PoDIM: A Language for High-Level Configuration Management

Thomas Delaet and Wouter Joosen – Katholieke Universiteit Leuven, Belgium

ABSTRACT

The high rate of requirement changes make system administration a complex task. This complexity is further influenced by the increasing scale, unpredictable behaviour of software and diversity in terms of hardware and software. In order to deal with this complexity, configuration management solutions have been proposed. The processes that many configuration management solutions advocate are kept close to manual system administration. This approach has failed to address the complexity of system administration in the real world. In this paper, we propose PoDIM: a high-level language for configuration management. In contrast to many existing configuration management solutions, PoDIM allows modeling of cross machine constraints. We provide an overview of the PoDIM notation, describe a case study and present a prototype. We believe that high-level languages are needed to reduce system administration complexity. PoDIM is one step in that direction.

Introduction

The fact is that configuration errors are the biggest contributors to service failures (between 40% and 51%). Configuration errors also take the longest time to repair [37, 36, 34]. As the complexity of computer infrastructures increases, the risk of configuration errors increases likewise and introduces even higher change costs. Changes to a configuration can be technically – such as software upgrades – or business oriented. A difficulty with configuration changes is the high number of dependencies between systems. Systems do not operate in isolation, but in a network. A change in the configuration of one networked service may cause a complex chain of changes in dependent services. Furthermore, infrastructural complexity is influenced by increasing scale, unpredictability in software behaviour and systems variety [21, 39, 4].

1. **scale**: The number of network devices, servers, desktops and laptops in a typical infrastructure is increasing significantly. New kinds of devices such as PDA's, mobile phones and sensor nodes are extending the scope of an organizational computer infrastructure.
2. **unpredictability in software behaviour**: Increasingly complex software systems tend to have more bugs, viruses and vulnerabilities. Bugs in software, viruses and vulnerabilities make full control over the system's behaviour an illusion [13].
3. **systems variety**: Computer infrastructures have a large variety in terms of hardware platforms, operating systems and application software. Our definition of infrastructures includes not only desktops, servers and laptops, but also embedded devices such as palmtops, mobile phones and network devices such as routers and switches. All of these devices run on a variety of operating systems and accompanying application software.

Using a network shell or a configuration management language whose process is close to manual system administration simply does not work in large and varied computer infrastructures with complex software systems. Indeed, the subtle interactions between (different versions of) software packages can make systems with the same operating system and hardware platform unique. According to [5], the cost per unit becomes excessively large when using manual management processes. More loose, higher-level, processes are necessary.

PoDIM abstracts from systems variety and allows, more than existing configuration management languages, expressing an administrator's intentions. Expressing intentions is clearly of a higher-level nature than expressing, for example, what lines in the sendmail.cf file need to be modified on a mail relay. The key concept of PoDIM's high-level language is that it allows modeling of cross machine constraints.

The remainder of this article is structured as follows. First, we introduce PoDIM as a high-level language for configuration management. Next, We elaborate on PoDIM in the sections on Configuration Descriptions and Rules. The run-time semantics of PoDIM are introduced in the Prototype Section. We also present a case study. We end with sections on related and future work.

Language Overview

The state of the art in configuration management allows assigning roles to machines and setting high-level parameters for those roles. "Configure machine X to be a web server" and "configure machine Y to be DHCP server" are examples of role assignments. "The web server must run on port 80" is an example of a parameter assignment. The PoDIM language aims for a higher level of abstraction. Instead of role assignments, we want to express things such as: "One of my

servers must be configured as a web server" and "On every subnet, there must be two DHCP servers." Instead of parameter assignments, we want to express things such as: "A web server must use a port number higher than 1024."

PoDIM's language consists of a rule language and a domain model. The distinction between domain model and rule language is a recurring theme in policy languages. A domain model provides a description of the domain in which a rule language solves problems. Since we are dealing with the domain of configuration management, the domain model contains descriptions for things such as DHCP servers and web servers. The rule language defines types of rules and how they interact with the domain model. A system administrator writes rules in PoDIM's rule language. These rules interact with the domain model and output a configuration for each managed device. The next section elaborates on the domain model. Subsequently, we describe PoDIM's rule language.

The basic principle of PoDIM's runtime is that each real world object is simulated in the system. The different classes of objects are defined in the domain model. Examples of object classes are devices, network interfaces and services such as DHCP servers and web servers. PoDIM's runtime takes a set of policy rules as input and tries to satisfy these rules by creating objects and setting parameters of objects. In doing so, it generates a configuration for each managed device. Existing tools, such as Cfengine [12, 10, 11, 14], can then be used to deploy the generated configuration on each real world device.

Configuration Descriptions

PoDIM's domain model is object-oriented. This means that the "things" in the domain model are coded as classes. Examples of classes are DHCP server and web server. We use an existing object-oriented programming language for coding classes, named Eiffel [30, 31].

Figure 1 shows a simplified graphical representation of the domain model in the BON notation [40]. All classes, such as DHCP server and web server, have one common ancestor: ENTITY. The ENTITY class is used for modeling common functionalities. The arrows denote inheritance relationships. The simplified example presented in Figure 1 uses only single inheritance relationships. In real life examples, multiple inheritance is often necessary. Eiffel supports this.

Classes define the interface of a software subsystem in PoDIM. A class has attributes that can be set, queries that can be executed and commands that can be executed. For example, the WEB_SERVER class has an attribute for setting its port, a query to find out the administrative mail address and a command to enable php support on the server. Attributes are set by the system administrator when writing rules. Queries are used by the system administrator and other objects to gather information about the runtime system. Commands are used as an inter-object communication mechanism. An example of the latter occurs when a webmail object commands a web server to enable php support.

In the rest of this section, we elaborate on the definition of classes. We start with attributes and queries. Attributes are an object's data structures. Queries define the questions one can ask an object. Next, we discuss commands. Commands define how objects can change each other's state. We end this section with a description on how dependencies are modeled between classes.

Attributes and Queries

Attributes define the data structures for objects of a class. Queries are methods which return a result. In Eiffel, all attributes are also queries by definition, i.e., objects can query each other's attributes. An object can only modify another object's attributes by using commands. The result of a query is computed based on the results of other queries or the values of attributes. The example in Listing 1 shows a partial web server class. It defines two attributes: "php_supported" and "domain"

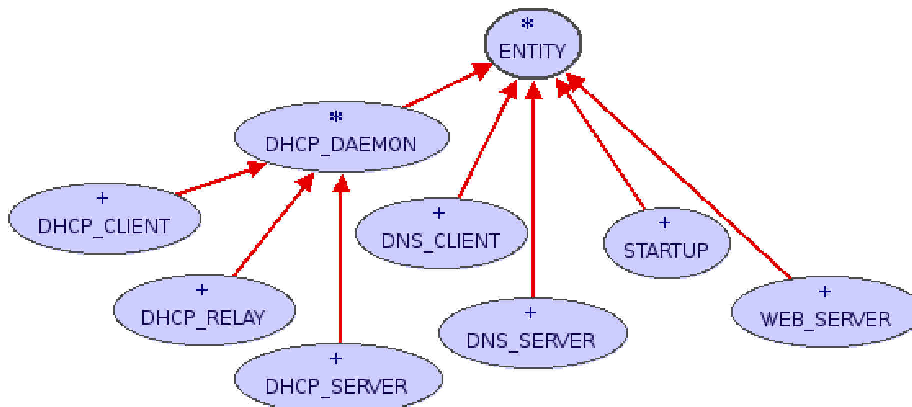

Figure 1: Domain model in BON [40] notation. All units of functionality such as mail servers and DNS clients are modeled as classes. All classes have one common ancestor: ENTITY. The arrows denote inheritance relationships.

and one query: "administrator_email". In the example, the "administrator_email" query is based on the attribute "domain". All attributes and queries have a type. For example, the "php_supported" attribute has type BOOLEAN. Note that the definition of WEB_SERVER is used as an illustration, not as an introduction to a real world WEB_SERVER class.

Commands

Commands are methods that change the state of an object (i.e., modify its attributes), but do not return a result. The example in Listing 1 contains a partial web server class with one command: "enable_php". The "enable_php" command changes the value of the "php_supported" attribute. Since a command can contain arbitrary code, its behaviour should be clearly documented. For this documentation, we use another feature of Eiffel: preconditions and postconditions. Preconditions express conditions that need to be true before the command is executed while postconditions express the effects of the command's execution. In our web server example, the precondition of the "enable_php" command is that php support is not yet enabled. Its postcondition expresses that php support will be enabled when the command is executed.

It is also possible to control access to commands, i.e., prohibit objects to execute commands on other objects. In the web server example, we could only allow objects that run on the same device to enable php support. In the Section on authorizing commands, we elaborate on how to specify access controls for objects.

Modeling Dependencies

A lot of dependencies exist between classes (and their real world software configurations). Imagine a startup class that is responsible for generating the /etc/init.d directory on Linux systems.[1] Both the web server and DHCP server classes depend on the startup system. Indeed, if these two network services have no hook into the startup system, they are not activated when we reboot a machine. Another example of a dependency is the relationship between the implementation of a service and its attributes. Imagine a web server class that supports two web server implementations: apache and publicfile. Apache supports php, publicfile does not. In this case, php support can never be enabled on a web server object if it uses publicfile as its implementation.

To make these kinds of dependencies explicit in our domain model, we use two language constructs of Eiffel: references and invariants. When declaring an attribute in a class, it will contain a reference to another object and not to the contents of the actual object. For example, the dependency of the web server on the startup system is modeled as a reference in Listing 2 on line 9. Invariants are arbitrary boolean expressions that are required to be true at all times during an object's lifetime. They provide a built in mechanism for modeling fine grained dependencies (and other restrictions on an object's attributes state). For example, the relationship between php support and the chosen implementation is modeled in Listing 2 on line 12.

Making dependencies that exist in an IT infrastructure explicit in the domain model has two advantages. First, they can be used as a documentation aid. Second, a dependency violation can be detected by the PoDIM runtime. For example, when one rule states that the attribute "implementation" of a web server class must be set to "publicfile" and another rule

[1] Classes can be used to abstract away from details like the operating system used and different software versions. For example, the same startup object results in other files being generated on Linux and BSD systems.

```
01 class WEB_SERVER
03 feature -- Attributes
05     php_supported: BOOLEAN
07     domain: STRING
09 feature -- Queries
11     administrator_email: STRING is
12         do
13              Result := "webmaster@" + domain
14         end
16 feature -- Commands
18     enable_php is
19         require
20             php_supported = False
21         do
22             php_supported := True
23         ensure
24             php_supported = True
25         end
27 end
```

Listing 1: This partial WEB_SERVER class defines two attributes: "php_supported" and "domain", one query: "administrator_email" and one command: "enable_php".

states that the attribute ''php_supported'' must be set to true, a dependency violation is detected. The default behaviour is to signal an error and abort.

Rules

The rule language is the user interface for the system administrator. It defines rules for expressing how the configuration of an infrastructure must look like. Remember that each real world object is simulated with PoDIM. For example, there exists an object for each device in your system, each network interface and every service that needs to be configured. The domain model presented in the previous section is a static description of the possible classes that can exist in the system. The rule language is used to create and manipulate objects.

A distinguishing feature of our rule language is that it allows the specification of constraints. We demonstrate the need for constraints with two examples.[2]

- When configuring a web server, the port is one of the attributes that can be set. A constraint allows expressing things such as ''the port should be set to 80 or a value higher than 1024.'' In contrast, a regular assignment only allows expressing things such as ''the port should be set to the value 80.''
- Servers typically have roles assigned which determine the services they must offer. For example, one can state that system X is going to be a web and mail server. By using constraints we can express things such as: ''A device should not provide more than four network services,'' ''I want two DHCP servers on each subnet,'' or ''One of my servers should configure itself as a web server.''

Since the domain model is object-oriented, the rule language contains rules to create objects and modify the attributes of objects. Remember that a class also defines commands for its objects. Commands allow objects to change each other's behaviour. The rule language also allows access controls between objects to be defined. In the rest of this section, we elaborate on the three types of rules: creating objects, modifying objects and authorizing commands.

[2]Since this work is about configuration management, we use examples from this domain. Nevertheless, the rule language is generic enough to apply to other domains.

Creating Objects

As a system administrator, you configure the network to offer services. This results in assigning a set of roles to each device in the network. For example, machine A acts as a web server and DHCP server. Machine B acts as a DNS server. All machines act as IPv4 nodes or routers. PoDIM's creation rules express role assignments precisely. Since every real world object is simulated in the PoDIM runtime, rules must exist for all real world objects to be created. In general, a creation rule instructs a set of objects to create other objects. For example, we instruct machine A to create a web server and a DHCP server. How do we know that a simulated object of machine A exists in the system? We can not assume this, so we have to create it with a creation rule. But then again, which object needs to create machine A? To get out this bootstrapping problem, we assume the presence of one object of a predefined class called SYSTEM_ENTITY.

Listing 3 shows a creation rule to create machine A. The rule contains three parts: The first part on line 1 specifies the rule type. In this case, we want to write a creation rule. The rule type can be extended with an optional rule identifier, in this case ''machine_A''. The second part states which object needs to be created. In this case, we want to create a DEVICE object (line 2). We also specify one initial attribute for the device object that is going to be created on line 3. The attribute ''name'' will be set to ''machine_A''. The third and last part on line 4, after the ''select'' keyword, specifies which object(s) need to execute this rule. In this case, we want all objects of class SYSTEM_ENTITY to execute this rule. By definition, only one SYSTEM_ENTITY object exists, so this rule will only be executed by one object. In plain English, the rule in Listing 3 reads as ''A device object with name machine_A must be created.''

```
01 creation machine_A
02     DEVICE
03         name "machine_A"
04     select SYSTEM_ENTITY
```

Listing 3: This creation rule reads as ''A device with name machine_A must be created.''

```
01 class WEB_SERVER
03 feature -- Attributes
05     implementation: STRING
07     php_supported: BOOLEAN
09     startup: STARTUP
11 invariant
12     implementation.is_equal("publicfile") implies not php_supported
13 end
```

Listing 2: This partial web server class illustrates the invariant mechanism to model the dependency between the implementation and php support and the reference mechanism to model the dependency between the web server and startup system.

Now that we can write rules to create objects for all managed devices, it is time to enable some functionalities on those devices. For example, all devices need to resolve host names to addresses. Consequently, we need to enable each machine's DNS configuration. To enable this, we assume the presence of a DNS_CLIENT class in the domain model. Listing 4 shows how to express that every machine should configure itself as a DNS client. The rule has "dns_clients" as its identifier (line 1). In this case, the object that needs to be created is of class DNS_CLIENT (line 2). The objects that need to create a DNS_CLIENT objects are all objects of class DEVICE (line 3). Enabling functionality on a device is thus equivalent to instructing a device to create an object that represents that functionality (in this case, a DNS client). In plain English, the rule in Listing 4 reads as "All machines must act as a DNS client."

```
01 creation dns_clients
02     DNS_CLIENT
03     select DEVICE
```

Listing 4: This creation rule reads as "All machines must act as a DNS client."

In many cases, a more fine grained mechanism is needed to describe which objects need to execute a rule. For example, how would you say that all machines you use as a server need to configure themselves as a web server? To enable this, the part of a rule that selects objects on which to apply the rule can be further refined with a boolean expression. This boolean expression filters the objects that apply the rule. For example, in Listing 5 the selection clause on lines 3-4 includes all DEVICE objects, except for those where the boolean expression on line 4 evaluates to false. In plain English, this rule reads as "Machines with label 'server' act as WEB_SERVER." Note that we assume the presence of a "labels" attribute in the class DEVICE. The contents of this attribute can be easily set when writing rules that create devices. This is done in the same way as we assigned the value "machine_A" to the "name" attribute in Listing 3.

```
01 creation mail_servers
02     WEB_SERVER
03     select DEVICE
04     where DEVICE.labels.has("server")
```

Listing 5: This creation rule reads as "Machines with label "server" act as WEB_SERVER."

The syntax of the select-clause – lines 3 and 4 of Listing 5 – is modeled after SQL SELECT statements [2]. The name of a table – class name in our case – follows the "select" keyword. The optional "where" clause excludes rows – objects conforming to the class name – where the boolean expression evaluates to false. All queries and attributes of a class can be used in a boolean expression. Operators are used to compose composite expressions. Listing 5 uses the feature call operator. Other examples of operators are: comparison operators, boolean operators and arithmetic operators.

In many cases, you want to express not only *what* objects need to be created on a DEVICE – or another object – but also *how many* need to be created. This is where creation constraint rules come into the picture. Listing 6 expresses the previously mentioned example that "a device should not provide more then four network services". Creation constraint rules have an extra keyword: "constraint". The name of the class to be created is also prefixed with an interval. In this case the interval expresses that a maximum of four server objects can be created. Note that we are using the inheritance features from the domain model in this example. We assume that all types of network services such as DHCP servers and web servers inherit from the SERVER class.

```
01 creation constraint server_objects
02     [ 0 : 4 ] SERVER
03     select DEVICE
```

Listing 6: This creation constraint rule reads as "A device should not provide more then 4 network services."

Often, you don't care which DEVICE will be your web server, as long as one – or more – devices are configured as web server. This can be expressed with the "group by" clause of the SQL SELECT syntax. The group by clause applies a rule to a group of objects rather than to of single objects. Listing 7 then reads as "One device with label 'server' must act as a web server."

We end with an often cited example in the context of configuration management: "I want two DHCP servers on each subnet." The rule for this example is shown in Listing 8. This example combines constraint rules and rules with "group by" clauses.

```
01 creation mail_service
02     WEB_SERVER
03     select DEVICE
04     where DEVICE.labels.has("server")
05     group by DEVICE.labels.has("server")
```

Listing 7: This creation rule reads as "One device with label "server" must act as a web server."

```
01 creation constraint dhcp_servers
02     [ 2 : 2 ] DHCP_SERVER
03     select NETWORK_INTERFACE
04     group by NETWORK_INTERFACE.subnet_interfaces
```

Listing 8: This creation constraint rule reads as "Each subnet must have two DHCP servers."

Before we explain the rule itself, we introduce the NETWORK_INTERFACE class. In the same way as we can create devices, DNS clients and web servers objects, we can create objects representing network interfaces. It does not matter if an object represents hardware (such as device and network interface) functionality or software functionality (such as DNS client and web server). The basic concept is that the SYSTEM_ENTITY object creates DEVICE objects. DEVICE objects can be instructed to create other objects such as DNS_CLIENT or NETWORK_INTERFACE objects. In the same way, NETWORK_INTERFACE objects can be instructed to create DHCP_SERVER objects, which is the functionality demonstrated in Listing 8.

The interval on line 2 limits the number of DHCP_SERVER objects to two. The "subnet_interfaces" query of the NETWORK_INTERFACE object returns a set of all subnet interfaces in the same subnet as the object on which the query is executed. The result of the "select" clause on lines 3-4 will be a set of network interface sets. Each inner set represents one subnet. On each of those inner sets, the rule to create two DHCP servers is executed, which results in two DHCP servers on each subnet.

Modifying Attributes

Once roles are assigned to devices, you want to tune the behaviour of those roles. Your web server needs a port to run on, your DHCP server needs to know whether it should serve fixed addresses, your DNS client needs to know what its domain is, and so forth. These examples can be expressed with PoDIM's attribute assignment rules. They change the value of an object's attributes.

Let's start with the simple case: how do we specify the search domain for our DNS clients? The rule that realizes this is shown in Listing 9. Rules dealing with attribute assignments are called assignment rules (hence the keyword "assignment" on line 1 of Listing 9). In general, an assignment rule consists of a series of attribute-value assignments that are applied to the objects in the select-clause. In our example, we show one attribute-value assignment, where the attribute is "search_domain" and the value is "mydomain.com". The objects on which this assignment is applied are, in this case, all DNS clients.

```
01 assignment dns_search_domain
02     search_domain    "mydomain.com"
03     select DNS_CLIENT
```

Listing 9: This assignment rule reads as "All DNS clients have mydomain.com as their search domain."

In some cases, you don't care what value an object's attribute has, as long as it's within a predefined range. For example, you might want to express that "the port of all my web servers should be set to 80 or a value higher than 1024." This is where assignment constraint rules come into the picture. Listing 10 shows the assignment constraint rule for our example. The attribute to be set is called "port". The valid values for this attribute are the union of the singleton 80 and all values greater than 1024.

```
01 assignment constraint webserver_ports
02     port    [ 80 ] + [ 1024 : ]
03     select WEB_SERVER
```

Listing 10: This assignment constraint rule reads as "A web server's port must be within the range 80 or a value greater than 1024."

Authorizing Commands

Many system administrators work in a team. In most teams, people have roles: Jack is our Linux server specialist, Greg is our networking guy and Bill is our desktop guy. In small teams, communication is easy – Jack, Greg and Bill are located in the same office. In larger teams, however, there is a need to specify roles more precisely and enforce those automatically.

Since Bill is our desktop guy, we do not want him to configure network services of any kind. How do we express this? Consider the SERVER class. All network services like DHCP servers and web servers inherit from this class. The SERVER class thus represents common functionality for network services. We want to express that Bill cannot modify the attributes of an object if it inherits from SERVER. To realize this, we first introduce two extra PoDIM features: a rule type to express access controls and support for writing rules about other rules.

Recall from the discussion of commands that we wanted to limit access to the "enable_php" command on a web server to objects that run on the same device as the web server. To allow this, we introduce a third type of rule: filter rules. Remember that we already introduced creation and assignment rules. Listing 11 shows a filter rule for the case where we want to limit access to the "enable_php" command of web servers. The filter rule blocks the execution of the "enable_php" command on a WEB_SERVER for every ENTITY that is not created by the same device as the WEB_SERVER.

Filter rules allow to block the execution of a command based on the caller object and callee. A filter rule starts with the "filter" keyword and has an

```
01 filter php_enabling
02     enable_php block
03     select ENTITY, WEB_SERVER
04     where not ENTITY.device.is_equal(WEB_SERVER.device)
```

Listing 11: This filter rule reads as "PHP support on web server can only be enabled by objects on the same device."

optional identifier. It contains one or more commands (with optional arguments) that need to be blocked. The selection part on lines 3-4 is a bit different than that of creation and assignment rules. A filter rule always selects pairs of objects to identify a caller/callee pair. In SQL terminology: the SELECT clause contains a join of tables named ENTITY and WEB_SERVER. The resulting tuple-set is then filtered with the "where" expression on line 4. In this case, we express that we want to block communications when the caller is any entity and the callee a WEB_SERVER (line 3). If the caller executes the "enable_php" command, it is blocked when the caller is not created by the same device as the web server.

The other feature we need are rules about other rules. When we want to express that Bill cannot configure network services of any kind, we need a filter rule that prohibits the modification of objects representing network services. The only way Bill can modify objects is to write assignment rules. So, we want to write a filter rule that blocks assignment rules written by Bill from being applied on network services. Remember that we defined a common class for network services in this example, called SERVER. Since a filter rule specifies a policy for the interaction between two objects and assignment rules are in this case part of the interaction, assignment rules themselves must be objects.

Let's go into more detail on how rules can be objects themselves. Take for example the assignment rule from Listing 9. The assignment rule contains a rule identifier, "dns_search_domain", an attribute that needs to be modified, "search_domain", the value for that attribute, "mydomain.com", and the set of target objects, all DNS_CLIENT objects. Looking at this rule as an object, we have an object with attributes "identifier", "attribute", "value", and "targets". In this example, the value of "identifier" is "dns_search_domain"; "attribute" is set to "search_domain" and so on.

Since rules exist as objects in our system, they must have a static definition (class) in the domain model. An updated graphical representation of our domain model is shown in Figure 2. PoDIM's three type of rules – creation, assignment and filter – are shown as classes in the domain model.

We have now introduced all features needed for expressing that Bill cannot configure network services. This policy is represented with a filter rule shown in Listing 12. The filter rule deals with the interaction between ASSIGNMENT_RULE objects and SERVER objects. By definition, the "execute_assignment_rule" command is used to execute an assignment rule on an object. Since we want to forbid Bill to configure network services, we must block the execution of this command for rules created by Bill on SERVER objects.

Note that we depend on the presence of the "creator" attribute of an ASSIGNMENT_RULE. This attribute can be set with another assignment rule. We will not delve into how we can be sure that identities can not be spoofed. For now, it suffices that this can be achieved with public key cryptography and rule signatures.

Notice from Figure 2 that rule classes have a common parent: RULE. RULE itself is a child of the common class MANAGED_OBJECT, as is the ENTITY class that was discussed previously. In the same way that you can not compare a DNS client with a web server, it is useless to compare rules with entities. They both have the same structure: they contain attributes, queries and commands, but they represent very different things: rules represent intentions on the part of the system administrators while entities represent real world objects such as devices, network interfaces and network services.

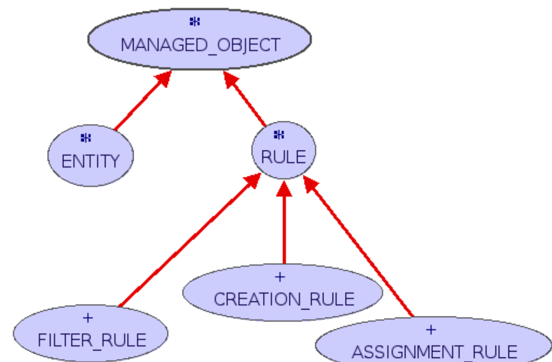

Figure 2: Domain model in BON [40] notation. This model includes RULE classes. RULE and ENTITY have a common parent: MANAGED_OBJECT. The arrows denote inheritance relationships.

Prototype

The prototype described below is available for testing from http://purl.org/podim/devel. First we describe the rule resolution process. Next, we describe how a configuration is deployed on a set of machines.

Rule Resolution

We have seen that the basic principle of PoDIM's runtime is that objects are created for each real world "thing". We have also discussed how a system administrator uses creation rules to specify which objects

```
01 filter bill_cannot_configure_services
02     execute_assignment_rule(rule) block
03     select ASSIGNMENT_RULE, SERVER
04     where ASSIGNMENT_RULE.creator = "BillsPublicKey"
```

Listing 12: This filter rule reads as "Bill cannot configure network services."

need to be created. The basic form of a creation rule is that it states that an object or objects of a particular class must be created by other objects. For example, we can assert that all devices must create a DNS client object. Remember that there is a bootstrapping problem with this approach. To solve this, we assumed the presence of one object of a predefined class, SYSTEM_ENTITY.

The component responsible for creating a SYSTEM_ENTITY object is the translation controller. The translation controller contains compiled versions of all domain model classes. At startup, it creates a SYSTEM_ENTITY object and then parses one or more policy files. Policy files contain one or more creation, assignment or filter rules. Remember that rules themselves are also objects in the system. Thus, the translation controller creates an object for each rule.

At this moment, there is one SYSTEM_ENTITY object and an object for each rule. The translation controller then iterates over all available objects and asks them to configure themselves. This configuration process is different for RULE and ENTITY objects. RULE objects check if there are new objects that conform to their selection clause. If there are, they attach themselves to those objects.

The configuration process of an ENTITY object starts with checking all attached creation rules. The creation rules are sorted by class name. Remember that classes represent things such as DHCP servers and web servers. For each class name, the intersection of all creation constraints is computed. Creation constraints are constraints on the number of objects of each class name. If the intersection of all creation constraints is empty, an error is generated. Else, the minimum number of objects is created to satisfy all creation rules.

Next, all attached assignment rules are checked and sorted per attribute name. For each attribute, the set of allowable values is computed. If this set is empty, an error is generated. Else, the number of elements in the set is computed. If there is only one element, the attribute's value can be assigned. Else, one value is chosen from the set. The algorithm that chooses one value from a set can be redefined in each class. For example, the algorithm for choosing a valid port on a web server will have to take into account ports chosen by other services on the same device. The algorithm for choosing a valid IPv4 address from a set will have to take into account the network address of its subnet and addresses already assigned to other devices on the same subnet.

After all objects have been asked to configure themselves for the first time, the whole process is repeated. In practice, SYSTEM_ENTITY will create a number of DEVICE objects based on its attached creation rules. In the next run of the configuration process, DEVICE objects will create other objects representing services like DHCP servers and web servers.

The configuration process continues until all existing objects reach a stable state. A stable state for an object is defined as follows: all rules attached to the object are satisfied. A class can extend the definition of a stable state. In the RULE classes, for example, the definition of stable state is extended with the requirement that a rule must be attached to all objects satisfying its selection clause. For a web server class, the definition can be extended with the requirement that the port attribute must have a value, even if no rules exist that set the port attribute. Determining values for attributes for which no rule exist is done by calling an extra method after the configuration process of each object. By default this method contains nothing, but objects can redefine it. For example, the web server class can define this method to set the port attribute to 80 if no rules exist for this attribute.

It is possible that a stable state is never reached. First, a class definition can be erroneous. The specification of what is a stable state can be ill-defined. The extra method that can be defined in each class for additional configuration can also contain errors that prevent objects from the class (or other objects) to reach a stable state. Second, because of the complex (multiple) inheritance relationships that can exist between classes it is possible that the creation rules are never satisfied.

The enforcement of filter rules is done when objects execute methods on each other. Before executing a method, the attached filter rules are checked. If a block policy exists, the execution is not allowed.

Configuration Deployment

When the translation controller notices that all objects have reached a stable state, the deployment process is started. The goal of the deployment process is to generate configuration files from the created objects and deploy these files on all managed devices.

The first phase is to output an XML-based representation of the in-memory objects. This is done by asking the SYSTEM_ENTITY object to output its configuration. The SYSTEM_ENTITY object asks all its children (which are DEVICE objects) to output their configuration. The DEVICE objects in turn, ask their children to output their configuration and so on. The result is that, for each DEVICE, a tree-structured XML profile is created. This profile consists of simple attribute-value assignments for all attributes and queries of an object. The format of this XML representation defined in Anderson's and Smith's LISA 2005 paper [8].

Next, the XML profiles are used as input for a template engine which generates configuration files and associated configuration instructions. Except configuration files themselves, everything that can be changed in a system is defined as a configuration instruction. Examples are: settings permissions and ownerships of files, installing packages, restarting software services

and creating links. The format of configuration instruction is XML-based and is derived from the internal XML format that Bcfg2 [18, 20, 19] uses. The grammar of the format can be found on http://purl.org/podim/devel . The configuration instructions are then translated to the languages used by one of the deployment backends. The prototype allows multiple deployment backends to be used. For example, it is possible to translate configuration instructions to Cfengine [12, 10, 11, 14], Bcfg2 [18, 20, 19] or Lcfg [7, 3, 6] specifications. It is also possible to add additional deployment backends.

Case Study

To validate our system, we use the IPv4 addressing policies for the Computer Science Department of the K. U. Leuven (CSNet). CSNet has a total of 600 machines in about 20 subnets. The 134.58.39.0-134.58.47.255 block of addresses is assigned to CSNet. CSNet has two connections to the university-wide network. The main connection is a subnet that contains, besides the external router for CSNet, switches from other departments and a router that connects to the main K. U. Leuven backbone. One lab is connected to the private network of the K. U. Leuven. We want to assign static addresses to all network interfaces. Some subnets need private addresses. Private addresses are used by the lab networks and the network of the departmental administration, since this is a Windows network which is safer behind a NAT device.

Besides classes for modeling devices and network interfaces, we need a class to model a static IPv4 address configuration. This class is shown in Listing 13. The class contains a reference to a network interface (line 6), the value for its address (line 9) and its network (line 12). The class also defines a query that returns the netmask (lines 17-20).

```
01 class
02     NETWORK_IPV4_STATIC_ADDRESS
04 feature -- Attributes
06     interface: NETWORK_INTERFACE
07         -- attached interface
09     address: IPV4_ADDRESS
10         -- IPv4 address
12     network: IPV4_NETWORK
13        -- subnet configuration
15 feature -- Queries
17     netmask: INTEGER is
18         do
19             Result := network.netmask
20         end
22 end
```

Listing 13: Class definition of a static IPv4 address.

The IPv4 addressing policies for CSNet are described in Listing 14. Because of space limitations, we omitted the creation of device and network interface objects, representing the hardware configuration of our infrastructure. Notice that, except for a few corner cases (the networks providing external access), all devices are managed with the first three constraint rules: one creation constraint rule that that creates static IPv4 address configuration and two rules for configuring the private and public address space. In plain English, the rules in Listing 14 read as follows.

1. **Rule on lines 3-8**: All network interfaces must have one static IPv4 address configured, except for the interface of "jasje" on the external access subnet (KULEUVENNET). "Jasje" is our network sniffer.
2. **Rule on lines 10-14**: All network interfaces that must be reachable from the Internet must have an IPv4 address in the range 134.58.39.0 - 134.58.47.255.
3. **Rule on lines 16-20**: All network interfaces on private subnets must have an IPv4 address in the range 192.168.0.0 - 195.168.255.255.
4. **Rule on lines 22-26**: The network interfaces on the PC_KLAS subnet must have an IPv4 address on the 10.2.15.0/24 subnet.
5. **Rule on lines 28-32**: The access switch on the PC_KLAS subnet must have the 10.2.15.254 address.
6. **Rule on lines 34-38**: All interfaces in the external access subnet – KULEUVENNET – must have an IPv4 address on the 134.58.254.64/29 subnet.
7. **Rule on lines 40-45**: The gateway of the external access subnet must have the 134.58.254.70 address.

As discussed previously, the translation controller reads the policy rules and tries to find a stable state. If the latter succeeds, configuration files are generated by the template engine. Based on the operating system of a device, a template file is chosen. This template then generates configuration files. For example, for OpenBSD devices, /etc/hostname.xxx files are generated. For Debian GNU/Linux devices, /etc/network/interfaces are generated. On Cicso routers and switches, one global configuration file is generated. Depending on the mechanics of the chosen deployment engine (Cfengine, Bcfg2, Lcfg, . . .), configuration files and are then transported to and deployed on a device.

Related Work

Related work of PoDIM's high-level configuration language includes configuration management tools. We also discuss how generic policy languages and a model finder are related to the problem PoDIM is trying to solve. We end this discussion with a characterization of application deployment frameworks.

Configuration Management Tools

Bcfg2 [18, 20, 19], Cfengine [12, 10, 11, 14], LCFG [7, 3, 6] and Puppet [26, 27] are the most cited

configuration management tools. As discussed previously, these tools can be used as a deployment backend for PoDIM. Bcfg2 and Cfengine are in the first place deployment engines. LCFG and Puppet include capabilities for modeling dependencies between configurations.

Other related work in the context of configuration management includes the work of Couch on closures [16]. Closures are defined as functional units that can accept commands from the user or other closures. Their internal mechanics are hidden. The classes from PoDIM's domain model can be seen as closures. Classes define commands that change the behaviour of their objects, but can also have queries and attributes.

Policy Languages

PoDIM separates the domain model and the policy specification language. Many other policy languages use this separation. It allows for reuse of both the policy specification language and domain model in different contexts. The PCIM [33, 32] (Policy Core Information Model) and CIM (Common Information Model) [15] initiatives define a generic model for representing policy specifications on one side and a set of domain classes on the other side. The generic model defines policy rules in a Condition-Action format. The domain model includes definitions for common network functionalities such as routing protocols, network configurations and IPSec configurations. The domain model itself is object-oriented and models relations between classes. The CIM domain model provides a valuable repository of existing domain knowledge, modeled as object-oriented classes. The CIM model is very similar to the PoDIM domain model. However, it has no support for specifying fine-grained dependencies. PoDIM uses Eiffel invariants for this. PCIM/CIM also does not support constraint handling.

Other frequently cited policy languages such as Ponder [17] and JRules [23] also offer an extensible domain model. The domain model of these languages

```
1  -- IPv4 Addressing
3  creation
4    -- Each interface has 1 IPv4 address, except "jasje"
5    [1-1] NETWORK_IPV4_STATIC_ADDRESS
6    select NETWORK_INTERFACE
7    where NETWORK_INTERFACE.device.name  /= "jasje" and
8      NETWORK_INTERFACE.labels.has("KULEUVENNET")
10 assignment constraint
11   -- Public address space
12   address [!!IPV4_ADDRESS.make("134.58.39.0"):!!IPV4_ADDRESS.make("134.58.47.255")]
13   select NETWORK_IPV4_STATIC_ADDRESS
14   where NETWORK_IPV4_STATIC_ADDRESS.interface.labels.has("PUBLIC_SUBNET")
16 assignment constraint
17   -- Private address space
18   address [!!IPV4_ADDRESS.make("192.168.0.0"):!!IPV4_ADDRESS.make("192.168.255.255")]
19   select NETWORK_IPV4_STATIC_ADDRESS
20   where NETWORK_IPV4_STATIC_ADDRESS.interface.labels.has("PRIVATE_SUBNET")
22 assignment
23   -- PC_KLAS IPv4 Address range
24   network !!IPV4_NETWORK.make("10.2.15.0",24)
25   select NETWORK_IPV4_STATIC_ADDRESS
26   where NETWORK_IPV4_STATIC_ADDRESS.interface.labels.has("PC_KLAS")
28 assignment
29   -- PC_KLAS external router
30   address !!IPV4_ADDRESS.make("10.2.15.254")
31   select NETWORK_IPV4_STATIC_ADDRESS
32   where NETWORK_IPV4_STATIC_ADDRESS.interface.device.name = "lswitch-cw"
34 assignment
35   -- KULEUVENNET external access
36   subnet !!IPV4_NETWORK.make("134.58.254.64",29)
37   select NETWORK_IPV4_STATIC_ADDRESS
38   where NETWORK_IPV4_STATIC_ADDRESS.interface.labels.has("KULEUVENNET")
40 assignment
41   -- Gateway configuration for KULEUVENNET
42   address !!IPV4_ADDRESS.make("134.58.254.70")
43   select NETWORK_IPV4_STATIC_ADDRESS
44   where NETWORK_IPV4_STATIC_ADDRESS.interface.device.name = "default-gateway" and
45     NETWORK_IPV4_STATIC_ADDRESS.interface.labels.has("KULEUVENNET")
```

Listing 14: Policy Specification for the network configuration of K. U. Leuven's CS department.

is object-oriented, as is the case with the PoDIM domain model. Policy rules are Event-Condition-Action based in these languages. The action that can be executed is an arbitrary operation of the domain model. Notice that differs with our approach. Our approach is less expressive in the sense that we do not allow the execution of arbitrary operations. However, we do allow to model constraints on attributes, which is something that is not supported by the current generation of policy languages.

Model Finding

In [35] a model finding approach is proposed for configuration management based on Alloy [24, 25, 1]. This approach is based on creating a model for an infrastructure based on first-order logic. Based on a number of inputs (such as the number of devices and network interfaces) an outcome is constructed that satisfies the model. The advantage of using a tool such as Alloy is that it allows very advanced reasoning over a configuration. The same model can be used to generate and validate configurations. The limitations are that constraints, as we discussed them in this paper, can not be modeled. It is also difficult to set specific attributes of ''things''. For example, it is difficult to set a human readable name for every device.

Application Deployment Frameworks

Application deployment frameworks like SmartFrog [29, 28], Spring [38] and JBoss Microcontainer [22] manage applications directly by tuning their parameters. Notice the difference with PoDIM: classes directly control real world things, while PoDIM classes representate real world things that are deployed when a stable state is reached. Because of this difference, application deployment frameworks listed above do not provide constraint resolution of the kind that PoDIM uses.

Future Work

There are various areas where PoDIM could be improved. We already mentioned the stable state problem: in some cases, it is impossible to reach a stable state and, as a consequence, generate a valid configuration. It is also possible that, on different runs of the translation controller, different configuration files are generated for the same input rule sets. This is because of the possibility that classes define random choices for choosing a value from a constraint set. Including information about previous runs could solve this problem.

A dry-run or analysis mode would also be useful. Currently, the prototype supports the actual translation and deployment process. Checking the validity of a rule set requires a simulation modus. Ponder, for example, uses Event Calculus [9] to do this.

We have not yet gathered data on the scalability of our prototype. In its current implementation, the translation from rules to configuration files is done on one central component. We are currently working on a decentralized version of the translation controller. In the decentralized version, each device can be made responsible for generating its own configuration and will need to communicate with other devices to find a globally valid state.

Other areas for improvement deal with usability.

- Extending the domain model requires knowledge of Eiffel. In many cases, a less expressive (and easier) notation suffices for creating new classes in the domain model. A program can then translate classes to the Eiffel notation.
- There is no support to track configurations throughout the translation process. Therefore, it is impossible to know what rules influence the generation of a configuration files or what changes in a configuration file are caused by a change in one of the rules. Both would be useful for debugging rules. In general, the translation controller must be able to explain why a configuration is generated.
- The rule language contains no structuring mechanisms. Working with large policy sets becomes cumbersome. Existing preprocessor systems can solve part of this problem, but specifying meta-rules will stay inconvenient. Instead of looking at PoDIM's rule language as a user interface, it is better to see it as a target format for more advanced interfaces which could be, depending on the case text-based, command-line programs, graphical user interfaces, web interfaces, and so on.

Conclusion

PoDIM tries to address the complexity of configuration management by abstracting from system variety and providing mechanisms for specifying cross machine constraints. The PoDIM language consists of a rule language and an extensible domain model. The current prototype translates high-level rules in low-level configuration files and subsequently uses existing configuration management tools to deploy the generated configuration on all managed devices. We believe that PoDIM provides an advancement of the state of the art. It provides a higher-level specification compared to what is currently available, which includes cross machine constraints and abstracts away systems variety.

Acknowledgments

This research has been supported by a grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (http://www.iwt.be). We would like to thank Sara Vermeylen and Nico Janssens for proofreading this paper. Thanks also to Ed Smith for his excellent work on shepherding this paper, and Paul Anderson for discussing configuration management ideas.

Author Biographies

Thomas Delaet is a Ph.D. student at the computer science department of the Katholieke Universiteit Leuven. His research is funded by the IWT, the Flemish institute for innovation in science and technology.

Wouter Joosen is a full professor in Distributed Systems at the Katholieke Universiteit Leuven, Belgium. He has been a professor in software engineering at Odense University, Denmark, from 1997 to 2001. Wouter's research interests include the development, deployment and management of distributed and secure software systems.

Bibliography

[1] *The Alloy Analyzer*, http://alloy.mit.edu .

[2] American National Standards Institute, *ANSI X3. 135-1992: Information Systems – Database* 1430 Broadway, New York, 1989.

[3] Anderson, Paul, *LCFG Homepage*, http://www.lcfg.org .

[4] Anderson, Paul, *Short Topics in system Administration 14: System Configuration*, USENIX Association, Berkeley, CA, 2006.

[5] Anderson, Paul and Alva Couch, "What Is This Thing Called "System Configuration?" LISA Invited Talk, November, 2004.

[6] Anderson, Paul, and Alastair Scobie, "Large Scale Linux Configuration with LCFG," *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, October 10-14, pages 363-372, USENIX, Berkeley, CA, 2000.

[7] Anderson, Paul and Alastair Scobie, "LCFG – the Next Generation," *UKUUG Winter Conference*, UKUUG, 2002.

[8] Anderson, Paul and Edmund Smith, "Configuration Tools: Working Together," *Proceedings of the Large Installations Systems Administration (LISA) Conference*, pages 31-38, USENIX Association, Berkeley, CA, December 2005.

[9] Bandara, A. K., E. C. Lupu, J. Moffett, and A. Russo, "Using Event Calculus to Formalise Policy Specification and Analysis," *Proceedings of the 4th IEEE Workshop on Policies for Distributed Systems and Networks*, 2003.

[10] Burgess, M., *Cfengine WWW site*, 1993, http://www.iu.hio.no/cfengine .

[11] Burgess, M., *GNU cfengine*, Free Software Foundation, Boston, Massachusetts, 1994.

[12] Burgess, M., "A Site Configuration Engine," *Computing Systems*, MIT Press: Cambridge, MA, Vol. 8, p. 309, 1995.

[13] Burgess, M., "Needles in the Cray Stack: The Myth of Computer Control," *USENIX ;login:*, Vol. 26, Num. 2, pp. 30-36, 2001.

[14] Burgess, Mark, "Recent Developments in Cfengine," *Unix.nl Conference Proceedings*, 2001.

[15] *Common Information Model (CIM) Standards*, http://www.dmtf.org/standards/cim/ .

[16] Couch, A., J. Hart, E. G. Idhaw, and D. Kallas, "Seeking Closure in an Open World: A Behavioural Agent Approach to Configuration Management," *Proceedings of the Seventeenth Systems Administration Conference (LISA XVII)*, USENIX Association, Berkeley, CA, p. 129, 2003.

[17] Damianou, Nicodemos C., *A Policy Framework for Management of Distributed Systems*, Ph.D. thesis, University of London, Department of Computing, 2002.

[18] Desai, Narayan, Rick Bradshaw, and Joey Hagedorn, *Bcfg2 Trac Homepage*, http://trac.mcs.anl.gov/projects/bcfg2 .

[19] Desai, Narayan, Rick Bradshaw, and Joey Hagedorn, *System Management Methodologies with Bcfg2*, *;login: The USENIX Association Newsletter*, Vol. 31, Num. 1, February 2006.

[20] Desai, Narayan, Rick Bradshaw, Scott Matott, Sandra Bittner, Susan Coghlan, Rémy Evard, Cory Lueninghoener, Ti Leggett, John-Paul Navarro, Gene Rackow, Craig Stacey, and Tisha Stacey, "A Case Study in Configuration Management Tool Deployment," *Proceedings of the Large Installations Systems Administration (LISA) Conference*, pp. 39-46, USENIX Association, Berkeley, CA, December, 2005.

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

[22] JBoss Group, *Jboss Microcontainer*, http://www.jboss.com/products/jbossmc .

[23] ILOG, *Jrules: Technical White Paper (Version 4.0)*, 2002, http://www.ilog.com/products/jrules/ .

[24] Jackson, Daniel, "Alloy: A Lightweight Object Modelling Notation," *ACM Transactions on Software Engineering and Methodology*, Vol. 11, Num. 2, pp. 256-290, 2002.

[25] Jackson, Daniel, *Software Abstractions: Logic, Language and Analysis*, The MIT Press, 2006.

[26] Kanies, Luke, *Puppet*, http://reductivelabs.com/projects/puppet/ .

[27] Kanies, Luke, *Puppet: Next-Generation Configuration Management*, *;login: The USENIX Association Newsletter*, Vol. 31, Num. 1, February, 2006.

[28] Hewlett Packard Laboratories, *The smartfrog Reference Manual*, 2007.

[29] Low, Colin and Julio Guijarro, *A smartfrog Tutorial*, Technical Report, Hewlett Packard Laboratories, 2004.

[30] Meyer, B., *Eiffel, the Language*. Prentice Hall, 1992.

[31] Meyer, Bertrand, *Object-Oriented Software Construction*, Prentice-Hall, Second Edition, 1997.

[32] Moore, B., *Policy Core Information Model (PCIM) Extensions, RFC 3460 (Proposed Standard)*, January, 2003.

[33] Moore, B., E. Ellesson, J. Strassner, and A. Westerinen, *Policy Core Information Model – Version 1 Specification, RFC 3060 (Proposed Standard)*; Updated by RFC 3460, February, 2001.

[34] Narain, Sanjai, *Towards a Foundation for Building Distributed Systems via Configuration*, 2004, http://www.argreenhouse.com/papers/narain/Service-Grammar-Web-Version.pdf .

[35] Narain, Sanjai, "Network Configuration Management via Model Finding," *LISA'05: Proceedings of the 19th Conference on Large Installation System Administration Conference*, p. 15, USENIX Association, Berkeley, CA, 2005.

[36] Oppenheimer, D., The Importance of Understanding Distributed System Configuration," *Proceedings of the 2003 Conference on Human Factors in Computer Systems workshop*, April, 2003.

[37] Patterson, D. A., "A Simple Way to Estimate the Cost of Downtime," *Proceedings of the Sixteenth Systems Administration Conference (LISA'02)*, pp. 185-188, USENIX Association, Berkeley, CA, 2002.

[38] *Spring Framework*, http://www.springframework.org .

[39] Strassner, John, "Policy management challenges for the future," Policy 2005 Keynote, 2005.

[40] Walden, Kim and Jean-Marc Nerson, *Seamless Object-Oriented Software Architecture*, Prentice-Hall, 1994.