Federated Access Control and Workflow Enforcement in Systems Configuration

Bart Vanbrabant, Thomas Delaet and Wouter Joosen
{*bart.vanbrabant, thomas.delaet, wouter.joosen*}*@cs.kuleuven.be*
DistriNet, Dept. of Computer Science,
K.U.Leuven, Belgium

Abstract

Every organization with more than a few system administrators has policies in place. These policies define who is allowed to change what aspects of the configuration of a computer infrastructure. Although many system configuration tools are available for automating configuration changes in an infrastructure, very little work has been done to enforce the policies dealing with access control and workflow of configuration changes. In this paper, we present ACHEL. ACHEL makes it possible to integrate fine-grained access control into existing configuration tools and to enforce an organization's configuration changes workflow. In addition, we prototype ACHEL on a popular configuration tool and demonstrate its capabilities in two case studies.

1 Introduction

Because the scale of modern computer infrastructure keeps increasing, so automation has become a crucial part of system configuration. Tools are important in system configuration for increasing the automation and autonomy of computer infrastructures. These tools have contributed to successfully scaling infrastructures without a linear growth in manual system administration [11].

A typical system configuration tool [9] translates a configuration specification to a per-system profile. Such a profile describes the desired state of a managed system. A local component of the system configuration tool checks whether the current state of the target system matches the intended profile and makes adjustments if necessary [10, 14, 19, 25, 32]. This local component is called the *deployment engine* of a system configuration tool. The configuration specification for the configuration tool is often retrieved from a central version-controlled repository [9, 21, 22, 33] such as CVS or Subversion. Such a repository provides a full history of the infrastructure's configuration. All configuration changes are directly checked into this central configuration specification repository.

Managing an infrastructure based on a centrally available configuration specification comes at a price. Because the central specification controls all aspects of all managed systems, control over the specification means

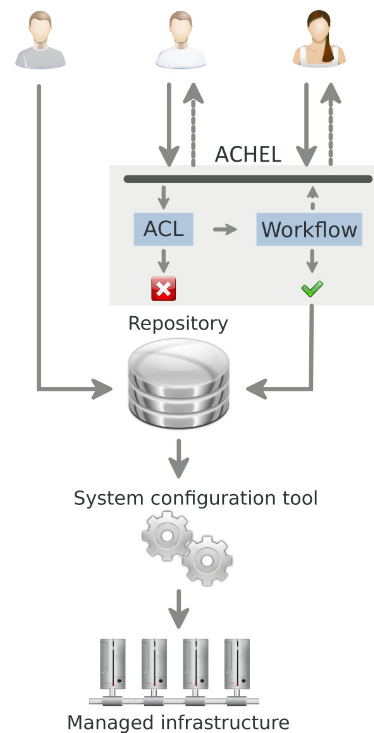

Figure 1: Conceptual overview of systems configuration without and with ACHEL.

control over all managed systems in the infrastructure. From a security perspective, the configuration specification repository requires very strict access control. Unfortunately, most existing tools do not provide any access control mechanisms, but rather reuse the access control available in the revision control repository [10, 14, 17, 19, 25, 32]. The access control systems of these revision control repositories default to allowing or denying full access based on the credentials of the user.

Although the hardware and the software in all infrastructures is quite comparable, the organization of system administration varies between different infrastructures [9, 22]. These differences can be attributed to system administration being organized either in a central or in a federated manner, as well as to the existence of different policies: certain infrastructures integrate changes directly into the configuration specification, others require changes to be approved by management, and yet others require rigorous quality control before a change is deployed. It is very hard to support these *workflows* in existing systems because of the limited workflow enforcement supportthat these systems provide [6, 16, 28].

In this paper we present ACHEL, a framework that enables the integration of fine-grained access control into existing configuration tools and enforces configuration change workflows in federated infrastructures. Figure 1 shows a conceptual overview of the process of updating a configuration specification, both with and without ACHEL. To improve the expressiveness of the access control rules, these rules are defined at the same abstraction level as the configuration specification language supported by the system configuration tool. ACHEL achieves this by taking the structure and the meaning of the configuration specification into account when generating *semantically meaningful changes*, instead of identifying changes line by line, such as the common diff-algorithm used in source code management does. ACHEL uses these semantically meaningful changes and the author history of each configuration statement to define fine-grained per user access control rules. Because a large portion of ACHEL is language agnostic, support for new configuration specification languages can be added with limited effort, as we did in our prototype.

ACHEL's second contribution is that it enforces workflow on configuration specification changes between repositories. We define a workflow as a set of rules that defines what steps a change should go through before it can be deployed on the managed infrastructure. ACHEL provides flexible workflows for centralized and federated infrastructures by building on a distributed version control system [2] and combining it with our fine-grained access control rules. A service or user signals its approval of a change by digitally signing the globally unique revision identifier that each distributed version control system provides. Access control rules are extended to enable them to require a signature before changes are allowed for inclusion. For example, ACHEL can enforce a company policy that requires changes to be signed off by a manager or an automated validation service before they are deployed.

The remainder of the paper is structured as follows: First, we discuss related work in section 2. Next, we discuss our design and how access control and workflow enforcement is applied. In section 4 we discuss the prototype we have developed, and in section 5 we evaluate our prototype.

2 Related Work

In the state of the art of system configuration tools, various levels of integration with version control systems and granularity of access control are available [10, 14, 18, 19, 24, 32]. But only very limited work on workflow enforcement of changes seems to be available [14, 25]. Using version control in configuration specification repositories is an idea that is used or recommended by most existing tools. Access control on these repositories is usually enforced by authenticating users, and it allows either full access or limited access to directories per user in a manner similar to that of a file system in an operating system. Some tools [16, 18, 24] are able to do a more fine-grained access control on a higher abstraction level, but none of them include any provisions for enforcing a workflow on configuration specification updates.

- BCFG2 [19] includes basic integration with version control repositories through a plug-in that gets the configuration specification from a SVN or Git [3] repository. BCFG2 uses the revision from the repository in reports about the configuration process, but does not provide any access control and relies fully on what the repository provides. Directing Change Using BCFG2 [21] details an approach to deploy complex configuration changes. These changes are split into steps, each of which needs to be deployed before the next can be executed. A script checks the BCFG2 reports for successful deployments before the next step is deployed.

- LCFG [10] does not integrate with version control systems, but the LCFG guide [8] does refer to using a CVS repository as the configuration database.

- Cfengine [13, 14] also recommends using version control systems as a configuration repository. [28] suggests using branches or tags to create different staging environments, for example for testing, production, and development, but no tools seem to be

available to enforce a workflow between these environments.

- Like most other tools, Puppet [25] suggests using version control repositories as a best practice on their wiki. It also contains examples for adding syntax validation before a change is accepted in the version controlled repository [4]. Additionally, the Puppet wiki suggests different branches for different environments, such as testing and production [5,6].
- DACS [32] includes tight integration with CVS or subversion. It hooks into the version control system to do basic checks such as syntax validation before a change is accepted.
- Devolved Management of Distributed Infrastructure with Quattor [16] describes how several European grid infrastructures manage large distributed infrastructures with sites under different administrative domains. They all use Quattor, a system configuration tool that uses the Pan configuration language [17]. In their workflow they include Subversion as version control repository. One of the problems with their current implementation is the inability to enforce fine-grained authorization. They handle this problem by modularizing the configuration specification using namespaces that the compiler enforces in the file name. This allows Subversion to enforce access control on file names, but the specification in one namespace can still access other namespaces, thus bypassing the Subversion access control.
- Machination [24] provides fine-grained access control based on manipulation primitives of the XML input language. Although at a higher level than providing access based on the file names, there is still an abstraction gap between the configuration specification and the access control. The manipulation primitives express what can be changed in the XML input and do not directly express what can be changed in the input specification, thus causing an abstraction gap between the access control rules and the input specification.
- PoDIM [18] includes rules to filter statements before they are applied to the network. These rules are specified at the same abstraction level as the source and apply directly to the statements in the source specification. However, there are no facilities to enforce a workflow: the specification becomes invalid and cannot be deployed if a change is added that depends on a change that is not approved.

Most tools rely on the coarse-grained access control available in version control repositories. Some tools, such as Machination [24], provide very fine-grained access control based on the configuration specification, but at a lower abstraction level than the specification a sysadmin writes. PoDIM [18] offers filtering of statements at the same abstraction level as the specification but lacks integration with workflow enforcement, thus making it hard to use. Cfengine [14] and Puppet [25] do include provisions to use different branches of version control repository for different stages in deployment in the same configuration server, but cannot enforce workflows between these stages. ACHEL solves these problems by performing access control based on semantically meaningful changes and it adds flexible workflow enforcement.

3 Design of ACHEL

ACHEL provides fine-grained access control which is applied on the semantics of configuration specification changes, as well as version tracking and workflow enforcement. Figure 2 shows a possible workflow and the access rules that can be enforced with ACHEL. This figure also provides an architectural overview of the distributed components of ACHEL. Each agent involved in the configuration of an infrastructure has his own ACHEL repository that is based on a distributed version control repository. The agents are not only system administrators or the system configuration tool, but can also be automated review or other validation services that are required by the company policy for deploying configuration updates. Section 3.1 explains the concepts and the operation of distributed version control systems.

Access control is applied on each of the repositories in Figure 2. Section 3.2 describes how this fine-grained semantic access control is implemented by analyzing the changes between versions at the language level. Finally, section 3.3 details how flexible workflows between the DVCS repositories in Figure 2 are enforced through combining features of distributed version control systems, digital signatures and fine-grained access control.

3.1 Distributed version control

ACHEL builds on a distributed version control system to provide flexible workflows [2] for updating a configuration specification. In contrast with traditional version control systems, where a central repository keeps track of all history, distributed version control systems (DVCS) use a different architecture. Instead of having to interact with a central repository to examine the history, commit changes or use branches, each user has his own local repository. This local repository not only contains a copy of the version he is working on, but also a complete

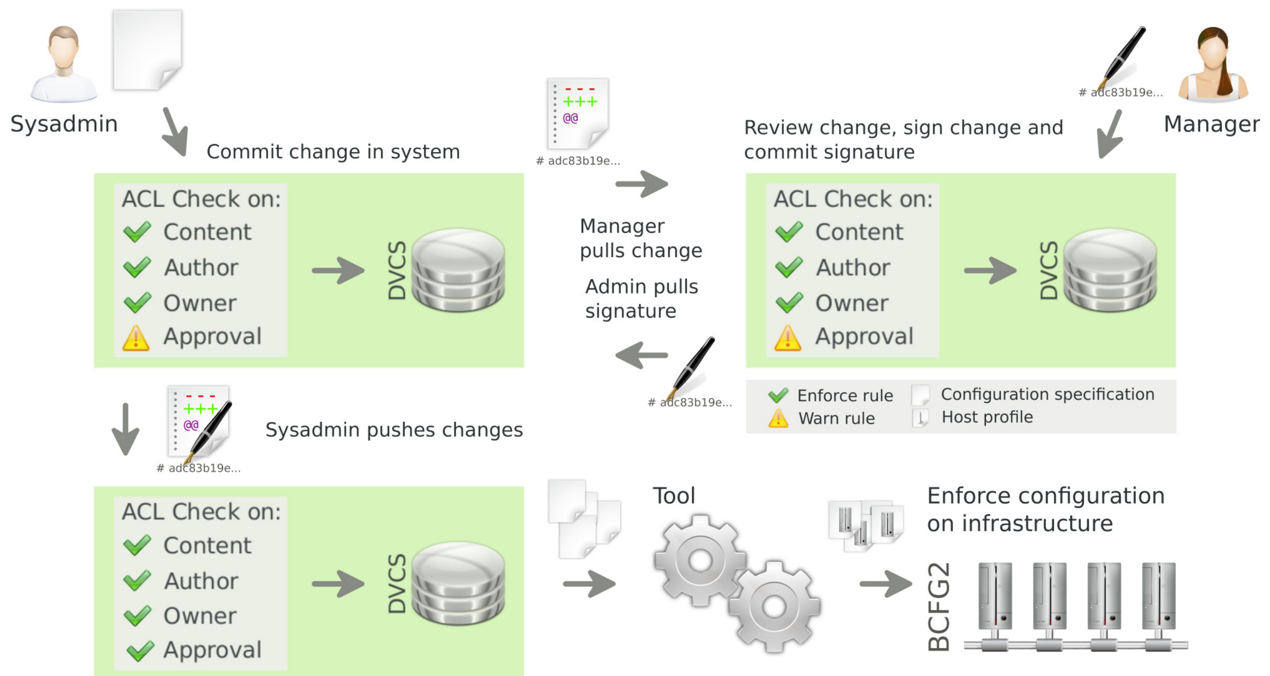

Figure 2: A possible workflow that ACHEL can enforce for the purpose of including a change in the configuration specification repository.

project history, branches, etc. All familiar version control operations such as examining the history, switching to other branches and even committing changes are local operations in a DVCS.

A DVCS enables flexible workflows because it can easily share information such as committed changes or new branches between individual repositories. Information is exchanged via push and pull operations. A push transfers local information to a remote repository, and a pull copies remote information to the local repository. When distributed repositories are used, there is no central repository, so a repository is only authoritative by convention. Another consequence of having distributed repositories is that a DVCS cannot use sequential revision identifiers as traditional version control systems do. Instead, DVCSs use a different mechanism to ensure that revision identifiers are globally unique. However, if two revisions in different repositories have the same history and introduce the same change, then the identifier needs to be equal. For example, Git [3] and Mercurial [27] both use SHA-1 hashes to identify revisions in the repository. The hash is based on the data in the files, on revision metadata and on the parent revisions. One useful consequence of this fact is that a revision identifier identifies and proves the integrity of a revision and all previous revisions. This hash is called the *revision identifier*.

ACHEL provides a sysadmin with flexible development workflows. ACHEL inherits these workflows from the DVCS it is built on. For example, a sysadmin can commit without interfering with changes from others. With authentication and authorization in the mix, flexible workflows become even more important: a user can commit changes that require authorization, without blocking the deployment of other consecutive changes that have already been approved. A DVCS can also be used in a more traditional manner whereby each sysadmin synchronizes his repository with a central *authoritative* repository, which in ACHEL is the repository the system configuration tool uses. In larger federated infrastructures a hierarchy of repositories can be used. Using a DVCS also enables sysadmins to share work with others directly. For example, two sysadmins who are preparing a new configuration for some service, share a common branch and share changes independently of the authoritative repository. Once their work is ready, it can be pushed to the authoritative repository for deployment.

3.2 Access Control

ACHEL enables fine-grained access control based on the semantics of the changes in the configuration specification. In contrast, most existing tools rely on the access control provided by the operating system or version control system. Access rules can be expressed as a function of three things: 1. the contents of a change, using semantically meaningful changes; 2. the owner of the configu-

ration statement that has been changed; 3. the author of the change. To perform access control based on semantically meaningful changes, the access control component of ACHEL needs to understand what the statements in a configuration specification are. The changes to which the access control rules are applied are generated by analyzing the differences between two statements and generating meaningful changes from these differences. In this section the access control approach used is explained; the next section elaborates on the workflow enforcement. The application of access control rules is split up into several steps, as shown in Figure 3.

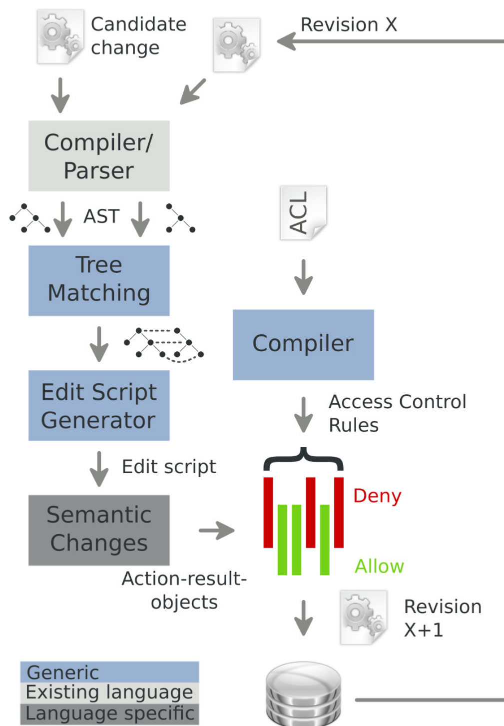

Figure 3: Architecture of the access control component that checks whether a new revision violates any access rules.

All version control systems use diff-like algorithms [30] that operate on flat files to generate changes between two versions of a file. These algorithms create what is called an edit script, which transforms the previous version of a file into the current version. Diff algorithms detect changed lines and create an edit script containing insert and remove line operations. Although this edit script is easy to generate and reapply again, it is impossible to define reasonable access control rules on the insert and remove operations. The abstraction level of these changes is too low because they are expressed in terms of adding and removing lines in a file, while the configuration specification is expressed in configuration related terms. In Listing 3, a diff generated edit script is displayed. The script transforms version 1 of the program in Listing 1 into version 2, which is listed in Listing 2.

Listing 1: Code example: version 1

```
1 var1 = 6
2 var2 = 6
3 prnt(var1 * var2)
```

Listing 2: Code example: version 2

```
1 var1 = 6
2 print(var1 * 7)
```

Listing 3: Edit script between version 1 and 2 generated the diff algorithm

```
1 @@ -1,3 +1,2 @@
2  var1 = 6
3 -var2 = 6
4 -prnt(var1 * var2)
5 +print(var1 * 7)
```

To generate semantically meaningful changes from the configuration specification, ACHEL uses an algorithm that analyses the abstract syntax tree of the configuration specification. System configuration tools build an abstract syntax tree of the configuration specification during compilation. An abstract syntax tree is a tree representation of the abstract syntax [29] of a file. The abstract syntax separates the syntax from the semantics of the specification.
An example of the abstract syntax trees of the program in Listings 1 and 2 is shown in Figure 4. The differences between two versions of a tree are used to generate changes at the right abstraction level, because the abstract syntax tree contains the structure and meaning of each statement, and the composing parts out of which the statement is built. Meaningful Change Detection in Structured Data [15] proposes an algorithm to generate these semantically meaningful changes. Several systems have been developed based on similar algorithms that calculate an edit script from unordered trees. For example, these algorithms analyze changes in source code [23], XML [7, 34], UML [31, 35], and HTML [26].

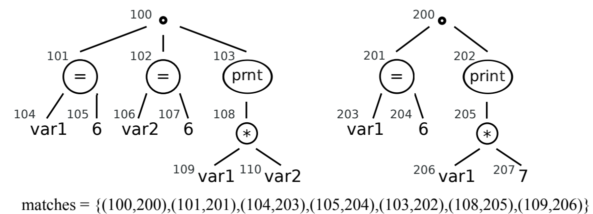

Figure 4: Matching the nodes in the abstract syntax tree of the two versions in Listing 1 and 2.

Listing 4: The edit script for transforming the first tree into the second tree in Figure 4.

```
add    = {node(text="7",parent=205)}
delete = {node(id=102),node(id=106),
          node(id=107),node(id=110)}
update = {node(id=103,text="print")}
```

ACHEL matches the nodes in two abstract syntax trees and generates a tree edit script to transform one tree into the other. It does this by parsing each revision of a configuration specification and generating an abstract syntax tree from it. Each revision of a tree is matched with its previous revision. Figure 4 shows the set (`matches`) of matched nodes obtained by applying algorithm [23] on these two trees. From these matched trees, an edit script is generated that contains a list of add, update and delete node instructions to transform the tree of revision X into the tree of revision $X + 1$. In Listing 4 the edit script of the abstract syntax trees in Figure 4 is shown. The algorithms used to generate an edit script from an abstract syntax tree are language agnostic, as shown in Figure 3.

The approach ACHEL takes until the generation of the edit script is similar to Machination [24]. ACHEL's edit script is comparable to the XML edit instructions of Machination, except that ACHEL can generate an edit script for any arbitrarily complex language. In contrast to Machination, ACHEL translates the edit script into semantically meaningful changes performed on the configuration specification. ACHEL then applies access control on these semantic changes instead of applying them on the edit script, which is of a lower abstraction level. An edit script is expressed in terms of operations on nodes in a tree, while a configuration specification is expressed in configuration related terms. Although an edit script can be generated for any tree, generation of the semantic changes is specific for each language. In section 4 we apply ACHEL's access control on a configuration specification language. The same access control system was used during prototyping to enforce access control on simple configuration files with parameters and sections, sometimes called *.ini* files.

ACHEL allows rules to be specified depending on the current owner of a statement in the configuration specification. Because all operations in an edit script are made by the same user, and this user is known to ACHEL through the DVCS repository, the owner of each statement can be determined. ACHEL determines the owner of each statement by starting at the tree of the first revision and applying the user information and the edit script until the last revision. Every time a node is added or modified, the user that made the change is used as the new owner of that node in the tree. For example, user A creates the first revision of the left tree in Figure 4, so A owns all nodes in that tree. User B makes the changes that result in the right tree in that figure. He removes nodes 102, 106, 107 and 110, updates node 103, and adds node 207. The algorithm will mark B as the owner of the new node and of the updated node 202.

The per node owner information in the abstract syntax tree is not very useful when access control rules are applied on semantic changes. The user information in the abstract syntax tree needs to be mapped onto the generated semantic meaningful changes. The owner information for each node from which a semantic meaningful change is built is used to determine the owner of a meaningful change. For example, in the multiplication statement that consists of node 205, 206 and 207, node 207 is owned by user B in Figure 4, but nodes 205 and 206 are still owned by user A. Depending on the meaning of the statement, ACHEL determines who the owner of the full statement is. In the multiplication from the example, the result of the multiplication will be different because of the change user B made, so he will be the owner of the statement. Because the author of a changeset is used to determine all ownership information, and the ownership and author are used in the access rules, the author information needs to be secure. ACHEL uses digital signatures and a PKI to ensure that all user information in the repository is authenticated.

An important step in developing an access control rule language is identifying the possible statements and their syntax in the configuration language input. Statements include for example assigning a value to a variable, calling a function or creating a new instance of a class or structure. On the statements, access control will be applied. To keep the structure of the needed patterns in the access control language simple, it is important to unify as many statements as possible in order to keep the grammar of the access control language limited. For example, the assign and multiplication statement in Listing 1 and in the abstract syntax tree in Figure 4 can be unified with a `<lhs> <op> <rhs>` structure. In this structure an operation (`op`) is performed on the left-hand side (`lhs`) using the argument on the right-hand side (`rhs`). Changes to a statement are split up into attributes. These attributes are the action performed (add, remove, modify, ...) on the statement, the type of unified statement that it is, and possibly additional attributes of a statement. In our prototype we developed an access control language for the configuration specification language we added to ACHEL.

3.3 Workflow

Each infrastructure has its own policy that defines workflows for deploying changes in the production configuration. For example, changes need to be tested in a test infrastructure before they are deployed in the production

configuration. Current configuration management tools cannot enforce these policies. ACHEL achieves this with a combination of the proposed access control system and the flexible workflows that DVCS's provide. To integrate the two, we extend the access control rules with a clause that requires authorization before a change is allowed.

This new clause specifies the number of authorizations required and a list of users that can authorize a change to be allowed in an ACHEL repository. A change is authorized by digitally signing its unique revision identifier. The identifier is signed with the user's private key. This key is also used by the author to sign information relating to a change. A signature on a revision signifies that a revision is approved by the person or service that is associated with the key used. The reader may recall from section 3.1 that the identifier in a DVCS is a hash that is based on the content and the parent revisions, so a signature also authorizes the full history of the specification.

Authorizing a revision is as simple as adding a signature to the repository metadata and committing the change. This approach generates a new revision in the repository for each new signature. This new revision can be fetched and merged by the user who requested the authorization. When the requesting user has the required number of signatures, he merges his change and the signature revisions into a new revision and pushes it to the repository of the system configuration tool. We rely on existing communication channels such as email or instant messaging for ACHEL notifications. These notifications are required when requests are sent out to authorize changes or to notify other users that a review is finished and a signature is available.

Workflows in ACHEL are based on exchanging changes between the local DVCS repositories that each user controls. ACHEL needs to enforce access control rules on each repository, even though all repositories could possibly have the same changes, because each user has full control over his own repository. This has two important advantages: First, each repository can determine who needs to approve changes before they are accepted. Second, authorization clauses can be set to only warn a user instead of denying access, because a repository revision is required to get a change authorized, and this revision is only available after a change has been included in a repository.

Figure 2 shows an example of a possible workflow. This workflow is similar to the one enforced in the access control rules in Listing 5. In this figure, the manager is a user named Alice, who is a member of the *admin* group in Listing 5. Whenever the sysadmin named Bob makes a change not related to variables named *dhcp_∗*, he needs the authorization of a user in the admin group. The system only warns about the authorization because Bob needs to be able to commit his change, although he does not yet have the authorization. After the change is committed to Bob's local repository, ACHEL enforces the following workflow to include his change in the authoritative repository:

- Bob cannot push his change to the system configuration tool repository, so he emails the users in the admin group that he needs authorization for the change with revision identifier `3d996986778d` in his repository at `http://bugatti:8000`.
- Alice, a user in the admin group, pulls change `3d996986778d` from Bob's repository into her own repository at location `http://ferrari:8000`.
- Alice reviews the change and signs `3d996986778d`, thus creating a new revision `ce5b84ef04a7` in her repository.
- Bob pulls `ce5b84ef04a7` from Alice's repository at `http://ferrari:8000` into his own repository.
- Bob creates a new merge revision `acdd701f412c` that now satisfies all access rules.
- Bob pushes `acdd701f412c` to the system configuration tool repository for the purpose of scheduling it for deployment onto the infrastructure.

Whenever a user pushes his specification changesets to another repository, other changesets that possibly conflict could have been included. When this occurs, these changesets need to be merged. During merging, two scenarios can occur. These scenarios are illustrated in Figure 5. If the changesets do not conflict with each other, a merge changeset can be created that does not introduce any additional changes. If all changesets prior to the merge satisfy all access rules, then the merge changeset can be accepted by ACHEL. Whether this changeset should be accepted is highly dependent on the configuration specification language to which ACHEL is applied. If the configuration language is fully declarative, then the order of configuration statements does not matter and changes can be applied in any order, as long as they do not conflict. If the order in which changes are applied does matter, then a special merge permission should be introduced so that only users with this permission can merge changesets.
In a second scenario some changesets can conflict, so the merge changeset needs to include additional changes to resolve the conflicting changesets. If this occurs, then these new changes need to satisfy all access control rules, just like any other normal change. Such a merge changeset can also include non-conflicting changesets, in which case the same restrictions apply as in the other scenario.

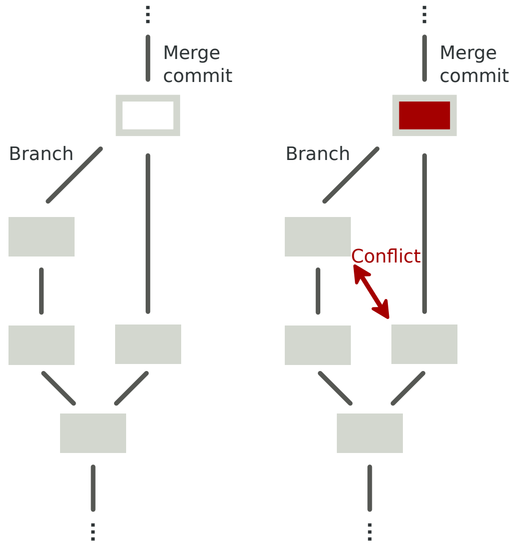

Figure 5: Merging branches of a configuration specification repository. Left: a branch is merged without any conflicts. Right: two changesets conflict and the merge changeset contains additional changes to resolve the conflict.

Conflicting changesets should not occur very often in a real infrastructure. Because responsibilities are mostly non-overlapping, modularizing the configuration specification in files that do not contain overlapping responsibilities will prevent conflicting changes, insofar as this is possible.

4 Prototype

This section describes a prototype configuration language and compiler: Westmalle. Westmalle is a simple configuration language that we extended with ACHEL support.

The Westmalle configuration language design is inspired by the functionality provided by LCFG2. The Westmalle configuration language is a declarative language with only three operations. It can *import* configuration directives from libraries, it can *add* values to lists and it can *assign* values to a variable name. Because of its declarative nature, variables can only be assigned once. Various values can be assigned to these variables, including string literals, functions that can interface with other systems such as template systems, and structures with named attributes.

Structures are an important part of Westmalle. Each structure has a list of named attributes. Structures are not declared and are created in the same way as classes are instantiated in languages such as Python. Attributes are added every time a value is assigned to an unknown attribute. Westmalle does not type check these structures, but therein lies its strength. Because classes are easily created, a user can create specifications that match what, for example, LCFG2 or BCFG2 require. When the compiler finishes resolving all variables in the specification, then these structures and their attributes are used to generate the output. The compiler can be used in two ways:

1. As a BCFG2 plug-in that exposes the return values of the BCFG2 client probes [20] as variables in the configuration language. These probes are used in BCFG2 to find information about the managed system.

2. Generating XML output that conforms to the structure proposed in Configuration tools: Working together [12].

We added support for ACHEL to Westmalle. The architecture of our prototype is shown in Figure 6. The right box is the Westmalle compiler, the left box is the ACHEL support infrastructure. The ACHEL support infrastructure contains a distributed version control repository at the bottom and a set of allow/deny access control rules. If a user wants to push a change to the repository, that change has to pass the access control rules before it is committed to the repository. Once the change is committed, the Westmalle compiler generates XML or BCFG2 specifications, which in turn can be used by a deployment engine to enforce the specification.

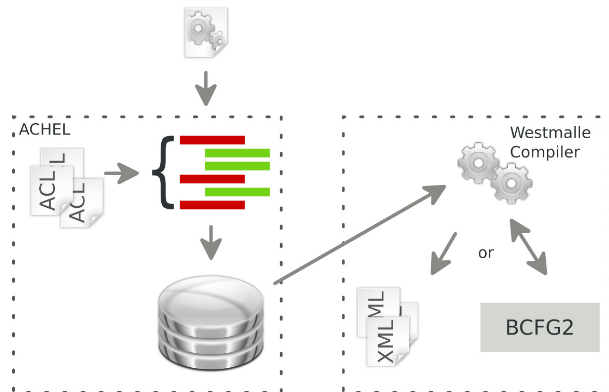

Figure 6: An architectural overview of our ACHEL prototype.

The access control rule enforcement is implement as show in Figure 4. The access control language defines pattern matching rules that match the attributes generated from the semantic changes in the configuration specification. Each rule contains an action that is taken if the patterns from the rule match the attributes of a change. Access control rules are evaluated in the order they are declared in the source file. When a rule matches, evaluation stops and the action of that rule is taken.

The rules in the access control language consist of two different parts. The first is a generic part that specifies the author, the owner, the required signatures and the action taken when a rule matches. The second part is language-specific and contains the structure the semantically meaningful edit instruction needs to match before an action will be executed. In the access control language, user groups can be defined and each user can be included in multiple groups.

Listing 5: Access control rules and group definitions

```
1  # list of senior admins
2  define admins as
       admin1@cs.kuleuven.be,
       admin2@cs.kuleuven.be
3
4  # allow everyone to create dhcp
5  # configuration
6  allow to:
7      add import dhcp.*
8      * assign * to dhcp_*
9
10 # senior admins can do anything
11 allow admins to:
12     *   *
13
14 # others can do anything if
15 # approved by a senior admin
16 allow to:
17     *   *
18     authorised by 1 admins
```

Listing 5 shows an example of possible access control rules.

- Line 2 defines a group of users. This group can be used in the access control rules.

- Line 6 starts a new rule with two language specific rules on the next two lines. The rule allows all changes that match one of the language specific sub-rules on the next lines. The first sub-rule on line 7 matches all changes that *add* an *import* statement. This import statement is constrained to libraries matching the wildcard string *dhcp.**. The second sub-rule on line 8 matches changes that *assign* any value (hence the second wildcard) to a variable that matches the wildcard string *dhcp_**. The first wildcard of this sub-rule signifies that these changes can be *added, modified or removed*.

- The rule on line 11 allows changes that match any of the sub-rules, if the author of the change is a user from the admin *group*. The sub-rule on line 12 matches every change.

- The rule on line 16 allows any change (the two wildcards on line 17) by any author, if it is authorized by one or more users from the administrator group (line 18).

A set of rules such as in Listing 5 starts with a generic header that expresses the action taken when a rule matches and lists the possible authors of the change. In the prototype, actions are limited to *deny* or *allow* the change. The action of the first matching rule is used and no other access control rules are checked. After the action, an optional list of authors of a changeset can be given, followed by the *to* keyword, a colon and a newline. The list of users is a comma separated list of email addresses or group names to allow role based access control.

The body of the rule can contain multiple directives that are divided into generic and language specific lines. The generic lines start with the *owned by* and *authorized by* keywords. The *owned by* clause specifies a list of owners of the original statement required for this rule to match. This list is identical to the author list described in the previous paragraph. The *authorized by* clause is followed by an optional number and a list of users. The number indicates how many signatures are required for a rule to match. If no number is provided, the number one is assumed.

The other rules in the body are language specific rules. These rules start with *add, modify or remove*, followed by a pattern to match a meaningful change. The first keyword specifies the change made in the changeset required for this rule to match. If multiple language specific rules are present in the body of a rule, they are treated as separate rules with a common generic part. For example, in Listing 5 for the rule on line 6, two rules with the same header are created because of the two language specific rules on the next lines. For each different structure in the configuration language, a different language specific rule syntax is required. The number of structures in a language depends on how similar statements in the configuration language are. Each rule syntax consists of matching patterns that are applied to the matching attributes of a semantic change. Each attribute is matched with a literal value, a string that may contain wildcards, or a regular expression.
For the simple configuration language we developed, we are able to unify all statements within the same structure. This means that we can match all statements in the configuration specification with the same rule syntax. Rule syntax contains at most three patterns to match the three attributes of each change to the configuration specification. These attributes are: the operation, the right-hand

side, and the left-hand side. The most important attribute of the configuration language is the *operation* of a statement. In the prototype there are only three available operations: import, assign and add. After the operation, a rule can also contain an optional right-hand side, as well as an optional left-hand side pattern.

ACHEL uses hg (short for Mercurial [27]) as the version control system in our prototype. It is a lightweight DVCS written in Python. It has been adopted by very big software projects such as OpenJDK and Mozilla. Like other well known DVCS control systems such as Git [3] and BitKeeper [1], hg uses content hashes as identifiers. Mercurial was selected for our ACHEL prototype because it is open source, it is written in Python and it has extensions for signing revisions.

ACHEL can merge changesets. A merge is handled differently depending on the scenario:

- If changesets have been merged without conflict and the merged branches introduce changes in different files, no additional permissions are required.
- If changes are introduced in the same file, a user that commits the merge changeset needs merge permissions. Merge permissions can be dependant on authorization, so they can be forced to go through a review process.
- If changesets conflict, then the new changes that solve the conflict need to satisfy the access control rules. The two previous rules apply to any non-conflicting changes that are also merged.

Digital signatures are used to verify the users in the author, owner, and authorization constructions in the access control rules. In the prototype, GPG is used to generate these digital signatures. The email addresses linked with the private key that is used to create digital signatures are used in the access control rules to identify users. GPG was chosen over x509 certificates for two reasons: First, because Mercurial already has support for signing revisions with GPG signatures. Second, because it is very lightweight and works well in a federated environment. To establish trust, GPG can be configured to only trust signatures from keys within a certain trust level.

ACHEL can support other DVCS and infrastructures for digital signatures. It uses the version control repository to report the email addresses of the users that created or signed a changeset and it relies on the ability of the repository to verify the user information on the basis of digital signatures. This way there is no direct coupling between the PKI used and ACHEL. Because the interface between ACHEL and the DVCS is small, new distributed version control systems are easy to add.

5 Evaluation

In this section we evaluate ACHEL in two case studies. The first case validates the improvements achieved in ACHEL for environments where several administrators manage the same infrastructure but are responsible for different aspects of the infrastructure. This case focuses mainly on the access control features with limited workflow enforcement. The second case validates the use of ACHEL in federated infrastructures with a focus on workflow, such as used in grid computing.

5.1 Case 1

This first case validates the prototype in an infrastructure managed by several system administrators with varying levels of seniority, each of whom has his own responsibilities. The update policy of this infrastructure is:

- Sysadmins can only make changes to the aspects of the infrastructure they are responsible for.
- Everyone can make any change if it is authorized by a senior sysadmin.
- Senior sysadmins can change anything.

The access control rules use group names to identify users, so users can be assigned according to their responsibilities. In Listing 6, an excerpt with access control rules is shown. The first rule on line 2 forces every change to encode the type in the variable name. For example, the variable with the configuration file for the dhcp server should be called `net_file_dhcpd_conf`. Six rules are declared for each type that can be used with the BCFG2 plug-in. If a variable does not match this convention, the change will be denied. The second rule on line 11 stipulates that every user in the *senioradmin* group can do anything. Statements in a change will only get to this rule if they conform to the previous rule. The rule on line 15 authorizes any change from any user if it has been approved by a user from the senioradmin group.

The next two rules are specific for users in the *netadmins* group. These users are only allowed to change the network configuration related to files located in `/etc/network` and services called `network` and `dhcpd`. The rule on line 26 allows a netadmin to import dhcp configuration libraries, to add entries to the global list of dhcp clients and to declare variables that are prefixed with `net_`. The rule on line 20 limits this to the files and services that are allowed. If none of the rules above matches, the change is denied.

Listing 7 shows a possible configuration an admin responsible for the network configuration has written in our configuration language. This code example creates a configuration file called

Listing 6: Case 1: Access control rules and group definitions

```
# enforce some conventions on everyone
deny to:
  * assign  File()          to  /^[^_]+_(?!file_)[\S]+$/
  * assign  Package()       to  /^[^_]+_(?!pkg_)[\S]+$/
  * assign  Service()       to  /^[^_]+_(?!service_)[\S]+$/
  * assign  Directory()     to  /^[^_]+_(?!dir_)[\S]+$/
  * assign  Symlink()       to  /^[^_]+_(?!ln_)[\S]+$/
  * assign  Permissions()   to  /^[^_]+_(?!perm_)[\S]+$/

# senior admins can do anything else
allow senioradmin to:
  * * *

# allow admins to do everything if a senior admins approves
allow to:
  * * *
  authorised by 1 senioradmin

# network related configuration
deny netadmins to:
  # deny files other then those in /etc/network
  * assign /^(?!\/etc\/network\/)\S+/ to /^net_file_\w+\.name$/
  # deny services other then dhcpd and network
  * assign /^(?!(dhcpd$|network$))\w+$/ to /^net_service_\w+\.name$/

allow netadmins to:
  * import  /^dhcp/
  # allow adding a list of values to the net_dhcp_clients list
  * add     /^\[[^\]]$/ to  /^net_dhcp_clients$/
  # allow only variables prefixed with net (ignore rhs)
  * assign  *           to  /^(?!net_)\S+$/
```

/etc/network/interfaces and enables the network service. This is allowed by the access control rules defined in Listing 6. The code starting from line 17 creates a /etc/hosts file. To do this, a user needs to be either a senior admin himself or else he needs to have the permissions of a senior admin.

A network administrator named Kris, who is not a senior administrator, wants to include his configuration specification in the main repository. To do this he needs permission from a senior administrator named Jean. Kris commits the change to his ACHEL repository and receives a warning that he needs permission from a member of the senioradmin group to satisfy the access control rules enforced on the main repository. Kris emails Jean and asks him to review the change with identifier c8d8c7780069 in Kris' repository, which is available at http://alpha:8000. Jean pulls this change from Kris' repository and ACHEL warns Jean that the change needs to be authorized. Jean reviews the change, signs it and commits the signature to his repository. Jean then emails Kris that he approves the change and that Kris can pull the signature with identifier 3a526359364e from his repository, which is available at http://beta:8000.

Kris pulls Jean's signature into his repository and ACHEL now shows that all changes satisfy the access control rules. Kris can now push his change and Jean's signature into the main repository in order to deploy it on the infrastructure.

5.2 Case 2

Quattor is used in federated infrastructures composed of multiple physical sites. Devolved Management of Distributed Infrastructures [16] explains how Quattor is used in a few different federated infrastructures. In this case we apply ACHEL to the BEGrid infrastructures described in [16]. BEGrid uses a model of highly-autonomous sites that loosely collaborate. Each site has its own configuration servers, but it gets its configuration from a central Subversion repository. In this repository, both common and site-specific configuration specifications are stored.

In this case we will focus on the application of ACHEL's workflow enforcement features on a BEGrid-like infrastructure. Each site configuration server uses its own repository instead of getting its configuration specification from the central repository. Each site has its own authoritative ACHEL repository from which the site configuration server gets its specification. The institution that coordinates the grid also maintains a repository from which each site updates their common configuration specification. In Figure 7 the flow of changes between repositories is shown. The workflow between the repositories in Figure 7, is already supported by any normal DVCS.

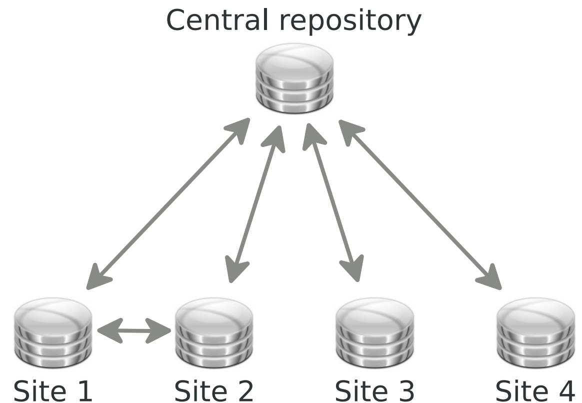

Figure 7: BEGrid repositories using ACHEL

In this case we will enforce policy rules on this workflow using ACHEL:

- Every site has its own ACHEL repository that is used by the site's configuration server.
- System administrators commit to the site repository, possibly with extra policy rules specific for their own site.
- Sysadmins can only change the configuration related to their own site.
- Changes by sysadmins from other sites have to be approved by a manager of the affected site.
- Changes to the common templates have to be approved by at least three out of five managers.

Our configuration language does not match the PAN [17] language used by Quattor in BEGrid, but this is not an issue for demonstrating the workflow capabilities of ACHEL. For the access control rules in Listing 8, we assume that the site-specific and common configuration can be identified by the first word in the variable name: common_ or site_. On lines 1-5 of the listing groups are defined containing the sysadmins of each site and all the managers. These groups are used in the access control rules we describe next.

The rules in Listing 8 implement the policy we described earlier in this section. The first rule on line 7 stipulates that changes to the *common* configuration need authorization from at least three users from the management group. Lines 11-16 define the access rules for the configuration of site 1. The first rule limits access to users in the site1 group, and the second rule stipulates that other users have access to site 1 configuration only if they have authorization from the site1 manager. The

Listing 7: Case 1: Network configuration specification

```
import base

# configure network interfaces
net_file_interfaces = File()
net_file_interfaces.name = "/etc/network/interfaces"
net_file_interfaces.owner = "root"
net_file_interfaces.group = "root"
net_file_interfaces.perms = "0644"
net_file_interfaces.content = source("net/interfaces.$hostname")

# network service needs to be enabled
net_service_network = Service()
net_service_network.name = "network"
net_service_network.status = "on"

# use template for /etc/hosts with loopback and host ip
net_file_hosts = File()
net_file_hosts.name = "/etc/hosts"
net_file_hosts.owner = "root"
net_file_hosts.group = "root"
net_file_hosts.perms = "0644"
net_file_hosts.content = template("net/hosts.tmpl")
```

rules on lines 18-37 provide for similar rules for the other three sites. Site 1 and site 2 are developing a new common feature defined under `common_new`. The last rule on line 39 allows users of both sites to work on it. This rule stipulates that users from the groups site 1 and site 2 have access to all configurations under `common_new`.

Each repository can have its own access control rules in ACHEL. For example, the last rule on line 39 only makes sense in the repositories of sites 1 and 2. This also holds for other the site-specific rules, which only need to exist at the site itself and at the main repository.

5.3 Limitations and Future Work

In this paper we have prototyped ACHEL on a simple configuration language. One area for future work involves the need to add ACHEL support to existing (more complex) configuration languages. Another area involves the need to improve the usability of ACHEL by adding support for processing authorization requests. A third area involves the need to provide support for meta-ACL's: i.e. to provide access control rules for specifying the access control rules.

Supporting existing configuration languagues

ACHEL itself is not a product, it is a generic framework that can be reused in existing configuration languages. The framework offers support for meaningful change detection, and for the enforcement of access control rules and workflow. To add support for ACHEL to an existing configuration tool, it must:

1. either add access control constructs to its language or use a separate access control language;
2. and provide ACHEL with an abstract syntax of its configuration specification.

The complexity of an access control language is directly linked with the number of different semantically meaningful change structures that need to be matched. The reader will recall that in our prototype we were able to unify all three language constructs within a single structure, but this will no longer be possible for more complex configuration languages. The expressions in Listing 6 are very powerful because they use regular expressions, but they are complex to use. If ACHEL were to be applied to more complex configuration languages such as Cfengine [14] or Puppet [25], the current access control language and matching model would probably become needlessly complex. These languages are more expressive and contain more than one structure that needs to be matched.

To support more complex languages and make the access control language easier, some enhancements are required. The first enhancement would be to use the type system of the configuration language to enforce rules, so that types and namespaces do not need to be encoded in the names of the variables as in the first case. Another required enhancement of the access control lan-

Listing 8: Case 2: BEGrid example access control rules

```
1  define management as
       director@begrid.be,
       manager@site1.begrid.be,
       manager@site2.begrid.be,
       manager@site3.begrid.be,
       manager@site4.begrid.be
2  define site1 as ...
3  define site2 as ...
4  define site3 as ...
5  define site4 as ...
6
7  allow to:
8    authorised by 3 management
9    * * * to /^common_/
10
11 allow site1 to:
12   * * * to /^site1_/
13
14 allow to:
15   authorised by
         manager@site1.begrid.be
16   * * * to /^site1_/
17
18 allow site2 to:
19   * * * to /^site2_/
20
21 allow to:
22   authorised by
         manager@site2.begrid.be
23   * * * to /^site2_/
24
25 allow site3 to:
26   * * * to /^site3_/
27
28 allow to:
29   authorised by
         manager@site3.begrid.be
30   * * * to /^site3_/
31
32 allow site4 to:
33   * * * to /^site4_/
34
35 allow to:
36   authorised by
         manager@site4.begrid.be
37   * * * to /^site4_/
38
39 allow site1, site2 to:
40   * * * to /^common_new_/
```

guage would be to include a mechanism for abstracting certain details: for example, a mechanism for using templates to match certain structures and to hide the complexity of the regular expressions used. A final enhancement would make it possible to combine rules using different operators.

The second requirement for ACHEL integration is that ACHEL should be provided with an abstract syntax tree. If a configuration tool includes a formal grammar definition of its language, this grammar can be reused. Another option is to dump the internal abstract syntax tree structures of a system configuration tool.

Supporting authorization request processing

Currently, ACHEL relies on existing communication for notifications related to authorization requests and signatures. To improve the usability of ACHEL in real infrastructures, a plug-in or tool is needed for automatically processing authorization requests by pulling the change, requesting a signature from the user, committing the signature and notifying the author of the signed change. ACHEL should also support multiple DVCS's and digital signatures for real world deployment in order to match existing practices and tools in an infrastructure. Bugtracker tools could also be useful in real world applications, because some bugtrackers include DVCS integration.

Meta-ACL's

Our prototype does not contain support to enforce access control on the access control language itself. The access control available in the underlying distributed version control system can in most cases be reused, because typically only a few users in an infrastructure are allowed to define policy related rules. Technically it is possible to apply the same change detection approach to the access control language itself and provide a meta access control language. This would also make it possible to implement a mechanism for delegating permissions.

6 Conclusion

ACHEL provides integration of fine-grained access control with existing configuration tools and it can enforce configuration change workflows in federated infrastructures. Moreover, ACHEL combines these capabilities with distributed version tracking and cryptographic secure authentication. It also makes access control rules easier to write because they are defined at the same abstraction level as the configuration specification language. And finally, because large parts of ACHEL are

language agnostic, support for new configuration languages can be added with minimal effort.

7 Acknowledgments

We would like to thank Wouter De Borger and Stefan Walraven for proofreading this paper. We also thank Mark D. Roth for his work on shepherding this paper this paper and the anonymous reviewers for their valuable feedback.

References

[1] BitKeeper Website. http://www.bitkeeper.com, 2007.

[2] Bazaar version control: Workflows. http://bazaar-vcs.org/Workflows, 2008.

[3] Git Website. http://git-scm.com/, 2009.

[4] Keep your Puppet manifests under version control. http://reductivelabs.com/trac/puppet/wiki/VersionControlPuppet, 2009.

[5] Puppet Change Management. http://reductivelabs.com/trac/puppet/wiki/ChangeManagement, 2009.

[6] Using Multiple Environments in Puppet. http://reductivelabs.com/trac/puppet/wiki/UsingMultipleEnvironments, 2009.

[7] AL-EKRAM, R., ADMA, A., AND BAYSAL, O. diffX: an algorithm to detect changes in multi-version XML documents. In *CASCON 05: Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research* (2005), IBM Press, pp. 1–11.

[8] ANDERSON, P. *The Complete Guide to LCFG*, 2003.

[9] ANDERSON, P. *Short Topics in System Administration 14: System Configuration*. Berkeley, CA, 2006.

[10] ANDERSON, P. LCFG: A large scale UNIX configuration system. http://www.lcfg.org, 2008.

[11] ANDERSON, P., AND COUCH, A. What is this thing called System Configuration? *LISA Invited Talk* (November 2004).

[12] ANDERSON, P., AND SMITH, E. Configuration tools: Working together. In *Proceedings of the 19th Large Installations Systems Administration (LISA) Conference* (Berkeley, CA, USA, 2005), USENIX Association, pp. 31–38.

[13] BURGESS, M. Cfengine: a site configuration engine. *USENIX Computing Systems 8*, 3 (1995), 309–402.

[14] BURGESS, M. Cfengine Website. http://www.cfengine.org, 2009.

[15] CHAWATHE, S. S., AND GARCIA-MOLINA, H. Meaningful change detection in structured data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data - SIGMOD 97 SIGMOD 97* (New York, NY, USA, 1997), ACM, pp. 26–37.

[16] CHILDS, S., POLEGGI, M. E., LOOMIS, C., MEJAS, L. F. M., JOUVIN, M., STARINK, R., DE WEIRDT, S., AND MELI, G. C. Devolved Management of Distributed Infrastructures With Quattor. In *Proceedings of the 22nd Large Installation System Administration (LISA) Conference* (Berkeley, CA, USA, 2008), USENIX Association, p. 175189.

[17] CONS, L., AND POZNANSKI, P. Pan: A high-level configuration language. In *Proceedings of the 16th USENIX Conference on System Administration (LISA)* (Berkeley, CA, USA, 2002), USENIX Association, pp. 83–98.

[18] DELAET, T., AND JOOSEN, W. PoDIM: A language for high–level configuration management. In *Proceedings of the 21st Large Installation System Administration (LISA) Conference* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–13.

[19] DESAI, N. Bcfg2: A Pay as You Go Approach to Configuration Complexity.

[20] DESAI, N., BRADSHAW, R., AND HAGEDORN, J. *Bcfg2 Manual*, July 2006.

[21] DESAI, N., BRADSHAW, R., HAGEDORN, J., AND LUENINGHOENER, C. Directing change using Bcfg2. In *Proceedings of the 20th Large Installation System Administration (LISA) Conference* (Berkeley, CA, USA, 2006), USENIX Association, pp. 215–220.

[22] DESAI, N., BRADSHAW, R., MATOTT, S., BITTNER, S., COGHLAN, S., EVARD, R., LUENINGHOENER, C., LEGGETT, T., NAVARRO, J.-P., RACKOW, G., STACEY, C., AND STACEY, T. A case study in configuration management tool deployment. In *Proceedings of the 19th Large Installation System Administration (LISA) Conference* (Berkeley, CA, USA, 2005), USENIX Association, pp. 39–46.

[23] FLURI, B., WUERSCH, M., PINZGER, M., AND GALL, H. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering 33*, 11 (2007), 725–743.

[24] HIGGS, C. Authorisation and Delegation in the Machination Configuration System. In *Proceedings of the 22nd Large Installation System Administration (LISA) Conference* (Berkeley, CA, USA, 2008), USENIX Association, pp. 191–199.

[25] KANIES, L. Puppet Website. http://reductivelabs.com/projects/puppet/, 2008.

[26] LIM, S.-J., AND NG, Y.-K. An Automated Change-Detection Algorithm for HTML Documents Based on Semantic Hierarchies. *Data Engineering 0* (2001).

[27] MACKALL, M. Towards a Better SCM: Revlog and Mercurial.

[28] MATES, J. Storing CFEngine configuration in CVS. http://sial.org/howto/cfengine/repository/, 2009.

[29] MCCARTHY, J. Towards a mathematical science of computation. *Information Processing 62* (1962), 21–28.

[30] MYERS, E. W. An O(ND) difference algorithm and its variations. *Algorithmica 1*, 1 (1986), 251–266.

[31] OHST, D., WELLE, M., AND KELTER, U. Differences between versions of UML diagrams. In *Proceedings of the 9th European Software Engineering Conference, held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering - ESEC/FSE 03 ESEC/FSE 03* (New York, NY, USA, 2003), ACM, pp. 227–236.

[32] ROUILLARD, J. Distribution and Configuration System. http://www.cs.umb.edu/~rouilj/DACS/, 2009.

[33] TRAUGOTT, S., AND HUDDLESTON, J. Bootstrapping an Infrastructure. In *Proceedings of the 12th USENIX Conference on System Administration (LISA)* (Berkeley, CA, USA, 1998), USENIX Association, pp. 181–196.

[34] WANG, Y. X-Diff: an effective change detection algorithm for XML documents. In *Proceedings 19th International Conference on Data Engineering (Cat No 03CH37405) ICDE-03* (Los Alamitos, CA, USA, 2003), vol. 0, IEEE Computer Society, p. 519.

[35] XING, Z., AND STROULIA, E. UMLDiff: an algorithm for object-oriented design differencing. In *ASE 05: Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering* (New York, NY, USA, 2005), ACM, pp. 54–65.