Automated Policy-Based Resource Construction in Utility Computing Environments

A. Sahai, S. Singhal, V. Machiraju
HP Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
{akhil.sahai, sharad.singhal, vijay.machiraju} @hp.com

R. Joshi[1]
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109
rajiv.joshi@jpl.nasa.gov

Abstract

A utility environment is dynamic in nature. It has to deal with a large number of resources of varied types, as well as multiple combinations of those resources. By embedding operator and user level policies in resource models, specifications of composite resources may be automatically generated to meet these multiple and varied requirements. This paper describes a model for automated policy-based construction of complex environments. We pose the policy problem as a goal satisfaction problem that can be addressed using a constraint satisfaction formulation. We show how a variety of construction policies can be accommodated by the resource models during resource composition. We are implementing this model in a prototype that uses CIM as the underlying resource model and exploring issues that arise as a result of that implementation.

Keywords

Policy, resource utility, utility computing, configuration, management

1 Introduction

Currently, there is a trend towards managing data center infrastructure and services within a utility model that provides resources on demand. HP's Utility Data Center product [1], IBM's "on-demand" computing initiative [2], Sun's N1 vision [3], Microsoft's DSI initiative [4], and work at Global Grid Forum [5] are examples of this trend. The intent in these initiatives is to create systems that provide automated provisioning, configuration, and lifecycle management of a wide variety of infrastructure resources. However much remains to be done in order to achieve the desired level of automation.

A utility infrastructure is dynamic in nature. The utility maintains an inventory of resources, and dynamically allocates resources to users. As needs change, the utility also adds or removes resources from its inventory. Typical resource management systems [6] limit the types of resources that can be requested by users to machines,

[1] Work done while author was at HP Laboratories.

clusters of machines, or space for storing data. However, as users move towards a utility computing environment, they will require much more flexibility in both the types of resources they want, as well as how those resources are configured. For example, a user may ask for a fully configured e-commerce environment from the resource provider. Hence, it is important to understand how requests for such complex environments can be met by constructing them "on-the-fly" from components such as software, firewalls, servers, load balancers, and storage devices. Automatically composing such "made-to-order" environments (e.g., a three-tier e-commerce application) from base resources (e.g., servers, storage, firewalls, subnets, software etc.) is a complex task, both because of the complexity inherent in resource compositions, but also because operators and users have requirements that need to be taken into consideration when doing the composition.

In this paper we describe a model for constructing such environments based on policies. In the model, the complex environments themselves are treated as higher-level resources that are composed from other resources. Policy constraints are used to guide and restrict the construction choices used to build these higher-level resources. These constraints can be embedded in the various resource types, specified by the operators of the resource pool, or by users as part of the requests for resources. We describe how such policies can by used by a utility to automatically compose resources that meet all policy constraints specified at all levels.

This paper is organized as follows. Section 2 describes the model for policy-based resource construction. Section 3 provides a number of examples to show how this model can be used to compose policies for a variety of resources and to undertake component selection. In section 4, we describe implementation issues we have considered in prototyping this model. This is followed by a summary of related work in Section 5 and conclusions in Section 6.

2 Policy-Based Model for Resource Construction

When resources are combined to form other higher-level resources, a variety of rules need to be followed. For example, when operating systems are loaded on a host, it is necessary to validate that the processor architecture assumed in the operating system is indeed the architecture on the host. Similarly, when an application tier is composed from a group of servers, it may be necessary to ensure that all network interfaces are configured to be on the same subnet, or that the same version of the application is loaded on all machines in the tier. To ensure correct behavior of a reasonably complex application, several thousand such rules may be necessary if the construction of such applications is to be automated. This is further complicated by the fact that a large fraction of these rules are not inherent to the resources, but depend on preferences (policies) provided by the system operator or indeed, by the customer as part of the request itself.

In this section, we propose a model for combining resources that allows specification of such rules in a distributed manner. By capturing the construction rules as part of the specification of resource types, and by formalizing how these rules are combined when resources are composed from other resources, we provide a very flexible model for policy-based resource construction.

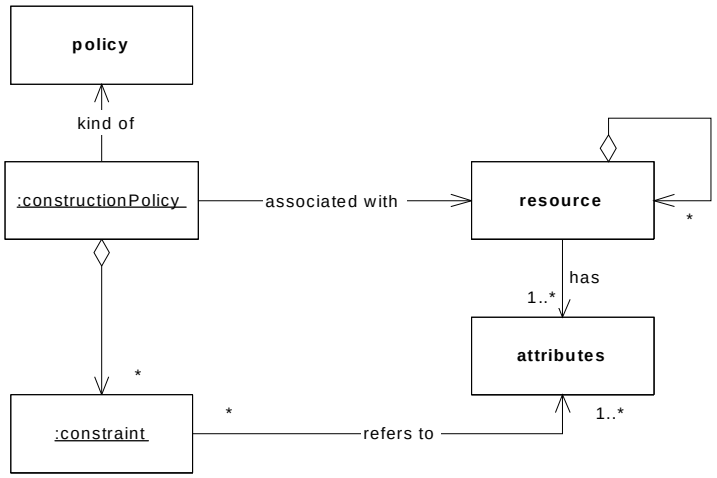

Figure 1: Conceptual model for resource construction

Figure 1 shows the basic conceptual structure of the model. Each resource is defined in the model by a resource type definition. We assume that resources are described by attributes that are part of the resource type. These attributes reflect configuration or other parameters that are meaningful for resource construction. We also assume that resources can be composed from aggregates of other resources, and that new resource types can be constructed by combining other resource types.

Instances of construction policy are associated with each resource type. Construction policy instances contain constraints that are defined using the attributes present in the resource type definitions. When a resource is instantiated, the resource management system ensures that all constraints specified for that resource are satisfied. Because resource types can be derived from other resource types, this implies that all constraints for all composing resources are also satisfied. Because the resource manager creates the union of all (relevant) constraints when instantiating resources, it can also accommodate a variety of operator and user level policies during instantiation. We discuss this further in Section 3.

Construction policy is modeled as shown in Figure 2. Every instance of a construction policy is associated with an instance of one or more resource types. Construction policy is modeled as an aggregate of one or more constraints that are defined using one or more attributes in policy. Policy attributes usually refer to attributes of the associated resource type, but may also be internal to the policy definition for convenience.

A policy applies to all the associated resource types. When the resource type is instantiated, the instantiated resource is defined to be *in compliance* with the construction policy if *all* policy constraints that refer to the attributes of the corresponding resource type are satisfied by the resource instance. Conversely, an instantiated resource is defined to be *in violation* of construction policy if *any* constraint that refers to an attribute of the corresponding resource type is violated by the resource instance. Note that it is possible for an instantiated resource to be in compliance, while the construction policy is in violation if the violation occurs as a result of a constraint that does not depend on any attribute of the resource. Additionally, if the constraint causing the violation refers to attributes in multiple

resources, it may not be possible to uniquely determine which resource is causing the policy violation. Unlike traditional [7] models of policy, no actions are defined in the construction policy model. We will discuss this further below.

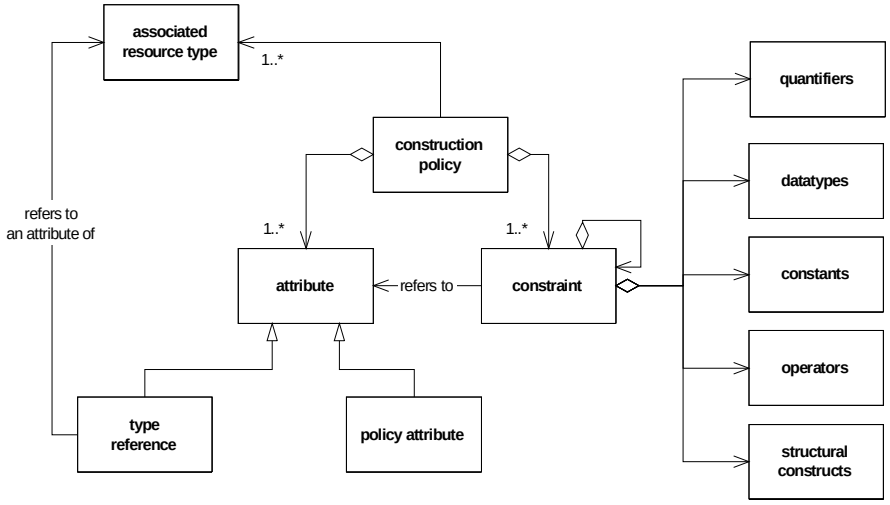

Figure 2: Elements of construction policy

Constraints form the core of the policy specification and are defined using expressions that use policy attributes as variables. During the instantiation process, the attribute values from the corresponding resource instances are used to validate policy. Constraints contain first order predicates, arithmetic and logical operators, and other structural constructs defined below:

Data types: Data types may be imposed on attributes as constraints that have to be satisfied by the corresponding attribute, e.g. constraints can specify if a particular attribute should be a String, integer, float etc.

Constants: Numeric or string constants may be used in constraints for defining the values or thresholds for attribute values.

Quantifiers: Quantifiers are often used in constraints, e.g. ∀ (for all), ∃ (there exists), etc.

Operators: A number of operators can be used to combine attributes in defining constraints. These operators fall in the following categories:

Arithmetic operators (+, -, *, /): These operators can be used for constructing arithmetic expressions on literals of the allowed data types.

Comparison operators (<, >, <= , >=, ==, !=): Comparison operators can be used to compare other expressions, and result in a Boolean value.

Boolean Operators (&&, ||, ! (unary not)): Provide logical expressions in constraints.

Implication Operators (==> (logical implication), <== (reverse implication), <=> (equivalence, or if-and-only-if)): Enforce a value for a given Boolean expression based on the result of another Boolean expression.

The instanceOf Operator: The <: operator is used to denote "an instance of" relationship. This allows constraints to be created that enforce data types on components or their attributes

Set Operator: The \in operator may be used to constrain values of an attribute to be always in a set.

Structural Constructs: Other structural constructs (e.g., let in, if then else etc.) are used mostly for syntactic convenience. These familiar programming constructs simplify the task of the constraint writer when complex constraints have to be expressed in policy.

Operationally, this model allows constraints to be specified in a distributed and hierarchical manner. The instantiation process is shown in Figure 3.

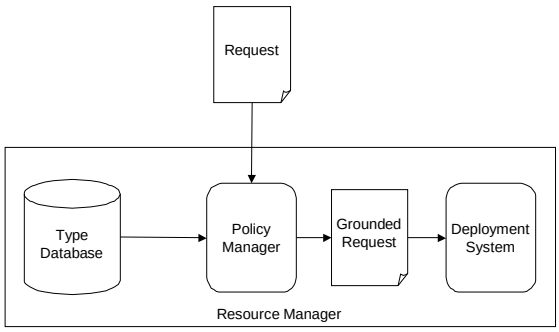

Figure 3: Resource construction process

The resource manager contains a type database containing the resource type definitions (and the associated construction policies). A user submits a request to the resource manager for some resources. The request may contain additional policy constraints desired by the user (using constructs similar to the ones in the resource type database). The resource manager extracts the corresponding types from the database, and sends the resource request and the types to the policy manager. The policy manager treats the constraints and types requested by the user as a goal to be achieved. It treats the problem as a constraint satisfaction problem, and uses a constraint satisfaction engine [8] to assign values to all attributes in the resource type definitions, such that all of the policy constraints are satisfied. The output of the constraint satisfaction engine is a request specification (a grounded request) where all attributes have been filled out. This grounded request is then handed to the deployment manager [18] for actual instantiation.

Note that because the policy manager assigns values to all the attributes such that all the constraints are satisfied, explicit condition-action pairs are not needed in construction policy. Thus, the model allows complex configurations to be built without requiring the user or the operator to pre-specify which combinations are valid and/or having to explicitly specify how such combinations can be achieved.

3 Examples of Applying Construction Policies

In this Section, we use a number of examples to highlight how this model of construction policy can be used in practice.

3.1 Resource Composition

When composing higher-level resources from other resources (e.g., an e-commerce site from servers), a variety of resources need to be put together. However not all possible combinations are valid. Policies can be attached to component types to ensure that the resulting construction is valid. To illustrate this principle, let us assume that we are trying to create servers. A server is simply defined as a computer system with an operating system on it. In order to create servers we need to select a computer and install an operating system image on it. A Server resource entity is thus constructed out of an underlying Computer resource and an OperatingSystem resource. However, not all computer types may be available in the resource pool, and not all operating system images would work on a given computer. Thus we need to define constraints that identify which computer and operating system image combinations are valid for constructing a Server. Figure 4 shows one possible way of doing this.

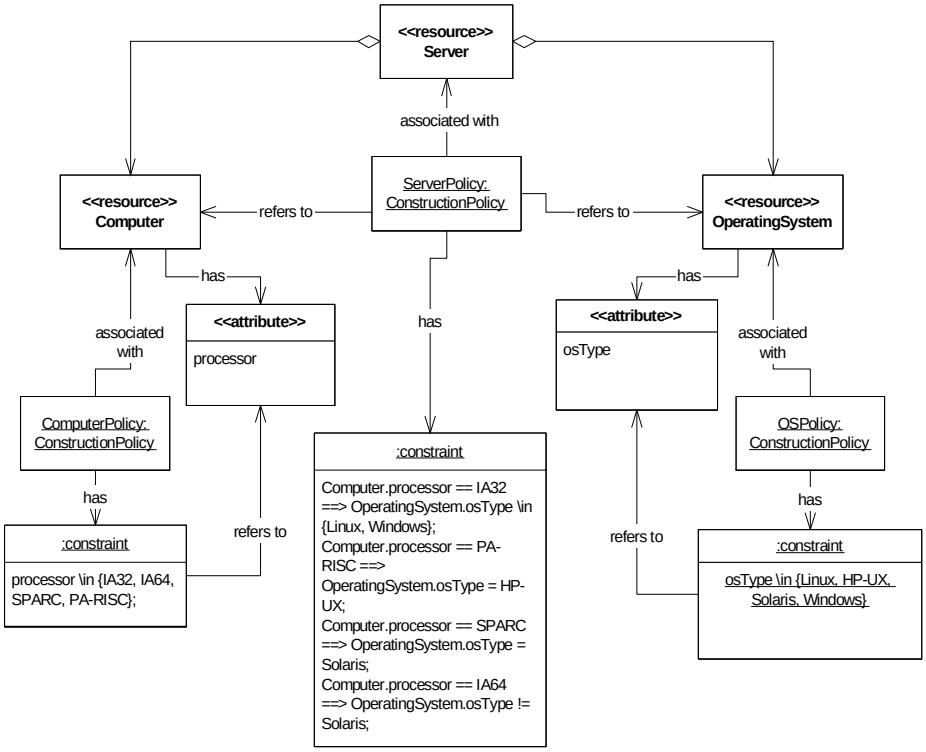

Figure 4: Example of a composition policy for a server

In the Figure, a Server is a resource type that is composed from two other resource types: a Computer, and an OperatingSystem. A Computer has an attribute processor while an OperatingSystem has an attribute called osType. A policy associated with the Computer type states that the attribute processor can only take values in the set {IA32, IA64, SPARC, and PA-RISC}. Note that this constraint is

specified by the system operator (perhaps because only these instances are available in the resource pool). Similarly, the construction policy associated with the OperatingSystem resource type states that the attribute osType can only take values in the set {Linux, HP-UX, Solaris, and Windows}. Again, the set is defined by the operator based on available operating systems within the utility. When the Server resource type is created, the type definition includes policy constraints that specify which osType can exist with which processor In order to create a valid instance of Server. Suppose a request specifies that it needs a resource of type Server with the additional constraint that Server.Computer.processor = PA-RISC. From the constraints specified as part of the Server, the policy system determines that the only valid value for OperatingSystem.osType = HP-UX, and automatically fills that value.

This example shows a number of aspects of constraint-based construction policies:

- By associating policy constraints with the individual component types, construction using those types can be controlled. By changing the allowed policy constraints, valid (or available) configurations can be maintained by the operators, and easily changed as needs change without extensive code or system-level modifications about how the new types are handled.
- Policy constraints for a resource type depend only on the attributes of that type, or the attributes of the underlying resource types. This simplifies hierarchical specification of types, because such dependencies can usually be localized in the type hierarchy. The designer of a new type only has to deal with the preceding types it is using and the corresponding constraints on them. The policy system automatically accounts for other dependencies that are created through transitional relationships.
- The policy system checks all constraints for validity when handling resource requests. Because it can locate constraints that are being violated, the request specification can be checked for "correctness" with respect to those constraints.
- The system can also fill in attribute values based on correctness of constraints. This means that the requestor has the freedom to only specify the attributes that are meaningful, and let the system fill in the gaps. Note that this is an important capability, because allows the requestor to provide minimally-specific requests, and does not require that the requestor know all of the parameters or constraints used for the construction a-priori.
- The requestor can add additional constraints for the policy system as part of the request. Because the policy system forms the union of all constraints when constructing the system, request-specific policy can be easily incorporated during construction.

3.2 Component Selection

In a resource utility, multiple components that offer the same base capability are often available in the resource pool. Examples may be different types of servers, firewalls, or network switches. However, these components frequently differ in the features (e.g., security, availability, throughput) offered by them. Such features can be captured by the attributes of the components, and depending upon the capabilities desired by the requestor, the appropriate components can be selected to meet the users' requests. In this section, we provide two other examples to highlight how policies can be used to provide construction choices for components.

Capability-based component selection

Suppose multiple types of switches are available in the resource pool. The switches offer different levels of security capabilities. For example, some switches may have Kerberos authentication or secure shell capabilities implemented, while others may not. Rather than forcing the user to understand all the details of the switches, the utility could allow the user to simply specify that she wants a switch fabric that supports secure shell access, and automatically select the appropriate components to meet that request. This is shown in the example below, where three different kinds of Cisco Catalyst switches are available, and the security capabilities needed are specified in the request.

```
// this component captures various switch attributes. Most likely extended from CIM
Switch {
        manufacturer: String;
        switchFamily: String;
        model: any;
        security: SwitchSecurityCapability
        // other attributes
}
// this component captures security capabilities of switches
SwitchSecurityCapability {
        PortSecurity: boolean;
        TACACSauthentication: boolean;
        AdvancedLayer3and4ExtendedAccessList: boolean;
        DynamicACL: boolean;
        PolicybasedRouting: boolean;
        NetworkAddressTranslation: boolean;
        SNMPv3: boolean;
        secureshell: boolean;
}
// this component defines the capabilities of available Catalyst switches
CiscoCatalystSwitch extends Switch {
        enum availableModels { WSC3750G24TSE, WSC2950ST24LRE, WSC4912G};
        satisfy (manufacturer == "Cisco");
        satisfy (switchFamily == "Catalyst");
        satisfy model \in availableModels;
        // security capability support in the available models
        satisfy (model == WSC3750G24TSE) ==>
                (security.portSecurity == true && security.ACL == true &&
                security.kerberos == true && security.TACASauthentication == true);
        satisfy (model == WSC2950ST24LRE) ==>
      (security.portSecurity == true && security.SNMPv3 == true &&
       security.ACL == true && security.TACASauthentication == true
       && security.secureshell == true);
        satisfy (model == WSC4912G) ==>
                (security.portSecurity == true && security.SNMPv3 == true &&
                security.TACASauthentication == true &&
                security.secureshell == true);
}
// request: This example shows how the security policy in a request
// selects a switch for the implementation. “main” is a special component
// that defines the system requested
main {
        sw: Switch;
        // this constraint specifies the desirable security
        // capabilities needed for switch
        satisfy (sw.security.ACL == true && sw.security.secureShell == true);
}
```

The example has been written in a pseudo-language similar to one used by our deployment manager [18]. Satisfy clauses identify policy statements in the type definitions. Note that:

- In this example, if switches with other capabilities become available, the operator can simply add those types to the system with their corresponding capabilities.
- Because the policy system finds attribute values that satisfy all constraints, an appropriate model of the switch is automatically selected.
- If multiple switches satisfy the user's request, the policy system is free to choose (an arbitrary) switch from the list of switches that satisfy the request. Other policies (such as cost of the solution) can be added in the policies to further restrict[2] how the selection takes place.

Class-of-service based component selection

In resource utility systems it is important to allocate resources to users based on certain criteria. These criteria may relate to the class of the user or the corresponding QoS implications. Often simple classes of users are defined (e.g. platinum, gold, silver etc) and users are provided with QoS guarantees based on that classification. In the next example, we demonstrate how different types of servers could be assigned to satisfy a request based on the user classification. In the example, the user class is contained in a context, which is modeled as a policy element that also contains the request. This enables the policy manager to account for the user classification during component selection.

```
// context defines the user context within which the user request is made
Context {
        userType: any;
        enum userTypes {platinum, gold, silver};
        request: any;
        // for silver users, satisfy server requests with basic servers
        satisfy (userType == silver) && (request <: RequestForServer) ==>
                request.grade == basic;
        // for gold users, offer a medium grade server
        satisfy (userType == gold) && (request <: RequestForServer) ==>
                request.grade == medium;
        // for platinum users, provide an advanced grade server
        satisfy (userType == platinum) && (request <: RequestForServer) ==>
                request.grade == advanced;
}
// The following maps different grades of servers to different requests
RequestForServer {
        enum grade {basic, medium, advanced};
        server: any;
        // a basic server is provided for basic grade requests
        satisfy (grade == basic) ==> (server <: Server) && (server.grade == basic);
        // A simple advanced server is provided for medium grade requests
        satisfy (grade == medium) ==> (server <: Server) && (server.grade == advanced);
        // A fail-over advanced grade server is provided for advanced grade requests
        satisfy (grade == advanced) ==>
                (server <: FailOverServer) && (server.grade == advanced);
```

[2] Note however, that the constraint satisfaction engine does not solve an optimization problem. Thus it is currently not possible to ask for a "minimum cost" solution, although "cost < 500" is an appropriate constraint. Merging optimization and constraint satisfaction problems is a subject of future research.

```
}
// base definition of a server
Server {
        grade: any;
        model: String;
        processor: any;
        numOfProcessors: int;
        memory: any;
        memtypes: any;
        powerSupply: any;
        powerSupplyTypes: any;
}
// Proliants can offer different capabilities to satisfy different grades
Proliant8500Server extends Server {
        satisfy grade \in {basic, advanced};
        // scaling up of ProliantServer
        satisfy numOfProcessors \in {1, 4, 8, 16};
        satisfy memtypes  \in
        {onlineSpareMemory, HotPlugMirroredMemory, HotPlugRAIDMemory};
        satisfy powersupplyTypes \in {usual, redundantPowerSupply, UPS};
        // advanced grade Proliants have the following capabilities
        satisfy (grade == advanced) ==>
                (memtypes \in {HotPlugMirroredMemory, HotPlugRAIDMemory}) &&
                (powerSupplyType <: UPS) && numOfProcessors == 16;
        //basic grade Proliants have the following capabilities
        satisfy (grade == basic) ==> (memtypes \in {onlineSpareMemory} ) &&
                ((powerSupplyType <: usual) ||
                (powerSupplyType <: redundantPowerSupply)) &&
                (numOfProcessors <= 8);
}
// a failover server actually contains two servers—one as a backup
FailoverServer extends Server {
        // a failover server
        server: Server;
        backupServer: Server;
}
// goal specified as a request
main{
        context: Context;
        satisfy context.userType == platinum;
        satisfy context.request <: RequestForServer;
}
```

Similar examples can be constructed for other capabilities such as availability, response time, throughput, processor speed based selection of components. These metrics have to be considered when constructing complex resources for different classes of users. Additionally, other metrics that have to be managed at run-time may be specified using a similar constraint language.

4 Implementation Issues

We are implementing a prototype resource manager (called Quartermaster) to validate the automatic construction concepts mentioned in this paper. The Quartermaster resource model extends the Common Information Model (CIM) [9], which is an object-oriented information model standard for IT systems from Distributed Management Task Force (DMTF) [10]. Because CIM defines information models for a large number of IT resources (including models for devices,

networks, databases, and users), all conforming to a single meta-model, it allows us to rapidly incorporate a large number of resources in our prototype without having to construct resource models from scratch. All resource type definitions map to classes in CIM (typically those under CIM_System class). Many types needed already exist as CIM classes, but others (e.g., application server, web server, tiers, e-commerce sites etc.) that are essential for the utility environments have been added by us in our prototype as extensions to the existing CIM classes.

Because construction policy is associated with the resource type definitions, it is necessary to associate the policy specification for resource construction with the type definition of the resource. In order to validate our approach, we have to define policy constraints for the existing CIM resource types in addition to the types we add to CIM. However, the CIM meta-model does not provide for associating policy instances with the type definitions. In our work, we are exploring how construction policy constraints can be specified within the framework provided by CIM.

One possible approach is to represent constraints as properties within CIM classes, but "tag" these special properties as constraints. CIM allows qualifiers for adding such special tags to properties, and a new qualifier called "constraint" can be added on a property to indicate that its value is a constraint as opposed to a typical property. The advantage of this approach is that it does not require a change in the CIM meta-model. All existing CIM and MOF tools (such as parsers, repositories, and browsers) will continue to work with this extension without any modification. However, they will not be able to interpret the meaning of a constraint nor would they be able to parse or check their syntactic correctness. For example, the MOF parser, which parses a MOF file and stores the defined classes into a CIM repository, would treat these constraints as strings and will not check for the correctness of the constraint expression.

Another approach would be to extend the CIM meta-model to incorporate constraints. A possibility is to use the Object Constraint Language (OCL), which is part of the UML standards [11], and is under consideration by DMTF for inclusion in CIM. However, any changes to the CIM meta-model require changes in existing CIM tools. In our work, we have used the former approach for modeling purposes, and added additional tools for checking constraint syntax.

We have implemented a CIMOM based resource inventory based on the SNIA CIMOM implementation [12]. In our inventory, we have extended the CIM resource model for defining new resource types that are required in a resource utility, have added constraints to the existing and new resource types and have populated the CIMOM with these resource type definitions.

We have also implemented the policy manager. When the request arrives at the policy manager implementation, it is parsed and checked against the resource model types available from the type database. A summarizer captures all relevant resource types that may be necessary to fulfill the request, collects all policy constraints from the different types, as well as from the request, and converts them into an intermediate representation. For convenience we used a representation based on the SmartFrog language [18] that is used by our deployment manager and preprocessed by our solver [21]. However, we believe that it should be possible to use other representations such as OCL [11] or Alloy [22], [23] depending on the underlying solver. This representation is handed to a formula generator which uses the constraints to compose logical formulae that represent those constraints. The solver

reasons on the formulae and does a feasibility analysis. If the solver determines that a solution is feasible it uses a concretizer to create a system specification that satisfies all the constraints.

The initial results of the implementation are encouraging. We have tried the policy manager on models comprised of hundreds of attributes and found the performance to be satisfactory with solution times in the order of seconds.

5 Related Work

Introducing constraints in UML specification of systems for configuration purposes is discussed in [13]. They define a set of construction rules at one place termed a domain. In that sense the approach is similar to expert systems. In our approach, we embed constraints hierarchically thus distributing constraints on to various resource types, and taking into account these constraints as the construction happens as opposed to creating a large number of constraints (rules) a priori. Our approach enables flexibility and extensibility in specification of constraint and in automatic construction depending on the user requirements. We have also applied the concept to CIM, which is the de-facto standard for management of infrastructure.

The ClassAds MatchMaking work [14] assumes that the match-maker matches the requestor entity's request against the provider entity's ClassAds (which are specifications in a semi-structured language). The assumption is that all the resources (like machines) exist a-priori and have been advertised. In a resource-utility environment however, some of the resource instances may not even exist a-priori (as is the case with transient/virtual resources) or may be logically constructed resources that have to be instantiated on-demand (e.g. appserver/tier/farm/e-commerce site). This causes a problem for approaches that undertake match-making only on instances. We enable construction on-the-fly by embedding constraints hierarchically in *the resource types* as described in this paper. This requires that the resource types be advertised by the resource manager, not just the instances.

A lot of work has been done in the community in terms of specifying, and associating events, conditions and actions for policies, namely IETF [15], CIM [9], PARLAY [16], PONDER [7] etc. Constraint satisfaction approachs have been used in goal-oriented systems [20] and in other application domains [19]. These bodies of work, to the best of our knowledge, have not looked at incorporating first-order logic and linear arithmetic based constraints in resource types for automatic construction of resources, and have not used a constraint satisfaction approach for arriving at a constructed resource specification.

6 Conclusion

Utility environment are fairly dynamic environments and deal with a large number of complex heterogeneous resources. These environments have to configure complex resources from other resource components while keeping in mind the user requirements specified as goals, resource requirements/constraints and operator policies. In this paper, we have discussed how automatic construction of such complex resources may be enabled by embedding constraints in the resource models themselves and using a constraint satisfaction approach to solve the relevant constraints coming from multiple sources. We are using a CIM-based representation

to specify resource-level construction policies. Once these policies have been specified, resource construction can be framed as a goal-satisfaction problem using the resource models for undertaking resource composition and component selection. In future, we plan to extend the work to also automatically specify configuration actions and automatically create workflows for deployment of resources.

References

[1] HP Utility Data Center (UDC) http://www.hp.com/solutions1/infrastructure/solutions/utilitydata

[2] IBM Autonomic Computing http://www.ibm.com/autonomic

[3] SUN N1 http://wwws.sun.com/software/solutions/n1/

[4] Microsoft DSI http://www.microsoft.com/management/

[5] Global Grid Forum http://www.ggf.org

[6] Platform LSF http://www.platform.com/products/LSF

[7] Nicodemos Damianou, Narankar Dulay, Emil Lupu, Morris Sloman: *The Ponder Policy Specification Language.* POLICY 2001: 18-38

[8] van Hentenryck, P. *Constraint Satisfaction in Logic Programming*, The MIT Press, Cambridge, Mass, 1989.

[9] CIM Modeling http://www.dmtf.org/standards/standard_cim.php

[10] DMTF: http://www.dmtf.org

[11] Object Constraint Language (OCL) http://www.ibm.com/software/awdtools/library/standards/ocl.html

[12] SNIA CIM Object Manager (CIMOM). http://www.opengroup.org/snia-cimom/

[13] Felfernig A, Friedrich G. E et al. *UML as a domain specific knowledge for the construction of knowledge based configuration systems.* In the Proceedings of SEKE'99 Eleventh International Conference on Software Engineering and Knowledge Engineering, 1999.

[14] Raman R, Livny M, Solomon M. *MatchMaking: Distributed Resource Management for High Throughput Computing.* In the proceedings of HPDC 98.

[15] IETF Policy. http://www.ietf.org/html.charters/policy-charter.html

[16] PARLAY Policy Management http://www.parlay.org/specs

[17] Moffet J, Sloman. *Policy Conflict Analysis in Distributed Systems.* In the proceedings of Journal of Organizational Computing,, 1993

[18] SmartFrog http://www-uk.hpl.hp.com/smartfrog/

[19] KrishnaKumar K.T, Sloman M. *Constraint based Network Adaptation for Ubiquitous Applications.* Proceedings of the 6th International EDOC Conference. Sep 2002, pp 258-271, Lausanne, Switzerland.

[20] LamsWeerde A, Letler E. *Handling Obstacles in Goal-oriented Requirements Engineering.* IEEE Transactions on Software Engineering. Special Issue on Exception Handling. 2000.

[21] Flanagan C, Joshi R., Ou, X, Saxe J., *Theorem Proving Using Lazy Proof Explication.* In Lecture Notes in Computer Science, Springer-Verlag Heidelberg, Volume 2725, pp. 355-367, Jul 2003.

[22] Jackson D. *Automating First-Order Relational Logic.* Proc. ACM SIGSOFT Conf. Foundations of Software Engineering, November 2000.

[23] Alloy http://sdg.lcs.mit.edu/alloy/