AutoSLAM - A Policy-driven Middleware for Automated SLA Establishment in SOA Environments

Mohan Baruwal Chhetri, Quoc Bao Vo and Ryszard Kowalczyk
Faculty of Information & Communication Technologies
Swinburne University of Technology
Hawthorn, Australia
{*mchhetri, bvo, rkowalczyk*}*@swin.edu.au*

***Abstract*—We propose a policy-based framework for the automated establishment of Service Level Agreements (SLAs) in Service Oriented Architecture (SOA) environments such as the cloud. The novelty of our proposed framework is the support for multiple SLA interaction models, giving entities the flexibility to choose the one that is most appropriate in a given context, while simultaneously participating in multiple concurrent SLA interactions using different interaction models. As part of the framework, we present AutoSLAM, a policy based middleware that uses policies described with the use of WS-SLAM, our new WS-Policy extension, that provides a domain-independent policy specification language for specifying conditional assertions over the supported SLA interaction models. We have implemented an AutoSLAM proof-of-concept prototype and evaluated it for purchasing computing resources on Amazon EC2 under different contexts.**

***Keywords*-SLA interaction models, policies, strategy assertions, context assertions, interaction protocol, policy-based middleware**

I. INTRODUCTION

When consuming or providing services in dynamic SOA environments such as the cloud, entities have to first reach agreements over the service usage terms and conditions. Given the diversity and dynamism of the cloud environment, using just a single interaction model for SLA establishment may not be appropriate in all scenarios and contexts, and service consumers and providers can benefit from supporting multiple interaction models. Some of the popular SLA interaction models include fixed-price selling, auctions, commodity markets (spot trading, forward contract, futures contract), and one-to-one and one-to-many negotiations. Support for multiple interaction models gives them the flexibility to choose the one that is most appropriate in a given context, while simultaneously participating in multiple concurrent SLA interactions using different interaction models.

In each SLA interaction model the interactions between the participating entities are governed by an interaction protocol which defines the "rules of procedure" for the conversation. Depending upon the interaction protocol used, the entities can use different decision-making strategies to try and reach an agreement. For example, if the interaction model is an auction based on the *first-price sealed-bid auction protocol*, then all the bidders submit a single sealed bid. For them, the decision-making strategy has to determine the best bidding price, while for the service provider, it has to determine the acceptable bid price. Similarly, if the entities are involved in bilateral negotiation using the *alternating offers protocol*, the decision-making strategy has to determine what initial offer to make, what counter-offer to make, when to accept an offer and when to terminate negotiation. Thus each SLA interaction model uses a specific interaction protocol and one or more decision-making strategies.

We propose a policy-based approach for supporting multiple SLA interaction models. Our policies are based on the popular *condition-action* rules paradigm, and allow the specification of conditional assertions over the supported SLA interaction models. The condition part captures the context surrounding the SLA interaction, while the action part specifies the executable SLA interaction model. A central part of our approach is our light-weight AutoSLAM (Automated SLA Management) middleware. In AutoSLAM, we formally specify SLA interaction policies in WS-SLAM, our novel extension of the WS-Policy framework [5]. In our proof-of-concept prototype, we make use of the Drools Rule Engine [6] for the evaluation of the WS-SLAM policies. We do this by first parsing the WS-SLAM policies into Drools rules which are then fed to the Drools Rule Engine. The AutoSLAM middleware intercepts each incoming request and determines which SLA interaction model to use for SLA establishment.

The rest of this paper is organized as follows. Section II discusses the Amazon EC2 service which we use to motivate our research. Section III elaborates on the AutoSLAM middleware architecture. Section IV gives an overview of the WS-SLAM policy language used to specify SLA interaction rules. Section V discusses the implementation of the proof-of-concept prototype of the AutoSLAM middleware using the Drools Rule Engine. Section VI presents an evaluation of the AutoSLAM middleware using the scenario of purchasing computing resources from Amazon EC2 under different contextual conditions. Finally Section VIII concludes the paper with an outlook of future work.

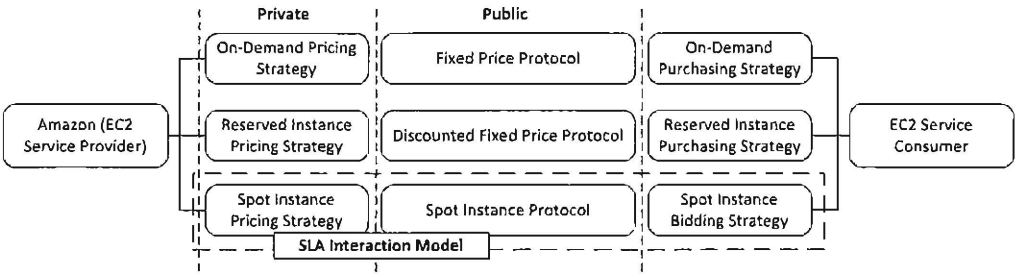


Figure 1. Amazon EC2 supports multiple SLA interaction models

II. Motivating Scenario - Amazon EC2

We consider the case of Amazon Elastic Cloud Compute (EC2) as a motivating scenario for our research work.

A. Amazon EC2 – Service Provider

The Amazon EC2 service offers cloud computing resources to its customers on-demand. At any given time, it provisions computing resources to thousands of customers across 190 countries simultaneously - a clear indication of its diverse customer base. These customers range from individual developers and startups to government agencies and large enterprises such as Amazon.com[1], the NY Times[2] and ESPN[3]. Some of these customers require dedicated resources with strict QoS requirements for their mission-critical applications, while others seek cheap computing options. Amazon offers its customers three different purchasing models (which we refer to as SLA interaction models) that they can choose from to purchase computing resources.

- **On-Demand Instances** – this model lets customers pay for compute capacity by the hour with no long-term commitments or upfront costs. Consumers can increase or decrease compute capacity on demand and pay the fixed hourly rate for the instances used.
- **Reserved Instances** – this model lets customers pay a small one-time, upfront payment for an instance, reserve it for a fixed period of time (one year or three years), and then pay a significantly lower fixed rate for each hour that the instance is used.
- **Spot Instances** – this model allows customers to bid for unused Amazon EC2 capacity. Amazon determines the *Spot Price* based on the bids received and the quantity of unused/idle resources. Customers have access to the requested resource as long as their bid price is above the spot price, which idirectly depends upon supply and demand.

Each of these three models has a specific interaction protocol. The *on -demand instance* model uses the *fixed-price protocol*, while the *reserved instance* model uses the *discounted fixed-price protocol*. With both these models, the customers have no flexibility in terms of the price they pay for the resources even though they can choose the instance type that meets their specific configuration requirements. However, they do have guaranteed and uninterrupted access to the computing resources. The *spot instance* model uses the *spot instance protocol*, which is based on a *uniform price, sealed-bid, market-driven auction. Uniform price* implies that all bidders pay the same price for the resource if they are successful in their bid. *Sealed bid* means that the bids are unknown to other participants and *market-driven* means that the spot price is set according to the client's bids. Using this model, consumers bid the maximum price they are willing to pay for the resource. If they are successful, they have access to the resource and are able to use it until either they choose to terminate it or the new Spot Price becomes higher than their bid. As the service provider, Amazon publicly advertises these three SLA interaction models. The corresponding interaction protocols are public knowledge and every participant has to abide by these rules. It has its own internal strategies to determine the fixed prices of the on-demand and reserved instances, and the dynamic prices of the spot instances.

B. Amazon EC2 Consumer

Consumers can choose any one of the three purchasing models to purchase computing resources on Amazon EC2. The chosen purchasing model depends upon their specific situation. As a simple illustrative example, let us consider the scenario where an entity executes jobs on behalf of its customers on the Amazon EC2 infrastructure. In order to to do so, it rents the computing resources as and when required. Each time the entity receives a request, it has to decide how many instances to rent and whether to purchase an on-demand instance or to go for a spot-instance. If purchasing spot instances, it also has to determine the best bid value to use. Depending upon the context, the entity can use a number of different strategies to rent the resources.

Let us look at four possible interaction contexts and the corresponding strategies that could be used to purchase

[1] http://www.amazon.com
[2] http://www.nytimes.com
[3] http://espn.go.com

computing resources from Amazon. The rules for strategy selection based on context take the form 'if *condition* then *action* and can be described as follows – *under a certain context (as specified by the condition part), use a specific strategy (as specified by the action part).* Strategies 2, 3 and 4 are currently being used by Amazon EC2 customers as explained in the video Deciding on Your Spot Bidding Strategy[4].

- **Scenario 1** – *Context:* Client wants immediate access to the resource. *Strategy:* Use on-demand purchasing model to purchase instance.

$$S_1 : P = P^i_{od} \quad (1)$$

where i denotes instance-type, P^i_{od} denotes on-demand price.

- **Scenario 2** – *Context:* Client wants to minimize the computing cost and job completion time is not a constraint. *Strategy:* Use spot-instance purchasing model and bid around the reserved instance usage price.

$$S_2 : P_{max} = \kappa \cdot P^i_r, \text{ where } 1 \leq \kappa \leq \frac{P^i_{od}}{P^i_r} \quad (2)$$

where κ is a constant, i denotes instance type, P^i_r denotes reserved instance price and P^i_{od} denotes on-demand price.

- **Scenario 3** – *Context:* Client wants to complete the job as quickly as possible and minimize the cost. *Strategy:* Use spot-instance purchasing model with *price history momentum* strategy which takes into account the previous trends in the pricing history.

$$S_3 : P_{max} = \kappa \cdot P^i_{avg_n}, \text{ where } \kappa \leq 1 \quad (3)$$

where κ is a constant and $P^i_{avg_n}$ is the average spot instance price for the last n hours.

- **Scenario 4** – *Context:* Client wants uninterrupted access to the resource for a long duration, but at a price lower than the on-demand price. *Strategy:* Use spot-instance purchasing model and bid a maximum price which is significantly higher than the on-demand price.

$$S_4 : P_{max} = \kappa \cdot P^i_{od}, \text{ where } \kappa > 1 \quad (4)$$

where κ is a constant and P^i_{od} is the on-demand price for the instance type i.

The above four simple scenarios show how consumers of the Amazon EC2 service can use different purchasing models and different decision-making strategies to purchase the same resources.

III. AutoSLAM Reference Architecture

In this section we present the AutoSLAM reference architecture that provides the foundation to build policy-driven automated SLA management systems such as the one we have implemented to purchase computing resources from Amazon EC2. The main benefit of our model is two-fold. On the one hand, it allows the reuse of existing elements of automated SLA establishment so that they can be freely integrated into the system. On the other hand, the model is flexible enough to adapt to the SLA interaction model that is best suited for each SLA interaction scenario.

A. Reference Architecture

We base our reference architecture on the XACML (eXtensible Access Control Markup Language) architecture [7]. The XACML framework is an authorization and access control framework that defines a declarative access control policy language and a processing model to evaluate authorization requests according to the rules defined in XACML policies. An XACML request is usually made by a *subject* to perform a certain *action* on a given *resource*. The output of the XACML policy processing model is a *permit* or *deny* decision based on which the authorization or access is approved or disapproved.

The AutoSLAM framework is a framework for the automated establishment and management of SLAs in SOA environments such as the cloud. The framework defines a declarative policy language WS-SLAM for specifying the supported SLA interaction models. It also defines a policy processing model which can evaluate incoming service requests (and the relevant context) against the SLA interaction policies to determine the most appropriate interaction model to instantiate. The main components of AutoSLAM are shown in Figure 2. The greyed box shows the AutoSLAM extension to the XACML architecture.

- **Policy Enforcement Point (PEP).** PEP is the entry point to the AutoSLAM policy processing middleware. Initially it receives the service request and forwards it to the Policy Decision Point (PDP). It then interprets the decision of the PDP and instantiates the appropriate SLA interaction model as shown in Figure 2.
- **Policy Decision Point (PDP).** PDP evaluates the incoming request and the relevant context against all the policies that are applicable in the current context. The outcome of the evaluation is the selected interaction model which is sent back to the PEP.
- **Policy Access Point (PAP).** PAP makes available to the PDP all the policies and rules that are in the policy database.
- **Policy Information Point (PIP).** PIP retrieves all the information about the relevant context surrounding the current service request.
- **Policy Administration Point (PAdP).** Policy authors manage the policies in the policy database through the PAdP. They can add new policies, and remove or edit existing policies to update the knowledge base of the AutoSLAM decision model.

[4] http://www.youtube.com/embed/WD9N73F3Fao

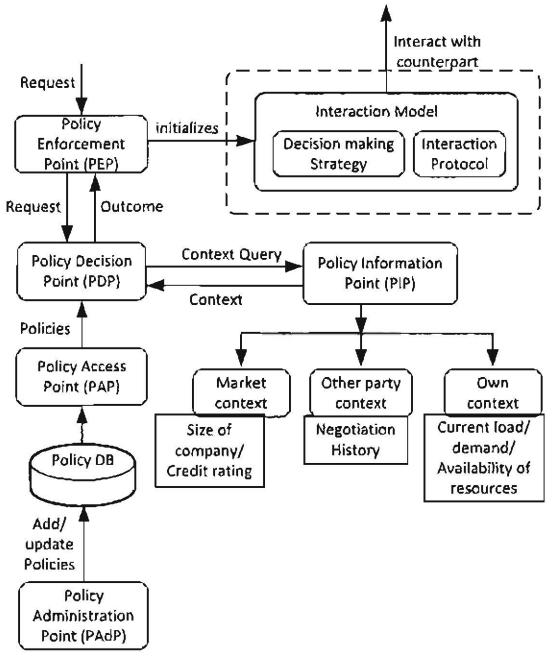


Figure 2. AutoSLAM Reference Architecture

As shown in Figure 2, when an entity initiates the SLA interaction process or responds to a request, the PEP forwards the request it receives to the PDP, which in turn retrieves all the current policies from the PAP, evaluates them against the contextual information retrieved from the PIP, and based on the evaluation, selects the appropriate SLA interaction model with the corresponding decision making strategy and interaction protocol. It then forwards this decision to the PEP which instantiates the selected SLA interaction model. Depending upon whether it is a one-round interaction or multi-round interaction, the interaction module exchanges messages with the SLA counterpart to try and obtain an outcome. If a common agreement is reached during the interaction, then the policy engine returns a decision to form a SLA and the service entity is provided access to the service. If an acceptable outcome is not achieved, then the PEP returns a failure decision.

IV. WS-SLAM Policy Language

In this section, we discuss the various aspects of the WS-SLAM policy language. We first provide a theoretical foundation for the proposed language followed by a discussion of the core elements of the language using a simple example. We refer readers to [1] and [2] for a more formal description of our policy model.

A. Theoretical foundation of WS-SLAM

We base the SLA interaction policies on the well-founded AI technique of *reactive planning* [4], which denotes a group of techniques for action selection by autonomous agents. A key feature of reactive planning is that they are highly suited for dynamic and unpredictable environments such as the cloud. One of the ways to represent reactive plans is the *Condition-Action (CA) rules paradigm*. A condition action rule (or if-then rule) is a rule of the form:

if `condition` **then** `action`

Such rules are also referred to as *productions* or *production rules*. The meaning of the rule is obvious - if the condition holds, perform the action. In the context of SLA interactions, the conditions refer to the contextual conditions and the actions refer to the executable SLA interaction models. The WS-SLAM policy language makes use of these concepts as follows. The **Condition:** captures the context surrounding the SLA interaction in the form of context assertions which are combined using the logical connectives `and`, `or` and `not`. The **Action:** refers to the executable SLA interaction model that is to be used to reach an agreement. It could refer to the executable strategy to be used in the interaction (strategy assertion), or the applicable interaction protocol (IP assertion) that has to be followed.

B. Core elements of WS-SLAM

Normal form of WS-Policy expression

```
<wsp:Policy>
  <wsp:ExactlyOne>
    (<wsp:All>(<Assertion... > ...<Assertion/> ) *
</wsp:All> )*
  </wsp:ExactlyOne>
</wsp:Policy>
```

Table I
NORMAL FORM OF WS-POLICY EXPRESSION

WS-SLAM is designed as an extension to the WS-Policy language [5] which allows Web Services to advertise their capabilities, requirements and general characteristics in a flexible and extensible grammer using XML format. In WS-Policy, a policy is essentially a collection of policy alternatives. Each policy alternative is in turn a collection of policy assertions. The policy assertion represents a specific requirement, capability, constraint or behaviour of the service. The policy assertions are not provided by the WS-Policy specification but instead can be provided for specific domains. WS-Policy operators (`wsp:Policy`, `wsp:All`, `wsp:ExactlyOne`) are used to group policy assertions into policy alternatives.

WS-SLAM provides a domain independent assertion model which combines the context assertions and strategy assertions in condition-action rules. The core element of WS-SLAM is the `Rule` element. Each rule has an `If` part which captures the contextual conditions specified in the form of Context Assertions which can be combined using the `and`, `or` and `not` logical operators. The `Then` part specifies the executable strategy that is to be invoked when the conditional part is satisfied. Each rule is identified by a

unique name and can have a number of optional attributes to provide additional information. The XML infoset representation of a WS-SLAM `Rule` is Table II. A number of WS-SLAM rules can be combined into a single policy alternative using the `All` policy operator.

WS-SLAM Rule

```
<slam:Rule name="xs:string" type="xs:string">
  <slam:RuleAttribute name="xs:string"
value="xs:string"/>*
  <slam:If>
    (<slam:Context/> |
     <slam:AndConditionalElement/> |
     <slam:OrConditionalElement/>)
  </slam:If>
  <slam:Then>
    <slam:Strategy.../>
  </slam:Then>
</slam:Rule>
```

WS-SLAM policy expression

```
<wsp:Policy>
   <wsp:All>
     (<slam:Rule.../>)*
   </wsp:All>
</wsp:Policy>
```

WS-SLAM Context Assertion

```
<slam:Context identifier="xs:string"
     objectType="xs:string">
   (<slam:FieldConstraint/> |
    <slam:AndConstraintConnective/> |
    <slam:OrConstraintConnective/>)
</slam:Context>
```

WS-SLAM Strategy Assertion

```
<slam:Strategy name="xs:string" >
   (<slam:StrategyAttribute name="xs:string"
      value="xs:string" />)*
</slam:Strategy>
```

Table II
WS-SLAM SYNTAX (NORMAL FORM)

A WS-SLAM rule makes use of three key assertions to declaratively specify the supported SLA interaction models. They are:

- **Context Assertion:** A context assertion captures the specific condition/s that determine the SLA interaction model to use in response to a service request.
- **Strategy Assertion:** A strategy assertion is a declarative specification of an executable strategy. There are two ways in which a strategy assertion can be made over the parametric strategy function:
  - *By reference* - in this case the WS-SLAM merely refers to an externally defined SLA interaction model that is to be invoked if the context holds true.
  - *By reference with values* - in this case, the strategy assertion not only refers to the externally defined strategy but also specifies the specific values for the strategy parameters.
- **Interaction Protocol Assertion:** An interaction policy assertion specifies the list of interaction protocols supported for SLA establishment.

In WS-SLAM, the context is represented by the `Context` element which can have an unrestricted number of fields (or context attributes). Constraints can be specified on the values these fields can take by using the `FieldConstraint` element. Multiple `FieldConstraint` elements can be combined using the logical `and` and `or` connectives. Atomic context assertions can be combined to compose complex context assertions using the `<slam:AndConditionalElement/>` and the `<slam:OrConditionalElement/>`. The XML infoset representation of the strategy assertions and the Interaction Protocol assertions are shown in Table II.

C. A basic example of WS-SLAM

Figure 3 shows a simple example of a policy document, which is compliant with the WS-SLAM policy language specification. In order to improve readibility, we have removed the namesspace declarations of both WS-Policy and WS-SLAM. The example policy shows three rules, where each rule specifies the SLA interaction model and decision making strategy to use in a given context. This example policy defines rules to make decisions for purchasing instances on Amazon EC2.

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:Policy>
 <tns:All>
  <slam:Rule name="Rule 1">
   <slam:If>
    <slam:Context identifier="context" objectType="Context" >
     <slam:AndConstraintConnective>
      <slam:FieldConstraint field-name="uninterruptedAccess">
       <slam:LiteralRestriction value="yes" evaluator="=="/>
      </slam:FieldConstraint>
      <slam:FieldConstraint field-name="minCost">
        <slam:LiteralRestriction value="yes" evaluator="=="/>
      </slam:FieldConstraint>
      <slam:FieldConstraint field-name="accessInFuture">
        <slam:LiteralRestriction value="true" evaluator="=="/>
      </slam:FieldConstraint>
     </slam:AndConstraintConnective>
    </slam:Context>
   </slam:If>
   <slam:Then>
    <slam:Strategy name="BlockPurchasingStrategy" />
   </slam:Then>
  </slam:Rule>
  <slam:Rule name="Rule 2">
   <slam:If>
    <slam:Context identifier="context" objectType="Context" >
     <slam:OrConstraintConnective>
      <slam:AndConstraintConnective>
       <slam:FieldConstraint field-name="minCost">
        <slam:LiteralRestriction value="yes" evaluator="=="/>
       </slam:FieldConstraint>
       <slam:FieldConstraint field-name="minCompletionTime">
        <slam:LiteralRestriction value="yes" evaluator="=="/>
       </slam:FieldConstraint>
      </slam:AndConstraintConnective>
     </slam:OrConstraintConnective>
    </slam:Context>
   </slam:If>
   <slam:Then>
    <slam:Strategy name="PriceMomentumStrategy" />
   </slam:Then>
  </slam:Rule>
 </tns:All>
  .........
</tns:Policy>
```

Figure 3. Example WS-SLAM Policy

V. AUTOSLAM PROTOTYPE IMPLEMENTATION

In order to validate our policy-based approach, we have implemented a proof-of-concept prototype of the policy middleware for automated SLA establishment. It has been implemented as an *Automated Purchasing Agent* for purchasing instances on Amazon EC2. It comprises of three key components

- WS-SLAM2DrlParser - A parser which parses WS-SLAM policies into Drools[5] rules.
- An embeddable Drools Rule Engine - which evaluates the incoming requests and the relevant context against the predefined WS-SLAM policies.
- A library of executable SLA interaction models that are used to purchase instances from Amazon EC2, including the decision-making strategies given in the motivating scenario in Section II.

A. WS-SLAM2Drl Parser

In the current version of the AutoSLAM middleware, we have implemented a WS-SLAM2Drl Parser, which parses WS-SLAM policies and rules into Drools rules as shown in Figure 4. The parser makes use of mapping rules to map from WS-SLAM constructs to the Drools constructs. As illustrated in the figure, a parser may be implemented to parse the WS-SLAM rules into Jess format in which case the JESS Rule Engine could be used to evaluate the request and relevant context against the rules.

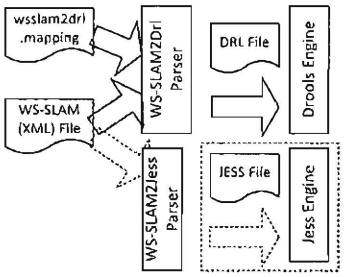


Figure 4. WS-SLAM2Drl Parser

Table III shows the correspondence between the main constructs in WS-SLAM and Drools. The WS-SLAM policy specification is a light-weight language which is intended to be used by non-technical policy authors and hence does not support low-level executable code expressions. On the other hand, in Drools the action part refers to executable actions and supports the insertion of executable Java code. Hence, there has to be a mapping file (*wsslam2drl.mapping*) which can map the abstract rules and constructs in WS-SLAM to more concrete executable classes and objects in Drools. A technical expert has to define the mapping between the abstract Context and Strategy names in WS-SLAM to the corresponding executable Java implementations which are inserted into the Drools file. Figure 5 shows a snippet of the mapping from WS-SLAM to Drools. Figure 6 shows the output of the WS-SLAM2Drl Parser when the input is the file shown in Figure 3.

[5]http://www.jboss.org/drools

Construct	Description	WS-SLAM	Drools
Rule	if CONDITION then ACTION end	IF CONDITION </IF> <THEN> ACTION </THEN>	rule name when CONDITION then ACTION end
Formula	Conditional part of the rule	atomic | and | or | not	atom | and | or | not | pattern
Atomic	Atomic member of formula	<CONTEXT> <CONTEXT> <STRATEGY> </STRATEGY>	
Action	Action refers to externally defined functions or methods		Objects imported into rule package and invoked from within rule

Table III
CORRESPONDENCE BETWEEN LANGUAGE CONSTRUCTS

```
# name of the properties file
drools-file=./conf/output.drl

# package-name - can be anything
package-name=au.edu.swin.cb

# Drools Rule Engine Implementation class
DroolsRuleEngine au.edu.swin.cb.DroolsRuleEngine;

# Context Assertion Mapping
Request=au.edu.swin.cb.context.Request;
Context=au.edu.swin.cb.context.Context;

# Strategy Assertion Mapping
OnDemand=au.edu.swin.cb.strategies.OnDemandStrategy;
PriceMomentum=au.edu.swin.cb.strategies.PriceMomentumStrategy;
MinimizeInterruption=
     au.edu.swin.cb.strategies.MinInterruptionStrategy;

# Executable code for the Strategy Assertions
OnDemand_Constructor=OnDemandStrategy odS = new OnDemandStrategy();
OnDemand_EntryPoint=DroolsRuleEngine.getInstance().addStrategy(odS);

PriceMomentum_Constructor=
     PriceMomentumStrategy pmS = new PriceMomentumStrategy();
PriceMomentum_EntryPoint=
     DroolsRuleEngine.getInstance().addStrategy(pmS);
```

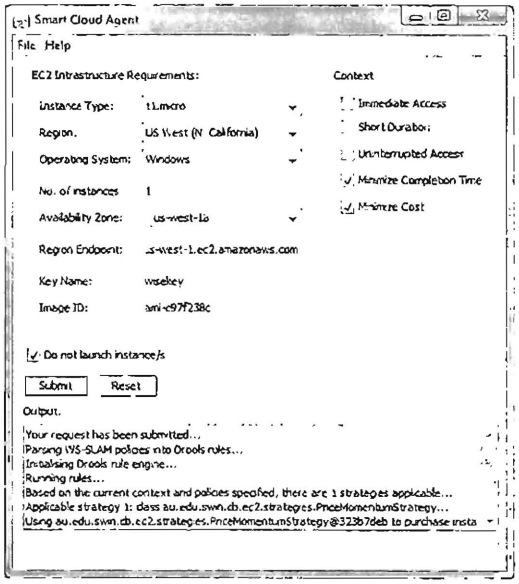
Figure 5. WS-SLAM to Drl Mapping

VI. VALIDATION

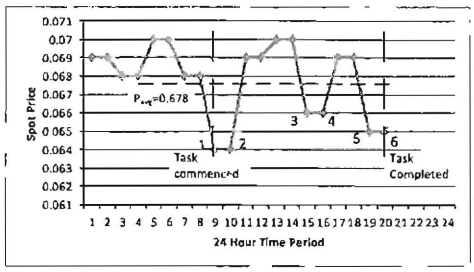
We have used the Amazon EC2 scenario described in Section II to validate our AutoSLAM policy-driven middleware for automated SLA establishment. In this scenario end-consumers submit their requests to the *Smart Cloud Agent* whenever they have a job to process on EC2. They know which instance type they want and how many instances of it. They have preferences and constraints over the task completion time and the total cost payable, which they specify when they submit their request. The cloud agent (policy engine) evaluates each incoming request against its policy base and determines the most appropriate purchasing model as well as the best bidding strategy. It then initiates the interaction with Amazon EC2 and if purchasing on-demand

```
package au.edu.swin.cb.core

import au.edu.swin.cb.context.Context;
import au.edu.swin.cb.strategies.BlockPurchasingStrategy;
import au.edu.swin.cb.strategies.PriceMomentumStrategy;
import au.edu.swin.cb.strategies.OnDemandStrategy;
import au.edu.swin.cb.strategies.CostOptimizationStrategy;
import au.edu.swin.cb.DroolsRuleEngine;

rule "Rule 1"
when
   context : Context( (uninterruptedAccess == "yes") &&
   (minCost == "yes") && (accessInFuture == "true") )
then
   BlockPurchasingStrategy tpmS = new BlockPurchasingStrategy();
   DroolsRuleEngine.getInstance().addStrategy(tpmS);
end

rule "Rule 2"
when
   context : Context( ((minCost == "yes") &&
   (minCompletionTime == "yes")) )
then
   PriceMomentumStrategy pmS = new PriceMomentumStrategy();
   DroolsRuleEngine.getInstance().addStrategy(pmS);
end

rule "Rule 3"
when
   context : Context( (immediateAccess == "yes") &&
   (duration == "short") )
then
   OnDemandStrategy odS = new OnDemandStrategy();
   DroolsRuleEngine.getInstance().addStrategy(odS);
end
```

Figure 6. Auto generated Drools Rule File



(a) Client UI



(b) Spot Price History

Figure 7. Smart Cloud Agent - Automated Selection of EC2 Purchasing Model

instances, initiates the process and starts up the instance. If going for spot-instances, it starts bidding for resources using the selected bidding strategy. If the bid is successful, it starts up the specific instance.

For the input request shown in Figure 7(a), the policy engine chooses the spot instance purchasing model and chooses the price momentum strategy. The policy engine computes the maximum bidding price as $0.678 based on the past 12 hours spot pricing history which is obtained by querying the Amazon EC2 web service. With the bid price of $0.678, the user is able to start and use the resource when the bid price is above the spot price as shown in the graph in Figure 7(b). The Smart Cloud Agent is able to make purchasing decisions on behalf of the end-users based on the domain knowledge captured in the form of strategy policies. Different mechanisms can be used to resolve deadlocks that result when more than one rule (and hence SLA interaction model) is applicable. The simplest solution is to choose the first applicable rule or a randomly selected rule. Alternatively, rules can be assigned individual scores and then the rule with the highest score is executed. Alternatively, we can choose the rule that satisfies the highest number of context attributes.

VII. Related Work

There are several research proposals on policy-based specification of decision-making strategies for automated negotiation. [13][14], [15], [16] and [17] propose the use of declarative rules to capture the decision-making strategies. [16] and [17] do not provide any formal models or concrete examples to illustrate how this can be done. The main limitation of defining strategies declaratively via rules is that while it is sufficient for simple strategies, it is not straightforward for complex strategies which could be based on a number of different approaches such as game-theoretic approaches [8][3], heuristic approaches [9][10] and evolutionary approaches [11]. There has to be a tradeoff between the expressive power of the policy language and the ease of usage. In [18], the authors have proposed the declarative specification of decision-making strategies using an extension of the WS-Policy specification language where the decision-making strategies are defined as parametric functions where the parameter values are specified via the strategy policy. While [19] has proposed the support for multiple negotiation models for a resources market, to the best of our knowledge, our paper is the first to support multiple interaction models through the use of a policy-based

approach. We allow the policy authors to specify which strategy to use under different contexts, so that the policy engine can autonomously make decisions that conform to these policies at run-time. Our approach also enables reuse of existing research results since we allow externally defined strategies to be referred to within our policies and separate the strategy reference from the actual implementation.

VIII. Conclusion

In this paper, we have presented AutoSLAM - our novel policy-based framework for the automated establishment of SLAs in open, diverse and dynamic environments such as the cloud. We have used the Amazon EC2 example to illustrate why service entities may require flexibility to choose the most appropriate SLA interaction model in a given context while at the same time participating in multiple concurrent interactions using different SLA interaction models. We have implemented a proof-of-concept prototype of our AutoSLAM middleware which makes use of WS-SLAM, our extension of the WS-Policy framework, to specify the SLA interaction policies. In our prototype, we have used the standard Drools Rule Engine to evaluate incoming requests against the WS-SLAM policies.We have validated our framework by implementing the Smart Cloud Agent, which evaluates incoming requests for computing resources on Amazon EC2 and the context surrounding the request against the SLA interaction policies to determine the best purchasing option.

While we have developed policy models for capturing preferences over the service usage terms and conditions [1], and for supporting multiple SLA interaction models [2], the two models are currently independent of each other. As future work the AutoSLAM middleware will be extended to combine the different types of policies i.e. preference policies, strategy policies and interaction protocol policies can be combined to provide a unified policy framework for automated SLA establishment in dynamic and diverse SOA environments such as the cloud.

Acknowledgment

This work was partially funded by the Service Delivery & Aggregation Project within the Smart Services CRC which is proudly supported by the Australian Federal Government's CRC Grant Program.

References

[1] M. Baruwal Chhetri, Q. B. Vo, R. Kowalczyk *A Flexible Policy Framework for the QoS Differentiated Provisioning of Services*, The 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID-11), Newport, California, USA, May 23-26, 2011

[2] M. Baruwal Chhetri, Q. B. Vo, R. Kowalczyk *Policy-Based Automation of SLA Establishment for Cloud Computing Services*, In Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID-12), Ottawa (Canada), 13-16 May 2012

[3] N. R. Jennings, P. Faratin, A. R. Lomuscio, S. Parsons, C. Sierra, and M. Wooldridge, *Automated Negotiation: Prospects, Methods and Challenges*, International Journal of Group Decision and Negotiation, 10 (2). pp–199-215, (2001)

[4] R. J. Firby *An Investigation into Reactive Planning in Complex Domains*, In Proceedings of the Association for the Advancement of Artificial Intelligence (AAAI) Conference, 1987

[5] *Web Service Policy 1.5 Framework*, http://www.w3.org/TR/ws-policy/

[6] *Drools Rule Engine*, http://www.jboss.org/drools

[7] M. Verma, *XML Security: Control Information Access with XACML*, available online at http://www.ibm.com/developerworks/xml/library/x-xacml/

[8] K. Binmore, and V. Nir, *Applying game theory to automated negotiation*, NETNOMICS, Vol.1, No.1 (1999)

[9] P. Faratin, C. Sieera, and N. R. Jennings: *Using similarity criteria to make trade-offs in automated negotiations*, Artificial Intelligence, Vol. 142, No. 2, pp–205-237 (2002)

[10] R. Kowalczyk, *Fuzzy e-negotiation agents*, Soft Computing, Vol 6. No. 5, pp–337-347 (2002)

[11] S. S. Fatima, M. Wooldridge, and N. R. Jennings, *A comparative study of game theoretic and evolutionary models of bargaining for software agents*, Artificial Intelligence Review, Vol. 23, No. 2, pp–187-208 (2005)

[12] N. Jain, I. Menache, O. Shamir, *On-demand or Spot? Learning-based Resource Allocation for Delay-Tolerant Batch Computing*, available online at http://research.microsoft.com/en-us/um/people/navendu/papers/Ibr infocom.pdf, accessed on 25 November 2011

[13] H. Gimpel, H. Ludwig, A. Dan and B. Kearney, *PANDA: Specifying Policies for Automated Negotiations of Service Contracts*, In Proceedings of ICSOC 2003, pp. 287-302 (2003)

[14] T. Skylogiannis, G. Antoniou and N. Bassiliades, *DR-NEGOTIATE - A System for Automated Negotiation With Defeasible Logic-Based Strategies* in Proceedings of IEEE International Conference on e-Technology, e-Commerce and e-Service, pp. 44-49 (2005)

[15] H. Li, S. Y. W. Su, and H. Lam, *On Automated e-Business Negotiations: Goal, Policy, Strategy, and Plans of Decision and Action*, Journal of Organizational Computing and Electronic Commerce, Vol. 13, No. 1, pp–1-29 (2006)

[16] F. Zulkernine, P. Martin, C. Craddock, et.al., *A Policy-Based Middleware for Web Services SLA Negotiation*, In Proceedings of IEEE International Conference on Web Services (ICWS2009), pp. 1043–1050, (2008)

[17] Z. Xiao, D. Cao, C. You and H. Mei, *A Policy-based Framework for Automated Service Level Agreement Negotiation*, In Proceedings of IEEE International Conference on Web Services, pp. 682–689 (2011)

[18] M. Comuzzi and B. Pernici, *An Architecture for Flexible Web Service QoS Negotiation*, In Proceedings of Ninth IEEE International EDOC Enterprise Computing Conferences, pp. 70–79 (2005)

[19] S. K. Garg S.K, C. Vecchiola, and R Buyya, *Mandi: a market exchange for trading utility and cloud computing services* In Journal of Supercomputing (JOC) (acceted 2011, in press)