events that may be available to the system at any point of time. Hence, even if the system configuration changes, and new kinds of events may be produced by the system, the policies can remain more or less stable.

It is possible, however, that as the system configuration changes, policies may need to be recompiled to take advantage of new kinds of events and processing elements, or to adapt to changes in event structure or semantics. The recompilation may result in the generation of a new processing graph for producing the high-level event streams specified in the policy. Even in this case, the policy itself does not change; just the way it is enforced changes.

The semantic nature of Eagle also allows policies to be written at a high level, independent of the actual entities in the system. Eagle policies are based on event streams defined using OWL/RDF graph patterns. This allows policies to identify entities based on their semantics. Many existing policy languages like Ponder [6] and DEN-ng [16] identify entities based on roles. The graph pattern based approach generalizes the role-based approach and allows expressing a rich set of conditions for identifying the entity. For example, the policy in Section 4 identifies all employees in EGS who are in VOIP calls to customers, and their managers. It also describes the conditions to be checked and the actions to be taken in terms of these employees and managers. Hence, the policy is independent of the actual employees and their managers in the company.

A key aspect of our framework is that all policies are defined based on terms described in domain ontologies. This simplifies the task of policy writers since the vocabulary is formally defined, using description logic, in OWL ontologies. At the same time, it brings up interesting issues with regard to policy maintenance as the ontologies evolve. In any large-scale dynamic system, the ontologies will change in order to define new terms (concepts, properties and in-

<table>
<thead>
<tr>
<th>PEs</th>
<th>Streams</th>
<th>Prereason</th>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>15</td>
<td>4.66</td>
<td>0.15</td>
</tr>
<tr>
<td>20</td>
<td>40</td>
<td>10.29</td>
<td>0.22</td>
</tr>
<tr>
<td>30</td>
<td>119</td>
<td>27.49</td>
<td>0.76</td>
</tr>
<tr>
<td>40</td>
<td>190</td>
<td>38.48</td>
<td>1.21</td>
</tr>
<tr>
<td>50</td>
<td>288</td>
<td>59.17</td>
<td>2.26</td>
</tr>
<tr>
<td>100</td>
<td>1,204</td>
<td>233.03</td>
<td>28.73</td>
</tr>
</tbody>
</table>
individuals), or to modify the definition of existing terms. These changes need to be percolated to policies that use these terms. This is a difficult problem, that must be solved in an automatic or semi-automatic manner to allow scalable policy maintenance.

We have experimented with our framework in a number of different domains including the call-center domain, a disaster recovery domain and a real-time traffic monitoring domain. For each domain, we developed domain ontologies and PEs, and used real or simulated event sources. We also specified high-level event stream patterns of interest and defined policies based on these stream patterns. From this experience, we are confident that our policy language and framework can be easily used in different domains. One of our main experiences from the use of this framework in different domains is the need for tools that can support collaborative ontology and policy editing and maintenance. Such tools should provide source control and versioning capabilities for ontologies and policies.

9 Related Work

Bandara et al [3] describe a goal based approach to policy refinement based on a formal representation of a system using Event Calculus. The Power prototype [12] allows refining policies in a semi-automatic manner. Our work focuses on a different problem, which is a way of specifying policies based on high level event streams, and automatically composing processing graphs for producing these streams.

One area of related work is in stream query languages and stream processing architectures. Systems like Aurora [1], TelegraphCQ [5], etc. support processing on streams based on extensions to SQL. However, most of these systems only operate on unstructured data and also do not consider the automatic composition of general operators for producing the high level streams.

Recently there has been interest in composing web services based on their WSDL interfaces, semantically enhanced interface specs (like OWL-S) or other service models (see [8] for a survey). Our model extends these ideas by describing the inputs and outputs using semantic graphs and using a powerful semantic graph-matching approach for connecting components.

10 Conclusions

We have proposed a semantics-based approach to specifying obligations policies. These policies specify high-level event patterns and actions to be performed for matching events. The main features of our approach are the ontology-based high-level policy specification and the use of semantic graph based planning for constructing optimized processing graphs that can produce high-level events. We have integrated the policy framework into the System S stream processing system.

We are working on extending the policy framework to detect conflicts in invoked actions through richer semantic action models, and to determine dependencies between policies. We are also extending the planning model to allow actions to be expressed as goals. Extensions to the policy language include specifying composite policies, as well as specifying authorization and delegation policies.

Another on-going work is on policy authoring tools. While Eagle is expressive and extensible through ontology extensions, authoring policies requires knowledge of the ontologies. We are developing tools to facilitate this authoring process by using technologies of ontology browsing/editing and of natural language processing.

References