Newcastle University

COMPUTING SCIENCE

An Architecture for Negotiation and Enforcement of Resource Usage Policies

Carlos Molina-Jimenez, Santosh Shrivastava and Stuart Wheater

TECHNICAL REPORT SERIES

No. CS-TR-1381 April 2013

TECHNICAL REPORT SERIES

No. CS-TR-1381 April, 2013

An Architecture for Negotiation and Enforcement of Resource Usage Policies

C. Molina-Jimenez, S. Shrivastava and S. Wheater

Abstract

Advances in Cloud computing are making it possible for service providers to offer computational resources such as storage and compute power (infrastructure as a service, IaaS) to sophisticated enterprise application services (software as a service SaaS) to remote clients for a fee on a highly dynamic basis. As in any business transaction, client access to a service is regulated by a legal Service Agreement (SA). A service agreement needs to be negotiated and agreed between the provider and the client before the latter can use the service. Then on, both the client and the provider will need assurances that service interactions are in accordance with the SA, and any violations are detected and their causes identified. There is thus a need for automated support for negotiation and enforcement of service agreements. This paper discusses key design issues for such a system, of which the main one is to ensure that the policies (termed also clauses) contained in an SA are logically sound and that they work in harmony with any private policies of the client and the provider. The paper presents an architecture and a proof of concept implementation.

© 2013 Newcastle University.
Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

MOLINA-JIMENEZ, C., SHRIVASTAVA, S., WHEATER, S.

An Architecture for Negotiation and Enforcement of Resource Usage Policies
[By] C. Molina-Jimenez, S. Shrivastava, S. Wheater

Newcastle upon Tyne: Newcastle University: Computing Science, 2013.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1381)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1381

Abstract

Advances in Cloud computing are making it possible for service providers to offer computational resources such as storage and compute power (infrastructure as a service, IaaS) to sophisticated enterprise application services (software as a service SaaS) to remote clients for a fee on a highly dynamic basis. As in any business transaction, client access to a service is regulated by a legal Service Agreement (SA). A service agreement needs to be negotiated and agreed between the provider and the client before the latter can use the service. Then on, both the client and the provider will need assurances that service interactions are in accordance with the SA, and any violations are detected and their causes identified. There is thus a need for automated support for negotiation and enforcement of service agreements. This paper discusses key design issues for such a system, of which the main one is to ensure that the policies (termed also clauses) contained in an SA are logically sound and that they work in harmony with any private policies of the client and the provider. The paper presents an architecture and a proof of concept implementation.

About the authors

Carlos Molina-Jimenez received his PhD in the School of Computing Science at the University of Newcastle upon Tyne in 2000 for work on anonymous interactions in the Internet. He is currently a Research Associate in the School of Computing Science at the University of Newcastle upon Tyne where he is a member of the Distributed Systems Research Group. He is working on the EPSRC funded research project on Information Coordination and Sharing in Virtual Enterprises where he has been responsible for developing the Architectural Concepts of Virtual Organisations, Trust Management and Electronic Contracting.

Professor Santosh Shrivastava was appointed Professor of Computing Science, University of Newcastle upon Tyne in 1986. He received his Ph.D. in computer science from Cambridge University in 1975. His research interests are in the areas of computer networking, middleware and fault tolerant distributed computing. The emphasis of his work has been on the development of concepts, tools and techniques for constructing distributed fault-tolerant systems that make use of standard, commodity hardware and software components. Current focus of his work is on middleware for supporting inter-organization services where issues of trust, security, fault tolerance and ensuring compliance to service contracts are of great importance as are the problems posed by scalability, service composition, orchestration and performance evaluation in highly dynamic settings. Professor Shrivastava sits on programme committees of many international conferences/symposi. He is a member of IFIP WG6.11 on Electronic commerce - communication systems, and sits on the advisory board of Arjuna technologies Ltd.

Dr Stuart Wheater: He graduated with First Class Honours in Computing Science and completed his PhD in 1990 at Newcastle University, UK. He is the Chief Architect and co-founder Arjuna Technologies. As a software designer and developer, he has played a central role in the design and implementation of several commercial products. In particular Agility, a Cloud Computing Platform for building Federated Clouds using Service Agreements mediated Policies. He was also leading figure in the construction of a CORBA based transactional workflow system (OPENflow) and the industry proven Transaction Service and Message Service from Arjuna. He also designed and developed the HP-ORB product, to replace a third-party incumbent ORB across HP's product range. As a researcher, he is interested in transactions, long-lived process support and cloud computing applications. He is the author and co-author of several publications in leading journals and conferences.

Suggested keywords

SERVICE ORIENTED COMPUTING
SERVICE AGREEMENT NEGOTIATION
UPDATING
TERMINATION AND ENFORCEMENT
POLICY CONSISTENCY

An Architecture for Negotiation and Enforcement of Resource Usage Policies

Carlos Molina–Jimenez
School of Computing Science
Newcastle University, UK
Carlos.Molina@ncl.ac.uk

Santosh Shrivastava
School of Computing Science
Newcastle University, UK
Santosh.Shrivastava@ncl.ac.uk

Stuart Wheater
Arjuna Technologies, UK
Stuart.Wheater@arjuna.com

***Abstract*—Advances in Cloud computing are making it possible for service providers to offer computational resources such as storage and compute power (infrastructure as a service, IaaS) to sophisticated enterprise application services (software as a service SaaS) to remote clients for a fee on a highly dynamic basis. As in any business transaction, client access to a service is regulated by a legal Service Agreement (SA). A service agreement needs to be negotiated and agreed between the provider and the client before the latter can use the service. Then on, both the client and the provider will need assurances that service interactions are in accordance with the SA, and any violations are detected and their causes identified. There is thus a need for automated support for negotiation and enforcement of service agreements. This paper discusses key design issues for such a system, of which the main one is to ensure that the policies (termed also clauses) contained in an SA are logically sound and that they work in harmony with any private policies of the client and the provider. The paper presents an architecture and a proof of concept implementation.**

***Keywords*-service oriented computing; service agreement negotiation, updating, termination and enforcement; policy consistency.**

I. Introduction

We consider a cloud computing environment that enables service providers to provision, in a rapid manner, on-demand network access to shared pool of compute resources (that can range from storage, compute power to applications and services) to consumers for a fee. As in any business transaction, consumer (client) access to a service will be underpinned by a contract, that we will refer to here as a Service Agreement (SA). A service agreement needs to be negotiated and agreed between the provider and the client before the latter can use the service. Then on, both the client and the provider will need assurances that service interactions are in accordance with the SA, and any violations are detected and their causes identified. There is thus a need for automated support for negotiation and enforcement of service agreements.

Electronic representation of the relevant parts of an SA is a pre-requisite for any such automation. Here we are most interested in *service description* part of an SA that specifies *resource usage* in terms of service delivery (dealing with quality of service) and consumption (dealing with usage pattern). For example, the SA might stipulate that a client is permitted to submit 100 requests per second and that the provider is obliged to respond within three seconds. Ideally, it should be possible to encode an SA as a set of executable business policies that can be evaluated by either party for controlling service interactions.

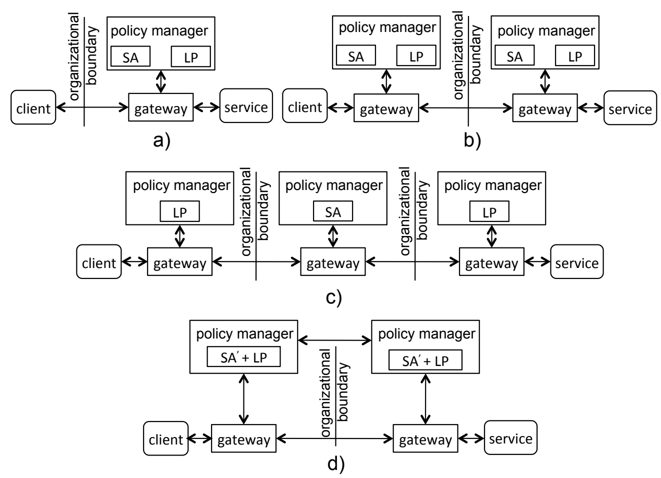

Figure 1. Deployment alternatives.

Fig. 1–a) shows a simple scheme where the provider uses a **Policy Manager (PM)** module (loaded with an executable version of the SA) for controlling access to the service by the client. The gateway acts as a policy enforcement point that either allows or prohibits access to the service as directed by the PM which is in essence a policy decision point. For example, let us consider the following policies from a simple SA about a service provided on a pre-paid basis:

1) *Clients can open an account by purchasing a single unit of prepaid time at the price of 10 euros.*
2) *A unit of prepaid time is considered consumed when the client consumes 100 minutes of connection time.*
3) *An open account can be topped up by the client by purchasing additional units of prepaid time.*
4) *Accounts with no prepaid time left will be declared closed by the service.*
    a) *The service is entitled to evict calls in progress that run out of prepaid time.*

Typically, a provider will have a set of local (private) business policies (LP) for customising an SA for different classes of clients. For example, the provider could be a bit lenient whilst dealing with valued customers ('gold clients') who exceed the prepaid time limit. Here is a sample clause of such a policy:

1) *Calls from gold clients that overrun their prepaid credits are granted up to ten minutes of discretionary time that can be used only by the call in progress.*
2) *Ignore the cost incurred by the use of discretionary time after evicting a call.*

In Fig. 1–a) we show that the PM uses both the SA and LP for controlling the gateway. The PM is the key component needed for automating negotiation and enforcement of service agreements and is the subject of discussion of this paper. Below we discuss the main requirements of a PM, and in the rest of the paper we describe the approach we have adopted and present a proof of concept implementation.

To begin with, we observe that policy managers and gateways can be deployed in several configuration alternatives, and not just as shown in Fig. 1–a). In Fig. 1–a), the decision whether the client's service access is compliant with respect to the SA is taken by the PM of the provider; however, there may be situations where the client's organization independently wants to perform such a compliance check, in which case, the symmetric deployment scheme of Fig. 1–b) is relevant. The client's organization might have its own local policies that put additional constraints on who/when service access is permitted (e.g., a local policy might be that only a senior manager is permitted access). Another deployment possibility is depicted in Fig. 1–c): here an independent third party is responsible for checking SA compliance, whereas the parties only check for their local policy compliance. The configuration depicted in Fig. 1–b) opens up the possibility of the two PMs being able to interact and negotiate to install a new SA on the fly. For example, a customer of a service might wish to upgrade to become a premier customer, in which case a new SA will come in force. This possibility is hinted at in Fig. 1–d) where $SA^{'}$ is under negotiation. In summary, the PM should be *modular* in structure and capable of being deployed in various configurations.

The machine interpretable language used for encoding SA and LP should be *expressive and usable*. By usability we mean that a technical person who understands SAs and LPs written in a natural language should be able to translate them into executable versions with relative ease. By expressiveness we mean that the language should be widely applicable. Finally, we require that the encoded versions of SAs and LPs be *amenable to formal analysis*, meaning there should be tools available for validating the logical consistency of an SA and LP taken individually and together. This is important as the intended meaning of clauses expressed in a natural language can be remarkably hard to capture and represent in a rigorous and concise manner for computer processing.

Our PM is based on the concept of *contract compliance checking* that we have developed earlier, and described in [1]. The concepts discussed in [1] also underpin the rule based contract specification language called EROP (for Events, Rights, Obligations and Prohibitions) and a contract compliance checking service (CCC) for contracts/service agreements written in EROP [1], [2]. The CCC essentially acts as the PM. The CCC is modular, as it has been developed for use as a third party service, so it can be deployed in any of the settings shown in Fig. 1. Using a number of examples, we show that the EROP language provides a uniform way of encoding SA and LP, satisfying the requirement of expressiveness and usability. The CCC is amenable to model checking, thereby providing a way for validation; we have indeed developed a high–level model checking tool for this purpose [3]. We have incorporated the CCC within the cloud management platform called Agility [4]. Agility is intended to assist two or more independently administered parties in sharing their IT resources in peer–to–peer or consumer–provider interaction. Agility provides basic support for negotiation. In Future Work section of the paper we describe how our work can be extended to support automated negotiation between PMs.

II. COMPLIANCE CHECKING

Clauses included in SAs and LP (Fig.1) stipulate the rights (something that a party is allowed to do) , obligations (something that a party is expected to do) and prohibitions (something that a party is not expected to do unless it is prepared to be penalised) of the parties. The clauses also stipulate when, in what order and by whom the operations are to be executed. Business partners exercise their rights, obligations and prohibitions by executing their corresponding business operations. As operations are executed, rights, obligations and prohibitions are granted to and revoked from business partners. At a given moment, each business partner can have several rights, obligations and prohibitions, in force. This idea is at the heart of the functionality of the CCC that we have implemented [2]: the CCC is an observer of execution of operations that determines and declares whether the operation is or is not contract compliant.

With each participant (role player), we associate a *ROP set*: the set of Rights, Obligations and Prohibitions currently in force. We use the set $B = \{bo_1, \ldots, bo_n\}$ to specify all the primitive business operations stipulated in a SA or LP. The CCC declares the execution of bo_i to be *contract compliant* if it satisfies the following three requirements and declares it *non–contract–compliant* if it does not:

- C1) $bo_i \in B$;
- C2) it matches the ROP set of its role player (meaning, the role player has a right/obligation/prohibition to execute that operation);
- C3) it satisfies the constraints stipulated in the contractual clauses.

The significance of the ROP sets in our model is that they allow to abstract the behaviour of the CCC as that of a conventional reactive system [5] with $m + 1$ states $S = \{s_0, \ldots, s_m\}$ where each state s_i represents the current state of the ROP sets. As a reactive system, the CCC remains in a given state s_i awaiting the arrival of events, when a contract–compliant event arrives, the CCC executes an action and progresses to state s_j. No state changes occur or actions are executed when the event is non-contract–compliant. The main action executed consists in updating the ROP sets: rights, obligations and prohibitions from state s_i are disabled and those that determine state s_j are enabled. The salient feature of this state–centric model is that it is intellectually manageable as there are well understood formal methods and software tools that can help reason about the correctness of both the model and its implementation. For instance, the CCC can be directly implemented as a conventional Event Condition Action (ECA) system.

In Fig. 2 we show how the CCC can be used as a PM to determine if the operations executed by a client are compliant with the policies specified in a client–provider SA. Only a single policy is shown in the figure. The event e_i represents the execution of an operation such as *call*. Upon evaluating the event against the ECA rules, the PM declares either e_i *is SA compliant* or e_i *is not SA compliant*.

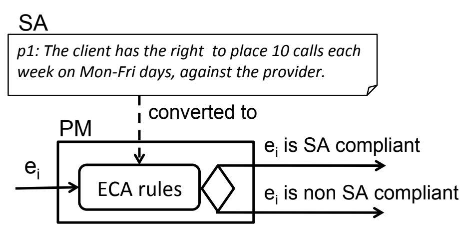

Figure 2. Abstract view of a policy manager.

Our current implementation of the CCC is based on JBoss Drools [6]. The rules that encode the SA clauses can be written either in EROP or Jboss drool language. Supportive functions such as event queues and time management, and examination of event logs are implemented as Java classes.

III. Expressiveness and Usability

To support our claims about the expressiveness and usability of the EROP notation, we will show how it can be used to encode typical and realistic examples of both SAs and LP. It should become apparent that modelling current rights, obligations and prohibitions explicitly—the core concept of EROP— makes our notation clear and intuitive.

We show examples of policies that regulate the execution of operations related to resource consumption (CPU, storage, etc.). In addition, to emphasis usability, we include examples of policies that regulate the execution of high level business operations such as submission of purchase orders and payments. In the latter, the execution of operations can be abstracted as the occurrence of events that indicates the initiator and responder of the operation, time stamp, and other parameters. However, in resource consumption policies, the execution of operations alone cannot determine the observance of a right, obligation or prohibition, equally important is the impact (e.g., amount of storage space consumed) of the operation. This information can be mapped to the occurrence of events, as well, for example, a storage monitor can be deployed to produce an event when a storage quota is exhausted.

We show the policies written in English and next we show their corresponding ECA rules in EROP notation. The rules are in pseudo-code; they abstract away several details, yet they include enough parameters to help appreciate the expressiveness of EROP. In the rules, lines that begin with # are commentaries. *R*, p and *e* stand for rule, policy and event, respectively. $R1- > p2$ means *rule 1 is related to policy 2*. The operator *in* verifies if the event is currently in the party's rights, obligations (obligs) or prohibitions. For example, *exeJob in math.obligs* will return *true* if the operation *exeJob* is currently in math's obligations, otherwise it returns *false*. The operators $+$ and $-$ grant and remove, respectively rights, obligations and prohibitions; thus $math.right+ = evictJob$ grants the right to execute operation *evictJob* to a party (role player) called *math*. The outcome of the evaluation of the event (*e*) is shown as *PCo* (Policy Compliant) This outcome should be regarded as a message sent by the PM to its gateway to instruct it to permit the operation under question. The assumption is that the gateway takes the absence of the *PCo* message as an instruction to deny the operation, but these are implementation details that we do not discuss here due to space constraints.

A. Condor example

The following policies are representative of Condor— a load managing system that allows a party (e.g, a university department) to share its idle resources [7]. Imagine they are deployed by the Math department's administrator willing to share his PC cluster with users from external departments. Each PC in the cluster works within these policies.

Math's local policies

1) *This PC is willing to execute jobs submitted by external users Mon–Sat from 8 pm to 9 am.*
2) *Jobs that exceed this time frame will be evicted immediately and without further notice.*
3) *External users are prohibited to submit jobs to this PC if its current average CPU usage is above 10%.*
4) *External users are prohibited from instantiating the the execution of more than three copies of a job, simultaneously.*
5) *This PC runs Linux and has 4 Gbytes of RAM and 850 GB of disk available for temporal files.*

To enforce its LP, Math can express them as EROP rules as shown by the next two examples and load them into its policy manager, for example, like in Fig. 1–a) and c).

```
#R1->p1: accept job
when e==exeJob && exeJob in math.obligs
 && e.user==external && e.ts==[Mon--Sat; 20--09 hrs]
 && Load<10%
then PCo; math.rights+=evictJob;
end
#R2->p2 evict jobs violating time frame
when e==evictJob; e.timeFrameViolation==TRUE
 && e.user==external && evictJob in math.rights
then PCo; math.rights-=evictJob
end
```

The second line of *R1* verifies if the event is *exeJob* (a request to execute a job) and that *exeJob* is currently in Math's obligations. The third and fourth lines verify conditions. The fift line produces a PCo message and grants Math's the right to evict, if necessary, the job. *R2* triggers to evict jobs violating the time frame, as stipulated by policy 2.

Imagine now that the Math and Biology administrators agree on the following SA.

Math–Biology SA policies

1) *Biology users have the right to execute jobs from 8 pm to 9 am.*
2) *Owners of jobs that threat to extend their execution beyond 9 pm will be notified by 8:30 and asked to remove their jobs.*
3) *If the owner takes no action by 8:45 am, his job will be evicted and queued into a dedicated cluster where its is likely to experience long delays.*
4) *Biology users are prohibited to submit jobs to a PC if its current CPU usage is above 10%.*
5) *Math is obliged to provide Linux and Windows machines with 4 Gbytes of RAM and 160 GB of disk available for temporal files.*

These SA policies can also be expressed in EROP and enforced by a policy manager, for example like in Fig. 1–a), b) or c). Here is the example for policy 1. Notice that the conditions in the third line restrict the submission time but not the day; this conflicts with the third line of Math's R1 which accepts submissions only on Mon–Sat.

```
#R1->p1: submit job
when e==exeJob && exeJob in math.obligs
 && e.user==external && e.ts==[20--09 hrs]
 && Load<10%
then PCo; math.rights+=evictJob;
end
```

B. Buyer–Seller example

The following policies are extracts from a SA between a buyer (inspired by [8]).

Buyer–Seller SA policies

1) *The* buyer *has the right to submit purchase orders (PO) to the seller, that shall include* itemName*, the desired delivery time (*dt*) and the payment (*pay*).*
2) Buyer *has the right to cancel his PO before* dt*.*
3) *A successful cancellation obliges the* seller *to reimburse 90%* pay *to the buyer.*
4) *The* seller *shall claim full payment when he delivers by* dt *and the* buyer *has not cancelled the PO.*

The ECA rules in EROP are shown next. Fig. 1–a, b and c show three possible alternatives to deploy them.

```
#R1->p1,p2,p4: accepts PO, grants buyer right to
#cancelPO and impose oblig to deliver on seller.
when e==PO && e.orig==buyer && PO in buyer.rights
then PCo, buyer.rights+=CancelPO;
     seller.obligs+=(deliver,dt)
end
#R2->p3:cancelPO imposes oblig. on seller to refund 90%
when e==CancelPO && e.orig==buyer &&
 CancelPO in buyer.rights && e.ts<dt
then PCo, buyer.rights-=CancelPO, seller.obigs+=90%refund
end
```

The fourth line of R1 declares the event PO policy compliant and grants the buyer the right to cancel the PO and the obligation to deliver by *dt* to the seller. The right to submit another PO is not removed from the seller. The third line of R2 checks if the buyers has the right to cancel a PO and the time stamp in the event. The fourth line produces an PCo message, removes the buyer's right to cancel and imposes the obligates the seller to refund 90%.

Imagine that the buyer operates under the following LP.

Buyer's private policies

1) *The issuer of the PO must have a budget assigned to it by a designated* budgetOfficer*.*
2) *A PO is allowed only if the balance in the issuer's budget exceeds the payment amount in the PO.*
3) *The issuer's budget will be charged upon the submission of the PO.*
4) *If the item is not delivered for whatever reason, the issuer's budget will be refunded.*

These LP can be expressed in EROP as shown below and deployed, for example in the client's policy manager of Fig. 1–b) or c).

```
#R1->p1,p2,p3: right to submit PO (POsub)
when e==POsub && e.orig==buyer && POsub in buyer.rights
  && e.pay<=BudgetBalance
then PCo, ChargeBudget; buyer.rights+=refund
  if BudgetBalance<=0 then buyer.rights-=POsub
end
#R2->p4: right to be refunded is delivery fails
when e==DlvFail && e.orig==seller && refund in buyer.rights
then PCo, refundBudget; buyer.rights-=refund
end
```

We do not discuss the seller's LP, but they can be treated in a similar manner.

IV. AMENABILITY TO FORMAL ANALYSIS

A policy is usually i) written in a natural language (e.g., in English) ii) converted into computer–amenable notation (e.g., ECA rule) and iii) deployed into an existing policy base. The maintenance of the logical consistency of the policy base is crucial and challenging. Careless addition, withdraw and edition of policies might result in syntactic and more subtle logical errors like redundancy, subsumption, incompleteness, unreachability, circularity and conflicts [9]. To prevent these problems, it is advisable to evaluate the logical impact of adding, editing or removing a policy, on a policy base, before altering it. With large policy bases (hundreds of policies) , this is only possible, when policies are expressed in notations that are amenable to logical examination with automatic tools.

A close examination of the policies of the Condor example will reveal that there are several conflicts between Math's LP and Math–Biology SA policies. There is a conflict about submission days (policies 1 and 1): SA allows submission of jobs on Sun whereas Math's LP prohibits that. Secondly, there is a conflict about the time window: The second SA policy specifies notification and re–allocation allowances, whereas Math's second policy specifies immediate eviction. Third, the omission of a clause in the SA to constraint the number of copies that can be instantiated conflicts with the constrain (no more than 3 copies) stipulated by Math's fourth policy. Finally, there is a conflict between policies number 5: SA specifies both Linux and Window machines, whereas the Math's LP offer only Linux.

As a second example, take an SA policy from Amazon Cloud drive [10] (a prepaid disk storage service) that stipulates that *Amazon will renew the client's plan automatically at the end of the prepaid period unless the client sends a cancellation message before the renewal date*. This policy could conflict with the LP of a company stipulating that *Employees need authorization from their managers to renew their Cloud drive accounts*. Conflicts like this are subtle and hard to detect without the assistance of mechanical tools. Thus numerous policy languages with their respective verification tools have been suggested that range from special purpose tools (see for example [11], [12]) implemented from scratch to existing general purpose logical verifiers such as conventional model checkers. In our research, we have taken the second alternative. In particular, we have explored the suitability of Spin [13] in the verification and testing of policies and have produced encouraging results [9], [14]. We have used Spin with both standard Promela and an extended version of Promela [3] to verify the logical consistency of SA policies before conversion into EROP rules [9]. In addition, we have used Spin as test case generator to validate the execution of EROP rules against errors introduced at conversion from English to EROP rules and by the execution environment [14]. As pointed out in[15], model–checking of large systems (say contracts with 50 or more clauses) can quickly result in state explosion; we argue that this issue can be prevented by means of abstraction techniques

It is worth emphasising that we are interested here in offline detection of logical errors. That is, the goal is to identify the situations (occurrences of an event or event patterns) that will, or are likely to drive the policy base out of consistency at run–time. Once the error, precisely the potential threat, has been identified several measures can be taken by the policy administrator to address the issue. For instance, trivial syntactic errors are immediately corrected after detection. Redundancy can be ignored in applications where it impacts efficiency but without logical implications. Likewise, the administrator might decide to introduce preventive measures against potential conflicts that are likely to happen frequently or have catastrophic impacts; alternatively, he or she might decide to live with the threat of a conflict and take corrective measures only when it actually materializes. *Metapolicies* (policies about policies) is a widely used technique to deal with conflicts. In the simplest case, a metapolicy can specify *precedence* between two or more potentially conflicting policies.

V. Proof of concept implementation

As a proof of concept that demonstrates how the PM operates, we have implemented the client's application, the service interface, the gateway and its integration to the PM as shown in Fig. 3. Notice that if we exclude the provider's LP from the policy manager, this implementation corresponds to the right side of Fig. 1–a).

Our implementation is RESTful based, thus communication between the components is realised as RESTful request and replies.

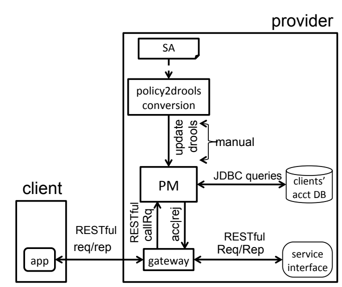

Figure 3. Proof of concept implementation.

The client's application is a HTML form that the client fills in with relevant personal information (name, account number, etc.) and submits to access the service. Client's request leaves the client as RESTful GET requests (*req*) to be intercepted by the gateway. The gateway extracts the client's personal information from the requests (precisely, from the query parameters), uses them to compose a GET request, sends it (*callRq*) to the PM for evaluation and waits for a reply. The PM extracts the personal information from the *callReq* and uses it for composing the event (event e_i in Fig. 2) that the drool engine needs to trigger the ECA rules that perform the evaluation of the *callReq*.

When the gateway receives a *rej* reply from the PM, it inserts a *rej* parameter in a RESTful reply and sends it (*rep*) to the client's *app*. More interestingly, when the gateway receives an *acc* from the PM, the gateway, composes a

RESTful request (*Req*) with the URL of the resource requested and forwards it to the *service interface*. Eventually, the *service interface* replies with a *Rep* to the gateway, who extracts the parameters of interest (for instance, the requested resource or a reference to it) to compose *rep* and sends it to the client's *app*.

We use the following SA in our experiment:

Client–Provider SA

p1 *The client can purchase call cards from the provider.*

p2 *A card entitles the client to place 10 calls (requests) against the provider, at any time.*

p3 *A card is considered consumed when its 10th call terminates.*

We admit that this SA is far from being complete; for instance, it does not specifies constraints on the length of the calls. Yet it is good enough to illustrate this discussion. Moreover, in this preliminary implementation, we converted the SA policies manually from English into standard drools instead of EROP rules.

We deployed two clients called (*Romeo* and *Juliette*), thus the PM is responsible for operating the gateway to permit or deny call requests related to Romeo–Provider and Juliette–Provider SAs. As shown in the figure, we use a conventional JDBC database to store the clients' accounts (*clients' acct*) which contain two parameters: client's name and its number of prepaid calls (*PrepaidCalls*).

p1 can be regarded as the signing of the agreements between the two parties. In a full implementation, the purchase of the call card would be taken by the provider as an indication to automatically update the PM (*update drools*) with policies *p2* and *p3* so that it can operate the gateway. In this experiment we update the PM manually, that is, we manually typed the following drools into a drools *drl* file. We inserted the numbers 01:, 02:, etc. in the code to help the discussion.

```
01: rule "Accept callRq"
02: when
03: $e: drools.Event(type=="callRq",
                     cli: originator)
04: eval(clientPC.getPrepaidCalls(cli)>0)
05: then
06: $e.setStatus("acc");
07: clientPC.updatePrepaidCalls(cli);
08: end
09: rule "Reject callRq"
10: when
11: $e: drools.Event(type=="callRq",
                     cli: originator)
12: eval(clientPC.getPrepaidCalls(cli)<=0)
13: then
14: $e.setStatus("rej");
15: end
```

The code contains two rules: *Accept callRq* and *Reject callRq*. *Accept callRq* triggers when a call request is to be accepted, whereas *Reject callRq* triggers when a call request is to be rejected. Both rules react to the event $drools.Event$ (lines 03 and 11, respectively). As defined in our Java classes, the event contains several fields; three of them are of interest here: Firstly, *type* identifies the type of event, for example, *callRq*; secondly, *originator* contains the name of the client that originated the event, for example, *Romeo*, *Juliette*; thirdly, *status* is used by the rules to store their decisions to accept or reject (*acc* or *rej*) the request expressed in the event under analysis. Upon receiving an event of type *callRq* (line 03) originated by a client (*cli*), rule *Accept callRq* evaluates its condition (a JDBC query in line 04) which verifies whether the client has prepaid calls in his or her account. If the condition is satisfied, the rule writes (line 06) *acc* in the status field of the event and access the JDBC database (line 07) to update (decrement by one) the number of calls consumed by the client.

Rule *Reject callRq* work similarly, except that it triggers when the client has no prepaid calls (line 12) in his account and writes (line 14) *rej* in the status field of the event.

The status of *drools.Event* is extracted by ancillary Java classes of the PM (a Servlet in current implementation) to compose the RESTful reply that the gateway is waiting for. The reply contains either *acc* or *rej* to instruct the gateway to accept or reject the client's request, respectively.

VI. Future Work

In Section I, we mentioned that the PM of two parties can interact with each other to negotiate the creation, updating and termination of SAs (Fig. 1–d)). We are currently in the process of integrating the monitoring facilities of the CCC with the negotiation facilities of Agility. Agility is a Cloud Management product developed by Arjuna Technologies Limited [4] to assist two or more independently administered parties in sharing their IT resources. It automates the creation, negotiation, updating and termination of SAs.

An abstract view of the negotiation components of agility's is shown in Fig. 4. The agility servers (AS_C and AS_P) are the core components of Agility and are responsible for establishing, storing, updating and terminating SAs negotiated through the *negotiation protocol*.

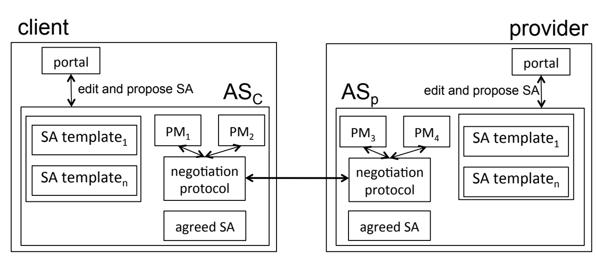

Figure 4. Agility's negotiation architecture.

To understand how the negotiation protocol works, imagine that the initiator and responder are, respectively, the client and provider. 1) The initiator wishing to create or update an SA uses its *portal* to edit an *SA template*. 2) The initiator consults its policy modules (PM_1 and PM_2) for approval of the SA proposal. 3) If the SA proposal is locally approved the initiator sends it to the responder). 4) The responder consults its local policies modules (PM_3, PM_4) for approval. 5) If the SA proposal is approved, the responder sends an acceptance message to the initiator; otherwise, it sends a rejection message.

Notice that each policy module contains one or more local policies and that decision to accept or reject a SA proposal is based on an implementation specific algorithm, expressed perhaps as a metapolicy on the policies in the policy modules.

Implementation issues aside, the challenge of this endeavour is the alignment of the LP of the parties and SA policies under negotiation. It seems that the techniques that we used in the analysis of the logical consistency of the SA policies can be applied in the analysis of LP. In the same way, these techniques can be used to reason about potential logical conflicts between the policies of the SA under negotiation (or already agreed upon) and the LP of the parties; namely conflicts between SA and client's LP and conflicts between SA and provider's LP.

We are aware that some conflicts are hard to detect offline. Yet we believe that some (or most) potential conflictive situations can be predicted by means of offline analysis. We speculate that Linear Temporal Logic (LTL) formulae can be used to reason about the temporal constraints— crucial information to predict run–time conflicts— on the clauses. For example, the requirement that *there should be no simultaneous permission and prohibition to submit a job for execution*, can be written in LTL as:

```
[](not( IS_permit(execute) && IS_prohibit(execute)))
```

This LTL reads that, *always it is not possible to be permitted and prohibited to execute*, and can be used for example to uncover conflicting situations where SA policies permit something that LP prohibit.

We feel that, this is a research direction worth exploring; good insights into this issue can be found in [16].

VII. Related Work

Research on contract regulated inter–enterprise interactions between parties subject to local and shared policies was pioneered by Minsky [17]. The notion of current right, obligations and prohibitions was introduced in [18]. A compact summary about the issues involved in contract management is provided in [19]. The author includes a list of 13 features that contract languages should provide. Within this context, we believe that our approach is particularly strong in capturing the dynamic of rights, obligations and prohibitions and contrary to duty obligations (contingency clauses) as our notation does this explicitly. Another salient feature is that it enables formal reasoning using existing general purpose tools like model–checkers both at design and implementation (testing) time. Intuitive mapping from notation used at verification time to actual implementation (not mentioned in [19]) is another salient feature of our approach. The need for automatic mechanisms for renegotiation (anticipated and exceptional updates) of legal agreements is recognised in [20], however, they focus on the protocol for the digital signatures and overlook the potential logical impact on the policies. Negotiation, deployment and monitoring of SAs is discussed in [21] but without accounting for logical conflicts. In [22] the authors discusses a policy management system called *MyPolMan* that can be used by administrators of Grid environments for editing (creating and updating) and disseminating policies (XML files). Though it is not explicitly discussed, the authors assume that the policy decision point is provided with a single XML document with logically consistent policies that satisfies both the policies of the Grid community and the administrator's local policies. They do not account for potential inconsistencies in the policies or clashes between Grid and local policies. A mechanism for granting access to Grid resources with the help of gateways controlled by policy decision points is discussed in [23]. Policy management here is centralised— in contrast, we deal with a multipolicy domain. A conceptual discussion on the use of metapolicies as a means of resolving policy conflicts that emerge in applications that involve several policy domains can be found in [24]. Though the focus is on security policies, her observations are applicable to other fields. This discussion is extended in [25] where it is suggested that metapolicies can be used to specify invariants, that is, to guard policies that cannot be overwritten by other policies, even in the event of conflicts. Policy conflicts are discussed in–depth in [26]. Special purpose tools for reasoning about policies (conflicts for instance) are suggested in [27], [11], [16], [28]. In contrast, in [9], [14] we suggest the use of existing model–checkers.

VIII. Conclusion

We have argued that service agreements (contracts) used in cloud computing are complex documents with a dynamic life–cycle that includes negotiation, conversion of policies (clauses) from English to executable code, deployment, enforcement, updates and normal or early termination. As a contribution, we discussed an architecture that includes automatic tools to help the designer at different stages. We raised the question about the desirable features that SA managing systems should provide. We focused our attention to the notation used to encode policies. We argued that it should be expressive enough to cover practical policies, clear, intuitive and implementable. We pointed out that the maintenance of the logical consistency of policies is not trivial and suggested that SA notations should be amenable to logical analysis with automatic tools to uncover potential problems. We discussed a proof of concept implementation based on RESTful and Jboss drools.

Acknowledgment

The first author was funded by EPSRC grant KTS-EP/H500332/1.

References

[1] C. Molina-Jimenez, S. Shrivastava, and M. Strano, "A model for checking contractual compliance of business interactions," *IEEE Trans. on Service Computing*, vol. PP, no. 99, 2011.

[2] M. Strano, C. Molina-Jimenez, and S. Shrivastava, "Implementing a rule–based contract compliance checker," in *Proc. 9th IFIP Conf. on e-Business, e-Services, and e-Society (I3E'2009)*. Nancy, France: Springer, 2009, pp. 96–111.

[3] A. Abdelsadiq, C. Molina-Jimenez, and S. Shrivastava, "A high–level model–checking tool for verifying service agreements," in *Proc. 6th IEEE Int'l Symposium on Service–Oriented System Engineering (SOSE'2011)*, 2011.

[4] Arjuna Technologies Limited, "Agility 1.2.0," 2011. [Online]. Available: www.arjuna.com/contact

[5] D. Harel and A. Pnueli, "On the development of reactive systems," *Logics and Models of Concurrent Systems*, vol. NATO ASI Series, F13, 1985.

[6] JBoss, "Drools," http://www.jboss.org/drools/.

[7] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor—a distributed job scheduler," 2001, chapter 15. [Online]. Available: www.cs.wisc.edu/condor/doc/beowulf-chapter-rev1.pdf

[8] N. H. Minsky and V. Ungureanu, "Scalable regulation of inter-enterprise electronic commerce," in *Proc. 2nd Int'l Workshop on Electronic Commerce*. Springer, 2001.

[9] C. Molina-Jimenez and S. Shrivastava, "Model checking correctness properties of a middleware service for contract compliance," in *Proc. 4th Int'l Workshop on Middleware for Service Oriented Computing (MW4SOC'09)*, Nov. 30, Urbana–Champaign, USA, 2009, pp. 13–18.

[10] Amazon, "Amazon cloud drive," 2011. [Online]. Available: www.amazon.com/clouddrive/learnmore

[11] D. Zhang and D. Nguyen, "Prepare: A tool for knowledge base verification," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 6, Dec. 1994.

[12] N. Dunlop, J. Indulska, and K. Raymond, "Dynamic conflict detection in policy-based management systems," in *Proc. Sixth Int'l Enterprise Distributed Object Computing Conf. (EDOC'02)*, 2002, pp. 15–26.

[13] G. J. Holzmann, *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[14] A. Abdelsadiq, C. Molina-Jimenez, and S. Shrivastava, "On model checker based testing of electronic contracting systems," in *12th IEEE Int'l Conf. on Commerce and Enterprise Computing(CEC'10)*, 2010, pp. 88–95.

[15] A. Paschke, "Verification, validation and integrity of distributed and interchanged rule based policies and contracts in the semantic web," in *2nd Int'l Semantic Web Policy Workshop (SWPW'06)*, 2006.

[16] N. Dunlop, J. Indulska, and K. Raymond, "Methods for conflict resolution in policy–based management systems," in *Proc. Seventh Int'l Enterprise Distributed Object Computing Conf. (EDOC'03)*, 2003, pp. 98–109.

[17] V. Ungureanu and N. H. Minsky, "Establishing business rules for inter–enterprise electronic commerce," in *Proc. 14th Int'l Symposium on Distributed Computing (DISC'00)*, 2000, pp. 179–193.

[18] H. Ludwig and M. Stolze, "Simple obligation and right model (SORM)-for the runtime management of electronic service contracts," in *Proc. 2nd Int'l Workshop on Web Services, e–Business, and the Semantic Web(WES'03), LNCS vol. 3095*, 2003, pp. 62–76.

[19] T. Hvitved, "A survey of formal languages for contracts," in *Fourth Workshop on Formal Languages and Analysis of Contract–Oriented Software (FLACOS'10)*, 2010.

[20] S. Angelov, S. Till, and P. Grefen, "Dynamic and secure B2B e-contract update management," in *Proc. 6th ACM Conf. on Electronic commerce(EC'05)*, 2005, pp. 19–28.

[21] C. Wang, G. Wang, H. Wang, A. Chen, and R. Santiago, "Quality of service (QoS) contract specification, establishment, and monitoring for service level management," in *Proc. 10th Int'l Enterprise Distributed Object Computing Conf. Workshops (EDOCW'06)*, 2006.

[22] J. Feng, L. Cui, G. Wasson, and M. Humphrey, "Policy-directed data movement in grids," in *Proc. 12th Intl Conf. on Parallel and Distributed Systems (ICPADS'06)*, 2006.

[23] G. Wasson and M. Humphrey, "Policy enforcement in virtual organizations," in *Proc. Fourth Int'l Workshop on Grid Computing (GRID'03)*, 2003, pp. 125–132.

[24] H. H. Hosmer, "Metapolicies I," *ACM SIGSAC Review*, vol. 10, no. 2–3, Special issue on Issues 91, pp. 18–43, Spring/Summer 1992.

[25] J. Schütte and T. Wahl, "Interdomain policy conflicts: Description logics-based handling," *IEEE Vehicular Technology Magazine*, vol. 5, no. 3, pp. 68–74, Sep. 2010.

[26] G. K. Giannikis and A. Daskalopulu, "Normative conflicts in electronic contracts," *Electronic Commerce Research and Application*, vol. 10, no. 2, pp. 247–267, Mar/Apr 2011.

[27] E. C. Lupu and M. Sloman, "Conflicts in policy–based distributed system management," *IEEE Transactions on Software Engineering*, vol. 25, no. 6, pp. 852–869, Nov/Dec 1999.

[28] S. Fenech, G. J. Pace, and G. Schneider, "CLAN: A tool for contract analysis and conflict discovery," in *Proc. 7th Int'l Symposium on Automated Technology for Verification and Analysis (ATVA'09), LNCS 5799*, 2009, pp. 90–96.