USENIX Association

Proceedings of LISA 2002: 16th Systems Administration Conference

Philadelphia, Pennsylvania, USA
November 3–8, 2002

Pp. 175-184 of the *Proceedings of LISA '02: Sixteenth Systems Administration Conference* (Berkeley, CA: USENIX Association, 2002).

Environmental Acquisition in Network Management

Mark Logan, Matthias Felleisen, and David Blank-Edelman
– Northeastern University

ABSTRACT

Maintaining configurations in heterogeneous networks poses complex problems. We observe that medium and large networks exhibit many contextual relationships, and argue that modeling these relationships explicitly simplifies configuration management. This paper presents a declarative data specification language, called Anomaly, that implements our ideas. Anomaly models containment relationships and uses a data aggregation technique called environmental acquisition to simplify system management. The interpreter for the language generates and deploys configurations from a source code description of the network and its hosts.

Introduction

Configuration management is an important task for system administrators, as it is often one of the largest portions of their job. In the last decade, a number of configuration management systems have emerged to help with this task. Some of this work is summarized in a paper by Remy Evard [1].

Our paper presents an attempt to take some of the best features of previous solutions and use them in a framework based on *environmental acquisition*, an abstraction mechanism borrowed from the object-oriented systems community. The result is a declarative language, called Anomaly. Anomaly is also intended to be useful in practice as well as in theory. It has a plug-in interface for adding extensions.

Previous Solutions

A survey of previous solutions to the problem of generating and managing configurations reveals three particularly important concepts: data aggregation techniques such as inheritance and class systems; logic programming techniques that reduce the complexity of configuration statements; and database-driven systems that store configuration data in a repository.

Cfengine [4] is one widely used solution. It is a language-based host configuration tool that uses a class system to aggregate configuration commands. Cfengine's class system allows an administrator to apply configuration statements to a class of machines. It uses techniques similar to logic programming to make its statements more concise. Couch and Gilfix demonstrated this in their 1999 paper, "It's Elementary Dear Watson..." [5]. Cfengine is not perfect, however. Its host-centricity does not lend itself well to other components of the environment, especially switches and routers.

Language-based configuration tools are an important contribution, but for large networks, configuration data can become unwieldy when stored in source code form. Database-driven approaches [2, 3] solve this problem by providing a repository for configuration data. This approach simplifies the maintenance of configurations and reduces the rate of errors by reducing the amount of source code that administrators have to deal with.

The paper by Couch, et al., [5] also presents an approach that leverages convergent processes based on logic programming (Prolog). The authors point out that previous approaches, especially Cfengine, already use convergent processes. An example of this is the Cfengine link command, which looks like:

```
links:
  /etc/sendmail.cf ->! mail/sendmail.cf
```

In Cfengine, the link command (like most other commands) hides a great deal of housekeeping from the administrator. The link command above takes care of checking whether /etc/sendmail.cf already exists as a link to mail/sendmail.cf, or if it already exists as a non-link file, and takes appropriate actions to make /etc/sendmail.cf a link to mail/sendmail.cf. To do the same in Bourne shell might take a dozen lines.

This process is called a convergent process because executing the link statement multiple times does not have side-effects after the first execution (i.e., it is idempotent). Also, the link statement behaves appropriately regardless of the initial state of the system. Therefore, the state of the system *converges* to the ideal state described by the source.

Environmental Acquisition

The fundamental insight of this paper is that all network objects exist in *contexts*. The idea of context dependency is both powerful and pervasive. Every computer, every switch, every printer in a network exists in a context that affects its desired behavior. In other words, the physical and logical location of a network object determines properties of that object. Here are some concrete examples of network objects and their context dependencies:

- **Access Controls**. A host in a computing lab must have less restrictive access controls than a host in a data center. Here, the physical location of the host is part of its context.
- **IP Configuration**. The netmask and default route of a host depend on the subnet to which the host belongs. Here, the subnet is the context of the host.
- **Switch Configuration**. Individual ports on network switches often require their VLAN membership to be set by an administrator. In networks where each subnet uses a separate VLAN, a switch port may depend on its context, i.e., the subnet to which it belongs, to determine its VLAN membership.
- **Printer selection**. Each computer lab or office in a building may have its own printer. This printer then becomes part of the context of each host in the room. Put simply, the most sensible thing for a host to do by default is to use a printer that has been assigned to the room in which the host is located. This can be accomplished by looking for a default printer in the physical context of each host.

System administrators use this contextual information almost every time they configure a network element such as a host, switch, or printer. To our knowledge, no tool exists to model this contextual information and automatically generate configurations based on it.[1] Therefore, the goal of Anomaly is twofold. First, Anomaly should encourage administrators to think actively about the contextual relationships in their network. Second, Anomaly should be a tool for modeling these relationships explicitly, in order to simplify the maintenance of configurations.

One method of modeling contextual information is to treat contexts as containers and construct a set of containment relationships regarding network objects. Using this idea, one notices relationships such as "the machine chorf is *in* room 201," and "the machine ambler *belongs* to subnet 192.168.7.0/24." From here, it is easy to observe that objects acquire properties from containers.

The name for this process is *environmental acquisition* [8]. Environmental acquisition, or simply acquisition, is an analogue of inheritance that operates on object/container relationships, rather than class/sub-class relationships.[2] The following example, paraphrased from the original paper on acquisition [8], illustrates the idea perfectly.

[1]Cfengine does have a notion of context, in that it conditions its actions based on probes of the filesystem and operating system. However, it does not emphasize *modeling* contextual relationships, nor does it propose a single method of examining context, such as the construction of containment graphs.

[2]The Zope application server (http://www.zope.org/), the most prominent use of environmental acquisition, uses environmental acquisition to build and manage web content.

Consider a red car. If someone asks you about the color of the car's hood, you will certainly tell them that the hood is red, unless you know otherwise. In this situation, the hood has *acquired* its color from the car. However, it would be wrong to model this relationship using inheritance, because a hood is not a type of car. A hood is a part of a car, that is, car and hood have an object/container relationship across which properties are acquired.

Through the use of acquisition, administrators can build models in which hosts acquire their access restrictions from their physical location, and their default routes from the subnet they reside in.

Many previous solutions use inheritance-like mechanisms for data aggregation. Although inheritance has worked well in previous approaches, it is not the most natural or useful data aggregation mechanism available.

First, the purpose of inheritance is not to aggregate data. At best, it can be considered a feature (or behavior) aggregation technique. Moreover, inheritance is used to establish *is-a* relationships. In the approaches discussed above, this rule is bent slightly (to great benefit, of course). For example, Cfengine's class system uses boolean set operations (logical AND and OR) to decide to which machines a given action should be applied. The following statement says "link /etc/passwd-link to /etc/passwd on all machines that are in classes solaris and guest."

```
solaris.guest:: # logical AND
     /etc/passwd-link -> /etc/passwd
```

This approach is loosely based around the idea of inheritance. Objects are instances of specific classes, and the class which an object belongs to determines its behavior. However, inheritance is not an appropriate technique for data aggregation in system management. Acquisition is a better approach to data aggregation for the following two reasons.

First, acquisition encourages system administrators to think about containment and contextual relationships. This is an improvement over inheritance, which encourages thinking in terms of *is-a* relationships. While it may be true that the host chorf *is-a* Solaris host, chorf also has interesting relationships with its surroundings. It is *in* a room, and it is *part-of* a subnet, but neither of these relationships lends itself to inheritance.

Second, acquisition has finer granularity than inheritance. Inheritance demands that a child class inherit all the features of its parent. A child class may choose to override certain features that it inherits, but it may not decline them outright. This is necessary because inheritance imposes sub-type relationships. A child class must have all the behavior of its parent, or else the *is-a* relationship is nullified. Because acquisition does not impose sub-typing, an object may pick and choose which features to acquire from its container.[3] This prevents objects from acquiring unexpected properties.

There is at least one other twist on inheritance that bears mentioning, namely *value inheritance*. Value inheritance is a data aggregation technique used in prototype systems such as ARK [6]. It works by making copies of a prototype object, and then making tweaks to the values inherited from the prototype. Most of the examples presented in this paper could also be implemented using value inheritance. We believe, however, that environmental acquisition is the better choice for the reasons discussed above. It encourages thinking in terms of contextual relationships, and it provides better granularity than value inheritance. David Ungar's paper on SELF [10] discusses prototyping and value inheritance in depth.

Logic Programming

Anomaly does not have the power of a full logic programming language such as Prolog. Cfengine's logic programming abilities also exceed that of Anomaly's. In Anomaly, the ability to specify the targets of configuration statements with 'facts' is replaced by the approach of context modeling and environmental acquisition. However, Anomaly keeps what we believe is the most important contribution of Cfengine, namely, the idea of convergent processes specified by logical statements. Therefore, rather than using imperative statements such as "add this user" or "put this interface in promiscuous mode," Anomaly uses logical assertions such as "this user must exist," or "this interface must be in promiscuous mode."

Anomaly

Our language, Anomaly, combines important elements of prior approaches with environmental acquisition. Anomaly has constructs to model containment relationships (i.e., which objects are contained in which objects), and constructs for describing the ideal state of individual objects.

Examples

The following examples illustrate the use of Anomaly and highlight its strengths, by examining several typical system administration scenarios. Several different types of contextual relationships are presented, in order to demonstrate the benefits of explicitly modeling context.

Host Access Controls

Configuration of host access controls is the most basic and obvious use of acquisition in system management. A system administrator can make access controls more maintainable by exploiting a contextual relationship in the environment. In a university setting, it may be appropriate to use the physical location of a host as the context. In a corporate setting, the appropriate context may be the ownership of the machines (e.g., departmental, individual). In this example, we use the physical location as the context.

[3]Acquisition comes in two flavors, implicit and explicit. When using implicit acquisition, any attempt to access a variable that is missing from an object results in an attempt to acquire that variable. When using explicit acquisition, no attempt is made to acquire variables unless they are listed explicitly as candidates for acquisition. Anomaly uses explicit acquisition.

The first consideration is the containment graph. It is specified in Figure 1. Figure 2 defines some of these objects. The object CullinaneHall is defined as a Building, and is given a default access policy that allows *only* users in the systems netgroup to log in. This means that any host in the building acquires a restrictive access policy, unless otherwise specified.

```
CullinaneHall contains {
    UnixLab;
    DeansOffice;
}

UnixLab contains {
    chorf;
    wharf;
    staypuff;
}

DeansOffice contains {
    deans-laptop;
}

Building CullinaneHall {
    AccessPolicy access;

    access.allows(systems);
}

Room UnixLab {
    AccessPolicy access;

    access.allows(students);
}

SolarisHost chorf {
    AccessPolicy access('/etc/passwd')

    acquire access
}
```

Figure 1: Controlling access policies in object definitions.

For the UnixLab object, that is exactly what is done. Since the UnixLab object represents a public computer lab, it must have an access policy that allows users from the students netgroup to log in. Therefore, any host placed in the UnixLab object acquires a permissive access policy, specifically one that allows students to use the computers.

From this example, it is easy to see the benefits of using this method across an entire environment. When every room in a building has a sensible access policy assigned to it, administrators hardly have to worry about individual hosts. This directly addresses the recurring problem of hosts moving from faculty desks to public labs (Or from similar restricted access locations to similar public access locations). If no one remembers to edit /etc/passwd, the result is that students cannot log into the machine.

The most important benefit of this scheme is that even if a forgetful administrator moves a machine from one place to another without consideration for its access policy, the machine acquires a sensible access policy from its new environment. Modeling the context of the machine has thus made the path of least work more likely to be the *correct* path. In short, the lazy way produces the best defaults.

```
SwitchPort zaphod-8-13 {
    SwitchNum switchnum(switch =
            'zaphod.ccs.neu.edu');

    switchnum.is('8/13');

    acquire vlan;
    acquire ether;
}

SolarisHost chorf {

    MAC ether;

    ether.is('01:1C:ED:C0:FF:EE');
}

Subnet UnixSubnet {

    NetworkAddr net;
    Netmask mask;
    VLAN vlan;

    net.is('10.10.116');
    mask.is('255.255.254.0');
    vlan.isnamed('116');
}

UnixSubnet contains {

    chorf;
}

chorf contains {
    zaphod-8-13;
}
```

Figure 2: Modeling a switch Port.

Switch Port Configuration

Switch port configuration often involves configuring VLAN membership and port security settings. The contextual relationship is not quite as obvious as in the previous example. Here, it is necessary to treat the switch port as an object contained in the host that is attached to it. From a physical standpoint, this seems inappropriate. However, from a logical standpoint it is easier to think of the host being part of the port's context. When using MAC-based port security, it is necessary to know which host the switch port is supposed to serve.

Figure 2 shows the containment and object declarations for chorf and its switch port. When the configuration in Figure 2 is built, the switch port acquires two fields: vlan and ether. The vlan field allows the switch port to set its VLAN membership properly, and the ether field allows it to set its port security settings properly. After the configuration is deployed, the switch will be usable only by chorf.

This figure also raises a question about the difference between the 'is' assertion and the 'isnamed' assertion. The former is used when an assertion describes the state of the variable. In this example, the ether field is completely specified by a 48 bit address, that is, the ether field *is* its address. The latter assertion, isnamed, is used when referring only to the identifier of a field. The vlan field in this example is only concerned with the identifier of the VLAN, not with configuration of that VLAN on the switch.

Beta-Test Environments

When upgrading subsystems such as daemons, kernels, or user applications, it is best to test the new software on a small subset of machines. It is possible to test upgrades on a single machine near the administrators' offices, but a single machine may not be representative of the rest of the environment (especially if host hardware is substantially varied throughout the environment). Furthermore, a designated test machine is not likely to have the same usage patterns as most machines in the environment. Anomaly can assist with this problem by providing a better method for organizing beta test systems, using environmental acquisition.

```
Platform Solaris {
    Packages packages;
    Patches patches;

    patches.has('sun-recommended');
    packages.has('openssh-3.0.2p1');
    packages.has('lprng-3.6.14');
}

Platform Beta {
    Packages packages;
    Patches  patches;

    patches.has('sun-recommended');
    patches.has('108604-18');
    packages.has('openssh-3.1p1');
}
```

Figure 3: Platform objects.

The containment relationships described in Figures 3 and 4 simplify beta testing. In order to test a new Solaris patch (108604-18), we add it to the Beta container. This causes the patch to be installed on all machines contained in Beta. When the patch is verified to work properly on all Beta machines, the patch can be moved from the Beta container up to the Solaris container, which causes it to be installed on all machines.

This approach offers several advantages:

- Because Beta is contained in Solaris, all other fields in the Solaris container are acquired by the machines in Beta. Therefore, the configurations of these machines will stay as close as

possible to the standard configuration being used by the rest of the environment.
- Because all hosts for testing the new configuration are aggregated into one container, it is easy to keep track of which machines are being used as test cases.

```
Solaris contains {
    north-star;
    dog-star;
    # and many, many others

    Beta;
}

Beta contains {
    # machines of various
    # hardware configurations
    chorf;
    emerald-city;
}
```

Figure 4: Beta test containment.

```
EthernetInterface Interface {

    acquire mode;    # declared in network
    acquire ethaddr; # declared in hosts
}

SwitchPort switchport {
    SwitchNum switchnum(switch =
          'zaphod.ccs.neu.edu');

    switchnum.is('8/13');

    acquire vlan;
    acquire ether;
    acquire smode;
}

Network CCSNetwork {
    IfMode mode;
    SwitchPortMode smode;
    mode.is('normal');
    smode.is('normal');

}
```

Figure 5: Interface declaration.

- The beta test containment relationship is orthogonal to other containment relationships such as physical location, network (logical) location, and ownership. Therefore, the inclusion of a machine into the beta test container does not affect the usage patterns of the machine.
- When no upgrades are being tested, the machines in the beta container acquire exactly the same settings that machines outside the beta container acquire. Therefore, the addition of the beta container does not affect the environment in any way when it is not being used. In other words, the container becomes completely transparent when nothing is being tested.

While it might seem that the 'has' assertion in this example must embody all of the functionality of Cfengine, it is not nearly that complex. It uses a directory that contains all the Solaris patches currently in use in the environment. In it there is a subdirectory named '2.8_recommended.' Other patches are listed individually. The has assertion simply checks if a patch is installed, and if it isn't, runs the patch's install script. The packages field works similarly. This method, of course, cannot handle all of the boundary cases handled by Cfengine.

Reparenting

Modeling context through containment is the central theme of Anomaly. The following example demonstrates how changes in context and containment result in appropriate changes in the network, with a minimum of reconfiguration.

Consider the containment graph in Figure 6. The object declarations for individual hosts and unmanaged containers are not shown, but are similar to the declarations used in Figures 8 and 1. The one unfamiliar object in this graph is Interface, which represents Ethernet interfaces on Unix hosts. Its declaration is shown in Figure 5; it is placed into its containers with copy containment. Figure 5 also shows the declaration of CCSNetwork, which specifies a mode (promiscuous or normal) which is acquired by all interfaces.

Suppose that the administrator of this network wishes to start running host-based IDS software. This can be accomplished without modifying the declarations of individual hosts or interfaces at all. The administrator simply adds an IDS container and reparents the appropriate objects as shown in Figure 7. Now, all IDS-related settings are aggregated into a single container. To make a machine an IDS machine, it needs only to be added to the IDS container. By acquiring settings from the IDS container, the machine receives the IDS packages, the machine's interface becomes promiscuous, and the switch port attached to the interface is put into spanning mode.

By modeling context through containment, we are able to aggregate control of three network components (host software, host interface, and switch port) into a single point of control. Examples such as this one make changes to configurations atomic (i.e., the desired change can be effected by moving one machine into one container), and thus more maintainable.

The Language

Anomaly is a simple object-oriented declarative language. Its basic components are:

- **Objects**. Objects in Anomaly are used to model components of a network. They can be divided into two categories:
  - *Managed Objects.* Managed objects represent elements of the network that are directly managed by system administrators, e.g., computers, switches, and printers.
  - *Unmanaged Objects.* Unmanaged objects are components of the network that are not

directly managed as part of the network, e.g., buildings, rooms, and research groups.

A simple Anomaly object appears in Figure 8.

```
# unix platform
Unix contains {
    Linux;
    Solaris;
    OpenBSD;
}
Linux contains {
    darkside;
}
OpenBSD contains {
    runningboar;
}
Solaris contains {
    chorf;
    ambler;
    north-star;
}
# a building
Cullinane contains {

    UnixLab;
    MrRoom;  # machine room
}
CCSNetwork contains {
    Subnet115;
    Subnet116;
    Subnet118;
}
Subnet115 contains {
    runningboar; # a host
}
Subnet116 contains {
    chorf;
    ambler;
    north-star;
}
Subnet118 contains {
    darkside;
}
ContainmentTemplate (
    QUERY = "SELECT hostname, \
           switchport FROM hosts;"
    NAME = hostname;
) contains {
    copy Interface contains {
        QUERY.switchport;
    }
}
```

Figure 6: Containment relationships in a small network.

- **Fields**. Fields store the configuration data for objects. In Figure 8, fields include hostname, ip, and accesspolicy.
- **Parameters**. Fields may be parameterized. In Figure 8, the field accesspolicy specifies that /etc/passwd is the location of the file it generates. The purpose of parameters is to tell Anomaly *where* data must go. Another example of a parameter is the name of a network interface object, e.g., hme0 or eth1.
- **Assertions**. In Figure 8, ip = '129.10.117.177'; is an assertion about the value of the field ip. It says: "The value of ip *is* 129.10.117.177." Assertions are implemented by the fields that use them, not by the core of Anomaly. Therefore, two different types of fields (e.g., AccessPolicy, Packages) may have completely different implementations of the 'has' assertion.

```
Platform IDS {
    IfMode mode;
    SwitchPortMode smode;

    packages.has('snort');
    packages.has('acid');
    mode.is('promiscuous');
    smode.is('spanning');

    acquire packages;
}
CCSNetwork contains {
    IDS;
}
OpenBSD contains {
    IDS;
}
IDS contains {
    runnnigbear;
}
```

Figure 7: IDS declarations.

```
SolarisHost chorf {
    Fqdn hostname;
    IPaddr ip;
    Access accesspolicy('/etc/passwd')
    ip = '129.10.117.177';
    acquire resolv;
    acquire accesspolicy;
    acquire mounts;
}
```

Figure 8: Example object.

- **Acquisitions**. In Anomaly, the keyword acquire signifies the acquisition of a field. Acquisitions tell Anomaly to search for the value of that field in the containers of an object. It is important to note that accesspolicy is an acquired field, yet it has a parameterized declaration. In this example, accesspolicy acquires its value from its containers, but retains the parameters specified by its object. The other acquired fields in this example result in the acquisition of the value *and* the parameters from the containers, since no declaration is specified in the object.
- **Containment declarations.** A simple containment declaration appears in Figure 9. The three

blocks in the example declare that chorf and ambler are *part of* Subnet116, chorf is *in* Room201, and ambler is *in* Room129.

- **Database templates.** While language-based configuration approaches have many merits, storing configurations for hundreds of machines in source code form can be unwieldy. Database templates are provided in order to integrate Anomaly with configuration database back-ends, by allowing for the creation of arbitrarily many objects in only a few lines of code.

```
Subnet116 contains {
    chorf;
    ambler;
}
Room201 contains {
    chorf;
}
Room129 contains {
    ambler;
}
```

Figure 9: Example containers.

Semantics of Containment

Objects in Anomaly may have multiple, non-nested containers. In fact, most do. For example, a host usually belongs (at minimum) to both a room and a subnet at the same time. Yet, the subnet is not in the room, and the room is not in the subnet.

During the design process, we considered requiring that the containment graph take the form of a forest whose trees meet only at the leaves. This decision turned out to be unduly restrictive. Therefore, the only requirement Anomaly places on the containment graph is that it be directed and acyclic, that is, objects may not contain themselves, directly or indirectly. Users of Anomaly can construct complex containment graphs using this rule.

When attempting to model certain containment relationships in this way, there are situations in which one would like to use the same object in many locations. For example, if network interfaces are represented by objects, they will most likely be identical across a wide group of machines. However, we cannot simply declare one interface object, and place it in each host. Doing so would create a containment graph like the one depicted in the left half of Figure 10, where the objects in the top layer all contain the same interface object. When the interface object attempts to acquire its netmask, it finds a netmask field in each of its parents, and compilation fails.

To remedy this situation, Anomaly offers *copy containment*. Copy containment reduces the number of duplicate object declarations by allowing a single declaration to be used in any number of places. When a containment relationship is declared as copy containment, the contained object is cloned, and the copy is placed into the container. This results in the containment graph in the right half of Figure 10, where the gray objects are all clones of an original object. This approach was inspired by *mixins*, an alternative approach to multiple inheritance [9].

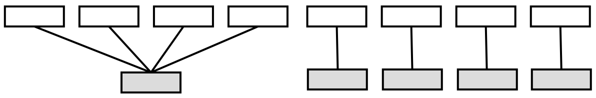

Figure 10: A simple containment graph with and without copy containment.

Semantics of Acquisition

Anomaly uses references to represent containment links between containee and container. When an object explicitly acquires a field, using the acquire statement, Anomaly searches for the field in the current object. If the field is not found, the containers of the object are searched recursively, until the field is found.

Because objects may have multiple containers, Anomaly must account for acquisition from multiple containers. If the acquisition search reveals that a variable can acquire its value from two different containers, Anomaly reports an error. This condition is called an acquisition conflict, because two equally valid contexts have been found.

As Anomaly attempts to resolve an acquisition, it may find that an object's container has attempted to acquire the same field. When this scenario occurs, the acquisition in the container is resolved first, and then the acquisition in the containee is resolved, using the value that has now been inserted into the container.

While this intermediate step looks superfluous at first glance, it is necessary to prevent ambiguities about the origin of a field's parameters. Recall that a field may specify its parameters, but acquire its value from the container. Consider three objects that are contained one within the other, like Russian dolls. Suppose that the outer object declares a field called accesspolicy, and that the inner two objects acquire that field. In the case where the outer object and the middle object specify different parameters (e.g., /etc/passwd and /etc/shadow), the semantics described in the above paragraph force the innermost object to acquire its parameters from the middle object rather than the outer object. In other words, this rule resolves any ambiguity about where a field's parameters are acquired from.

The Dependency Graph

In large configurations, it is both time consuming and inconvenient to re-deploy every configuration on the network when only a few objects have been modified. Therefore, Anomaly uses a simple strategy to deploy configurations only to those objects that may have changed since the last update. This strategy exploits the structure of the containment graph.

The containment graph, in addition to modeling contextual relationships, also models dependency relationships. Using acquisition, a change to the configuration of one object can affect only that object, and any objects contained within. It is not possible for changes in an object to have any effect on its containers. To exploit this invariant, Anomaly keeps track of which objects have been altered since the last successful attempt to deploy configurations. When another attempt is made to deploy the configurations, Anomaly discards all objects that have not been modified, and are not contained within objects that have been modified.[4]

Figure 11 illustrates a modification to the access policy of a computing lab. The left half shows the network before the change is made; the right half shows the network after the change is made. Only objects in the shaded area require re-deployments of their configurations. The shaded area is computed by a simple mark and sweep algorithm.

Checking Types

Anomaly enforces some type constraints during compilation. In particular, it checks assertions for type correctness in the obvious manner. For example, if we declare that some variable represents an IP address and make an assertion about the variable, then Anomaly ensures that the assertion associates a well-formed IP number with the variable. Anomaly does not, however, attempt to enforce constraints at run-time. That is, for all those actions for which it cannot check type constraints during compilation, the configuration transport mechanisms must enforce the coherence of the data access operations (e.g., file access, SNMP set commands). In practice, this means that if a type constraint is violated while Anomaly is generating configurations (i.e., after the source code has been processed), the configuration will either fail or produce incorrect results. Making Anomaly truly type safe – indeed, exploring what type safety precisely means in this context – is future research.

[4]It is also possible to construct an acquisition graph from the containment graph, in which each edge represents the acquisition of some value. Using this graph, the minimal set of modified machines can be computed. Currently, Anomaly does not compute this minimal modified set.

Templates

Templates allow Anomaly to instantiate many objects at once, using data from a configuration database. Figure 12 shows an example template.

```
Template (
 QUERY = "SELECT hostname FROM hosts \
        WHERE os == 'solaris'";
 NAME  = hostname;
) {
  Fqdn hostname;
  hostname = QUERY.hostname;
  acquire access;
}
```

Figure 12: Example template.

Templates consist of two parts. The first part is the query declaration, which appears between the parenthesis at the top of the template declaration. It specifies a query to be issued to the database, and specifies what the identifier (NAME) of each new object will be. In this example, one object is instantiated for each row returned by the specified SQL query, and the object is given the name contained in the 'hostname' column of that row.

The second part of the template declaration is the object declaration. It follows all the same rules as a regular object declaration, except that any column of the query result can be referenced by the keyword 'QUERY,' as seen on line 8 of Figure 12.

There is also a second type of template, called a containment template, which is used to declare containment relationships. An example of this appears at the end of Figure 12. It follows the same rules as a normal containment declaration, except that it can reference columns in the query result, just like the template above.

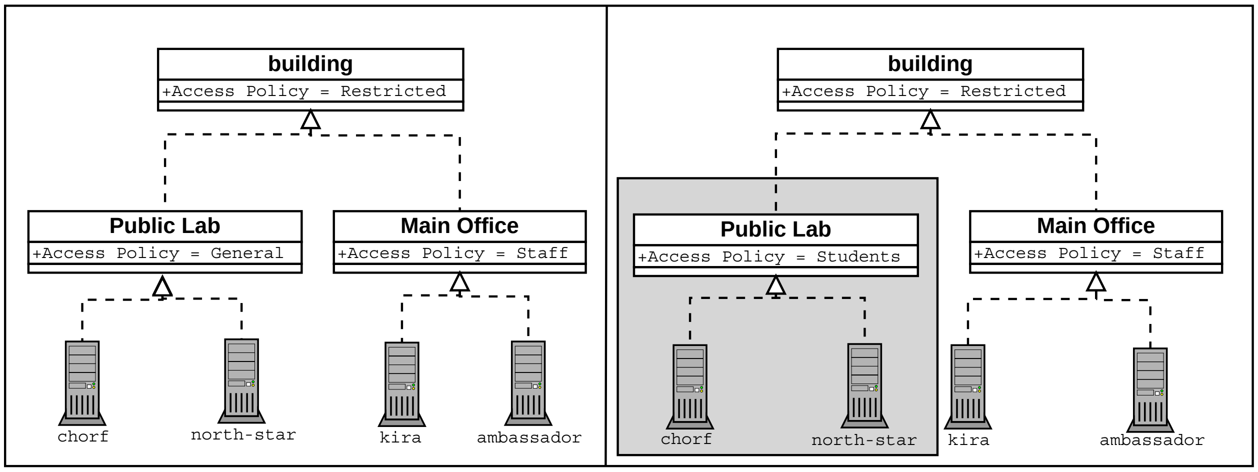

Figure 11: Calculating dependencies after changing access policy for a lab.

Practical System Administration with Anomaly

To see how Anomaly can fit into normal administrative practices, we list several sub-categories of system management, describe them briefly, and show how Anomaly can fit into each area.

Host management is the area of system management embodied by Cfengine. Most of the examples in this paper have focused on this area. Host management tasks lend themselves well to a context-based approach, so Anomaly is best suited to this area of system management. For Anomaly to succeed, however, it must be able to conform itself to existing environments, rather than expecting environments to be built around its notions of containment and context. Fortunately, typical computing environments are ripe with contextual relationships that can be exploited by Anomaly. Therefore, we believe that Anomaly has the potential to be an appropriate addition to existing environments, rather than a foundation for environments that are being rebuilt from the ground up.

Service management includes tasks such as the generation of DNS zone files and DHCP configurations, to name a few. Database driven approaches lend themselves well to these tasks, because the configuration files being produced are often nothing more than a listing of the contents of the database in an obscure format. Anomaly is not particularly well suited to this area of system management, because it often requires data about every object in the environment to be brought together in a single location. A simple query to a configuration database is the appropriate tool for this job. To do the same with an acquisition-based tool would be awkward and inefficient.

User management falls partially under the previous category, because it may involve services such as NIS, LDAP, and Kerberos, but deserves its own category because it also involves resource allocation, specifically home directories and mail spools. User accounts are laden with contextual information, such as the account owner's position within the organization. Anomaly could be used with an existing user management system as a means to model these contextual relationships. Doing so could simplify tasks such as disk quota assignment and account expiration.

Software (or package) management is the area embodied by systems such as Depot, Stow, and RPM. If software packages are installed locally on individual hosts, this category is partially subsumed by host management. However, software is often installed on globally accessible filesystems (e.g., NFS), so software management must be considered separately.

Anomaly is not intended to be a software management system by itself, but it can interface with existing software management tools. For example, an RPM extension to Anomaly would need only to wrap logical assertions around RPM's query based interface. The `has` operator, in this case, would issue a query to see if a given package was installed on a target machine, and install it if necessary. Here we see that Anomaly can coexist with other systems. There would be no reason to stop using the existing system and rely only on Anomaly. Anomaly would simply be used to model and enforce the contextual relationships affecting software installation (e.g., if a machine is one of the mail servers for an environment, it must have an MTA installed).

Another practical consideration is that of Anomaly's configuration transport mechanism. In practice, Anomaly uses three types of configuration mechanisms: configuration files (e.g., `/etc/passwd` and `/etc/resolv.conf`), configuration scripts (e.g., a script that executes `link` statements), and SNMP set commands. The configuration files are transported to the appropriate machines using ssh and a small helper script that writes the file contents to the appropriate locations. Configuration scripts are copied to a temporary directory and executed on the target machine. SNMP set commands are executed as Anomaly interprets the code. In this sense, Anomaly is primarily a ''push'' tool, that is, configurations represented in Anomaly are generated on a single host, and then distributed to the managed objects.

A Caveat

Acquisition is not a panacea for all system administration tasks. In particular, an acquisition-based approach does not provide any assistance with unique information in a network. For example, when a host is moved from one subnet to another, its IP address must change,[5] in addition to its netmask and default route. A natural suggestion is that hosts acquire their IP addresses from their subnet containers, which are responsible for ensuring the uniqueness of each address.

Anomaly's implementation of acquisition cannot facilitate this design, and acquisition in general does not lend itself to this approach. This is true for a number of reasons, the most important one being that to implement such a system, containers would have to retain state about the values that had been handed out to sub-objects. Storing this state would make it impossible to cull unmodified objects by constructing a dependency graph.

We do not believe, however, that the inability to manage unique information constitutes a genuine weakness. The goal of Anomaly is to make it easier to manage information that is common to a group of objects. System administrators are already good at managing unique information, but they do need support to keep common information consistent.

Some First Experiences

We have used Anomaly to manage a lab of Unix workstations, and the switch ports to which they are

[5]Changing IP addresses via a remote configuration management system is problematic for other reasons. Nonetheless, this example illustrates a range of problems in which unique information must be managed.

connected. The workstations run Solaris 2.8, and the switch is a Cisco Catalyst 5500. Using this experimental setup, we were able to construct real containment graphs that effectively modeled our test network. By moving machines to different positions in the containment graph, i.e., by changing their context, we were able to verify how quickly an Anomaly-administered network can be reconfigured when the components of the network change.

The experiments also revealed some weaknesses in Anomaly's configuration transport mechanisms. First, because Anomaly is intended to work with a wide variety of hardware (not necessarily UNIX machines), restricting all objects to an rsh transport mechanism is not feasible. For Anomaly to be extended to other operating systems, we need to design a universal system for configuration transport. Second, to overcome the difficulties of performing changes to networking configurations (e.g., IP addresses, netmasks), Anomaly should produce floppy disks (or other removable media) to transport basic configuration data. We hope to address these problems with future research.

Project Status and Source Code Availability

As of this writing, Anomaly is not ready for widespread use. Its most pressing need is for the development of a large and portable suite of administrative modules. Currently, Anomaly has modules for managing Solaris hosts in an NIS/NFS environment, and a module for managing certain aspects of Cisco Catalyst series switches (specifically, VLAN membership and port security).

Some of the source code listings in this paper show components of Anomaly that are not stable as of this writing for illustrative purposes. Specifically, database templates, the 'packages' field, and the 'patches' field are only partially implemented.

The latest source code, along with current information about Anomaly, is available at http://www.ccs.neu.edu/home/mlogan/anomaly/.

Author Information

Mark Logan is a recent graduate (BS) of Northeastern University's College of Computer Science. The work presented in this paper is his senior thesis. While studying for his degree, he spent approximately three years working for CCS in various capacities, primarily as a system administrator. He can be reached electronically at mlogan@ccs.neu.edu.

Matthias Felleisen received his PhD in 1987 from D. P. Friedman (Indiana University), spent 14 years at Rice University as a professor, and joined Northeastern University as a Trustee Professor of Computer Science last year. He is interested in all aspects of program design and programming languages.

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer Science and the author of Perl for System Administration (O'Reilly). He has spent the last 15 years as a system/network administrator in large multi-platform environments and has served as Senior Technical Editor for the Perl Journal. He has also written many magazine articles on world music.

References

[1] Evard, R., "An Analysis of UNIX System Configuration," *Proceedings of the Eleventh Systems Administration Conference (LISA XI)*, USENIX Association, Berkeley, CA, p. 179, 1997.

[2] Anderson, P., "Towards a High-Level Machine Configuration System," *Proceedings of the Eighth Systems Administration Conference (LISA VIII),* USENIX Association, Berkeley CA, p. 19, 1994.

[3] Finke, J., "An Improved Approach for Generating Configuration Files from a Database," *Proceedings of the 14th Systems Administration Conference (LISA XIII)*, USENIX Association, Berkeley CA, p. 29, 2000.

[4] Burgess, M., "A Site Configuration Engine," *Computing Systems*, Vol. 8, No. 1, MIT Press, Cambridge MA, p. 309, Winter 1995.

[5] Couch, A. and M. Gilfix, "It's Elementary, Dear Watson: Applying Logic Programming to Convergent Systems Management Processes," *Proceedings of the 15th Systems Administration Conference (LISA XV)*, USENIX Association, Berkeley CA, p. 123, 2001.

[6] Holgate, M. and W. Partain, "The Arusha Project: A Framework for Collaborative Unix System Administration," *Proceedings of the 15th Systems Administration Conference (LISA XV)*, USENIX Association, Berkeley CA, pp. 187-198, 2001,

[7] Hagemark, B. and K. Zadeck, "Site: A Language and System for Configuring Many Computers as One Computer Site," *Proceedings of the Workshop on Large Installation Systems Administration III*, USENIX Association, Berkeley CA, p. 1, 1989.

[8] Gil, J. and D. Lorenz, "Environmental Abstraction – A New Inheritance-Like Abstraction Mechanism," *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 214-231, October 1996.

[9] Flatt, M., S. Krishnamurthi, and M. Felleisen, "Classes and Mixins," *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 171-183, January 1998.

[10] Ungar, D. and R. Smith, "Self: The Power of Simplicity," *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pp. 227-242, December 1987.