Service Level Agreements for Semantic Web Agents

Nir Oren, Alun Preece, Tim Norman[*]

Department of Computing Science
University of Aberdeen, Scotland, UK

[*] *noren,apreece,tnorman@csd.abdn.ac.uk*

Abstract

We propose an RDF-based language called SWCL to represent contracts, with a focus on service level agreements. The language came about due to a requirement in our research concerning reliable service delivery in domains containing virtual organizations, and we describe one such domain. Our language is built around a production rule system using RDF bindings and containing some contract-specific resources. A number of other approaches to contracting have been proposed, and we first describe these, and then compare them to SWCL. Finally, we suggest a number of extensions to this work, as well as ideas regarding how contracting can form part of a framework for increasing agent reliability in an open environment.

Introduction

Agents operating in an unregulated open environment do not know that other agents sharing this environment with them are entirely trustworthy. When one offers a service to another, the service consumer has no guarantee that the provider will fulfill its end of the bargain. Similarly, the provider has no way of ensuring that it will be paid for its services.

Mechanism design (Dash, Jennings, & Parkes 2003) can ensure that rational agents do not attempt to renege on their obligations. However, programming errors or simple maliciousness can render this approach ineffective. Trust and reputation systems (Yu & Singh 2002) can prevent agents from approaching unreliable providers but have difficulty dealing with agents about which nothing is known. Contracts between agents are able to overcome some of these difficulties, and have the additional benefit of allowing agents to agree on exactly what service is to be provided, and how. Furthermore, contracts are declarative, allowing an agent to autonomously decide how best to fulfill its obligations. Contracts do however have the disadvantage of requiring a contract execution monitoring mechanism, as well as some external body to implement sanctions against agents breaking contracts. Contracts can thus be seen as one of the tools required to create a trusted environment between agents operating within open systems.

Robust service delivery in CONOISE-G

This research is being done as part of the CONOISE-G (Norman *et al.* 2004) project. The project is focused on investigating technologies to enhance VO reliability, and contracting forms one of the foundations for this goal.

Inter-agent contracts allow us to reinforce proper behavior in two ways. First, an agent is reluctant to violate a contract due to the penalties it will incur. Second, when a contract is violated, the contract monitoring agent informs the trust component, leading to a reduction in the violating agent's reputation. A key consequence of this is that the agent is less likely to successfully bid for work in the future. Thus, even an agent which doesn't care about the penalties it receives for breaking a contract will have difficulty affecting the marketplace after not meeting their obligations in a few contracts.

In this paper we describe a machine understandable language for contracts. We begin by describing the scenario in which our contracting language will be evaluated. We then examine other contract representation languages and look at their strengths and weaknesses. After this, our contracting language is described. Finally, we look at possible extensions to this research.

Scenario

We envision a marketplace where agents can temporarily pool their resources to form a virtual organization (VO) so as to achieve goals that they would be unable to achieve individually, or to provide services more efficiently than they would be able to do alone. Once a VO is formed, it should act as a single cohesive unit in the context of service provision.

While our research is designed to operate in a generic service provision domain, we have focused on a mobile multimedia scenario.

The domain consists of a number of service provider agents (SPs), each of which is able to provide a number of multimedia services to a set of clients. Currently defined services include text-messaging, news clippings, streamed movies and phone calls. Each SP has a limited number of resources it can provide at any one time. Client agents represent the user, and are able to request services from the SPs. The environment provides a number of infrastructure services, instantiated as agents and agent components:

- Yellow pages agents (YPs) store SP contact details, as well as information concerning the services advertised by SPs.
- A clearing agent (CA) allows an agent to determine which set of bids best fulfills its requirements, based on the agent's request and various parameters associated with each bid (such as the bid price).
- A quality agent (QA) is able to assign a quality rating to the services offered by an agent. This quality rating can be used by the CA to determine the overall utility of an agent's bid.
- A monitoring agent abstracts the various sensors in the system, using a publish-subscribe pattern. Agents interested in observing the environment contact the monitoring agent and tell it what services (and which service properties) should be monitored, and for what type of events. When such an event occurs, the monitoring agent sends a message to the interested agent, informing it of the event.
- A trust component models the levels of trust between the various agents in the system. The trust value can be used as one of the parameters passed to the CA so as to allow it to make a decision regarding which bids should be accepted.
- A contract monitoring component generates inter-agent contracts and tells the monitoring agent what service attributes should be monitored for compliance. When a message from the monitoring agent is received,the contract monitor computes whether a contract has been violated, and if so, what penalties should apply.

An agent may also take on one of a number of roles:

- It may take on the role of a client and thus ask other agents to provide it with services.
- An agent may be the VO manager (VOM), in which case it represents the entire VO to the client.

A client is unable to tell whether it is dealing with a single provider agent or with a VO fronted by a VOM. VOs are thus opaque and hidden from other agents by the VOM handling their tasks.

A VO's lifecycle consists of three distinct phases: VO formation, VO service provision and VO dissolution.

VO formation

The VO formation process is illustrated in Figure 1 and proceeds as follows: the client contacts a SP and asks for a package of services. The SP takes on the VOM role, and contacts the YP asking it which other SPs can provide the services requested by the client. Once the YP responds, the VOM sends a request for bids to the other SPs. Once these bids are received, a clearing algorithm computes which SPs should form part of the VO. In our system, the messages exchanged between the agents take the form of RDF fragments. The agent's knowledge base is also internally stored as RDF, and our agents are thus able to easily parse, reason and integrate the messages into their internal knowledge structures.

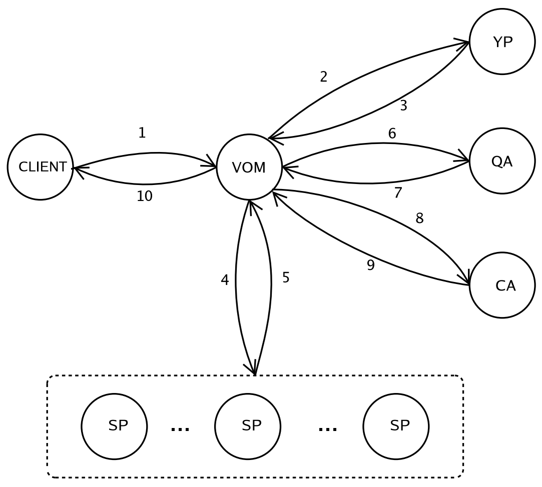

Figure 1: Some of the agents making up the VO and the environment, together with the order of messages passed between them during the initial part of the VO formation episode. For clarity, message passing between the VOM and the contracting component, as well as between the VOM and the monitoring component has been omitted. The numbers next to the edges indicate the order in which messages are generated and passed.

Contract negotiation is outside the scope of our work. Thus, when successful SPs are informed that they have been chosen to provide a service, they send the VOM a contract that describes the what they will be providing. Similarly, the VOM sends a completed contract to the client containing details about the package it will be providing.

In the final stage of VO formation, the VOM and client ask the monitoring agent to monitor the values of the attributes for the services that they will be receiving. Alerts from the monitor agent are used in the next stage of the VO lifecycle.

VO service provision

The VO service provision stage runs until a contract is terminated, either successfully or due to a fatal contract failure. Within this stage, the client asks the VOM for a service. The VOM then transmits the client's request to the appropriate SP(s) (and, given that SPs may themselves be VOMs, this request may filter down through multiple VOs). The SP(s) then begins to provide the service to the client. Various events can perturb a service provision episode.

One of the simplest perturbations occurs when the SP providing a service is unable to fulfill its contract conditions. Other possible perturbations include a failure by the VOM to forward messages from a client (also leading to a contract failure), the VOM deciding to reorganize the VO (possibly to take advantage of new market conditions such as a new agent appearing), and having the VOM or an SP leave the environment during service provision.

Whether recovery is possible depends both on the exact perturbation type, as well as the contents of the contract. For example, if an SP providing a critical service leaves, and cannot be replaced, recovery is impossible. This paper will not provide a detailed taxonomy of perturbation types, their effects and possible recovery methods. It should, however, be noted (and will be discussed in greater detail below) that a contract should be able to specify how recovery from a failure can take place.

VO Dissolution

A VO may dissolve due to a contract successfully ending, contract termination due to a failure, or the VOM deciding to dissolve the VO. Our contracting language must be able to specify what actions the VOM and SPs should take when the VO is dissolved.

Contracting in CONOISE-G

As should be clear from the previous sections, contracting is a critical component of the CONOISE-G environment. Since we are implementing a technology demonstrator, rather than a fully functioning prototype, we do not use a central contract repository. Instead, agents keep copies of their own contracts (which, in a real environment would be digitally signed to avoid tampering), and update their internal contract state as appropriate. Contracts are enforced by having an agent's contract monitoring component compare the state of the contract to that of the real world. If a discrepancy is detected, an investigation can then take place to determine what penalties should come into force. Only one centralized component is needed in such a framework, namely the sanctions enforcer. If the only sanction is loss of trust, then a distributed reputation system can overcome even this limitation, leading to a completely decentralized system.

We believe that the CONOISE-G domain is complex, but contains many properties required of any system operating in the real world. Furthermore, we claim that, as described in more detail in the next section, existing contracting languages would have great difficulty operating in such environments.

Contracts and contracting languages

In this section, we look at the requirements for a contracting language in more detail. We examine the desiderata of a contracting language, and look at how existing contracting languages fulfill these desiderata.

Broadly, a contract is a legally binding agreement between two (or more) parties. We extend this definition somewhat: to us, contracts are legally binding agreements between two or more parties, specifying the norms in effect between the parties for the duration of the contract.

Since a contracting language must be able to describe anything talked about by a contract, it should be able to :

1. Identify the parties involved in the agreement.
2. Refer to the temporal aspects of the contract, including durations and time instants to allow for the description of
   (a) When the contract comes into force.
   (b) When the contract terminates.
   (c) When certain clauses are in effect.
3. Provide support for the concept of an agent undertaking an action.
4. Allow for the description of norms (obligations, prohibitions, permissions and the like) imposed on each party.
5. Provide a way of representing rewards and penalties.
6. Allow for contrary to duty obligations (e.g. since rule a was broken, rule b is now in effect).
7. Refer to the state of other agents, contracts, and the environment.
8. Allow for the conditional evaluation of obligations and contract elements, based on temporal, action and state elements.

These desiderata were selected by examining the features provided by a number of existing contracting languages, as well as by the domains they appear to be intended to function in (usually identified by looking at the examples provided in their descriptions). While we cannot in any way prove that they completely describe all the requirements for a contracting language, we were unable to find further requirements based on our survey of existing contracting languages.

Note that many of these requirements are interrelated, and may be required at different stages of contract execution. As an example, assume the following (simplified) contract:

This contract is in effect for 24 hours beginning at midnight on the 31st of December 2005. The provider is to provide a movie every three hours, each of which must be between one and two hours long. For each three hour period during which a movie is not supplied, the provider must pay the consumer a penalty of twenty dollars. A payment of $20 is to be made at the end of the contract.

It is clear that temporal, deontic and action concepts can come into play at multiple levels: they describe the contract start and end conditions, the way in which actions are executed, how often (and, more infrequently, how) penalties must be assessed, the manner in which payment is made and are an integral part of determining whether an action is valid.

Contracting languages are normally designed to fulfill any of a number of possible roles. Some languages focus on the contract negotiation phase. Courteous Logic Programs (Reeves *et al.* 1999) are an example of such a language. CLPs extend standard logic programming approaches by introducing priorities between rules. Reeves *et al.* provide an ontology for use with CLPs that facilitates contract creation and negotiation from base contract templates. SweetDeal (Grosof & Poon 2004) extends CLPs by situating them (allowing them to invoke methods) and presents a concrete DAML+OIL ontology for contracting.

Another possible goal for a contracting language is the ability to perform contract consistency checking. Given a contract written in such a language, it should be possible to determine whether the contract contains any inconsistencies (for example, an agent being obliged to both achieve and not achieve a certain state of affairs simultaneously). (Daskalopulu 1997) proposed a contracting language that performs this task. DPL (Milosevic & Dromey 2002) is a deontic logic based contracting language that is also intended to allow for consistency checking.

The majority of contracting languages are designed to allow for automated contract execution monitoring, that is, the determination of what state a contract is in, and which contract rules are in effect given the current contract state.

Contracting languages can be broadly divided into two groups: formal approaches have very well specified semantics, and are often based on some logic. Ad hoc approaches on the other hand often have only "commonsense" semantics, and are often constructed by abstracting the features of real world contracts. Each approach has its own advantages and disadvantages.

The well defined semantics of formal contracting languages, exemplified by approaches such as LCR (Dignum *et al.* 2002), reduces contract enforcement in fully observable domains to theorem proving. However, formal languages have a number of shortcomings: first, they require that the environmental state can be translated to a contract state. Tied to this is the requirement that the environment be fully observable. Generating contracts in a formal language is difficult, and important features are often not included in the language for the sake of tractability (for example, specifying repeated actions is difficult in LCR).

Ad hoc languages, including DLP (Dimitrakos *et al.* 2003) and NoA (Kollingbaum & Norman 2002), also borrow ideas (such as obligations) from logic, most notably deontic logic. These language often support many of the desired attributes of a contracting language, but their weak semantics means that contract enforcement is more difficult (and often implementation dependent). Without very careful consideration to the design of the language, other tasks, such as contract validation, can be impossible for an ad hoc language. However, specifying contracts in these languages is relatively easy, due to their English or programming language like syntax.

Formal languages appear to take a "top down" approach to contracting: first, define semantics, and then see what type of contracts can be represented using the approach. Ad hoc techniques attack the problem in a"bottom up" manner: first, one determines what features a contract requires, then one constructs a language that provides these features, and finally, semantics are added at the very end.

Since the effects of a rule within a contract can be arbitrarily general, a contracting language should be as expressive as a general purpose programming language. By focusing on specific classes of contracts, it may be possible to reduce the descriptive capabilities of the contracting language, thus simplifying its design.

One such class of contracts are service level agreements (SLAs). Service level agreements are contracts that specify a set of services to be provided, and ranges in which the various attributes of each service must remain. Penalties are assessed when an attribute strays outside of its required boundaries. Rewards can also be given when an attribute stays within its required range. Normally, penalties take the form of a payment from the provider to the consumer, with repeated failures being grounds for the cancellation of the contract. SLAs are found in myriad environments, for example in the telecommunications sector, outsourcing agreements and service provision environments. Work has started on a standard for machine readable SLAs, called WS-Agreement[1]. Since CONOISE focuses on the service provision domain, our language also focuses on being able to represent SLA type contracts. We have named our language "SWCL", standing for "Semantic Web Contracting Language".

SWCL

Our language is RDF based, and borrows many ideas from the work being done on the WS-Agreement standard. Our reasons for creating a new language, rather than using WS-Agreement, are discussed in a later section. We will illustrate aspects of SWCL by referring to fragments from a sample contract. The fragments are written in pseudo-RDF, with namespaces left out when unimportant so as to enhance readability. We will not attempt to give a full, in-depth description of our contracting language in this paper.

```
<RDF>
<SLA rdf:about="contract1">

<provider>
  <Actor rdf:about="uri:a123"/>
</provider>
<Consumer>
```

[1]http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap-agreement.pdf

```
  <Actor rdf:about="uri:b321"/>
</Consumer>
```

Following the standard RDF document preamble, we begin by identifying the parties participating in the contract (fulfilling the first desideratum). The ¡Actor¿ tag originates in an ontology used within CONOISE and uniquely identifies an agent in the system. All the tags used to describe services and attributes are specified in a CONOISE-G service ontology, adapted from DAML-S[2] and intended to be compatible with OWL-S. Providers and consumers need to be identified so that penalties and rewards can be dispensed as appropriate by the contracting framework.

```
<monitoredService>
  <Service rdf:about="movieService1">
    <monitoredAttribute>
      <Attribute rdf:about="frameRate">
        <monitorMethod><QoSC/></monitorMethod>
      </Attribute>
    </monitoredAttribute>

    <monitoredAttribute>
      <Attribute rdf:about="genre">
        <monitorMethod><QoSC/></monitorMethod>
      </Attribute>
    </monitoredAttribute>
  </Service>
</monitoredService>

<monitoredAttribute>
  <Attribute rdf:about="currentTime">
    <monitorMethod><System/></monitorMethod>
  </Attribute>
</monitoredAttribute>

<monitoredAttribute>
  <Attribute rdf:about="frameRateDropCounter">
    <monitorMethod><System/></monitorMethod>
  </Attribute>
</monitoredAttribute>
```

The next section of the contract specifies the services and attributes that are to be monitored, as well as where the attribute values are obtained. The QoSC is the monitoring agent within our system, and provides us with all our environmental data. Some environment variables (such as currentTime) are also specified. Attributes monitored by the QoSC are listed within the service as a contract may specify the same attributes for different groups of services (for example, two movie services may be required, each of which has a different genre or frame rate). Attributes monitored by System are related to, and kept track of by, the contracting environment.

[2]http://www.daml.org/services/

```
<hasClause>
  <Clause rdf:about="minimumFrameRate">
    <evaluationExpression>
      <swrl:IndividualPropertyAtom>
        <swrl:propertyPredicate
              rdf:resource="swrlb:greaterThanOrEqual"/>
        <swrl:argument1 rdf:resource="frameRate"/>
        <swrl:argument2> 24 </swrl:argument2>
      </swrl:IndividualPropertyAtom>
    </evaluationExpression>
    <assesmentInterval>
      <DurationDescription>
        <minutes>1</minutes>
      </DurationDescription>
    </assesmentInterval>
    <effect>
      <Payment>
        <from rdf:resource="uri:a123"/>
        <to rdf:resource="uri:b321"/>
        <amount>5</amount>
      </Payment>
    </effect>

    <effect>
      <CounterIncrement>
        <counter rdf:resource="frameRateDropCounter"/>
        <amount>1</amount>
      </CounterIncrement>
    </effect>
  </Clause>
</hasClause>

</SLA>
</RDF>
```

The last section of the contract contains the rules for contract execution and monitoring. Borrowing terminology from real world legal documents, we call each rule a clause. As with most other rule driven systems, each clause contains an evaluation component and an action component. The evaluation component is made up of an evaluation expression and an evaluation frequency fragment. A clause's action component is executed if the evaluation expression becomes true. The evaluation expression is reevaluated (and the action component may thus be run again) based on information from the evaluation frequency section.

Clause evaluation expressions are built up of SWRL[3] expressions. They may reference any attributes declared in the monitored attributes section, as well as query the state of other contracts and clauses.

A clause can have multiple effects. These effects may alter the contract or the environment in many ways. Apart from altering the values of certain variables (notably counters and Boolean conditionals), a contract

[3]SWRL is the Semantic Web Rule Language. More details can be found at http://www.daml.org/2003/11/swrl/

effect may specify a payment is to take place. An effect clause may also change the state of the contract, possibly triggering other clauses.

To allow for contracts to refer to other contracts and clauses, it is important to keep track of contract state. A contract may be in one of the following states:

- Running. In this state, the contract is in force, and no violations have taken place.
- Violated. Here, the contract is still in force, and some clauses have been violated.
- Terminated. This state occurs when the contract is no longer in force, due to entering a failure state from which recovery was impossible.
- Complete. This state occurs when a contract is no longer in force due to successful contract completion.

Clauses can either be in a running state, in which case their evaluation expression is under observation, or active, in which case their effect is triggered. One-off clauses, once triggered, are considered to be permanently active.

The second desideratum requires the language to be able to specify when the contract starts and ends. At the moment, SWCL contracts come into force as soon as all parties agree to them. We can, however, alter when individual clauses come into effect by adding a temporal condition to their evaluation expression. An earlier version of language did support a ¡startTime¿ tag, but we discovered it was not often used. Contract termination in SWCL comes about as an effect of a clause.

SWCL fulfills the desiderata for the concept of an action (3), norm description (4) and contrary to duty obligations (6) in the contract ¡Clause¿ sections. Conditional evaluation of obligations (desideratum seven) is done by allowing evaluation expressions to refer to the state of other clauses.

Rewards and penalties (desideratum five) are viewed as actions by many contracting languages. Our explicit ¡Payment¿ comes from the CONOISE-G environment, and we allow any sort of effect to represent a reward or penalty.

Most languages supporting contract execution allow for conditional obligations (desideratum eight). Allowing clauses to come into effect only as a result of some other clause failing is how we cater for this feature.

The reflective features (i.e. the ability to refer to other contracts and clauses) of SWCL are intended to allow one to refer to other contracts (part of requirement seven). These features were not shown in the previous example, but could appear as follows:

```
<ReplaceClause>
  <replace>
    <Clause rdf:resource="minimumFrameRate"/>
  </replace>
  <with>
    <Clause rdf:resource="newMinimumFrameRate">
  </with>
</ReplaceClause>
```

The ReplaceClause object would normally be embedded as the object of another clause's effect. Other, similar effects are the removal and insertion of clauses. Another aspect of reflection is the ability, within an evaluation expression, to examine a contract's state.

Comparison with other contracting languages

In this section, we compare the features of SWCL with those provided by other approaches. Table 1 summarizes the various approaches, showing how they fulfill the desiderata described previously. Note that while a language may not explicitly support a feature, it may be possible to still implement it, at the cost of additional complexity. For example, while a language might not explicitly offer support for temporal features, by allowing for the invocation of methods from a programming language, support for such features may be provided.

As was mentioned previously, and is highlighted in the table, formal approaches lack the expressive power required to represent real world contracts. Ad hoc languages, with the exception of NoA, have much greater expressive power. NoA does not explicitly support a number of contracts, but due to its integration with the Java language, additional features can easily be added. However, these additions, would require modifications to the language and its interpreter.

SweetDeal was probably the first contracting language which attempted to represent contracts in a format appropriate for the Semantic Web. While it shares a number of similarities with SWCL, its main focus was the extension of CLPs to deal with contrary to duty obligations. Thus, while it is more capable than CLPs in representing the various desiderata, we believe that certain concepts can be more neatly represented using our approach.

WS-Agreement tackles a similar domain to SWCL, so in the remainder of this section, we will compare these two approaches.

The WS-Agreement standard is still in its early stages of development. Furthermore, the standard is currently mostly concerned with building a standard syntax for specifying SLAs, giving users latitude as to which language should be used to represent the semantic details of the contract. WS-Agreement does not provide any RDF mappings for its tags, since the rest of our system makes extensive use of Semantic Web technologies, using WS-Agreement would have led to unneeded complexity in our framework. Other differences arise because of a difference in focus between WS-Agreement and SWCL. For example, the WS-Agreement draft specification states that WS-Agreement will make no attempt to cater for contract negotiation, or describe a condition expression language for specifying clauses. While we do not cover contract negotiation in this paper, the ability to easily generate contract templates with SWCL, and convert these into concrete contracts, allows one to perform at least some form of contract negotiation. We also specify a condition expression language as we believe that allowing for other languages to be inserted causes unnecessary complexity in the final framework.

We feel that one other advantage of our language over other existing contracting languages is the addition of reflective capabilities, allowing parts of a contract to refer to other contracts or clauses within itself, and modify these. It is our belief that this requirement has been noticed by other language designers, hence the prioritized clauses in CLP, but that we cater for this need in a more explicit manner.

Lang.	1	2a	2b	2c	3	4	5	6	7	8
Formal Approaches										
LCR	P	N	Y	Y	Y	Y	N	Y	N	Y
CLP	P	N	N	N	N	Y	P	P	P	P
Ad hoc Approaches										
DLP	Y	N	N	N	Y	Y	Y	Y	N	P
NoA	Y	P	P	Y	Y	Y	P	P	N	Y
Sweet	P	N	N	Y	N	Y	P	P	P	P
WS-A	Y	Y	Y	Y	P	P	Y	P	N	P
SWCL	Y	Y	Y	Y	Y	P	Y	Y	Y	Y

Desideratum	Description
1	Identify involved parties
2a	Agreement start time
2b	Agreement end time
2c	Temporal constraints in clauses
3	Action representation
4	Application of norms to agents
5	Describe rewards and penalties
6	Contrary to duty obligations
7	Refer to other agents, contracts and clauses
8	Conditional evaluation based on temporal, action and state elements

Table 1: The desiderata (described in the section on contracts) catered for by each language. "Y" means that the language has explicit constructs allowing for this feature. A "P" means partial support, while "N" means no support for the feature.

Future work and conclusions

Our short term goal is to create a component that is able to take in such a contract, and, by monitoring the environment, triggers the correct clauses. In the longer term, our future work consists of two streams, namely enhancing the contract representation language and improving the contract monitoring environment.

While descriptive, the contracting language still has a number of shortcomings. Contract clauses should have more power to alter other sections of the contract during the execution phase. For example, given a scenario in which a service provider has agreed to provide movie and text services to a client, and has then reneged on the text services, it should be possible to modify the contract so that the service provider is still responsible for providing movie services. We also intend to provide the language with more concrete formal semantics in the future. Since SWCL is RDF-based, it should be possible to do this with relative ease[4].

The current contract execution environment has two sources of information which it can use to monitor contract execution, namely the QoSC monitoring agent and the contract execution framework (referred to as System in the RDF fragments). This means that our environment is fully observable. SWCL will still function in partially observable environments. The contracting language is intended to be one of the foundations for a more generalized policing framework. As described in (Oren *et al.* 2004), we are investigating an argumentation based approach to failure detection and blame assignment in such complex environments. We require contracting for our research as contracts form the basis for what an agent is obliged to achieve.

Other open questions include extending the language to provide support for contract negotiation, as well as methods for performing contract validation on contracts written with it.

In this paper, we described a RDF-based contracting language that focuses on the SLA domain. While many contracting languages have been suggested, it is our belief that each is missing some desirable features. SWCL is our first attempt at creating a language that, we believe, provides enough structures to model complex, real world contracts. SWCL is designed to allow for automated contract enforcement.

Acknowledgments

The CONOISE-G project is jointly funded by the DTI/EPSRC E-Science Core Programme and BT Exact – British Telecom's research, technology and IT operations business. The project is a collaboration between BT Exact, and Aberdeen, Cardiff and Southampton Universities.

References

Dash, R. K.; Jennings, N. R.; and Parkes, D. C. 2003. Computational–mechanism design: A call to arms. *IEEE Intelligent Systems* 18(6):40–47.

Daskalopulu, A. 1997. Logic-based tools for legal contract drafting: Prospects and problems. In *Proceedings of the 1st Logic Symposium*, 213–222.

Dignum, V.; Meyer, J.-J.; Dignum, F.; and Weigand, H. 2002. Formal specification of interaction in agent

[4]A specification for the semantics of RDF can be found at http://www.w3.org/TR/rdf-mt/

societies. In *Proceedings of the Second Goddard Workshop on Formal Approaches to Agent Based Systems*, 37–52.

Dimitrakos, T.; Djordjevic, I.; Milosevic, Z.; Josang, A.; and Phillips, C. I. 2003. Contract performance assesment for secure and dynamic virtual collaborations. In *Proceedings of the Enterprise Distributed Object Computing Workshop*, 62–75.

Grosof, B., and Poon, T. C. 2004. Sweetdeal: Representing agent contracts with exceptions using semantic web rules, ontologies, and process descriptions. *International Journal of Electronic Commerce* 8(4):61–98.

Kollingbaum, M. J., and Norman, T. J. 2002. Supervised interaction – creating a web of trust for contracting agents in electronic environments. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi–Agent Systems (AAMAS'02)*, 272–279.

Milosevic, Z., and Dromey, R. G. 2002. On expressing and monitoring behaviour in contracts. In *Proceedings of the Sixth International Enterprise Distributed Object Computing Conference (EDOC'02)*, 3–14.

Norman, T.; Preece, A.; Chalmers, S.; Jennings, N.; Luck, M.; Dang, V.; Nguyen, T.; Deora, V.; Shao, J.; Gray, W.; and Fiddian, N. 2004. CONOISE: Agent-based formation of virtual organisations. *Knowledge-Based Systems* 17(2–4):103–111.

Oren, N.; Norman, T.; Preece, A.; and Chalmers, S. 2004. Policing virtual organizations. In *Proceedings of the 2004 European Workshop on Multi-Agent Systems (EUMAS 2004)*.

Reeves, D.; Grosof, B.; Wellman, M.; and Chan, H. 1999. Toward a declarative language for negotiating executable contracts. In *Proceedings of the AAAI-99 Workshop on Artificial Intelligence in Electronic Commerce*, 39–45.

Yu, B., and Singh, M. 2002. An evidential model of distributed reputation management. In *Proceedings of the first international joint conference on Autonomous Agents and Multi–Agent Systems (AAMAS'02)*, 294–301.