From Representational State Transfer to Accountable State Transfer Architecture

Joe Zou
IBM Australia
Macquarie University
Sydney, NSW, Australia
joezou@au1.ibm.com

Jing Mei
Semantic Integration
IBM Research
Beijing, China
meijing@cn.ibm.com

Yan Wang
Department of Computing
Macquarie University
Sydney, NSW, Australia
yanwang@science.mq.edu.au

***Abstract*—Since Representational State Transfer (REST) architecture was proposed by Fielding in early 1990s for distributed hypermedia systems, it has become a popular architectural style of choice in various computing environments. However, REST was not originally designed to support enterprise requirements, in particular the accountability requirements that are crucial for the business services offered through the Software as a Service (SaaS) and Cloud Computing environments. In this paper, we propose an Accountable State Transfer (AST) architecture to bridge the accountability gap in REST. With AST, service participants can be held accountable for each representational state transfer during service consumption. A formal service contract model with a hybrid reasoning mechanism and a novel accountable state transfer protocol are designed as the mechanisms underpinning the AST architecture. Moreover, we implement a Credit Check service prototype based on AST, demonstrating the practicality of such architecture. Inheriting REST's scalability, AST architecture provides the much needed accountability capabilities for the virtual service delivery environment.**

Keywords: REST, SOA, Accountability, AST, service contract

I. INTRODUCTION

REST architecture was originally designed by Fielding in early 1990s to support the high performance and scalability requirements of the hypermedia environment [1]. Since then, its application has gone beyond the original environment and made further inroads into the e-business arena. Due to its simplicity and scalability, it also emerges as a strong alternative to SOAP-based Web Services for building the Service-Oriented Architecture (SOA).

SOA is the key architectural foundation that turns traditional IT focused services into business services, enabling dynamic service delivery and consumption. Based on the principles of SOA, Cloud Computing is emerging as the latest trend that offers the promise of massive scaling of service delivery and consumption. While the current focuses in Cloud computing are mainly in the technology areas such as virtualization, workload management and Web 2.0 style of interfaces, the crucial business issue of accountability is often ignored in the IT industry. Fundamentally, Cloud computing is just a new business model that provides a flexible delivery model for transacting business services, thus the reputations of the service provider and consumer's confidences on the services are the key successful factors for the Cloud computing business model. In a commercial environment, reputations and confidences are built upon the accountability of the services. Accountability in this context means that obligations of service participants are fully **disclosed**; actions can be **justified**; the disclosed obligations are faithfully **honored**, or else **liability** is assumed by the misconduct party [2]. In a Cloud computing environment, service accountability becomes even more critical as the virtualized environment is inherently less trustworthy.

Traditionally accountability can be best enabled and enforced by an implicit or explicit legal contract. To enable accountability in a Cloud computing environment, the underlying SOA needs to have a formal service contract construct, plus the ability to monitor the contract execution and reason the contract state. Currently, neither REST nor the existing WS-* specifications support the concept of service contract as well as the performance tracking of that very contract. REST in particular has a large accountability gap compared to SOAP-based services, as it was not originally designed to address the enterprise requirements such as security, reliability, transaction ability and manageability. As REST is increasingly becoming a popular choice for implementing Cloud services, it is imperative to build accountability mechanisms in the REST architecture.

In this paper, we propose a novel architecture called Accountable State Transfer (AST), which extends REST architecture to bridge the accountability gap. Our approach is innovative as it seamlessly integrates *service contract* semantics into the syntactic-based REST service to enable accountability, yet still retains the RESTful characteristics and therefore inherits the scalability of REST architecture. The AST architecture supports accountability by enabling obligation disclosure; action justification and obligation tracking for each contracted party plus contract state reasoning. We also implement a Credit Check prototype service based on AST to demonstrate its practicality.

The rest of the paper is structured as follows. Section II reviews the related work. Section III outlines AST's guiding principles and architectural decisions. This is followed by section IV that describes the AST architecture design. A prototype implementation is discussed in section V. Finally contributions and future work are summarized in Section VI.

II. RELATED WORK

According to Fielding, REST behaves like a virtual state machine, where the state transition happens when the user selecting links, resulting in the next state of the application being transferred to the user [1]. REST lays down the foundation for the Web architecture. There are a number of efforts to extend REST to address certain aspect of requirements. In [3], the authors suggest the concept of "Computational Transfer" and propose Computational REST architecture (CREST). The idea is to use AJAX and

mashups as mechanisms for framing responses as interactive computations or for "synthetic redirection" and service composition. CREST is essentially the Web2.0 style of Web architecture. To strengthen REST's capability in supporting enterprise requirements, the authors in [4] extend REST to induce four properties: events, routes, locks and estimates. They derive four new REST styles (ARREST, ARREST+E, ARREST+D and ARRESTED) optimized for each of the above four types of resources. However, currently REST and its extensions do not address the accountability requirements. In particular, they do not support the concept of service contract and also do not have any ability to provide justification on the action of state transfer.

Various definitions on accountability can be found in the IT literature. Refer to [5] for a detailed review on accountability literature. In this paper, we adopt Schedler's definition of accountability: "A is accountable to B when A is obliged to inform B about A's (past or future) actions and decisions, or justify them and to be punished in the case of misconduct" [2]. We further distill the essence of accountability as: obligation **disclosure**; action **justification** based on disclosed obligations and **evidence** produced; and resuming **liability** for non-fulfillment of obligation. In the e-services context, accountability is driven by the underlying business contract established between the service provider and the service consumer, which reflects the business reality in the traditional commercial services space. Therefore to support accountability in SOA, we need to introduce the notion of service contract at the architecture level.

From an accountability solution perspective, the support for accountability in IT is limited. Currently it is mainly provided in monitoring systems such as Business Activity Monitoring (BAM) and IT infrastructure and application monitoring (ITM). The former focuses on business process monitoring while the later focuses on QoS or service level-agreement (SLA) monitoring. While SLA is a kind of contract, it is normally used to record the non-functional aspect of obligations of the involved parties rather than the functional one. Moreover, the current ITM and BAM solutions do not provide obligation disclosure and action justification capabilities.

On the other hand, e-Contract is an extensively researched area in the IT literature. Most of the existing e-Contract models are either represented by some form of XML documents or some kind of logic models. One notable XML contract is IBM's Trading Partner Agreement (TPA), which stipulates the general contract terms, conditions, participant roles, communication and security protocols, and business process [6]. TPAml had been submitted to OASIS and used as a basis for developing ebXML Collaboration Protocol Profile (CPP) and Collaboration Protocol Agreement (CPA). TPAml and ebXML CPP/CPA are designed for business to business (B2B) process integration. As both TPAml and ebXML CPP/CPA require a full stack of infrastructure and protocol support on both sides, they are too heavy weight and thus not suitable for REST services. In [7], Xu proposes a multi-party e-Contract model that maps a paper-based contract into contract actions and contract commitments. Xu's model representation is based on a First-Order Temporal logic programming model, which is not accessible to a REST service. In [8], an e-Contract model based on Modal Action Logic, Deontic Logic and Subjective Logic is presented. In [9], a Business Contract Language (BCL) and Formal Contract Language (FCL) are proposed using Defeasible Logic and Deontic Logic. Another approach is to derive e-Contract from business process [10]. Some other e-Contract models include Event Calculus/ecXML [11], CrossFlow [12], e-Contract based on Description Logic (DL) [13], SweetDeal (based on RuleML and DAML+OIL) [14], etc.

While the existing models provide significant contributions in various aspects of e-Contract literature, they do not allow seamless integration into REST architecture; nor provide service contract meta-data during service invocation. Furthermore, most e-Contract models favour expressiveness over decidability in order to mimic a legal business contract.

III. GUIDING ARCHITECTURAL PRINCIPLES

A. *RESTful Principles*

The key characteristic of the REST architecture is that it takes a "resource view" of the world. The RESTful principles described in [1] and elaborated in [3] are:

Table I. RESTFul Principles

P1: Resource can be identified by an URI;
P2: Separation of the abstract resource and its concrete representations;
P3: Stateless interaction, each interaction contains all the necessary context information and meta-data;
P4: Small number of operations, with distinct semantics based on HTTP methods: safe operations (Get, Head, Options, Trace); non-safe, idempotent operations (Put, Delete); and non-safe, non-idempotent operation (Post);
P5: Idempotent operations and representation metadata support cache;
P6: Promote the presence of intermediaries such as proxies, gateways or filters to alter or restrict request and response based on metadata.

Following these principles ensures our AST architecture retaining REST's scalability and performance.

B. *Service Contract as Foundation for Enabling Accountability*

The key accountability concerns addressed in this paper are: obligations disclosure; execution status tracking based on evidence; and the ability to provide justification and explanation of actions in relation to a pre-established contract. Compared to the existing accountability models in IT literature, a key differentiation of our approach is that we position *service contract* as the foundation underpinning the accountability concerns in an SOA environment.

C. Principles for the Service Contract Model

Most of the e-Contract models in literature attempt to mimic a legal contract. Consequently the underlying logic model needs to have strong expressiveness power at the expense of computation completeness and decidability. In this paper, a key principle is that the logic framework underlying the *service contract* model must be computationally complete and decidable; in the meantime, it should have enough expressiveness to represent the key obligations in a traditional contract. Thus our *service contract* model does not aim at replicating all the information in a traditional contract; instead it focuses on facilitating runtime disclosure, monitoring and management. We now define its scope used in this paper:

Definition 1: Service contract is an electronic representation of a traditional contract that captures the essential contractual information including involved parties, domain specific terms, obligations for each party, contract execution states and rules that determine those states.

D. Architectural Decisions in AST Implementation

In contrast to a lot of existing e-Contract models in the literature that only provide theoretical models, a key principle of our approach is to ensure the practicality of model implementation. To achieve that, trade-offs need to be made in implementation decisions. Followings are the architectural decisions for our AST architecture:

Decision 1: Implement *service contract* as an ontology and store *service contract execution* instances in a knowledge base (KB); In addition, add rules to allow reasoning of the contract execution state in the KB.

An ontology describes the concepts in the domain and the relationships held between those concepts. Building upon an ontology, our *service contract* model further needs rule capability for contract state reasoning.

Decision 2: Take a resource view on *service contract* and use URI to uniquely identify elements in the service contract.

Thus the elements in a *service contract* ontology can be referred via URIs during service invocation.

Decision 3: Separate the *service contract* and *service contract* execution concepts.

The rationale behind decision 3 is that in a SaaS or Cloud Computing environment, a service can be executed multiple times during the valid period of the underlying business contract. For example, a Credit Check service contract may last for one year; during the year the service can be executed for multiple times. Each execution is an execution instance of the service contract. The separation of contract and contract execution concepts allows contract execution tracking, which is not seen in most of the existing e-Contract model.

Decision 4: Adopt a hybrid reasoning approach that leverages strengths from different formalisms and technologies.

The last decision applies in the area of designing the reasoning mechanism for our *service contract* KB. We need to consider the expressiveness power and computation complexity of the underlying formalisms such as OWL-DL and SWRL; also take into account the availability of the tooling support to make the optimal design decision.

IV. Accountable State Transfer Architecture

A. Extending REST to Support Accountability

In a traditional REST service, both the consumer and the provider can not be held accountable for their actions during the representational state transfer. At the client side, the client consumes the provider's services by following the URL links to get representational states from some resources under the provider's control. But the client does not know precisely the linkage between the state transfer and the provider's obligations. The provider also does not know exactly why the client makes a particular request. In an e-services environment like SaaS or Cloud Computing, fundamentally each e-service is linked to a pre-established business contract between the service provider and the service consumer. Therefore, to establish accountability in the REST interaction, it is important to link the interaction to a particular contract context. So both the consumer and the provider can track the performance of the contract, understand the reasons behind each request and response w.r.t. that very contract. For example, suppose a SaaS provider provides a credit check service by exposing some REST interfaces. A service consumer needs to establish a binding contract with the service provider before he/she can consume the service. Once the contract is established, the service consumer can invoke the REST service to check a particular customer's credit score. However under the traditional REST, both the service consumer and service provider have no ways to know which contract the service is related to in runtime, let along tracking the progression status and determining which party breaches the contract.

In order to address the above problem, we extend the REST architecture by bringing in the *service contract* context as the meta-data during the interaction between the consumer and provider. The contract context information includes the name of the *service contract*, the current contract execution instance status and the overall contract progression status. In REST, each element of the *service contract* information can be treated as a resource, identified by an URI. Therefore the contract context meta-data can be simply referred to by URIs in HTTP headers. Also the contract progression and performance can be monitored by a trusted-third party (TTP). We call this style of the REST extension as Accountable State Transfer architecture.

B. Service contract Structure and Service contract Execution Structure

Definition 1 provides a high-level scope of *service contract*. We now list a rigorous definition [15]:

Definition 2: A *service contract* is a tuple $SC = (s, D, P, Op, Oc, Seq, st, R, T)$, where:

- s is a non-trivial service offered through Cloud platform;
- D is a finite set of domain specific contract term definitions: $D = \{d_1, d_2, \ldots, d_n\}$;
- P is a pair of involved parties (provider *pr* and consumer *pc*);
- Op (Provider Obligation) is a finite set of (Action, Evidence) pair: $Op = \{(ap_1, ep_1), (ap_2, ep_2),\ldots, (ap_n, ep_n)\}$, where Action $Ap = \{ap_1, ap_2, \ldots, ap_n\}$, Evidence: $Ep = \{ep_1, ep_2, \ldots, ep_n\}$;
- Oc (Consumer Obligation) is a finite set of (Action, Evidence) pair: $Oc = \{(ac_1, ec_1), (ac_2, ec_2), \ldots, (ac_k, ec_k)\}$;
- In Op and Oc, Action is a tuple: $a = (input, output, pre, post)$, where $input$, $output \in D$, both *pre* and *post* are binary condition expressions that are evaluated to *true*;
- Evidence is a finite set of triple: $E = \{(o_1, t_1, c_1), (o_2, t_2, c_2), \ldots, (o_n, t_n, c_n)\}$, where $o_i \in D$, t_i is the creation timestamp of o_i, c_i is a binary condition expression, $1 \leqslant i \leqslant n$;
- Seq is a finite set of sequence of actions, $Seq = \{s_1, s_2, \ldots, s_n\}$, where s_i is a sequence of actions;
- Contract State st: $st \in S$, $S = \{st_1, st_2, \ldots, st_n\}$, where st_i is one of user defined contract states, for example, *initialisation*, *in progress, provider breaching contract*, etc;
- Rules: $R = \{ r_1, r_2, \ldots r_n \}$, where $r_j (1 \leq j \leq n)$ is a horn clause: *consequent* ← *antecedent*, and
- Time Period $T = \{contract\_start\_time, contract\_end\_time\}$.

Definition 2 defines a generic structure for a two-party *service contract*. In theory, multi-party *service contract* can always be decomposed to multiple two-party service contracts. As explained in the rationale of architecture Decision 3 in Section III, we need to define the concept of *service contract execution* to capture execution information in each contract execution instance:

Definition 3: A *service contract execution* is a tuple $SCE = (sc, E, Op, Oc, se, R)$, where:

- sc is an individual of *service contract SC*;
- E is execution information, $E = (start\_time, complete\_time, timeout\_value)$;
- Op is a set of obligations that are successfully completed by the provider; (See Definition 2 for obligation definition);
- Oc is a set of obligations that are successfully completed by the consumer;
- Contract Execution State: $se \in SE$, $SE = \{se_1, se_2, \ldots, se_n\}$, where se_i is one of the user defined contract execution states, for example, *in progress*, *complete*, *pending*, etc;
- Rules: $R = \{r_1, r_2, \ldots, r_n\}$, $r_j (1 \leq j \leq n)$ is a horn clause: *consequent* ← *antecedent*.

C. The Accountable State Transfer (AST) Architecture

AST architecture introduces two extra components called *sContractMonitor* and *sContractManager* in addition to the traditional REST architecture components. *sContractMonitor* monitors the interactions between the consumer and the provider, and then feeds events to *sContractManager* through a low-coupling queuing mechanism. *sContractManager* determines the current contract execution instance's status based on the rules prescribed in the *service contract* and the events fed from *sContractMonitor*. It also maintains a *service contract* KB for all the contract execution instances so it can reason the overall contract status.

AST architecture can be classified into two categories. One is a centralised AST and the other is a peer-to-peer AST. In a centralised AST, it can be further categorized to two styles, one is an *in-line* TTP AST and the other is an *on-line* TTP AST. An *in-line* TTP AST's *sContractMonitor* acts as a HTTP proxy, seeing through all the interactions between the consumer and the provider. Based on the contract meta-data on the HTTP headers and the body message, it can verify whether the obligations have been met by checking the prescribed evidence. Then it generates the assertion events to *sContractManager*. *sContractManager*'s reasoner component determines the *service contract* execution status and maintains an up-to-date *service contract* execution KB based on the rules defined in the service contract. Figure 1 illustrates the *in-line* TTP AST model.

In an *on-line* AST Model, the service contract monitor remotely monitors the consumer and the provider separately. In a *peer-to-peer* AST model, each party will have its own *service contract* monitor and manager, and needs an arbitrator reasoner for dispute resolution.

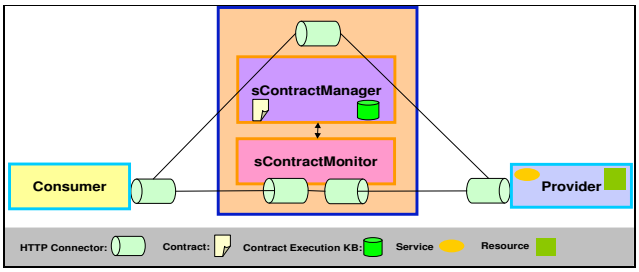

Figure 1. In-Line TTP AST Architecture

D. AST Protocols and an Example

In order to bring in *service contract* context information during REST interactions, the following AST protocols are proposed for communicating the contract meta-data:

Table II. Accountable State Transfer Protocol

AP1: Refer to service contract during service invocation: HTTP Header: sContract: *sContract_URI*;
AP2: Add service contract meta-data to a service request: HTTP Header: Required-Obligations: *action_URI list*;
AP3: Add service contract meta-data to a service response: HTTP Header: Met-Obligations: *action_URI* list:
AP4: Query on contract execution status: GET *sContract_KB_URI*?queryString;
AP5: Notify contract breach or execution abnormality: POST *involvedParty_URI* with XML payload indicating *sContract* or *sContractExecution* status.

The HTTP header in AP1 can be used in each REST interaction to establish linkage to a *service contract*. AP2's HTTP header can be used when the consumer sends out a request, indicating the request is relating to the provider's obligation as specified in the *service contract*. AP3 can be

used when the provider replies with a response, indicating that the response is relating to the fulfilled obliged actions. AP4 enables REST client and server to query the *sContractManager* for contract execution status. AP5 allows *sContractManager* to notify a client or the server on the contract execution status.

Now we use an example to illustrate how the AST works. Suppose a service provider *pr* ($pr \in P$) has signed a contract *SC* with a consumer *pc* ($pc \in P$) to provide a Credit Check service defined by *s* for a period of *T*. The contract defines the terms relating to Credit Check in a definition set *D*. The contract prescribes the provider's obligation as *Op* and the consumer's obligation as *Oc*. *Op* = {(*P_checkCredit*, *E_creditEvidence*), (*P_returnError*, *E_ErrorEvidence*)}. The consumer's obligation *Oc* = {(*C_provideInput*, *E_inputEvidence*), (*C_payFee*, *E_feeEvidence*)}. The valid action sequences are defined as either "*C_provideInput*, *P_checkCredit*, *C_payFee*" or "*C_provideInput*, *P_returnError*". Also the contract defines a set of rules *R* to determine the contract status *st* ($st \in S$) based on the evidence of the fulfilled obligations.

There is no concept of service contract in traditional REST. Both service provider and service consumer rely on other means (mostly off-line and manual) to know the performance status of the contract. With AST, the contract performance can be tracked while executing the service. Moreover, both client and server understand the "why" behind the request and response (representational state transfer) from a service contract perspective. For example, when the consumer invokes the Credit Check service, he/she issues the following request with the Http headers below:

```
GET
/credit_chk.jsp?fname=jon&lname=bond&id=102
435 HTTP 1.1
Host www.creditcheck.com
HTTP headers:
sContract:
http://sContractManager.com/creditcheck.owl
Required-Obligations: #P_checkCredit,
#P_returnError
```

With these HTTP headers, the consumer links the request to a pre-established contract, also states that the request is related to the provider's obligations *P_checkCredit* and *P_returnError* as prescribed in the contract.

When the server responds, it adds the HTTP headers to further explain the response in relation to the contract:

```
HTTP/1.1 200 OK
HTTP headers:
sContract:
http://sContractManager.com/creditcheck.owl
Met-Obligations: #P_checkCredit
Required-Obligations: #C_payFee
```

V. Implementation of AST Architecture

A. Languages for Specifying the Service Contract Model

The ontology underpinning our service contract model can be specified using Web Ontology Language (OWL), which is recommended by W3C as the standard for representing ontologies on the Web. OWL provides three sub-languages with increasing level of expressiveness: OWL-Lite (corresponding to $\mathcal{SHIF}(\mathcal{D})$ [16]); OWL-DL (corresponding to $\mathcal{SHOIN}(\mathcal{D})$ [16]); and OWL-Full which is an extension to Resource Definition Framework (RDF). Both OWL-Lite and OWL-DL provide computation completeness and decidability [17], whereas OWL-Full has maximum expressiveness but no computational guaranteed. As per the guiding principles in section III, OWL-DL is chosen to specify our *service contract* model since it has the better trade-off between expressiveness and decidability. Other benefit of using OWL-DL is that the consistency of the *service contract* can be validated using proven DL reasoners such as RACER, KAON2, PELLET, etc. However, OWL-DL has limitations. In particular it has the well known "*hasUncle*" problem; i.e. it is impossible for OWL-DL to describe the role chain of *hasParent* and *hasBrother* leading to the *hasUncle* role. To address this limitation, we leverage Semantic Web Rule Language (SWRL) for defining rules to do contract state reasoning on top of OWL-DL. SWRL is a W3C submission, extending OWL-DL axioms with a set of horn clause rules. It is basically a combination of OWL-DL and OWL-Lite with the unary/binary Datalog sublanguages of the Rule Markup Language (RuleML) [18].

While OWL-DL is decidable, SWRL is proven not decidable [17]. To solve this problem, we further restrict SWRL to DL-Safe rule. A rule *r* is called DL-Safe if each variable in *r* occurs in a non-DL-atom in the rule body. A program *P* is DL-Safe if all its rules are DL-Safe (see [17] for details). The DL-Safe restriction is exposed to ensure that the variables in the rule body are bound to only explicitly existing individuals in the KB. Our model complies with the DL-Safe restriction, which means that anonymous individuals are disregarded in reasoning on rules.

B. Service Contract Representation

We now formally define the representation for our *service contract* model.

Definition 4: The service contract execution knowledge base *Ksc* can be defined using DL's Tbox $\mathcal{T}$, Abox $\mathcal{A}$, adding the SWRL DL-Safe rules $\mathcal{H}$, thus $Ksc = (\mathcal{T}, \mathcal{A}, \mathcal{H})$, where:

- A TBox $\mathcal{T}$ consists of a finite set of concept inclusion axioms of the form $C \sqsubseteq D$, a finite set of role inclusion axioms of the form $R \sqsubseteq S$ and transitivity axioms $Trans(R)$, where *C* and *D* are concepts, *R* and *S* are roles;
- An ABox $\mathcal{A}$ consists of a finite set of concept and role assertions and individual equalities/inequalities C(*a*), R(*a*, *b*), $a = b$, and $a \neq b$, respectively;
- A horn rule set $\mathcal{H}$ consists of a finite set of horn axioms. A horn axiom consists of an antecedent (body) and a consequent (head) in the form of: $a \leftarrow a_1 \wedge a_2 \wedge \ldots \wedge a_n$, where *a*, a_i $(0 \leq i \leq n)$ are atoms in rules that can be of the form $C(x)$, $P(x, y)$, $Q(x, z)$, sameAs(x, y) or differentFrom(x, y),

and *C* is an OWL concept; *P* is an object-valued property; *Q* is a data-valued property; *x*, *y* are either variables or individuals; and *z* is either a variable or a data value. Variables *x*, *y*, *z* must be bound to named individuals in the KB to satisfy the DL-Safe rule criteria.

C. *Service contract Ontology and Axioms*

Based on OWL-DL and DL-Safe criteria, we define a *service contract* model that captures the fundamental aspect of a service contract. Figure 2 shows a simplified version of the ontology. A *scContract* class has contract term definitions (*Defintions* class); it involves *Party* class, which has subclasses of *Provider* and *Consumer*. Each party has *Obligation* which consists of multiple *Action* and *Evidence* pairs. A domain specific contract class like *CreditCheckContract* inherits from the generic *scContract* class. Such domain specific contract instance may be executed multiple times. Each execution is an instance of *scContractExecution* class. The *scContractExecution* instance executes *Obligations* as defined in the *scContract* and produces *Evidence* instances. If each *Action* instance can be proven by the respective *Evidence* instance, then the obligation is fulfilled. Otherwise either *Provider* or *Consumer* may breach the contract depending on the specific contract rules, which can be defined as the axioms for a domain specific service contract model. A detailed analysis on our model's action semantics and the model validation technique using Coloured Petri-Nets are documented in [15].

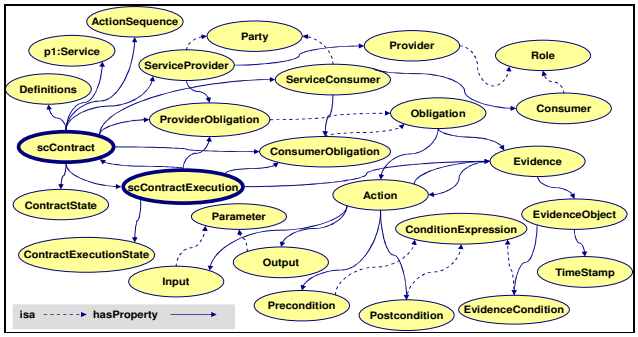

Figure 2. A Simplified Version of the *Service Contract* Ontology

We here list some axioms that determine the contract state for our generic *service contract* model. For domain specific service contracts, these axioms can be extended; overwritten or new axioms can be developed based on the specific terms and conditions of the underlying contract.

Table III Partial Generic Contract Axioms

scContract(?*x*) ∧ *hasExecutionInstance*(?*x*, *false*) → *inState*(?*x*, *INIT*)	(1)
scContract(?*x*) ∧ *executedBy*(?*x*, ?*y*) ∧ *isContractExpired*(?*x*, *false*) → *inState*(?*x*, *IN_PROG*)	(2)
scContractExecution(?*x*) ∧ *startsAt*(?*x*, ?*y*) ∧ *noEvidenceSupportObligations* (?*x*, ?*z*) ∧ *isTimeOut* (?*x*, *false*) → *inExecutionState*(?*x*, *EINIT*)	(3)
scContractExecution(?*x*) ∧ *execute*(?*x*, ?*y*) ∧ *specifiesObligation*(?*y*, ?*z*) ∧ *mustDo*(?*z*, ?*a*) ∧ *verifiedBy*(?*a*, ?*b*) ∧ *produceEvidence*(?*x*, ?*b*) → *fulfilledObligations*(?*x*, ?*z*)	(4)

Axiom 1 states that if a *scContract* instance does not have any execution instance, then the *scContract* is in the initial (*INIT*) state. Axiom 2 states that if the *scContract* instance is executed by some execution instances and the contract is not expired, then the service contract state is in-progress (*IN_PROG*). Axiom 3 says if the instance of *scContractExecution* starts at a particular time, but no evidence produced to prove the fulfilment of obligations, and the execution is not time out yet, then the contract execution instance is in initial (*EINIT*) state. Axiom 4 determines whether a particular obligation is fulfilled based on the collected evidence.

D. *Prototype Implementation*

1) *Overall Prototype Architecture*

Figure 3 depicts the Credit Check service prototype that implements the in-line TTP AST in Figure 1. The main components are described below:

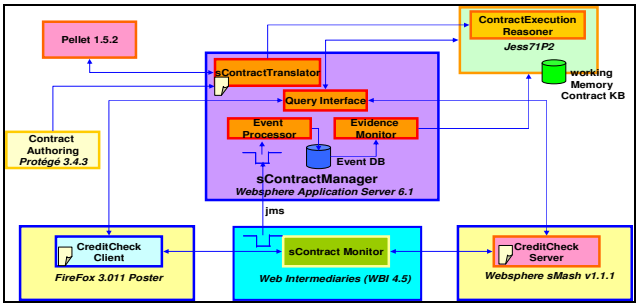

Figure 3. Prototype in-line TTP AST Implementation

CreditCheck Client: We used Firefox Poster to simulate a Credit Check client. Firefox Poster provides an intuitive interface for sending REST requests with user defined HTTP Headers.

sContractMonitor: *sContractMonitor* is built on a RESTful component – Web intermediary. We used IBM's Web Intermediaries Development Kit 4.5 (WBI DK) [19] as the underlying Web intermediary platform, creating a Monitor plug-in to track the HTTP messages. The messages will be sent to *sContractManager* through JMS.

CreditCheck Server: A Credit Check server is developed and hosted in Websphere sMash [20], which provides an environment for developing and hosting REST applications.

Contract Authoring: Protégé 3.4.3 [21] is used as the service contract authoring tool. SWRLtab in Protégé is used for SWRL rule authoring.

Pellet 1.5.2: Pellet 1.5.2 [22] is the DL reasoner that is used to classify the terms in *service contract* and check contract consistency at design time.

sContractManager: *sContractManager* is implemented in a Websphere Application Server (WAS). It consists of

sContractTranslator, *QueryInterface*, *EventProcessor* and *EvidenceMonitor*. *sContractTranslator* converts the *service contract* from OWL-DL / SWRL to Jess facts / rules using XSLT, then sends them to *ContractExecutionReasoner*. *EventProcessor* picks up the raw monitoring data, storing evidence data into an event database. Then *EvidenceMonitor* checks if the evidence is valid, if so, it sends assertions to *ContractExecutionReasoner*. Finally the *QueryInterface* allows contract status query from both the *client* and the *server*.

ContractExecutionReasoner: This component receives Jess facts or rules, and then invokes Jess71p2 [23] to do reasoning, maintaining the contract execution KB in Jess's working memory.

2) Implementation of Hybrid Reasoning Mechanism

Based on architectural decision 4 outlined in Section III.D, we adopt a hybrid approach to reasoning. In contract authoring stage, DL reasoner like Pellet will be used for normal TBox and ABox reasoning in design time. After the service contract is developed, the OWL-DL ontology will be translated to Jess facts via one XSLT file, while the SWRL rules will be translated to Jess user-defined rules via another XSLT file. Additionally, we need to import pre-defined Jess rules, which are transformational implementations for OWL semantics [24]. Then the Jess facts and (pre-defined and user-defined) rules will be fed into the Jess engine, taking advantage of the fast Rete algorithm for contract state reasoning at runtime. In our prototype, we created a *CreditCheckServiceContract* based on *scContract* in Figure 2. The contract is between service provider CreditBureau and consumer MortgageBank. CreditBureau*'s* obligation is to complete actions *P_checkCredit* or *P_returnError* if exception occurs. MortgageBank's obligation is to complete actions *C_provideInput* and *C_payFee*. The actions need to be proven by evidences which are also defined in the *service contract* ontology. This contract instance will be executed multiple times during the valid contract period. Each execution instance is an instance of *CreditCheckExecution* class. In addition to the generic axioms listed in Table III, domain specific axioms can be defined to reason *Credit Check* specific execution state. Two example rules used to determine if the service participants breach the obligation is listed below. Other axioms are omitted due to space limit.

Table VI Credit Check Specific Axioms

CreditCheckServiceExecution(?*x*) ∧ *isTimeOut*(?*x, true*) ∧ *fulfilledObligations*(?*x, OC_ProvideCustomerDetails*) ∧ *noEvidenceSupportObligations*(?*x, OP_ProvideCreditScore*) ∧ *noEvidenceSupportObligations*(?*x, OP_ReturnError*) → *inExecutionState*(?*x, EP_NOPF*)	(5)
CreditCheckServiceExecution(?*x*) ∧ *isTimeOut*(?*x, true*) ∧ *fulfilledObligations*(?*x, OP_ProvideCreditScore*) ∧ *noEvidenceSupportObligations*(?*x, OC_PayServiceFee*) → *inExecutionState*(?*x, EC_NOPF*)	(6)

Axiom 5 states that if MortgageBank has provided input for credit check, but CreditBueau hasn't provided credit score nor returned error; and the execution is timeout, mark the current contract execution instance as status *EP_NOPF* (Service Provider Non-Performing obligations). Simular rule defined in Axiom 6 to determine consumer non-performing obligation. Note that the reasoning power is limited by the expressiveness of OWL DL and SWRL, so normal programming logic is still needed to address the limits of DL reasoner and rules engine. For example, the predicate *isTimeOut* in Axiom 5 is very difficult for reasoners to decide because that there is no current time concept in OWL-DL, nor is provided in the SWRL's temporal built-in. However, it can be easily done in a Java program by checking the current time, and producing an assertion triple to the Jess engine. So our hybrid approach can be simply described as: DL reasoning at design time, Jess Rule reasoning at runtime, with input assertions produced by a Java program.

3) Results and Discussions

The test environment is based on a PC with a duo-core 2.4 GHz Intel CPU, 2GB RAM running on Windows XP. After completing the design of Credit Check *service contract* in Protégé, the Pellet reasoner is invoked to check the consistency of the ontology. It took 1.68 seconds to classify the taxonomy and 3.81 seconds to check the consistency of the service contract model. Then through *sContractTranslator*, both OWL-DL and SWRL rules are translated into Jess facts and rules. It took 1562 milliseconds for running *sContractTranslator* to translate OWL facts to Jess facts, generating 1948 asserted triples. Jess took less then 1 second to reason the input jess facts and jess rules, generating 2621 inferred triples in its working memory. As Jess' Rete algorithm is linier to the number of rules and polynomial to the number of objects [25], when the KB grows, we need to scale up the underlying environment to cater for the load. Once the translation is done, the *sContractManager* is waiting for event collected by the *sContractMonitor*. Once *evidenceMonitor* picks up an event, it validates if it is an evidence for a particular obligation.

There is no noticeable performance impact on both client and server, mainly due to the decoupling of *sContractManager* and *sContractMonitor*. The *sContractMonitor* is just a read-only plugin installed in a Web proxy; which is a widely adopted pattern in today's internet environment.

The limitation of translating OWL-DL to Jess facts is documented in [26]. In our model, since we only use OWL-DL reasoner at design time to verify the consistency of concepts in the *service contract*, in addition we apply DL-Safe restrictions in our model; and we use Jess rule engine for run time reasoning, hence our reasoning is sound and complete in each reasoning stage. Theoretically, we acknowledge the loss of information when combining the two reasoning paradigms with interfaces for translating OWL-DL to Jess facts. However, the loss information in our

model is about reasoning on anonymous individuals, and such anonymous individuals in rules are disregarded due to our adoption of the DL-safe restriction.

Another interesting issue is about negation. OWL-DL is based on an open world assumption and thus can not reason "negation as failure". In our model, we work around this problem by defining properties like *noEvidenceSupportObligations*. The evidence monitor *EvidenceMonitor* is responsible to generate a Jess assertion on this property if no evidence is found. Therefore we can use Jess rule engine to reason non-fulfilled obligations. This demonstrates the strengths of our hybrid reasoning approach.

VI. Conclusion and Future Work

REST is increasingly becoming a key architectural style, thanks to the growing popularity of the Web 2.0 technology. REST services also form a major part of the services offered through SaaS or Cloud Computing. Thus building accountability mechanism in the REST architecture is crucial for the long-term viability of these new business models. In this paper, we address the accountability gap in REST by proposing an innovative architecture extension AST to enable accountability in RESTful services. Our contributions can be summarized as: Firstly, we outline the architectural principles and decisions for enabling accountability in an e-Services environment. Secondly, guiding by those principles and decisions, we propose a novel AST architecture with an accountable state transfer protocol to enable service accountability, yet retaining scalability of REST architecture. The new architecture seamlessly integrates *service contract* semantics into the traditional syntactic-based REST services. Thirdly we apply the formal *service contract* model in [15] to design a Credit Check domain specific *service contract* with a hybrid reasoning mechanism that leverages strengths from formalisms like DL, Rules and traditional programming language. The hybrid reasoning mechanism provides capabilities like temporal reasoning and "negation as failure" that are not found in normal DL and SWRL. Moreover, it separates reasoning in design-time stage and runtime stage, taking into account of both expressiveness and computational complexity of the underlying logic formalisms. Lastly we provide a prototype implementation for a Credit Check service that demonstrates the practicality of AST architecture, proving that the new AST architecture can be implemented with existing products and technologies. All these are not covered by [15].

The new architecture allows service obligation disclosure, obligation tracking, and action justification in a stateless service environment. With such capabilities provided at the architectural level, effectively service participants can be held accountable for each representational state transfer during service consumption.

Finally we observe that the future work entails applying the *service contract* model to SOAP-based Web Services model and Enterprise Service Bus (ESB) solutions.

References

[1] R.T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Dissertation, Doctor of Philosophy, University of California, Irvine, 2000.

[2] A. Schedler, Self-Restraining State: Power and Accountability in New Democracies, pp. 13-28. Lynne Reiner Publishers, 1999

[3] J.R. Erenkrantz, M.M. Gorlick and G. Suryanarayana, From Representations to Computations: The Evolution of Web Architectures, ESEC/FSE'07, ACM, 2007.

[4] R. Khare and R.N. Taylor, Extending the REpresentational State Transfer (REST) Architectural Style for Decentralized Systems, Proceedings of the 26th Intl. Conf. on Software Engineering, 2004.

[5] Kwei-Jay Lin, Joe Zou and Yan Wang, Key Note, Accountability Computing for e-Society, The International Conference on Advanced Information Networking and Applications, IEEE, 2010.

[6] A. Dan and et al.: Business-to-Business Integration with TPAML and a Business-to-Business Protocol Framework, IBM System Journal, 2001

[7] L. Xu, Monitoring Multi-party Contracts for E-business, Dissertation, Doctor of Philosophy, University of Toronto, 2004

[8] A. Daskalopulu, Logic-Based Tools for the Analysis and Representation of Legal Contracts, Dissertation, Doctor of Philosophy, University of London, 1999.

[9] G. Governatori and Z. Milosevic, A Formal Analysis of a Business Contract, Language, Proc. Int'l J. Cooperative Info. Sys., vol. 15, no. 4, 2006, pp. 659–685, 2006.

[10] M.J. Carlos and et al., Run-Time Monitoring and Enforcement of Electronic Contracts, Electronic Commerce Research and Applications, vol. 3, no. 2, 2004, pp. 108–125, 2004.

[11] A.D.H. Farrell and et al., Performance Monitoring of Service-Level Agreements for Utility Computing Using the Event Calculus, Proc. 1st Int'l Workshop Electronic Contracting, pp. 17–24, IEEE Press, 2004.

[12] P. Grefen and et al., CrossFlow: Cross-Organizational Workflow Management in Dynamic Virtual Enterprises, J. Computer Systems Science and Eng., vol. 15, no. 5, 2000, pp. 277–290

[13] H. Liu and et al., Modeling and Reasoning about Semantic Web Services Contract using Description Logic, The Ninth International Conference on Web-Age Information Management, IEEE, 2008.

[14] B. Grosof and T. Poon, SweetDeal: Representing Agent Contracts with Exceptions Using XML Rules, Proc. 12th Int'l Conf. World Wide Web, pp. 340–349, ACM Press, 2003.

[15] J. Zou, Y. Wang and K.J. Lin, A Formal Service Contract Model for SaaS and Cloud Services, paper submitted to SCC 2010, 2010.

[16] I. Horrocksand P.F. Patel-Schneider, Reducing OWL Entailment to Description Logic Satisability, 2nd International Semantic Web Conference, Florida, USA, October, 2003.

[17] B. Motik, U. Sattler and R. Studer, Query Answering for OWLDL with Rules, The SemanticWeb ISWC: 3rd International Semantic Web Conference, Hiroshima, Japan, 2004

[18] I. Horrocks and P.F. Patel-Schneider, A proposal for an OWL rules language. In The Thirteenth International World Wide Web Conference, New York, May ACM Press, 2004.

[19] IBM, Web Intermediaries DK, http://www.almaden.ibm.com/cs/wbi/

[20] IBM, Available at http://www.projectzero.org/

[21] Protégé, Available at http://protege.stanford.edu/

[22] Pellet, Available at http://clarkparsia.com/pellet/download/

[23] Jess, Available at http://www.jessrules.com/download.shtml

[24] J. Mei, E. Paslaru Bontas and Z. Lin, OWL2Jess: A Transformational Implementation of the OWL Semantics, ISPA Workshops 2005, LNCS 3759, pp. 599–608, Springer-Verlag Berlin Heidelberg, 2005.

[25] C.L. Forgy, On the Efficient Implementation of Production Systems, Ph.D. dissertation, Carnegie-Mellon University, 1979.

[26] J. Mei, and E. Paslaru Bontas, Reasoning Paradigms for SWRL-enabled Ontologies,Protégé With Rules Workshop, Madrid, 2005.