



A Systematic Framework Enabling Automatic Conflict Detection and Explanation in Cloud Service Selection for Enterprises

Chunqing Chen, Shixing Yan, Guopeng Zhao, Bu Sung Lee, Sharad Singhal

HP Laboratories

HPL-2012-104

Keyword(s):

Cloud Service Selection; Conflict Detection; Explanation; SMT; Constraint Programming;

Abstract:

The fast growth of cloud service offerings has attracted more enterprises to migrate their IT applications into cloud. Nonetheless, complex enterprise user requirements, especially interdependent relations across them, raise new challenges of cloud service selection. In addition, a major concern for these enterprises is ensuring compliance with their policies on the use of cloud services. In this paper, we present a systematic framework, based on formal verification and constraint solving techniques, to help enterprises tackle problems when adopting cloud computing. Our framework enables automatic detection of conflicts covering violation of enterprise policies and inconsistency of user requirements, and explanation generation which identifies problematic user requirements. The framework next select automatically cloud services which satisfy all enterprise policies and user requirements (with interdependent relations). We have prototyped and successfully applied our approach to projects which manage heterogeneous cloud infrastructure services for large enterprises.

External Posting Date: May 21, 2012 [Fulltext]

Approved for External Publication

Internal Posting Date: May 21, 2012 [Fulltext]

A Systematic Framework Enabling Automatic Conflict Detection and Explanation in Cloud Service Selection for Enterprises

Chunqing Chen*, Shixing Yan*, Guopeng Zhao*, Bu Sung Lee*[†]

* HP Labs, Singapore [†] SCE, NTU, Singapore

Email: {chunqing.chen, shixing.yan, guopeng.zhao, francis.lee}@hp.com

Sharad Singhal[‡]

[‡] HP Labs, Palo Alto, CA, USA

Email: sharad.singhal@hp.com

Abstract—The fast growth of cloud service offerings has attracted more enterprises to migrate their IT applications into cloud. Nonetheless, complex enterprise user requirements, especially interdependent relations across them, raise new challenges of cloud service selection. In addition, a major concern for these enterprises is ensuring compliance with their policies on the use of cloud services. In this paper, we present a systematic framework, based on formal verification and constraint solving techniques, to help enterprises tackle problems when adopting cloud computing. Our framework enables automatic detection of conflicts covering violation of enterprise policies and inconsistency of user requirements, and explanation generation which identifies problematic user requirements. The framework next select automatically cloud services which satisfy all enterprise policies and user requirements (with interdependent relations). We have prototyped and successfully applied our approach to projects which manage heterogeneous cloud infrastructure services for large enterprises.

Keywords—Cloud Service Selection; Conflict Detection; Explanation; SMT; Constraint Programming

I. INTRODUCTION

The cloud computing paradigm [1], [2] provides users a wide range of services, from infrastructure (IaaS), through computing platforms (PaaS), to software (SaaS). A cloud service can be characterized by multiple features, such as service availability, price, and the location of the physical machine hosting the service. Particular cloud services have their own metrics; for example, IaaS services include processing, memory, network, and storage. Recently, a number of approaches [3]–[5] have been proposed to manage heterogeneous cloud services from different providers.

The increasing scale and dynamics of cloud services have attracted more enterprises to migrate their IT applications into cloud. Nonetheless, the complexity of enterprise applications, especially with *interdependent relations*, leads to new challenges for selecting appropriate cloud services. For example, when deploying a three-tier application into cloud, a virtual machine (VM) at the application tier and a VM at the data tier need to locate at the *same* areas to minimize network latency, and on the other hand a VM as the backup database server must be distributed to *different* areas for the sake of fault-tolerance; those VM locations are *interdependent* on each other. Expressive interdependent

relations can increase the difficulties of finding appropriate services since one user requirement may influence or be influenced by others. Also, these relations can easily introduce subtle conflicts; for instance, users may specify inappropriate values for interdependent requirements which cause these requirements to be impossible to satisfy.

In addition, a major concern for enterprises wishing to adopt cloud computing is compliance with IT policies mandated by the enterprise or government [6], such as service availability and geographic constraints (many countries prohibit storing personal information about its citizens on any machine outside their border).

Thus, when choosing cloud services, a user from an enterprise needs to consider interdependencies that span across his/her requirements, and also the compliance of his/her requirements with relevant enterprise policies. Unfortunately, it is troublesome and impractical to manually detect various conflicts from a number of requirements, and even harder to identify problematic requirements that cause conflicts. It is hence important to provide enterprise users with comprehensive support which enables conflict detection and explanation for their cloud service selection.

Existing work on cloud service selection [7]–[12] has limited support for interdependent relations and enterprise policies, particularly the generation of diagnostic information when there is no solution satisfying user requirements.

In this paper, we present a generic and formal framework to overcome the limitations of existing solutions. The framework integrates *conflict analysis* and *service selection* in a systematic and automatic manner as shown in Figure 1.

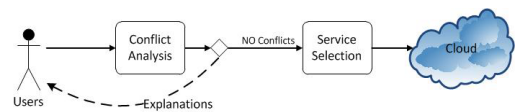


Figure 1. Framework Generic Workflow

Our framework *first* checks the existence of conflicts based on user requirements. We categorize conflicts into two types based on their causes, and prioritize them according to their impact scopes. When a conflict is detected, a form containing an explanation of the conflict is generated to assist users to quickly understand and resolve the conflict. The

process of analyzing conflicts is *interactive* and *iterative*. Moreover, to cope with interdependent relations, we adapt Satisfiability Modulo Theories (SMT) [13] techniques which can automatically determine the satisfiability of formulas expressed in first-order logic and have been widely applied to electronic design automation and software verification.

After conflict analysis, our framework can assist users to derive *all* solutions and *optimal* ones (under some criterion like minimum cost) over *finite* domains. A solution consists of a set of cloud services, where each service satisfies a user requirement and complies with enterprise policies, and these services also fulfill all interdependent relations. We model the service selection as a constraint satisfaction program (CSP) [14] and automate the solving procedure using constraint programming (CP) techniques [15]. Specifically, interdependent relations and dependent domain constraints are encoded into a mathematical matrix which represents a selection problem; the matrix is then processed by a solver developed upon Choco [16], a Java CP library.

The aforementioned execution order between the conflict analysis and the service selection can avoid unnecessary computation caused by conflicts and offer users diagnostic information when there is no solution. Our framework supports models expressed in *first-order logic* and *linear arithmetic*, which are capable to represent various enterprise policies, user requirements, and cloud services. To the best of our knowledge, our approach is the *first* attempt to provide enterprises with a wide range of support covering conflict detection and explanation generation for problematic user requirements, and flexible solution provisioning.

The rest of the paper is organized as follows. Section II gives a brief on cloud services, SMT and constraint programming. The architecture of the approach is elaborated in Section III. Section IV illustrates how we explore SMT techniques to detect and explain conflicts involving interdependent relations. Section V describes our modeling method based on the constraint programming paradigm. Section VI shows experimental study of our approach. Related work is discussed in Section VII. Section VIII concludes the paper.

II. BACKGROUND

A. Cloud Services

Cloud computing is a model for enabling ubiquitous and on-demand network access to configurable resources [1], which encompass hardware, programming environments, and applications. Cloud computing shifts the IT paradigm from owning and operating IT resources to purchasing *everything as a service* [2].

Cloud services are usually characterized by multiple features such as service availability, cost, and locations of physical machines hosting services. Domain-specific features are used as metrics to further distinguish services in a particular domain; for example, public IaaS providers like Amazon, GoGrid, and Rackspace can be distinct by

the VM capacity they offer (e.g., CPU type, RAM size, OS version, and storage access type). An emerging trend is to manage heterogeneous cloud services from multiple providers [3]–[5]; Lee et al. [5] defined common interfaces to broker cloud infrastructures from different commercial IaaS providers. This trend enables users to combine various services to accomplish complicated tasks in clouds.

Despite the interest in cloud computing, many concerns remain about security, privacy and compliance to enterprise/government IT policies [6]. Geography constraints about data location are common examples: for instance, data about European citizens cannot be stored outside EU borders. It is crucial for an enterprise to ensure that their user requirements of cloud services comply with such policies.

B. Satisfiability Modulo Theories (SMT)

SMT [13] has been used widely in many industrial applications such as electronic design automation and software verification. It is defined as the problem of determining the satisfiability of a *first-order logic* formula over some decidable theories which constrain the interpretation of special symbols; for example, the *arithmetic* theory restricts the meaning of symbols $+$, $\leq$, 0 .

An SMT problem is solved by finding a *valuation* for the variable, function, and predicate symbols that makes the formula true. A practical approach for solving SMT problems is to involve *lazy* techniques [17] to integrate SAT solvers [18] that can check the satisfiability of *propositional logic* formulas with dedicated theory interpretation. Specifically, each atom of an input formula is abstracted by a distinct propositional variable; next, a SAT solver is applied to finding a valid valuation of this propositional formula; lastly, the valuation is verified against corresponding theory interpretation. When a propositional formula is *unsatisfiable*, namely no valid valuation, several algorithms [19], [20] are available to extract an unsatisfiable core that is a (often small) subset of the input clauses that are not able to satisfy; such a core can be used to elaborate the cause of the unsatisfiability. These algorithms have been implemented in several SAT solvers, zChaff [21], for instance.

C. Constraint Programming

Constraint satisfaction has been shown to be generally applicable to a wide variety of problems such as planning and management [22]. A constraint satisfaction problem (CSP) consists of a set of variables and constraints where constraints restrict allowable value combination of variables in the form of logic and arithmetic expressions. A solution to the CSP is an assignment of all variables such that all constraints are satisfied. CSP on finite domains are typically solved using search techniques such as backtracking, constraint propagation, and local search [14].

Constraint programming (CP) [15] enables users to concentrate on modeling a problem without worrying about the

way the problem is solved. This is achieved by embedding constraints into a programming language which is called the *host language*. Common host languages include Prolog (in declarative style) and Java (imperative).

III. FRAMEWORK ARCHITECTURE

Our conceptual architecture is shown in Figure 2. We first illustrate the inputs of the framework, followed by two phases: conflict analysis and solution derivation.

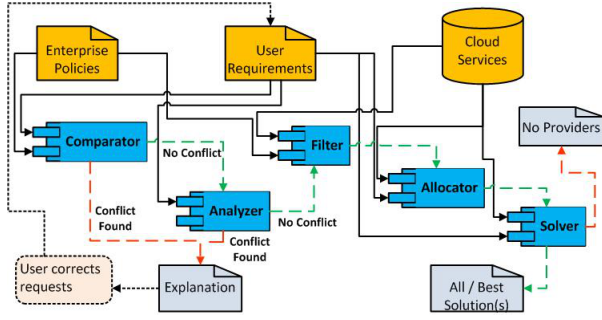


Figure 2. Conceptual Architecture and Workflow

A. Input Models

Although enterprise policies on the use of cloud services, user requirements, and cloud services can vary in representation form, we capture their intrinsic semantics in terms of *measurable* features such as service availability and locations of data centers hosting services. We construct their models using first-order logic and linear arithmetic.

A cloud service $CService_i$ consisting of multiple features is modeled by a set of attributes $\{A_{cs}^{i,1}, \dots, A_{cs}^{i,n}\}$. The range of an attribute $A_{cs}^{i,j}$, say $R(A_{cs}^{i,j})$ where R is a function returning values of the attribute, can be a *constant* of basic type like *integer* or *string*, a *set* of basic types, or a *function*. For example, the following set expression describes a *compute* service from *RackSpace* (*RS*) with relevant information: the type of supported OS, locations of the service, and the *dependency* between price and OS with the RAM size (e.g., the price is \$0.12/hr with 2G RAM using the Linux OS).

$$\{RS, Compute, \{USA, \dots\}, \{Linux, Windows\}, \\ \{(Linux, 2) \mapsto 0.12, (Windows, 1) \mapsto 0.08, \dots\}\}$$

An enterprise policy towards the use of cloud services can be modeled by a *Boolean-valued function* over the attributes that denote restricted service features. For instance, a policy $Policy_1$ requesting the availability of a compute service to be greater than 99% can be captured by a function $Policy_1(A_{pol}^{1,1}, A_{pol}^{1,2}) = A_{pol}^{1,1} == \text{"Compute"} \Rightarrow A_{pol}^{1,2} \geq 99$, where a logic implication ($\Rightarrow$) is used to capture the relation between the service type ($A_{pol}^{1,1}$) and availability ($A_{pol}^{1,2}$).

A user requirement Req_i usually comprises a set of sub-requirements $\{SubReq_i^1, \dots, SubReq_i^k\}$ where each sub-requirement constrains some service features, and *interdependent relationships* $\{IDR_i^1, \dots, IDR_i^k\}$ across other user

requirements. For example, a user may require a compute service to be hosted at USA or UK with the availability above 99.5%, and to *collocate* with an associated database service (also in cloud); namely, this requirement contains two sub-requirements and one interdependent relation. We represent a sub-requirement $SubReq_i^j$ by a *Boolean-valued function* over constrained attributes, $SubReq_i^j : (A_{req}^{i,1}, \dots, A_{req}^{i,l}) \rightarrow \mathbb{B}$ where $j \in 1 \dots l$ and $\mathbb{B} = \{\text{true}, \text{false}\}$. An interdependent relation (e.g., a collocation relation) is also modeled by a Boolean-valued function $IDR_i^j : (A_{req}^{i,1}, \dots, A_{req}^{i,m,1}) \rightarrow \mathbb{B}$, although the input attributes involved are from *different* user requirements.

Our input models provide an unambiguous interpretation expressed in mathematics and logics for a variety of enterprise policies, user requirements and cloud services. Our focus on measurable features allows us to conduct quantitative analysis over them.

B. Conflict Analysis

Conflict analysis is the first phase of our framework. This phase consists of the components *Comparator* and *Analyzer* (in Figure 2). It takes input models of enterprise policies and user requirements and outputs explanations when there are conflicts. Solid lines in Figure 2 depict the dataflow among components, while the control flow is indicated by colored dashed lines; for example, control shifts from the *Analyzer* to the *Filter* when there is no conflict, and otherwise the control moves to the process of explanation generation.

We categorize conflicts into two types based on the levels of the parties involved: a *vertical* type of conflict is caused by parties at *different* levels, and a *horizontal* type of conflict is caused by parties at the *same* level. For example, compliance between user requirements and enterprise policies is a vertical type of conflicts, and inconsistencies among user requirements are a horizontal type of conflicts. Moreover, we prioritize conflicts according to the *highest* level of the parties involved, specifically, based on the impact scope of the conflict; conflicts are analyzed following their priorities.

The *Comparator* checks the compliance between user requirements and enterprise policies with a *higher* priority. Formally, a sub-requirement $SubReq_i^j$ of requirement Req_i is said to comply with an enterprise policy $Policy_k$ if *every* value accepted by $SubReq_i^j$ is also accepted by $Policy_k$. Note that both $SubReq_i^j$ and $Policy_k$ are modeled as Boolean-valued functions. This compliance relation can be specified by the following function *Com* which returns value *true* if the application of $SubReq_i^j$ and the application of $Policy_k$ to their *semantically equivalent* attributes (indicated by $\simeq$) fulfill the implication relation¹. Two attributes are *semantically equivalent* if they are from different domains while representing the *same* service feature. For example, if both attribute $A_{req}^{2,3}$ from user requirement Req_2 and attribute $A_{pol}^{1,3}$

¹The $\bullet$ operator can be interpreted as *such that*, i.e., “s.t.”.

from policy $Policy_1$ denote the service availability feature, we have $A_{req}^{2,3} \simeq A_{pol}^{1,3}$.

$$\begin{aligned} Com(SubReq_i^j(A_{req}^{i,1}, \dots, A_{req}^{i,l}), Policy_k(A_{pol}^{k,1}, \dots, A_{pol}^{k,l})) = \\ \forall v_1 : R(A_{req}^{i,1}); \dots; v_l : R(A_{req}^{i,l}) \bullet \\ SubReq_i^j(v_1, \dots, v_l) \Rightarrow Policy_k(v_1, \dots, v_l) \\ \text{where } A_{req}^{i,1} \simeq A_{pol}^{k,1} \wedge \dots \wedge A_{req}^{i,l} \simeq A_{pol}^{k,l} \end{aligned}$$

Thus, the compliance of Req_i with $Policy_k$ is valid if *every* sub-requirement of Req_i complies with $Policy_k$, namely, $\bigwedge_{j=1}^l Com_i(SubReq_i^j, Policy_k)$. If Req_i fails to comply with $Policy_k$, the *Comparator* returns automatically *all* problematic sub-requirements which fail to satisfy the compliance relation, namely, the set $\{SubReq_i^j : Req_i \mid \neg Com(SubReq_i^j, Policy_k)\}$ as explanations.

On the other hand, the *Analyzer* checks inconsistencies among user requirements associated with interdependent relations. The *Analyzer* discovers and explains such a conflict one at a time, as this type of conflict may affect one another: resolving one conflict may also correct other conflicts. We will detail the way of analyzing these conflicts in Section IV.

C. Solution Derivation

After the execution of the conflict analysis phase, the Solution Derivation phase derives appropriate cloud services by components *Filter*, *Allocator*, and *Solver* (in Figure 2).

First, the *Filter* identifies *potentially valid* cloud services by removing services which violate an enterprise policy. A cloud service $CService_i$ is potentially valid for an enterprise policy $Policy_j$ provided every attribute constrained by $Policy_j$ can be assigned a valid value from its semantically equivalent attribute from $CService_i$. Namely,

$$\begin{aligned} \exists v_1 : R(A_{cs}^{i,1}); \dots; v_l : R(A_{cs}^{i,l}) \bullet Policy_j(v_1, \dots, v_l) \\ \text{where } A_{cs}^{i,1} \simeq A_{pol}^{j,1} \wedge \dots \wedge A_{cs}^{i,l} \simeq A_{pol}^{j,l} \end{aligned}$$

The reason of imposing these *loose* constraints (using existential quantifier $\exists$) rather than requesting *all* values from cloud service attributes to be valid for a policy is because some attributes, such as service locations and prices, can have multiple values and some of them may be *invalid* against the policy; the *Filter* further deletes these invalid values from potentially valid cloud services.

Next, the *Allocator* deduces a set of potentially valid cloud services for *every* user requirement based on the result from the *Filter*. Specifically, a cloud service is potentially valid against a requirement provided that there is an assignment of the service attributes such that all sub-requirements are valid with respect to that assignment.

Last, the *Solver* derives appropriate solutions by considering interdependent relations and dependent domain constraints. A solution is a set of cloud service, where each service serves one user requirement and complies with all enterprise policies, and all interdependent relations are valid

for the solution. We will extend the discussion of applying constraint programming for the *Solver* in Section V.

In this section, we have demonstrated the conceptual architecture and workflow of our approach. The advantages of the execution order and the way of handling various conflicts were also discussed. We remark that the execution order between two phases *avoids* unnecessary computation caused by conflicts and also provides *more* diagnostic information to users when there is no solution.

IV. APPLYING SMT FOR INTERDEPENDENT RELATIONS

Interdependent relations that span across user requirements are important when combining cloud services. Location distribution and software/platform compatibility are two common types of interdependent relations in the deployment of multi-tier applications into cloud; the former can be further divided into *collocation* and *anti-collocation* relations. With the growing user requirements, it becomes easy for users to introduce conflicts by inappropriate specification, although it is hard to manually detect and identify the causes of conflicts, especially when multiple types of interdependent relations are involved. Therefore, it is desired to develop automatic conflict detection and explanation dedicated to interdependent relations.

We first define the *valuation of user requirements* as an assignment of *all* requirement attributes with values from corresponding user-specified ranges, namely, $Val_{req} : A_{req}^{i,j} \rightarrow R(A_{req}^{i,j})$. Next, an interdependent relation IDR_i is *valid against user requirements* if there *exists* a valuation of *involved* user requirement attributes such that the relation is *true* under the valuation that can be modeled as follows²,

$$Val_{IDR}(As : \mathbb{P}(\mathbb{U}_{reqAs})) = \{v_{i,j} \mid v_{i,j} = Val_{req}(A_{req}^{i,j})\}$$

where $A_{req}^{i,j} \in As$ and As is the set of involved attributes, and $\mathbb{U}_{reqAs} = \{A_{req}^{i,j} \mid i \in 1 \dots n \wedge j \in 1 \dots m\}$ is a set containing *all* user requirement attributes. Thus, the validity of an interdependent relation IDR_k is:

$$Valid(IDR_k) = \exists Val_{IDR}(As) \bullet IDR_k(Val_{IDR}(As))$$

A set of interdependent relations $\{IDR_1, \dots, IDR_k\}$ is *conflict-free* if there *exists* a valuation of all involved requirement attributes such that the valuation fulfills *all* requirements. In the expression below which depicts the conflict-free property, the *generalized union* operator ($\bigcup$) merges sets of input attributes (As_i) from different interdependent relations into one set, and thus an attribute shared by multi-interdependent relations is assigned the *same* value.

$$\exists Val_{IDR}(\bigcup_{i=1}^{i=k} As_i) \bullet \forall j : (1 \dots k) \bullet Valid(IDR_j)$$

When there is a conflict caused by interdependent relations, *no* valid valuation exists. That is to say, there exists an

²The *powerset* operator $\mathbb{P}$ generates the set of all subsets of the input set, e.g., $\mathbb{P}(\{1, 2\}) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$.

input attribute $A_{req}^{i,j}$ constrained by an interdependent relation IDR_k such that $A_{req}^{i,j}$ has no valid value to satisfy IDR_k . An *explanation* of such a conflict (involving interdependent relations) is a *set* of interdependent relations comprised of: (1) IDR_k , (2) *directly* relevant relations which share some semantically equivalent attribute(s) with IDR_k , (3) *indirectly* relevant relations which share some semantically equivalent attribute(s) with either directly relevant relation(s) or other indirectly relevant relation(s), and (4) *remotely* relevant relations which contain some attribute(s) that can affect a relation from the aforementioned three types, where the influence relation is captured by some sub-requirement(s).

For example, assume five interdependent relations with their constrained user requirement attributes, $IDR_1(A_{req}^{1,2}, A_{req}^{2,2})$, $IDR_2(A_{req}^{3,3}, A_{req}^{2,2})$, $IDR_3(A_{req}^{4,3}, A_{req}^{3,3})$, $IDR_4(A_{req}^{4,2}, A_{req}^{1,2})$, $IDR_5(A_{req}^{5,1}, A_{req}^{2,1})$, and a sub-requirement $SubReq_2^1$ that captures an effect between $A_{req}^{2,1}$ and $A_{req}^{2,2}$: when $A_{req}^{3,3}$ in IDR_2 has no valid value, the explanation is $\{IDR_1, IDR_2, IDR_3, IDR_4, IDR_5\}$, because IDR_1 and IDR_3 are directly relevant relations (sharing $A_{req}^{2,2}$ and $A_{req}^{3,3}$ with IDR_2 , respectively), IDR_4 is an indirectly relevant relation (sharing $A_{req}^{1,2}$ with IDR_1), and IDR_5 is a remotely relevant relation due to $SubReq_2^1$.

We can obtain these explanations by adopting the SMT [13] solving techniques. Specifically, we construct a formal model using first-order logic and linear arithmetic to represent interdependent relations and relevant sub-requirements which share attribute(s) with some interdependent relations. For example, a *collocation* relation can be represented by equality over locations (which can be represented by a set of strings) and an *anti-collocation* relation by inequality over locations. Thus, we convert the checking of the conflict-free property into a SMT problem: we create a CNF (conjunctive normal form) formula by replacing every interdependent relation and relevant sub-requirement with a Boolean-valued variable/clause; this CNF formula is then analyzed using SAT methods [19], [20] which can generate either a satisfying assignment or an unsatisfiable core that is a subset of clauses of the CNF formula causing the unsatisfiability. In the former case, the assignment of each clause is examined based on the underlying theories of that clause, the linear arithmetic theories in our case; if the assignment is invalid, its inverse is inserted back to the CNF formula and we apply the SAT methods again. In the latter case, we can derive an explanation from the unsatisfiable core by converting those Boolean-valued variables appeared in the core back to their corresponding interdependent relations.

V. CONSTRAINT PROGRAMMING FOR SELECTION

Our approach provides users with the flexibility to find appropriate cloud services on finite domains: one feasible solution, all solutions, or optimal solutions. We classify our solution selection problem as a constraint satisfaction

problem (CSP), and illustrate below how to build the *Solver* (in Figure 2) based on constraint programming (CP). We remark that the SMT techniques used in Section IV have strong strengths of detecting conflicts, although they have *inadequate* support for finding *all* or *optimal* solutions.

The input of the *Solver* consists of two groups: interdependent relations and *potentially valid cloud services*. Every user requirement, say req_i , is assigned a set of potentially valid cloud services, $\{CService^{i,1}, \dots, CService^{i,j}\}$, obtained from the *Filter* and *Allocator* (in Section III-C); each service may have multiple attributes. We denote a service attribute by $A_{cs}^{i,j,k}$, meaning “a service attribute indexed k in a cloud service indexed j for a user requirement indexed i ”.

When there is more than one potential cloud service for a requirement, *dependent domain constraints* on service attributes exist. For example, the service cost for a user requirement depends on the particular service provider chosen for that requirement. We describe below an effective method to model our selection problem.

First, we construct a two-dimensional mathematical matrix with a size of $n \times m$ for n user requirements and m attributes of each potentially valid cloud service. Each row indicates a valid cloud service for one user requirement; the value of the i th row is a set of potentially valid cloud services for the i th requirement. Each column denotes an attribute shared by all cloud services. For example, the first column may represent the name of service provider, and thus cell $C_{(1,1)}$ stores provider name for the *first* user requirement. Note that we here analyze the case where cloud services share the same number of attributes; nonetheless, the analysis can be easily extended to handle services having various numbers of attributes.

Next, a dependent domain constraint over cloud service attributes for a user requirement is modeled by *enumerating* the dependency over relevant cells that denote service attributes involved in the constraint, along a row that indicates the requirement. Such a constraint is represented by a set of conjunctive implications where each implication constrains values of involved attributes.

For example, let $CService^{1,1}$ be the potential cloud service for req_1 and $CService^{2,1}, CService^{2,2}$ for req_2 ; each service contains two attributes. Specifically, $A_{cs}^{1,1,1}, A_{cs}^{2,1,1}, A_{cs}^{2,2,1}$ denote provider names, and $A_{cs}^{1,1,2}, A_{cs}^{2,1,2}, A_{cs}^{2,2,2}$ service costs, we can model a dependent domain constraint between providers and costs in req_2 by two implications below.

$$C_{(2,1)} = R(A_{cs}^{2,1,1}) \Rightarrow C_{(2,2)} = R(A_{cs}^{2,1,2}) \\ \wedge C_{(2,1)} = R(A_{cs}^{2,2,1}) \Rightarrow C_{(2,2)} = R(A_{cs}^{2,2,2})$$

Last, an interdependent relation can be captured by constraining involved cells. Reusing the previous example, an interdependent relation that restricts the total service cost for req_1 and req_2 to be less than a constant T is modeled as: $C_{(1,2)} + C_{(2,2)} < T$ where $C_{(1,2)} = R(A_{cs}^{1,1,2})$ and $C_{(2,2)} = R(A_{cs}^{2,1,2}) \vee C_{(2,2)} = R(A_{cs}^{2,2,2})$.

We apply classic constraint solving techniques, in particular the depth-first-search with backtracking approach [14], to analyzing our matrix representation. As CSP search algorithms usually consider *possible assignments* for *only one* variable at *each* node in their search trees, we developed the following two heuristics dedicated to our models to improve the solving efficiency.

One heuristic is regarding the order of exploring variables. We use *dependent domain constraints* to determine the order. Attributes whose domains depend on other attributes have lower priority than others. In addition, attributes which are involved in more dependent domain constraints to decide other attributes' domains have higher priority. When attributes have the same priority, they are selected randomly.

The other heuristic focuses on the value selection of a variable which may influence other unassigned variables. This is important for *interdependent relations* where attributes may affect each other. We adopt the *least-constraining-value* heuristic to begin an assignment with the value that leaves *maximum* flexibility for subsequent variable assignments; for example, the assignment of an attribute denoting a service cost starts with the smallest value from its range when looking for minimum total cost.

VI. EXPERIMENTAL STUDY

We have prototyped our framework in Java. Specifically, the *Analyzer* (in Section IV) is implemented based on Cauldron [23], an SMT tool developed by HP labs using the zChaff solver [21]. The execution of our *Analyzer* consists of three steps: 1) automatically transform all interdependent relations and sub-requirements that contain attributes which are involved by those relations into Cauldron models, 2) execute Cauldron to process the transformed models, 3) automatically identify and classify relevant interdependent relations from the output obtained at Step 2. In addition, we have implemented our modeling methods and solving heuristics (in Section V) based on Choco [16], a Java CP library. Our solver automatically constructs mathematical matrixes where interdependent relations and dependent domain constraints are encoded, with the support from Choco on variables and constraints of arithmetic and logics.

We have successfully applied our approach to *Monsoon* system [24], an enterprise solution for managing hybrid clouds. Monsoon provides a common interface to manage cloud infrastructure services from multiple public cloud providers like Amazon, Rackspace, and GoGrid, and private cloud computing systems such as HP CloudSystem and others developed using OpenStack, Eucalyptus, or OpenNebula.

Common concerns from Monsoon's customers are usually from two groups. IT administrators/governors are concerned for 1) the compliance of the cloud services required by IT developers, the other group, with IT policies, and 2) the ability to predict the influence on existing user requirements and applications in clouds when changing a policy. On

the other hand, IT developers/architects would like to 1) ensure that their requirements satisfy enterprise policies; and otherwise, diagnostic information of the violation is desired, 2) find a valid solution, or the optimal solution under specific criterion like minimum cost if there is more than one, and 3) identify possible causes of a conflict which leads to an infeasible solution, especially when the conflict involves various interdependent relations such as software/platform compatibility and service geography distribution.

We embed our framework into Monsoon to address those customer concerns with a high degree of automation. Users specify their enterprise policies and requirements by filling up pre-defined forms which cover 15 common cloud IaaS service attributes; new service attributes can be easily added, and complex policies or requirements can be formed by combining those service attributes using conventional logic and mathematical operators such as implication and disjunction. Our framework then conducts conflict analysis *before* derives appropriate solutions from the cloud services managed by Monsoon. When a conflict is detected, a user-friendly explanation is generated for conflict resolution.

As an illustrative case, a multinational corporation as our customer aims to develop and deploy web applications with three-tier architecture into clouds. Its IT administrators define general policies for the applications including minimum availability and security for back-end services like database servers, geography constraints (at the country level), and preferred cloud providers for specific type of services (e.g., compute or storage). Its IT developers specify technical requirements for particular applications, such as service type, compute capability (e.g., RAM size), storage capability (e.g., the size and access type), OS with versions (e.g., Windows Server 2008R2, Enterprise version), software with versions and *compatible* platform (e.g., Microsoft IIS Server 7.0 is only compatible with windows OS, while Oracle iPlanet Web Server 7 compatible with all OS), availability, security, load balancing, location, and distribution type (i.e., collocation or anti-collocation). In practice, an enterprise application requires tens of VMs or less.

We below present six simplified user requirements (in Table I) for a web application and compute services from two providers (in Table II³). Here each requirement, Req_i where i is its ID value, shows user-specified RAM size, areas, and optional collocation ID ($CoID$); e.g., service for Req_5 needs to collocate with service for Req_1 . A service price depends on provider name, area, and RAM size.

Table I
USER REQUIREMENTS OF RAM SIZE, AREAS AND COLLOCATION

ID	Size	Areas	ID	Size	Areas	CoID
1	(0.5..2)	SG, USA	4	(2..8)	JP, USA	2
2	(0.5..3)	USA, UK	5	(1..2)	UK, SG	1
3	(0.5..2)	JP, UK	6	(2..8)	JP, UK	2

³Prices are for the Linux OS and obtained from provider websites; the normalized prices of AWS over different USA sites are used in this instance.

Table II
AMAZON (AWS) AND RACKSPACE (RS) CLOUD SERVICES AND PRICES

	Name	Areas	Price (per hour)		
	AWS		1.7G	7.5G	
		USA	\$0.08	\$0.34	
		SG, IE	\$0.09	\$0.38	
		JP	\$0.1	\$0.4	
Name	Areas	Price (per hour)			
RS	USA, UK, HK	1G	2G	4G	8G
		\$0.06	\$0.12	\$0.24	\$0.48

Our framework first detects a conflict and produces an explanation “*Check relations across requirements 2, 4, and 6*” to identify possible problematic user requirements involved in collocation relations: Req_2 and Req_4 have USA in common, and Req_2 and Req_6 share UK, although these requirements cannot have a common area at the same time.

Assuming users add UK to Req_4 and USA to Req_6 to resolve the conflict, our framework next selects appropriate cloud services. Table III shows two *optimal* solutions: one for the minimum cost of 100 hours’ usage, and the other for the maximum RAM usage. Each row returns specific values of cloud service attribute for a requirement. Note that a requirement can be assigned different services under different criterion: e.g., Req_2 is assigned a 1G RAM at UK in one solution, but a 2G RAM at USA in the other. In addition, all collocation relations are fulfilled in both solutions: Req_2 , Req_4 , and Req_6 are always assigned the same area.

Table III
SOLUTIONS W.R.T. MINIMUM COST & MAXIMUM RAM USAGE

ID	P	S	Area	\$	ID	P	S	Area	\$
1	AWS	1.7	SG	9	1	AWS	1.7	SG	9
2	RS	1	UK	6	2	RS	2	USA	12
3	RS	1	UK	6	3	AWS	1.7	JP	10
4	RS	4	UK	24	4	AWS	7.5	USA	34
5	AWS	1.7	SG	9	5	AWS	1.7	SG	9
6	RS	4	UK	24	6	AWS	7.5	USA	34

When the number of user requirements and the type of interdependent relations increase, manual conflict detection and explanation become more difficult. In a larger case where 80 user requirements are specified and among them there are 31 collocation relations and 56 anti-collocation relations (10 requirements are involved in both types of relations), our framework can detect subtle conflicts and generate corresponding explanations. For example, one detected conflict has an explanation “*Check relations across requirements 15, 21, 22, 24, 38, and 61. Check relations across requirements 19, 21, and 22*”. Table IV displays relevant user-specified attributes: requirement ID, areas, collocation ID (CoID) and anti-collocation ID (ACoID) where value 0 means no constraint on distribution. With the explanation, we can deduce that Req_{15} , Req_{22} , and Req_{24} share SG, and Req_{21} , Req_{38} , and Req_{61} also share SG, by their respective collocation relations, although Req_{21} and Req_{22} should reside separately by their anti-collocation relation.

Remark: Our explanations for policy violation can demonstrate ill-defined values of relevant requirement attributes,

Table IV
RELEVANT USER REQUIREMENTS IN A CONFLICT

ID	Areas	CoID	ACoID
21	JP, SG, USA	0	19
22	SG, UK	15	21
24	HK, SG, USA	15	0
38	HK, JP, SG	21	0
61	SG, UK, USA	21	0

and explanations for conflicts involving interdependent relations can identify problematic requirement IDs. In the latter case, problematic requirements are further grouped based on the types of interdependent relations, as shown in our previous case of 80 user requirements. We are currently developing advanced techniques to provide more in-depth explanation by analyzing the values of requirement attributes constrained by different types of interdependent relations. Separately, our solver performs acceptably for the scale of use cases we have tested. For larger application development (e.g., more than 100 VMs), we can exploit a hierarchical approach to the selection optimization.

VII. RELATED WORK

With the booming growth of the cloud services, there are several approaches [7]–[12] focusing on automatic cloud service selection. Wright et al. [7] developed a multi-cloud marketplace to manage services which comprise user-required applications, and services are selected based on locality and cost constraints. Similarly, in the RESERVOIR model presented by Rochwerger et al. [11], the deployment of virtual execution environment instances considers various constraints including resource consumption cost and placement restrictions like platform compatibility. Both approaches allow users to specify the collocation constraints, one type of interdependent relations (captured in the infrastructure metadata [7] and the *affinity* constraints [11]), although conflict detection and explanation are not covered.

In addition, existing work [8]–[10], [12] lacks the support of interdependent relations. Wu et al. [9] presented resource allocation algorithms to minimize infrastructure cost for enterprise users by considering various quality of service (QoS) parameters. Zeng et al. [12] defined a hierarchical architecture of cloud services based on service types and developed a two-step selection algorithm. Rehman et al. [10] discussed the unification issue of a variety of cloud services and proposed a multi-criteria service selection methodology. Garg et al. [8] also treated the cloud service selection as a multi-criteria decision-making problem, and extended the cloud service measurement index consortium to measure the quality and to prioritize cloud services. Separately, Person and Sander [25] tackled the selection problem from the privacy and security aspects; they encoded user’s policies and regulatory into a rule-based system and automatically created questionnaires for assessing providers’ properties.

Several approaches [26]–[28] emphasized the evaluation of adopting cloud computing. Khajeh-Hosseini et al. [26]

implemented a modeling tool that can estimate cost of migrating IT systems into cloud and users need to determine preferred cloud providers by themselves. Gmach et al. [27] introduced and compared three cost models for apportioning costs in shared resource environment. Zardari and Bahsoon [28] proposed a framework based on a goal-oriented approach to help users refine requirements in adopting a cloud; some general guidelines were presented to evaluate cloud service providers and analyze the matches/mismatches of user requirements against cloud provision.

There has been considerable effort on the conflict analysis of network system management policies. Charalambides et al. [29] expressed QoS policies using Event Calculus for managing DiffServ networks, and their conflict analysis is conducted in a pairwise comparison fashion. Samak et al. [30] also worked on identifying conflicts among policies managing QoS in DiffServ networks, but emphasized the resource management at the router level. Dunlop et al. [31] proposed to specify policies of permission, prohibition and obligation in a *temporal* logic language that can reason about the sequences of events. Our approach targets different type of policies from the above work: enterprise policies are modeled as constraints without involving events or actions.

VIII. CONCLUSION

We have prototyped and applied a systematic framework in a practical cloud managing system (i.e., the *Monsoon*) for enterprises to tackle the emerging difficulties of cloud service selection, particularly the involvement of enterprise policies and interdependent relations that span across user requirements. Our framework automatically checks various conflicts covering the violation against enterprise policies and inconsistency within user requirements, *followed* by selecting appropriate cloud services that satisfy user requirements and comply with enterprise policies; explanations of detected conflicts are generated to identify problematic user requirements. Our experiments of real-life cases have indicated the effectiveness of our approach to satisfy dynamic and complicated requirements from enterprise customers.

Currently, we are enhancing our input model of cloud services by encoding 85 common services from the HP cloud service catalog⁴. Handling selection criteria associated with priority/preference and conducting more in-depth investigation of the complexity of the framework (w.r.t. requirements and policies) are some of our future work.

REFERENCES

- [1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.
- [2] P. Banerjee, R. Friedrich, C. Bash, P. Goldsack, B. A. Huberman, J. Manley, C. D. Patel, P. Ranganathan, and A. Veitch, "Everything as a service: Powering the new information economy," *IEEE Computer*, vol. 44, no. 3, pp. 36–43, 2011.
- [3] B. Rochwerger, D. Breitgand, A. Epstein, D. Hadas, I. Loy, K. Nagin, J. Tordsson, C. Ragusa, M. Villari, S. Clayman, E. Levy, A. Maraschini, P. Massonet, H. Muñoz, and G. Tofetti, "Reservoir - when one cloud is not enough," *IEEE Computer*, vol. 44, no. 3, pp. 44–51, 2011.
- [4] D. S. Milojicic, I. M. Llorente, and R. S. Montero, "OpenNebula: A cloud management tool," *IEEE Internet Computing*, vol. 15, no. 2, pp. 11–14, 2011.
- [5] B. S. Lee, S. Yan, D. Ma, and G. Zhao, "Aggregating IaaS service," HP Laboratories, Tech. Rep. HPL-2011-22, February 2011.
- [6] D. Catteddu, "Security and resilience in governmental clouds," European Network and Information Security Agency, Tech. Rep., 2011.
- [7] P. Wright, T. J. Harmer, J. Hawkins, and Y. L. Sun, "A commodity-focused multi-cloud marketplace exemplar application," in *CLOUD*, 2011, pp. 590–597.
- [8] S. K. Garg, S. Versteeg, and R. Buyya, "Smicloud: A framework for comparing and ranking cloud services," in *UCC*, 2011, pp. 210–218.
- [9] L. Wu, S. K. Garg, and R. Buyya, "SLA-based resource allocation for software as a service provider (SaaS) in cloud computing environments," in *CCGRID*, 2011, pp. 195–204.
- [10] Z. ur Rehman, F. K. Hussain, and O. K. Hussain, "Towards multi-criteria cloud service selection," in *IMIS*, 2011, pp. 44–48.
- [11] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. S. Montero, Y. Wolfsthal, E. Elmroth, J. A. Cáceres, M. Ben-Yehuda, W. Emmerich, and F. Galán, "The reservoir model and architecture for open federated cloud computing," *IBM Journal of Research and Development*, vol. 53, no. 4, p. 4, 2009.
- [12] W. Zeng, Y. Zhao, and J. Zeng, "Cloud service and service selection algorithm research," in *GEC*, 2009, pp. 1045–1048.
- [13] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, 2009, ch. 26.
- [14] S. Russell and P. Norvig, "Constraint satisfaction problems," in *Artificial Intelligence: A Modern Approach*, 2002, ch. 5.
- [15] K. Apt, *Principles of Constraint Programming*, 2003.
- [16] C. Team, "Choco: an open source java constraint programming library," Ecole des Mines de Nantes, Tech. Rep. 10-02-INFO, 2010.
- [17] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe, "Theorem proving using lazy proof explication," in *CAV*, 2003, pp. 355–367.
- [18] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract davis-putnam-logemann-loveland procedure to DPLL(T)," *Journal of the ACM*, vol. 53, pp. 937–977, November 2006.
- [19] L. Zhang and S. Malik, "Extracting small unsatisfiable cores from unsatisfiable boolean formulas," in *SAT*, 2003.
- [20] R. Gershman, M. Koifman, and O. Strichman, "An approach for extracting a small unsatisfiable core," *Formal Methods in System Design*, vol. 33, no. 1-3, pp. 1–27, 2008.
- [21] Y. S. Mahajan, Z. Fu, and S. Malik, "Zchaff2004: An efficient SAT solver," in *SAT*, 2004, pp. 360–375.
- [22] S. Asaf, H. Eran, Y. Richter, D. P. Connors, D. L. Gresh, J. Ortega, and M. J. Mcinnis, "Applying constraint programming to identification and assignment of service professionals," in *CP*, 2010, pp. 24–37.
- [23] L. Ramshaw, A. Sahai, J. Saxe, and S. Singhal, "Cauldron: A policy-based design tool," in *POLICY*, 2006, pp. 113–122.
- [24] S. Yan, B. S. Lee, G. Zhao, D. Ma, and P. Mohamed, "Infrastructure management of hybrid cloud for enterprise users," in *SVM*, 2011.
- [25] S. Pearson and T. Sander, "A mechanism for policy-driven selection of service providers in SOA and cloud environments," in *NOTERE*, 2010, pp. 333–338.
- [26] A. Khajeh-Hosseini, I. Sommerville, J. Bogaerts, and P. B. Teregowda, "Decision support tools for cloud migration in the enterprise," in *CLOUD*, 2011, pp. 541–548.
- [27] D. Gmach, J. Rolia, and L. Cherkasova, "Resource and virtualization costs up in the cloud: Models and design choices," in *DSN*, 2011, pp. 395–402.
- [28] S. Zardari and R. Bahsoon, "Cloud adoption: a goal-oriented requirements engineering approach," in *SECLoud*, 2011, pp. 29–35.
- [29] M. Charalambides, P. Flegkas, G. Pavlou, J. Rubio-Loyola, A. K. Bandara, E. C. Lupu, A. Russo, N. Dulay, and M. Sloman, "Policy conflict analysis for diffserv quality of service management," *IEEE Transactions on Network and Service Management*, vol. 6, no. 1, pp. 15–30, 2009.
- [30] T. Samak, E. Al-Shaer, and H. Li, "QoS policy modeling and conflict analysis," in *POLICY*, 2008, pp. 19–26.
- [31] N. Dunlop, J. Indulska, and K. Raymond, "Dynamic conflict detection in policy-based management systems," in *EDOC*, 2002, pp. 15–26.

⁴<https://portal.saas.hp.com/site/html/coral.mss>