Automated Policy-Based Resource Construction in Utility Computing Environments

Akhil Sahai, Sharad Singhal, Rajeev Joshi, Vijay Machiraju
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2003-176
August 21st , 2003*

E-mail: {firstname.lastname}@hp.com

policy,
resource utility,
utility computing,
configuration,
management

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 construction. 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.

* Internal Accession Date Only

Approved for External Publication

© Copyright Hewlett-Packard Company 2003

Automated Policy-Based Resource Construction in Utility Computing Environments

Akhil Sahai, Sharad Singhal, Rajeev Joshi, Vijay Machiraju
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA 94034
{firstname.lastname}@hp.com

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 construction. 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], and Microsoft's DSI initiative [4] are examples of this trend. In addition, much of the work in the Global Grid Forum [5] [23] is targeted at developing a web-services based infrastructure that can offer resources (e.g. servers, software, storage etc.) to users when needed. The intent in these initiatives is to create systems that provide automated provisioning, configuration, and lifecycle management of a wide variety of infrastructure resources.

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, 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, loadbalancers, 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 restrict the construction choices used to build these higher-level resources. Policy choices that guide the construction 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 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 which 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.

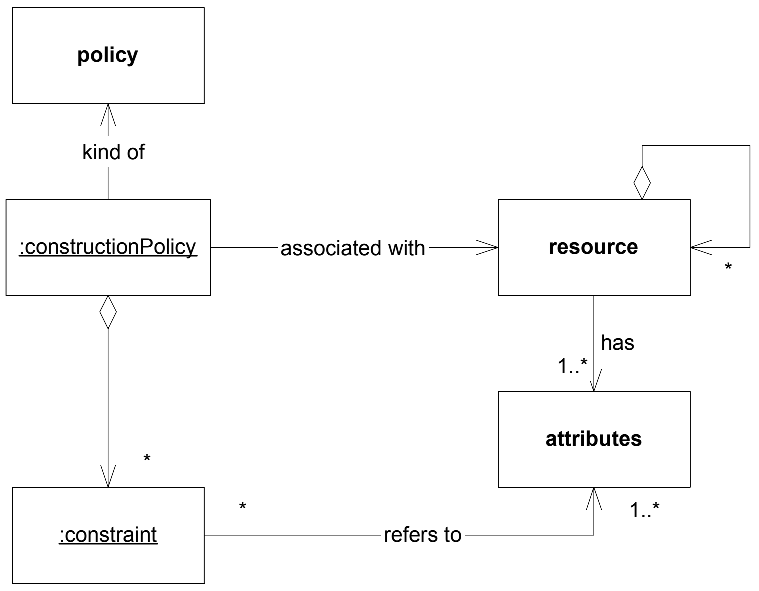


Figure 1: Conceptual model for resource construction

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.

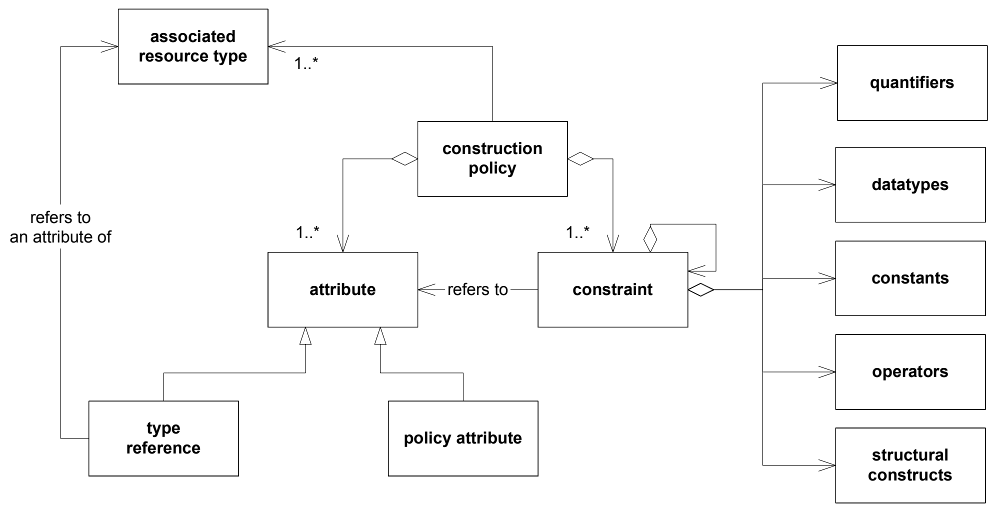


Figure 2: Elements of construction policy

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. This allows validation of data types when different underlying components are used to provide similar functionality.

Constants: Numeric or string constants may 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)): These operators allow expression of dependencies between attributes, e.,g. (name == Solaris) ==> version \in {5.7,5.8};
- *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, e.g., // ensure that component "server" is an instance of type Appserver server <: AppServer;
- *Set Operator*: \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.

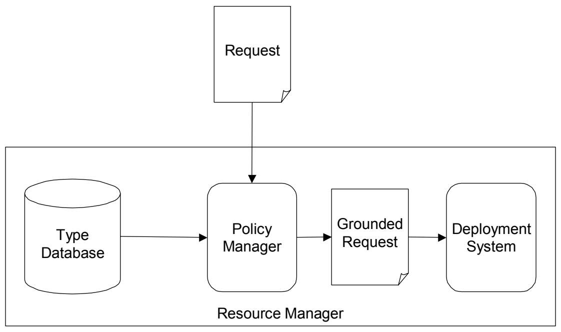


Figure 3: Resource construction process

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 [21] 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.

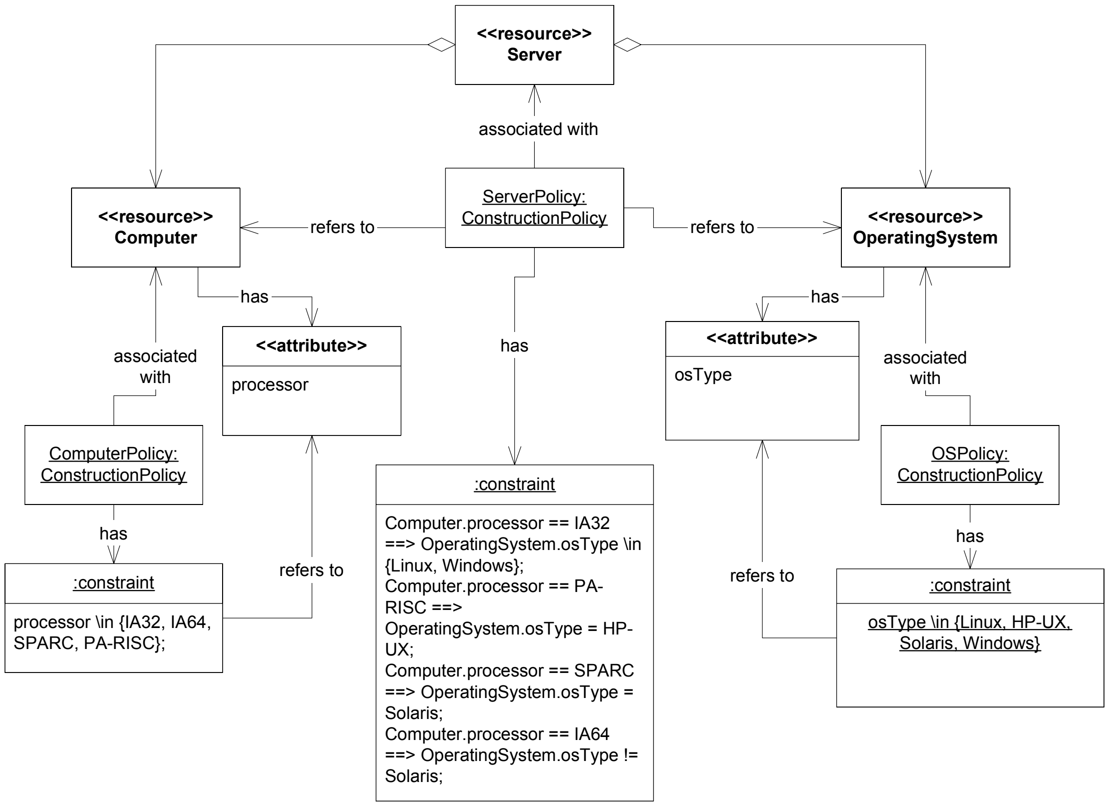
3 Examples of Applying Construction Policies

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 5 show one possible way of doing this.

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`.



<<resource>> Server
associated with
ServerPolicy: ConstructionPolicy
<<resource>> Computer
refers to
refers to
<<resource>> OperatingSystem
has
has
has
<<attribute>> processor
<<attribute>> osType
associated with
associated with
ComputerPolicy: ConstructionPolicy
OSPolicy: ConstructionPolicy
has
has
:constraint
processor \in {IA32, IA64, SPARC, PA-RISC};
refers to
:constraint
Computer.processor == IA32 ==> OperatingSystem.osType \in {Linux, Windows};
Computer.processor == PA-RISC ==> OperatingSystem.osType = HP-UX;
Computer.processor == SPARC ==> OperatingSystem.osType = Solaris;
Computer.processor == IA64 ==> OperatingSystem.osType != Solaris;
refers to
:constraint
osType \in {Linux, HP-UX, Solaris, Windows}

Figure 3: Example of a composition policy for a 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. Similarly, during the instantiation of new resource types, the policy system can aid the designer by validating that at least one valid instance of the new type can be created.
- 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. This simplifies the requests.
- 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.

Many additional constraints can be added to this simple example to account for items such as licenses, software versions, or other attributes such as memory and CPU speed. In addition, higher level resources can be constructed in a similar manner. For example, a `Server` may take the role of a `webServer`, an `applicationServer`, or a `databaseServer` if the corresponding images are installed on it. Furthermore, by adding topology constraints (e.g. web server tier precedes an app server tier which in turn precedes a data base tier), one can construct much more complex resource types like a three-tier architecture or a highly available server and treat them as higher-level resources. These complex resource types may then be instantiated and deployed.

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 capabilities (e.g., security, availability, throughput) offered by them. Such capabilities 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 four 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 using the grammar specified in Appendix A to show how the components (and the request) could be written. Again, note that:

1. In this example, if switches with other capabilities become available, the operator can simply add those types to the system with their corresponding capabilities.

2. Because the policy system finds attribute values that satisfy all constraints, an appropriate model of the switch is automatically selected.

3. 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[1] how the selection takes place.

Transient (virtual) resources

Transient or virtual resources (e.g., virtual machines [9]) have to be managed by a resource utility. Transient resources are instantiated, allocated and removed just like other resource instances. However, unlike other resources, transient resources do not have underlying a-priori resource instances in the resource pool. Let us consider an example related to server partitioning. A `Server` not only can be allocated and used as a single resource; it may be partitioned for better resource utilization. Server partitions enable multiple applications to run on the same server while maintaining isolation among them, thus improving server utilization. A `Server` can support both hardware and software partitions. Multiple software partitions may be supported within a hardware partition. A hardware partition is termed an nPartition (nPar) and a software partition is termed a Virtual Partition (vPar) as shown in Figure 4. Usually, there are restrictions on the number of nPars and vPars that may be created on top of different types of servers. For example, an nPar may be restricted to have 2, 4, 8 or 16 CPUs in it.

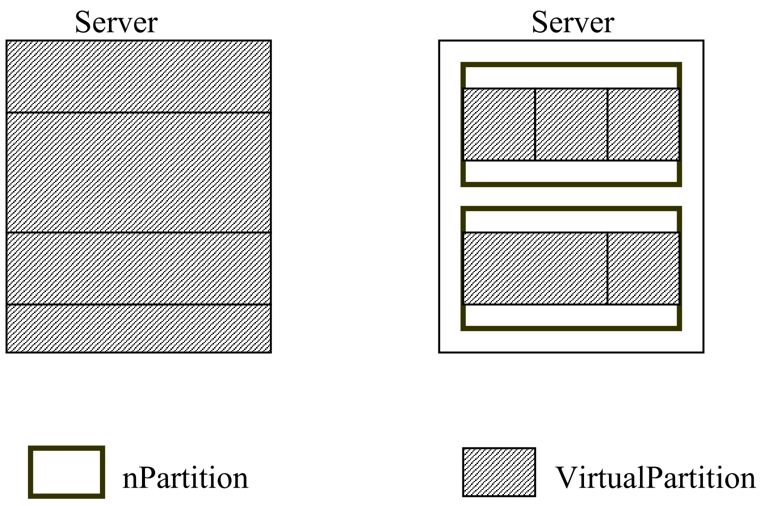


Figure 4: Hardware and software partitions in a server

In the following example, we show how such restrictions can be coded as policy constraints in the types defining virtual servers.

```
// A virtual partition
VirtualPartition extends Partition{
            // at least one CPU in the virtual partition
            int numCPU;
```

[1] 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.

```
            cpu : CPU[numCPU]; satisfy numCPU >= 1;
            // A virtual partition has its own oS Image
            osImage: OperatingSystemImage;
            // software images may be installed in a virtual partition
            swImage: SoftwareImage;
}
// a nPartition can have virtual partitions
nPartition extends Partition{
            // at least 4 CPUs in a nPartition
            int numCPU;
            int numOfvPars;
            cpu: CPU [numCPU];satisfy numCPU >= 4;
            vpars: VirtualPartition [numOfvPars];
            satisfy numOfvPars > = 0;
            // total number of CPUs in the virtual partitions
            // and the nPartition is the same
            satisfy (numOfvPars > 0) ==>
                        (numCPU == (+ i: 0<=i< numOfvPars: vpars[i].numCPU)) ;
}
// A Server can have a number of either nPars or vPars
Server{
            int numCPU ;
            cpu: CPU[numCPU]; satisfy numCPU > 0;
            manufacturer: String;
            memory: int; // in GB
            model: any;
            int numOfPartitions;
            partitions: any[numOfPartitions];
            parttitionType: any;
            satisfy partitionType <: nPartition || partitionType <: VitualPartition;
            satisfy if partitionType == nPartition then partitions <: nPartition[];
                        else partitions <: VitualPartition[];
            satisfy numOfPartitions >= 0;
            // if there are partitions
            satisfy numOfPartitions > 0 == >
                        (numCPU == (+ i:0 <= i < numOfPartitions: partitions[i].numCPU) ;
}
HPrpServer extends Server{
            enum hprpTypes {rp8400, rp7410};
            satisfy model \in hprpTypes;
            // only nPartition can be created on this type of server
            satisfy (partitionType <: nPartition);
            // At max of 2 nPartitions may be created
            satisfy (numOfPartitions >= 0 && numOfPartitions <= 2);
            // maximum 16 virtual partitions
            satisfy (model == rp8400) ==>
                        ((+ i: 0<= i < numOfPartitions: partitions[i].numOfvPars) <= 16);
            // maximum of 8 virtual partitions
            satisfy (model == rp7410) ==>
            ((+ i:0 <= i< numOfPartitions: partitions[i].numOfvPars) <= 8);
}
Superdome extends Server
{
            enum SuperdomeTypes {16way, 32way, 64way};
            satisfy model \in SuperdomeTypes;
            satisfy (model == 16way) ==> (numCPU==16) && (memory == 64);
            satisfy (model == 32way) ==> (numCPU==32) && (memory == 128);
            satisfy (model == 64way) ==> (numCPU == 64) && (memory == 256);
            satisfy (partitionType <: nPartition) || (partitionType <: VirtualPartition);
            // number of CPUs in a partition is recommended to be 8
            satisfy numOfPartitons > 0 ==>
                        ((+ i: 0<= i< numOfPartitions: partitions[i].numCPU) == 8);
            // number of virtual partitions in an nPartition is recommended to be 8
            satisfy (numOfPartitions > 0 && partitionType <: nPartition) ==>
            (forall i :0<= i < nPartition: partitions[i].numOfvPars == 8));
}
main {
            server: Server;
            satisfy server.partitionType <: VirtualPartition;
            satisfy server.numOfPartitions == 4;
}
```

This example shows how relatively complex constraints on creation of resources can be easily specified as construction policies. Additionally, because the rules specify the maximum number of partitions that can be created on an underlying resource, the policy system will automatically restrict the number of such transient resources that are allowed by the system.

Multi-function and polymorphic resources

Multi-function devices and polymorphic devices are resources that need special consideration in a resource utility. Multi-function devices are devices that have multiple capabilities and are capable of performing many functions simultaneously, while polymorphic devices are those that are reconfigurable so that at any given point in time, they can appear as one of a number of different resources. For a given request, multi-function and polymorphic devices may be configured to take on the appearance of one or more resources required for that request. In the example below we describe a Virtual Service Switch (VSS) that is a multi-function device uses Virtual Service Modules (VSM) to offer multiple capabilities. VSMs are software modules that can be loaded dynamically an existing VSS.

```
// The software virtual service modules that may be installed on a
//VirtualServiceSwitch are Firewall, Intrusion Detection and Prevention
//(IDP), VPN, Virtual Global Server Load Balancer, Virtual Server Load
//Balancer, Virtual SSL, Web Acceleration
VirtualServiceModule extends ManagedElement {
            enum types {firewall, idp, vpn, vgslb, vslb, VirtualSSL, webAcceleration};
            vsmType: types;
            vsm: any;
            // In the most generic case, assuming that Firewall, IDP, VPN, VGSLB,
            // VSLB, VirtualSSL, WebAcceleration Resource Types have been already
            // defined separately and they all extend ManagedElement
            satisfy (vsmType == firewall) ==> (vsm <: Firewall);
            satisfy (vsmType == idp) ==> (vsm <: IDP);
            satisfy (vsmType == vpn) ==> (vsm <: VPN);
            satisfy (vsmType == vgslb) ==> (vsm <: VGSLB);
            satisfy (vsmType == vslb) ==> (vsm <: VSLB);
            satisfy (vsmType == VirtualSSL) ==> (vsm <: VirtualSSL);
            satisfy (vsmType == WebAccel) ==> (vsm <: WebAcceleration);
}
// A virtual Service Switch is a polymorphic device and may have any
// combination or all of the modules installed e.g. Inkra Virtual Service Switch
```

```
VirtualServiceSwitch {
            number : int ;
            installedVSM: VirtualServiceModule[number];satisfy number > 0;
}
// A Virtual Service Switch may be configured in whichever way, in
// that sense it behaves like a multi-function device. If there was a
// resource type defined that was often used we can pre-define the
// Virtual Service Modules on it
FirewallVPNServiceSwitch extends VirtualServiceSwitch {
            satisfy number >= 2;
            satisfy (installedVSM[0].vsmType == firewall);
            satisfy (installedVSM[1].vsmType == vpn);
}
```

Class-of-service based resource 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);
}
// 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) [10], which is an object-oriented information model standard for IT systems from Distributed Management Task Force (DMTF) [11]. 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., appserver, webserver, tiers, e-commercesites etc.) that are essential for the utility environments have been added on top of the existing CIM classes.

Because construction policy is associated with the resource type definitions, it is convenient to combine 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 following example demonstrates how a constraint such as "If the total visible memory size of a machine is less than 10 MB, then its OS cannot be OS/390"[2] using this approach.

```
Qualifier Constraint : boolean = false,
   Scope(property),
   Flavor(DisableOverride);
Qualifier ConstraintLanguage : string = NULL,
   Scope(property),
   Flavor(DisableOverride);
class CIM_OperatingSystem : EnabledLogicalElement {
   [Static, Constraint, ConstraintLanguage("Text")]
   string constraint1 =
      "if TotalVisibleMemorySize < 10000 then OSType != 60";
}
```

This example is written in Managed Object Format (MOF) – a language used to describe CIM models. The first two Qualifier declarations declare two qualifiers – "Constraint" and "ConstraintLanguage". This has to be done once so that the MOF parser understands these qualifiers when we use them to tag properties. The declaration of the class "CIM_OperatingSystem" includes a property called constraint1, whose value contains the constraint expression. The "Static" qualifier indicates that this constraint holds true for all instances of the class. The "Constraint" qualifier indicates that this property has to be interpreted as a constraint. For simplicity, the constraint itself is expressed in plain text (as indicated through the "ConstraintLanguage" qualifier), although more formal constraint languages such as the one described in this paper or the Object Constraint Language (OCL) [1220] can be used to represent the constraint expression.

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

[2] This is simply a sample constraint and is not intended to represent any real life scenario.

constraint expression. Given that we anticipate a large number of constraints will be necessary, this approach would make creating and maintaining policies difficult.

Another approach to incorporate constraints into CIM is to change CIM's meta-model. In addition to classes containing properties and methods (as is currently the case), we would need to add constraints as well as first class entities. This would require changes to the MOF syntax as the following example demonstrates:

```
class CIM_OperatingSystem : EnabledLogicalElement {
  [ConstraintLanguage("Text")]
  satisfy  "if TotalVisibleMemorySize < 10000 then OSType != 60";
}
```

The example shows a constraint expression added to a class in a more natural manner using a "satisfy" keyword. Qualifiers can be used to add additional information about the constraints, just like they are used to add additional information to properties and methods. This approach would require changes to existing CIM tools, but is nevertheless a more elegant and complete solution.

In addition, we are exploring how the constraints themselves will need to be represented. One possibility is to use OCL. OCL is being proposed to DMTF as an extension to CIM for defining constraints. OCL offers most of the constructs we envision as necessary for construction policy; however, like CIM, it is complex. We are currently using a simpler language (with the grammar shown in the Appendix) for our initial work. However, we anticipate that we will be able to incorporate OCL constraints in our prototype in the future.

We have implemented a CIMOM based resource inventory based on the SNIA CIMOM implementation [13]. 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. In addition, we are also in the final stages of constructing the policy manager that solves the constraint satisfaction problem such that, given a set of constraints written in first-order logic with linear arithmetic, we can determine if the constraints are feasible, and if so, produce a resource description that satisfies the request.

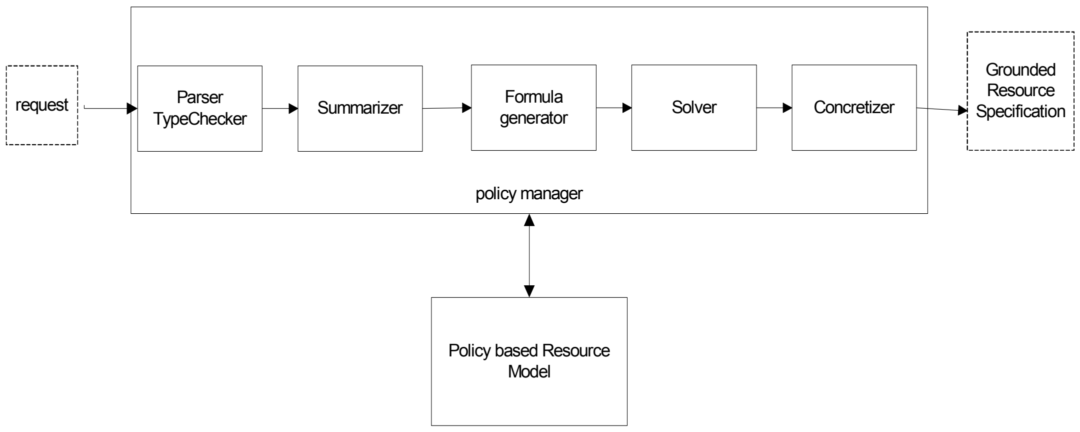


Figure 5: Architecture of the Policy Manager

Figure 5 shows the basic architecture of the policy manager. When the request arrives at the policy manager, it is parsed and checked against the resource model types available from the type database. The 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. The formula generator uses these constraints to compose logical formulae that represent the constraints. The solver reasons on the formulae and does a feasibility check. If the solver determines that a solution is feasible it uses the concretizer to create a system specification that satisfies all the constraints.

5 Related work

Introducing constraints in UML specification of systems for configuration purposes is discussed in [14]. 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. The differing user requirements may result in one construction being different from another. We have also applied the concept to CIM, which is de-facto standard for management of infrastructure.

The ClassAds MatchMaking work [15] 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. The same concepts are extensible to resource instances as well. It is also not clear whether the ClassAds language supports first-order logic and linear arithmetic. As we have shown in the examples, it is important to have notions of quantifiers, implications, equivalences and other first order-logic expressions for reasoning.

There is lot of work that has been done in the community in terms of specifying, and associating events, conditions and actions for policies, namely IETF[16], CIM[17], PARLAY [18][19], PONDER [7] etc. Additional work relates to using policy for SLA management [22]. 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 constructions of resources and have not used a constraint satisfaction approach for arriving at a constructed resource specification. The WS-Policy [20] work at OASIS has focused on generic schemas for specifying arbitrary policy assertions on web services. The constraints as specified in this article may be embedded inside these assertions.

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. We have proposed an intermediate language based on first-order logic with linear arithmetic extensions to enable reasoning and analysis on the policies and goals. As shown in the examples, the language proposed by us is expressive enough to capture a wide variety of resource models and policy behaviors. 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.

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. VMWare Virtual Machine http://www.vmware.com/
10. CIM Modeling http://www.dmtf.org/standards/standard_cim.php
11. DMTF: http://www.dmtf.org
12. Object Constraint Language (OCL) http://www.ibm.com/software/awdtools/library/standards/ocl.html
13. SNIA CIM Object Manager (CIMOM). http://www.opengroup.org/snia-cimom/
14. 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.
15. Raman R, Livny M, Solomon M. MatchMaking: Distributed Resource Management for High Throughput Computing. In the proceedings of HPDC 98.
16. IETF Policy. http://www.ietf.org/html.charters/policy-charter.html
17. DMTF-CIM Policy http://www.dmtf.org/standards/documents/CIM/CIM_Schema26/CIM_Policy26.pdf
18. PARLAY Policy Management http://www.parlay.org/specs
19. Moffet J, Sloman. Policy Conflict Analysis in Distributed Systems. In the proceedings of Journal of Organizational Computing,, 1993
20. OASIS WS-Policy WG. http://www.oasis-open.org
21. SmartFrog http://www-uk.hpl.hp.com/smartfrog/
22. Verma D, Beigi M, Jennings R. Policy based SLA Management in Enterprise Networks. Workshop on Policy, POLICY 2001.
23. Foster I, Kesselman C, Tuecke S. The Anatomy of Grid: Enabling Scalable Virtual Organizations. In the Proceedings of International Journal of Supercomputer Applications, 2001.

Appendix A: The Policy Language BNF

List of reserved keywords:
any boolean char const else end enum extends false float if in \in int let satisfy String struct then true uint16 uint32

Grammar:

```
<file> ::= <typeDecl>*
<typeDecl> ::= <enumDecl> | <structDecl>
<enumDecl> ::= "enum" <IDENTIFIER> "{" <idList> "}" ";"
<idList> ::= <IDENTIFIER> ( "," <IDENTIFIER> )*
<structDecl> ::= <IDENTIFIER> [ "extends" <IDENTIFIER> ] "{"
        <componentDecl>*
        "}"
<componentDecl> ::= <enumDecl> | <elementDecl> | <constraintDecl>
<elementDecl> ::= <idList> ":" <type> [ "=" <exprList> ] ";"
        | <constDef>
        | <funDef>
<type> ::= <baseType> ( "[" [ <INTEGER_LIT> ] "]" )*
<baseType> ::= <primitiveType> | <rangeType> | <IDENTIFIER>
<primitiveType> ::= "boolean" | "float" | "char" | "String" | "datetime" | "any"
        | <primitiveIntType>
<primitiveIntType> ::= "int" | "uint8" | "uint16" | "uint32" | "uint64"
        | "sint8" | "sint16" | "sint32" | "sint64"
<rangeType> ::= <INTEGER_LIT> ".." <INTEGER_LIT>
<constDef> ::= "const" <idList> ":=" <exprList> ";"
<funDef> ::= <IDENTIFIER> "(" <idList> ")" ":=" <expr> ";"
<exprList> ::= <expr> ( "," <expr> )*
<constraintDecl> ::= [ <IDENTIFIER> "::" ] "satisfy" <constraintExpr> ";"
<constraintExpr> ::= <simple Constraint> | <quant Constraint>
<quant Constraint> ::= "(" <quantOp> <idList> ":" <simpleConstraint> ")"
<quantOp> ::= "forall" | "exists"
<simple Constraint> ::= <expr>
<expr> ::= "let" <defList> "in" <expr> "end"
        | <ifThenExpr>
<defList> ::= <def> ( "," <def> )*
<def> ::= <ID> ":=" <expr>
<ifThenExpr> ::= "if" <expr> "then" <expr> "else" <expr> "end"
        | <iffExpr>
<iffExpr> ::= <impExpExpr> [ "<=>" <impExpExpr> ]
<impExpExpr> ::= <andOrExpr> [ ( "==>" | "<==" ) <andOrExpr> ]
<andOrExpr> ::= <equalityExpr> ( "&&" <equalityExpr> )*
        | <equalityExpr> ( "||" <equalityExpr> )*
<equalityExpr> ::= <relationExpr> [ ( "==" | "!=" ) <relationExpr> ]
<relationExpr> ::= <addExpr> [ "<"    <addExpr> ]
        | <addExpr> [ "<="   <addExpr> ]
        | <addExpr> [ ">"    <addExpr> ]
        | <addExpr> [ ">=    <addExpr> ]
        | <addExpr> [ "<:"   <addExpr> ]
```

```
            | <addExpr> [ "\in"  <addExpr> ]
<addExpr> ::= <multExpr> ( ( "+" | "-" ) <multExpr> )*
<multExpr> ::= <unaryExpr> ( "*" <unaryExpr> )*
<unaryExpr> ::= "+" <unaryExpr> | "-" <unaryExpr>
            | "!" <unaryExpr>
            | <postfixExpr>
<postfixExpr> ::= <prefixExpr> <suffixExpr>*
<prefixExpr> ::= "(" <expr> ")"
            | <IDENTIFIER>
            | <literal>
            | <setExpr>
<literal> ::= <INTEGER_LIT> | <FLOAT_LIT> | <CHAR_LIT> | <STRING_LIT> | <TRUE> |
<FALSE>
<setExpr> ::= "{" <exprList> "}"
<suffixExpr> ::= "[" <expr> "]"
            | "(" <exprList> ")"
            | "." <IDENTIFIER>
```