View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Edinburgh Research Explorer

THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Managing Real-World System Configurations with Constraints

Citation for published version:
Delaet, T, Anderson, P & Joosen, W 2008, Managing Real-World System Configurations with Constraints. in Networking, 2008. ICN 2008. Seventh International Conference on. IEEE Computer Society, pp. 594-601. DOI: 10.1109/ICN.2008.38

Digital Object Identifier (DOI):
10.1109/ICN.2008.38

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Networking, 2008. ICN 2008. Seventh International Conference on

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. Apr. 2019

Delaet, T., Anderson, P., & Joosen, W. (2008). Managing Real-World System Configurations with Constraints. In Networking, 2008. ICN 2008. Seventh International Conference on. (pp. 594-601). IEEE Computer Society. 10.1109/ICN.2008.38

Managing real-world system configurations with constraints

Thomas Delaet
Department of Computer Science,
Katholieke Universiteit Leuven, Belgium
thomas@cs.kuleuven.be

Paul Anderson
School of Informatics,
University of Edinburgh, UK
dcspaul@inf.ed.ac.uk

Wouter Joosen
Department of Computer Science,
Katholieke Universiteit Leuven, Belgium
wouter@cs.kuleuven.be

Abstract

Managing large computing infrastructures in a reliable and efficient way requires system configuration tools which accept higher-level specifications. This paper describes an interface between the established configuration tool LCFG, and the experimental configuration tool PoDIM. The combined system is used to generate explicit real-world configurations from high-level, constraint-based specifications. The concept is validated using live data from a large production installation. This demonstrates that a loosely-coupled, multi-layer approach can be used to construct configuration tools which translate high-level requirements into deployable production configurations.

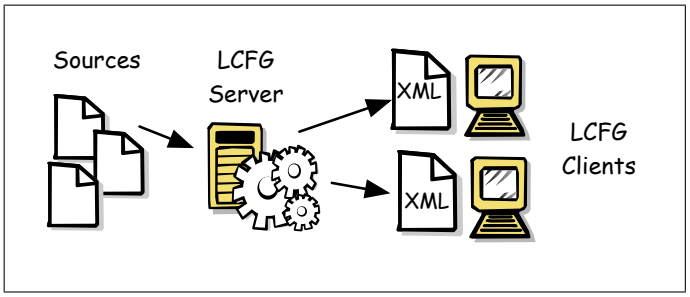

Figure 1. The LCFG Architecture

1 Introduction

The Computing Research Association's report on "Grand Challenges in Information Systems" [9] identifies reliability and the control of complexity as two of the major current challenges to Information Systems research. Many modern systems require sufficiently complex configurations that are beyond the ability of system administrators to reliably configure by hand; configuration errors are the biggest contributors to service failures (between 40% and 51%), and these errors take the longest time to repair [22], [21], [20]. Autonomics [18] offers the promise of self-configuring systems which will help to reduce these costs, but there is a considerable gap between the vision of a fully autonomic system and the practical tools currently available for configuring the infrastructure of a computing site [4]. Smith and Anderson argued in [23] that a modular approach with autonomic capability at several levels provides a route towards this vision.

This paper describes a practical example of the modular approach; the high-level experimental configuration tool PoDIM [12] is interfaced to the the well-proven (but essentially low-level) configuration tool LCFG [7]. This creates a configuration system which automatically generates correct host configurations from specifications of site-wide constraints.

Section 2 describes the LCFG and PODIM tools. Section 3 describes the architecture used to combine the two systems. Next, we describe a detailed example of a real configuration problem and show how it can be solved by the combined tool in Section 4. Further examples are presented in section 5. Some of the limitations of the prototype system are described in Section 6, and we end with a section on related work 7 before presenting the conclusions in Section 8.

2 Background

2.1 LCFG

LCFG [7] is an established configuration framework for managing large numbers of Unix systems. It is particularly concerned with specifying and managing the relationships between clients and servers in large installations. The underlying principles are described in [4], and full documentation is available from the web site [3]. Figure 1 shows a simple version of the LCFG architecture which operates as follows:

- The configuration for an entire site is described by a set of declarative *source files*, held on a master server. These source files may be managed by different people and describe different *aspects* of the overall configuration, such as a "web server", a "student machine" or a "laptop".
- The *LCFG compiler* continuously monitors changes to these source files and immediately recompiles the configuration for all hosts affected by any changes.
- The result of the compilation is one XML *profile* for each client node. The profile defines explicit values for all of the configuration parameters (*resources*) of that host[1].
- The client includes a modular set of *components*, each responsible for a self-contained subsystem, such as `inetd` or `apache`. When the client receives a new profile, it notifies all those components whose resources have changed, and they immediately reconfigure the subsystem to correspond to the new configuration.

A number of properties of LCFG are important to note for the discussions which follow:

- The LCFG source descriptions are *declarative*. They describe the *desired state* of the system configuration, rather than some process for achieving that state. The LCFG client components automatically monitor this desired state and take any appropriate action to synchronise the *actual* state with the desired state.
- The LCFG source files contain specifications for different aspects of the configuration. It is common for these aspects to overlap – for example, a file specifying the security requirements for machines at a particular site may contain parameters which are also specified in the source file for some other aspect of a particular machine, such as "laptop". The LCFG compiler resolves these conflicts – either by using specific precedence rules, or by demanding human intervention.
- LCFG supports a concept known as a *spanning map*. This allows inter-machine dependencies to be collated, and presented as part of the configuration for a particular machine. For example, the configuration for a firewall may be automatically constructed from the configuration of all the machines at a site which are running external services. This ensures that the firewall state is never out of step with the running services.
- LCFG is capable of fully *prescriptive* configuration. This means that clients can be completely configured (even installed from scratch) without any manual configuration.
- The input language to LCFG has a very simple syntax and can easily be generated by other processes.
- The configuration process is centralised and takes place on a single server.

Its simple syntax and the fact that it takes care of the lower level details of configuration management make LCFG particularly suitable as a testbed for experiments in higher-level configuration automation (see, for example [6]); some process can inspect configuration parameters, and automatically generate other configuration parameters (both as simple key-value pairs). The generated parameters can be written to a simple source file and the compiler will take care of composing them with the manually specified parameters and deploying the reconfiguration.

2.2 PoDIM

PoDIM is a recent prototype for configuration management which translates high level configuration management specifications to lower level parameters. It can be best described as a configuration management compiler; it does not deploy configurations, but generates a lower level representation which can be used as input to other tools for further processing and deployment. We limit our description of PoDIM to the features that are relevant for this paper; a full description and justification of the input language and runtime environment can be found in [12].

PoDIM is based on an object-oriented model of configuration, which supports multiple-inheritance. Examples of classes include "DHCP server", "DNS client" and "web server". Instances of these classes are created to represent real configurable objects such as services or interfaces; every configurable physical object must be represented by a simulated instance in the PoDIM model. The input to the PoDIM compiler consists of a set of rules that defines the objects to be created, their types, and their relationships.

[1] A typical profile may contain about 5000 resources.

A system administrator uses the PoDIM rule language to configure the network 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.

A distinguishing feature of the rule language is the ability to specify constraints instead of explicit values; for example, we can express high-level requirements such as: "A device should not provide more then 4 network services", "I want two DHCP servers on each subnet" or "One of my servers should configure itself as a web server".

Before illustrating the use of constraints, we show how regular role assignments are performed in PoDIM. Listing 1 shows a creation rule to express that every machine should configure itself as a DNS client.

```
creation
        DNS_CLIENT
        select DEVICE
```

Listing 1. "All machines must act as DNS clients".

In many cases, we want to express not only *the type of* objects need to be created, but also *how many*. We can do this using constraints; for example, listing 2 shows the specification for: "A device should not provide more then 4 network services".

```
creation constraint
  [ 0 : 4 ] SERVER
  select DEVICE
```

Listing 2. "A device should not provide more then 4 network services".

Often, we don't care *which* DEVICE will act as the web server, as long as one device is configured to do so. This can be expressed with a *group by* clause as illustrated in Listing 3. This applies a rule to a whole group of objects. Listing 3 states that "One device with the label "server" must act as a web server".

```
creation constraint
  [ 0 : 4 ]  WEB_SERVER
  select DEVICE
    where DEVICE.labels.has("server")
    group by DEVICE.labels.has("server")
```

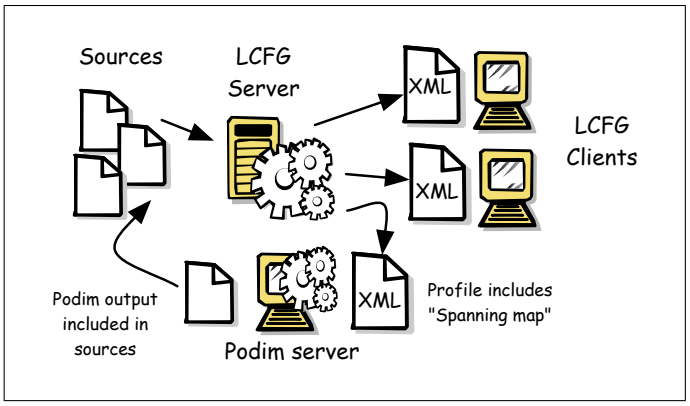

Figure 2. The LCFG/PoDIM Interface

Listing 3. "One device with the label "server" must act as a web server".

The syntax of the select-clause 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. The *group by* clause is used to apply a rule to a group of objects.

It is important to remember that PoDIM does not directly manage real world objects; it manipulates representations of the real world objects and generates a per-device XML file for each object. This file includes the objects (roles) that are assigned to a device. Other tools can then use this XML file for further processing and deployment on the physical devices.

3 Combining LCFG and PoDIM

This section describes how LCFG has been provided with an experimental interface to PoDIM which allows PoDIM's constraint-resolution process to be tested and evaluated on real-world configurations. Figure 2 shows the architecture for the combined system:

- One host runs the PoDIM server which is managed by a custom LCFG component (`lcfg-podim`).
- The LCFG schema for the PoDIM component specifies that certain configuration parameters should be collated from all other machines and passed to PoDIM (using a *spanning map*). For example, the hostnames, subnets and configured services of all other machines can all be easily collected.
- Whenever any of this information changes, the component is notified and PoDIM runs to re-evaluate the constraints. This may result in a re-allocation of services to hosts, for example.

- The output from PoDIM is passed back to the LCFG compiler as simple LCFG source files. The compiler notices any changes to these files and recompiles the configuration for any affected hosts, thus implementing the computed configuration.

Notice that this loose coupling does have a number of side-effects which may be undesirable in a production system; for example, the reconfiguration occurs in two phases – the compiler actually deploys the manual changes before the constraints can be evaluated and the computed changes can be deployed. It would clearly be preferable to deploy all of these changes as a single atomic action. This latency also provides a potential for oscillation if the automatic changes have consequences for the input to the constraint process. However, these problems would be easily solved in a production implementation, and they do not affect the validity of these experimental results.

The custom LCFG component includes all the necessary code to convert from the LCFG spanning map to the PoDIM input, and from the PoDIM output to the LCFG source.

4 An Example Application

The current state of the art in configuration management supports explicit assignment of roles to machines; for example: "Configure machine X as a DHCP server". Our experimental system is capable of defining *looser* constraint-based assignments, such as: "Configure 2 DHCP servers on each subnet". In this example, roles (a DHCP server) are assigned to groups of machines (every subnet).

Specifying the desired configuration in a looser way has several advantages:

- Looser specifications allow dynamic adaption (autonomic reconfiguration) when the network setup changes. For example: suppose machine X is a DHCP server. A rule that states "Configure machine X as a DHCP server" results in a non-functional network if machine X goes down. The looser rule - "Configure 2 DHCP servers on each subnet" - allows the DHCP server role to be automatically assigned to another machine if X goes down.

- Looser specifications are closer to the high level requirements of the system and require less manual translation. For example "We want a redundant DHCP setup" can be translated directly into a rule such as: "Configure 2 DHCP servers on each subnet".

- There is less chance of specifications conflicting because different administrators are not forced to overspecify their requirements.

Suppose we want to enforce the specification "Configure 2 DHCP servers on each subnet". Listing 4 shows this rule in PoDIM's rule language. It specifies that exactly two objects of the DHCP_SERVER class need to be created. Since DHCP servers are bound to a network interface, the NETWORK_INTERFACE objects are responsible for creating DHCP server objects. Finally, the *group by* statement specifies that the rule should not be applied to every NETWORK_INTERFACE object, but to *groups* of NETWORK_INTERFACE objects. The groups are defined as the set of interfaces in the same subnet. The "layer3_network" is an attribute of the NETWORK_INTERFACE class and is populated by every network interface object with the set of network interfaces in the same subnet.

```
creation constraint
  [2:2] DHCP_SERVER
    select NETWORK_INTERFACE
    group by NETWORK_INTERFACE.layer3_network
```

Listing 4. "Configure 2 DHCP servers on each subnet".

Besides the rule shown above, rules for representing the infrastructure are also needed; for example, rules that create representations of the real devices and network interfaces. These are generated automatically by the LCFG podim component from the spanning map containing the information about hosts and IP addresses (the full rule specification can be found on the website).

The process used by PoDIM to translate these high-level rules into deployable configurations is described in detail in [12]. Briefly, all objects that conform to the SELECT statement are monitored - in this case all NETWORK_INTERFACE objects. Based on the number of DHCP_SERVER objects in the GROUP BY statement, NETWORK_INTERFACE objects are asked to take the role of DHCP_SERVER or give it back. Note that the algorithm itself is not dependent on the type of service - in this case DHCP_SERVER.

If all rules are satisfied, PoDIM outputs a per-device XML-representation. An example of such a representation - containing only the relevant information for this example - is given in Listing 5. Each device configuration contains a tree-structure of objects and every object can contain a set of attribute-value pairs. In this example, the tree starts with an object of type "PC_DEVICE" and has an attribute "name". This device has one child - it's Ethernet interface. The Ethernet interface has one child, a DHCP_SERVER. Based on these kind of files, we can determine - for each host - if it is assigned the role of DHCP server.

```
<configuration xmlns="http://purl.org/podim/cm_il">
```

```
  <object type="PC_DEVICE">
    <attributes>
      <name>chard.inf.ed.ac.uk</name>
    </attributes>
    <children>
      <object type="ETHERNET_INTERFACE">
        <children>
          <object type="DHCP_SERVER" />
        </children>
      </object>
    </children>
  </object>
</configuration>
```

Listing 5. XML representation of a device configuration.

Finally, the LCFG `podim` component extracts the DHCP_SERVER information from the XML and creates the necessary LCFG source statements to deploy the DHCP server on the corresponding machine.

Our prototype system has been used to implement this example using live data for 120 devices over 14 subnets.

5 Further Examples

The "2 DHCP servers on each subnet" rule is one example of how roles can be assigned to machines. In this Section, we classify and illustrate the different types of rules for role assignments. and we show how sets of role assignments interact.

5.1 Types of role assignments

Roles can be assigned to individual machines (the most basic case) and to one or more groups of machines. In the same way, constraints can be assigned to individual machines and to one or more groups of machines.

5.1.1 Assigning roles to machines

The most basic form of a role assignment is to assign a role to a machine. This is the type of role assignments that is supported by the current state of the art. An example of such a role assignment is shown in Listing 6. The rule reads as "machine X must be a mail server". This rule will result in a the activation of a mail server on machine X.

```
creation
  MAIL_SERVER
    select DEVICE
    where DEVICE.name = "machineX"
```

Listing 6. "Machine X must be a mail server".

5.1.2 Assigning roles to groups of machines

As we have seen, explicit assignment of roles to machines is often not desirable; in the previous example, if machine X fails, clients will be unable to sent mail. We would like to specify a looser rule that assigns the mail server role to a group of machines. If the machine currently running the mail service becomes unavailable, our system will then automatically assign the mail server role to another machine in the group. Listing 7 illustrates this type of role assignment; the rule is applied to one group of devices (those in the mydomain.com domain) and only the devices with the label "server" are candidates for hosting the mail service.

```
creation
  MAIL_SERVER
    select DEVICE
    where DEVICE.domain = "mydomain.com"
        and DEVICE.labels.has("server")
    group by DEVICE.domain
```

Listing 7. "Configure a mail server for my domain".

5.1.3 Assigning constraints to machines

An extension of the previous rule allows the assignment of a role to multiple groups of machines. Imagine that our company policy is to have a mail server for each department. The rule shown in Listing 8 implements this policy; it assigns the mail server role to each group in the *group by* statement, which defines the different departments.

```
creation
  MAIL_SERVER
    select DEVICE
    where DEVICE.labels.has("server")
    group by DEVICE.department
```

Listing 8. "Configure a mail server for each department".

5.1.4 Assigning constraints to groups of machines

The need for constraint rules arises when we do not want to limit role assignments to exactly one role. i.e. we want to assign multiple roles or we want to specify that the number of roles of a specific type must be within an interval. We can use this to limit the number of services permitted on any host, and hence limit the loading. Listing 9 shows an example in the PoDIM notation. Note how we make use of the inheritance hierarchy of the defined classes - i.e. rule types. We assume that all types of servers - mail, web, DHCP, ... - inherit from a common class "SERVER".

```
creation constraint
  [0:2] SERVER
    select DEVICE
```

Listing 9. "A host can not run more than 2 services".

A constraint can also be assigned to a group (or multiple groups) of machines. The "2 DHCP servers on each subnet" rule was an example of a constraint rule that is assigned to multiple groups - the subnets - of machines. Another example would be "there should be a minimum of 4 mail servers active on our company network". This rule is illustrated in Listing 10. Note that the *select* clause of this rule is the same as the rule in Listing 7 ("we want one mail server for our domain"). In this rule, we state that there must be at least four mail servers, without specifying an upper limit.

```
creation constraint
  [4:] MAIL_SERVER
    select DEVICE
    where DEVICE.domain = "mydomain.com"
        and DEVICE.labels.has("server")
    group by DEVICE.domain
```

Listing 10. "There must be at least 4 mail servers serving mydomain.com".

5.2 Interactions between role assignments

A real life configuration contains sets of rules which interact with each other. Listing 11 shows a typical set of rules, derived from some of the previous examples, and we will discuss some of the interactions which arise.

```
creation constraint
  [4:] WEB_SERVER
    select DEVICE
    where DEVICE.domain = "mydomain.com"
        and DEVICE.labels.has("server")
    group by DEVICE.domain

creation constraint
  [2:2] FILE_SERVER
    select DEVICE
    group by DEVICE.department
            and DEVICE.labels.has("server")

creation
  MAIL_SERVER
    select DEVICE
    where DEVICE.domain = "mydomain.com"
        and DEVICE.labels.has("server")
    group by DEVICE.domain

creation constraint
  [0:2] SERVER
    select DEVICE where DEVICE.labels.has("server")
```

Listing 11. An example set of role assignments.

Listing 11 illustrates the following policy:

- Configure at least 4 web servers in my network
- Configure 2 file servers on each subnet
- Configure a mail server in my network
- No host can run more than 2 services.

Notice the different interactions between rules: each device can be a member of one or more different groups and for each group, multiple rules need to be satisfied. The (multiple) inheritance hierarchy further complicates the role assignment process. In our example, WEB_SERVER, FILE_SERVER and MAIL_SERVER all inherit from the generic SERVER class.

We want to deploy three types of services (web, file and mail) with the extra constraint that no device can run more than two services. What happens when these rules are fed to our system? The system will try to find an allocation of services for each device that satisfies all rules. If such an allocation can not be found, our prototype signals an error and requires manual intervention from a system administrator. Possible actions could include changing the rules or extending the number of available servers.

If the set of available servers changes, the system tries to find a new allocation that satisfies all rules. Suppose that machine Y has been assigned the role of file server. If we take machine Y out of service, the file server role will be assigned to another server in the same subnet that has less than two server roles assigned to it.

If an extra rule is added, a new allocation is computed with the new rule taken into account. Suppose that we want to have two database servers in our network. Our system tries to find servers which have not yet been assigned two network services and assigns the role database server to them.

6 Limitations

The prototype system described above has been implemented with a very loose coupling which demonstrates the feasibility of this approach. However, for a practical system, there are a number of disadvantages; for example, different languages are required to specify the fixed parts of the configuration (LCFG) and the constraints (PoDIM). This is clearly undesirable, and we would like to provide a more

unified language which would be sufficiently powerful to generate both of these underlying tool languages. The latency of the combined system in evaluating constraints may also cause spurious transient configurations to be published during evaluation; and it is currently possible to create specifications which produce non-terminating behaviour (configuration oscillation). Both of these are undesirable require further work to eliminate.

Performance is a problem in the current prototype; the 120 node example is at the limits of practicality for the existing implementation. However, the main bottleneck is the PoDIM compiler,and profiling shows that Eiffel's reflection library is by far the largest contributer to this. The performance overhead is thus not caused by the algorithm for resolving constraints but by implementation decisions, and this is something which we intend to address.

The current system is capable of reconfiguring autonomically to replace failed nodes; if a node providing a service is removed from the configuration system, a replacement will automatically be configured to maintain the constraint satisfaction. However the system does not currently make use of any monitoring information to automatically remove nodes which appear to have failed. Given accessible monitoring information, this should be simply a matter of adding an additional "liveliness" clause to the appropriate constraints.

7 Related Work

There is a wide range of system configuration tools in use at production sites, and the capabilities of newer tools have not advanced significantly from the survey described in [5]. Examples of such tools include Bcfg2 [13], Cfengine [8] and Puppet [17]. These tools are essentially *low-level*, and ad-hoc automatic generation of their input is common in practice. However, there is little published work, and we are not aware of any similar attempts to generate practical configurations from high-level constraints.

There is a large body of work on constraint satisfaction [10] originating from the domain of artificial intelligence.

We are aware of two projects that use a generic constraint solver to solve configuration management problems:

- In [19], Narain uses Alloy [15, 16, 1] to generate configurations. Alloy is an object-oriented constraint solver. Narain's approach is based on creating a model for an infrastructure based on first-order logic. Using 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.

- In [14], Hinrichs et al. use Epilog, the Stanford Logic Group's library of automated reasoning tools, to model the configuration of an E-commerce site as an Object-oriented constraint satisfaction problem. The approach used is roughly the same as Narain's approach with Alloy. However, the work is limited to the specification of constraints over individual parameters and relationships between parameters. In our work, we model constraints at the level of roles.

As far as we are aware, none of these general solutions have been validated on a large-scale real-world infrastructure; the examples discussed in this paper generally involve simpler constraints over very large datasets, and there are concerns about the performance of a more general solver in this case. We believe that using a tool tied to the context of configuration management has the potential for a more scalable implementation.

8 Conclusions

We have created code to translate LCFG configurations into PoDIM's rule language, and to translate the PoDIM output back in to LCFG specifications. This has been encapsulated in an LCFG component and deployed at a real live site. The resulting framework has been used to implement and test constraint-based specifications involving up to 120 live nodes.

This has demonstrated that it is possible to create a configuration tool which translates high-level constraint-based specifications into practical, low-level configurations which can be deployed on a real-world infrastructure. Despite their complexity, the generated configurations are guaranteed to be *correct* with respect to the simpler constraint-based specifications. Moreover, the system is built from a loose coupling between separate tools, and this validates the concept of a modular approach to autonomic configuration.

Availability

LCFG [3] and Podim [11] are both available under the GNU Public License. The custom LCFG component, the rule set for the DHCP server example, and detailed documentation for the PoDIM/LCFG interface are available on the PoDIM website [11].

References

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

[2] American National Standards Institute. *ANSI X3.135-1992: Information Systems — Database Language — SQL (includes ANSI X3.168-1989).* American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1989.

[3] P. Anderson. LCFG. http://www.lcfg.org.

[4] P. Anderson. *System Configuration*, volume 14 of *Short Topics in System Administration*. SAGE, 2006.

[5] P. Anderson, G. Beckett, K. Kavoussanakis, G. Mecheneau, and P. Toft. Technologies for large-scale configuration management. Technical report, The GridWeaver Project, December 2002.

[6] P. Anderson, P. Goldsack, and J. Paterson. SmartFrog meets LCFG - autonomous reconfiguration with central policy control. In *Proceedings of the 2003 Large Installations Systems Administration (LISA) Conference*, Berkeley, CA, 2003. Usenix.

[7] P. Anderson and A. Scobie. LCFG - the Next Generation. In *UKUUG Winter Conference*. UKUUG, 2002.

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

[9] Computing Research Association. *Grand Research Challenges in Information Systems*, June 2002.

[10] Constraint satisfaction. http://en.wikipedia.org/wiki/Constraint_satisfaction.

[11] T. Delaet. PoDIM. http://purl.org/devel/podim.

[12] T. Delaet and W. Joosen. PoDIM: A language for high-level configuration management. In *Proceedings of the Large Installations Systems Administration (LISA) Conference*, Berkeley, CA, November 2007. Usenix Association.

[13] N. Desai, R. Bradshaw, and J. Hagedorn. Bcfg2 Trac homepage. http://trac.mcs.anl.gov/projects/bcfg2.

[14] e. a. Hinrichts, T.L. Using object-oriented constraint satisfaction for automated configuration generation. In *Proceedings of Utility Computing: 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM*, 2004.

[15] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

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

[17] L. Kanies. Puppet. http://reductivelabs.com/projects/puppet/.

[18] J. Kephart. Technology challenges of autonomic computing. Technical report, IBM Academy of Technology Study, November 2002.

[19] S. Narain. Network configuration management via model finding. In *LISA'05: Proceedings of the 19th conference on Large Installation System Administration Conference*, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.

[20] S. Narain, T. Cheng, B. Coan, V. Kaul, K. Parmeswaran, and W. Stephens. Building autonomic systems via configuration. In *Proceedings of Autonomic Computing Workshop*, June 2004.

[21] D. Oppenheimer. The importance of understanding distributed system configuration. In *Proceedings of the 2003 Conference on Human Factors in Computer Systems workshop*, April 2003.

[22] D. A. Patterson. A simple way to estimate the cost of downtime. In *Proceedings of LISA '02: Sixteenth Systems Administration Conference*, pages 185–188. Usenix, Usenix, 2002.

[23] E. Smith and P. Anderson. Toward broad-spectrum autonomic management. In *Proceedings of ICN 2007, The Sixth International Conference on Networking*. IEEE Computer Society Press, April 2007.