Nr.: FIN-004-2010

Utilizing Internal Domain-Specific Languages for Deployment and Maintenance of IT Infrastructures

Sebastian Günther, Maximilian Haupt, Matthias Splieth

Very Large Business Applications Lab

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Technical Report

Nr.: FIN-004-2010

Utilizing Internal Domain-Specific Languages for Deployment and Maintenance of IT Infrastructures

Sebastian Günther, Maximilian Haupt, Matthias Splieth

Very Large Business Applications Lab

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG):

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Prof. Dr. Hans-Knud Arndt
Postfach 4120
39016 Magdeburg
E-Mail: hans-knud.arndt@iti.cs.uni-magdeburg.de

http://www.cs.uni-magdeburg.de/Preprints.html

Auflage: xx

Redaktionsschluss: 31.03.2010

Herstellung: Dezernat Allgemeine Angelegenheiten,
Sachgebiet Reproduktion

Bezug: Universitätsbibliothek/Hochschulschriften- und
Tauschstelle

Utilizing Internal Domain-Specific Languages for Deployment and Maintenance of IT Infrastructures

Sebastian Günther, Maximilian Haupt, and Matthias Splieth

University of Magdeburg, School of Computer Science
sebastian.guenther@ovgu.de,
maximilian.haupt@st.ovgu.de,
matthias.splieth@st.ovgu.de

Abstract. The deployment and maintenance of IT Infrastructures is a complex task. Systems and components need to be selected, configured, installed and maintained in different configurations spanning multiple physical and virtual machines. Further requirements are the growing demand in computational performance, cost pressure to migrate physical machines to virtual ones, and increased variant setups to support testing. These challenges require adequate tool support.
In this paper, we present and use a special tool: Domain-Specific Languages (DSL). DSLs comprise a domain as an executable language. Using appropriate domain-specific expressions and notations, they express even complex conditions. Several tools in the context of IT Infrastructure deployment and maintenance exist, but none of them provide an integrated support for the following tasks: (1) To initially setup physical or virtual machines, (2) to configure and install arbitrarily software packages on the machines, and (3) to support the user in selecting appropriate software packages. We analyzed the tools and the particular requirements of each task, and created an internal DSL to support them. Integrating the DSL into each other leads to a declarative description of IT Infrastructures. How to use the DSLs is shown in a case study where we deploy two servers and several web applications.

1 Introduction

IT Infrastructures are an important cornerstone: servers provide real-time messaging, support business operations, and deliver electronic books and documents. However, infrastructure deployment and maintenance is not a trivial task [15]. Different applications, technologies, and protocols, implemented with several programming languages, have to be managed in a consistent way. Additional demands, like increasing computer performance or cost pressure to switch from physical to virtual machines, imply further burdens. These challenges require adequate tool support. In this paper, we choose to use Domain-Specific Languages (DSL) as the particular tool.

DSLs are specialized languages which comprise a specific application area [26]. They express domain knowledge in the form of an executable language. Their characteristic feature is the use of domain-specific notations and abstractions [41]. Sophisticated DSLs express their domain with a clear, readable language.

DSLs have matured into a common software engineering technique. Earlier DSL examples support financial products [1], signaling installations for rails [21], and video device drivers [39]. Recent domains where DSLs have been applied successfully are healthcare systems [33] and web applications [42]. DSLs can also be used to support software development itself, for example to model software product lines [23] or to ease feature-oriented programming [24].

Our research background is engineering internal DSLs that are built on top of another programming language. These DSLs benefit from the existing language infrastructures such as editors or code optimizers [32]. We deal with the question how DSLs are designed, implemented, and utilized in software engineering. In a recent contribution we collected our insights and practical experiences to form an engineering process for internal DSL using dynamic programming languages [22]. We use this process to design three distinct DSLs that support deploying and maintaining IT Infrastructures. The starting point is to identify or install physical and virtual machines with different UNIX-like operating systems. This is supported by the *Boot-DSL*. Having the machines operable, the next task is to configure the software packages that are needed for the infrastructure. With the help of the *Configuration Management DSL* (CM-DSL), package objects are provided with operating system independent configurations detailing their installation, several configuration parameters and metadata. Having a broad set of packages, the final task is to choose the particular application set, resolve all dependencies, and deploy the applications within the infrastructure – this is the task of the *Software Deployment Planning DSL* (SDP-DSL). Once the infrastructure change is finished, CM-DSL and SDP-DSL can be used to update packages or install new ones. The combination of Boot-DSL and SDP-DSL expands or consolidates the infrastructure's machines.

The paper's goal is to present a DSL utilization case study in the context of IT Infrastructure deployment and maintenance. The particular focus is first to show how to abstract the complex domain by using DSLs, and second how tasks concerning the deployment and maintenance of infrastructures are supported by DSL expressions. The DSL utilization is shown with an infrastructure case-study that deploys two servers and several web applications.

Section 2 provides a concise summary of the utilized DSL-engineering process. Section 3 explains the infrastructure deployment and maintenance domain with its terminology, tools and existing languages. Section 4 explains the design and implementation of the three developed DSLs. Section 5 shows and explains how the languages work together by deploying an architecture containing two servers and several applications. Section 6 summarizes the paper.

2 Engineering Internal Domain-Specific Languages

From a top-down perspective, literature proposed several processes how to engineer Domain-Specific Languages. A general approach is described by Mernik et al. His process consists of the following five phases [32]:

- **Decision** – The first point is to decide whether a DSL should be implemented generally. On the one hand, there are criteria like the required investment in terms of man hours, budget, and possible tool support. And on the other hand, there are enhanced productivity, future reuse and better domain understanding. Both parts are weighted against each other, and finally the decision is made.
- **Analysis** – The analysis is a common part in software engineering. In the case of DSL engineering, its role is the understanding and gathering of domain knowledge. This knowledge is represented in the form of a domain model that scopes the domain and defines the used vocabulary.
- **Design** – Development continues with *designing* the DSL. One can use an existing language and specialize or extend it, or design a language from scratch with its own syntax and semantics expressed with formal methods.
- **Implementation** – Once the language design is established, the implementation is the final step to complete the development. The available options are to use an interpreter, compiler or embedding the DSL into another language.
- **Deployment** – This phase is only briefly mentioned by Mernik et al., we understand it as the actual DSL usage and adjoining tasks like developer training.

In our view, this process provides valuable insights in the required effort to implement a DSL. The following case-studies for DSL engineering highlight details of language design. They complement this general framework with specific techniques.

For external DSLs, the early PSG [2] and PAN [3] language development environments use a complex set of formal languages to describe the syntax and semantics of DSLs. The tools even generate stand-alone development environments to work with the designed languages. Consel and Marlet explain a formal approach to language design too. They define the static and dynamic semantics in a devotional form using an abstract machine [9].

For internal DSLs, Cunningham presents a DSL for surveys (as inspired by [4]) written in Ruby. The design phase uses a form of commonality and variability analysis. In the implementation, metaprogramming mechanisms like evaluating strings containing source code at runtime or creating non-existent methods at runtime [13]. Dinkelaker and Mezini use a detailed analysis and design process. The language's syntax is specified in a Backus-Naur-Form. Then the non-terminals are implemented successively, starting with the overall internal interpreter, and continuing to provide all Domain Objects and Domain Operations. They use the *Groovy* Programming Language and its support for closures [16].

By further consolidating the work of [14], [43], [38], [32], and by applying our own DSL-engineering knowledge [24], [23], we developed a specialized process for engineering internal DSLs.

The process consists of several iterations in the phases of *domain design*, *language design*, and *language implementation.* It has an *agile nature* – iterations and their tasks are run in an arbitrary order, but follow the goal to provide a working implementation at the end. Automated tests and *constant refactoring* provide the necessary infrastructure for satisfying requirements and features. *Patterns* capture recurring DSL-engineering problems and their solutions and thus further leverage developer knowledge. Since the full details of the process are contained in [22], we will here just explain the three process steps and the three most important principles. ►Figure 1 graphically represents this process.

2.1 Domain Design

The first goal in DSL-Engineering is to understand the domain. We gather *domain material* – handbooks, documentation, systems and stakeholder expressions. The material is studied to produce either formal or informal expressions about the domain. We could use domain engineering techniques like Feature-Oriented Domain Analysis [14], or variability and commonality analysis to collect statements about the domain in natural language [13]. When no written documentation exists and stakeholder are the sole source of knowledge, techniques like brainstorming or more formal questionnaires [12] should be used.

We refine the knowledge and represent it as a *domain model.* This model defines the *concepts*, *attributes*, and *relationships* that are important in the domain. The model supports two goals: To better understand the domain and to provide an initial vocabulary used in development and supposed as expressions in the DSL. Therefore, contradicting statements and possible *language defects* like synonyms, homonyms and more [29] should be fixed immediately.

While the domain model can be seen as the *static structure* of the domain, the *dynamic structure* is important too. The conditions that change the domain model's status have to be expressed too. Typically, interaction of different domain concepts – respectively their instances called *domain objects* – are studied to define further *domain operations.*

Our process does not prescribe concrete modeling techniques or tools. We see it as vital that the developers use those techniques and tools they already know. The less modifications the process imposes, the more likely can it fit into existing mindsets and processes, and thus the process success is likelier. For example, if the developers are versatile with UML, they can use UML class diagrams for the static structure of entities, attributes and relationships, and the UML state diagram to represent the different status of the domain.

2.2 Language Design

The language design phase is a creative process in which a number of language expressions are collected. The expressions reflect the domain model and the overall vocabulary. Furthermore, expressions need to be valid statements in terms of the host language.

Two principal approaches are available. The first one is to design expressions without the host language in mind, and to make them host language compatible afterwards. A useful metaphor is that of a *language game*. The philosopher WITTGENSTEIN used language games to determine the grammatical correctness of natural-language expressions [27]. Such language games can be used with a compiler or interpreter. If a DSL expression raises only semantic errors, then it is syntactically valid according to the host language. The second approach works vice versa – taking host language expressions, and simplifying them.

The chosen host language is of great importance in this stage: It forms and constraints the semantic and syntactic capability of the DSL. An important DSL characteristic is *language expressiveness*. We speak of high language expressiveness if the DSL only uses tokens that have a meaning in the domain. Conversely, tokens and expressions like semicolons, certain brackets, or statement modifiers, which are germane to the host language, but have no meaning in the domain, reduce the language expressiveness. Of course we should choose a host language that has a high expressiveness for our domain.

2.3 Language Implementation

Once we have collected a working set of language expressions we can provide the implementation. Our process uses a behavior-driven development style. Tests describe the DSL's behavior or capabilities. In the first iterations of DSL engineering, *example expressions as test cases* can be used to build an implementation that provides the semantics for the used domain objects and operations. In later development stages, tests are written to extend the language capabilities or to cover errors. All tests together can be seen as a complete specification of the DSL – a very valuable development artifact.

Patterns also play a vital role in the implementation. In computer science, patterns are understood to explain "... an important and recurring design" [20] in applications. Designs address specific problems in application development, like providing alternatives for different sorting algorithms or creating customized objects. The combination of the problem and its design solution is presented in an abstract pattern description to foster reusability [19].

We use patterns for three concerns of DSL-Engineering: *Language Modeling* (provide executable form of the domain model), *Language Integration* (integrate the DSL into the application framework and other languages), and *Language Purification* (reduce domain-foreign symbols and tokens to increase language expressiveness). The patterns help to identify and solve core concerns of the DSL. Therefore, they are an important leverage to ensure strong quality of the implementation.

2.4 Process Summary

We want to emphasize three important principles of this process: *agile steps*, *constant refactoring*, and *pattern knowledge*.

The first cornerstone are agile process steps. Iterations typically start with understanding the domain, then designing an expression, and finally implementing it. While a traditional top-down approach is useable in principle, our experience shows that small iterations work better. The focus on one requirement or feature at a time incrementally extends the DSL. A production-ready and tested version of the DSL is available after each iteration. This approach has proven very valuable especially if the requirements are not clear or cannot be made clear enough upfront.

Constant refactoring is the second cornerstone. At the end of each iteration, the DSL is refactored with two goals in mind: to provide an accurate representation of the domain's vocabulary and to have a clean [31] and minimal codebase. A complete set of test suites only suffice for extensive refactorings – they are an important prerequisites. Once refactored, the code is an ideal starting point for the next iteration.

The third cornerstone is patterns. With sufficient experience in DSL engineering, we encountered similar problems like providing an executable model of the domain or extending DSL functionality in a modular way. The problems and their solutions form a pattern catalogue. Once we have this understanding, we can make development decisions upfront. Further DSL engineering will refine the catalogue and add novel patterns.

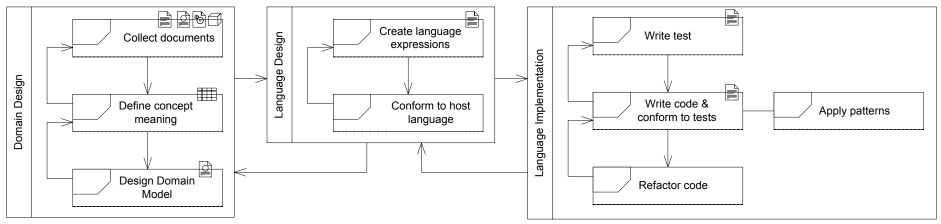

Fig. 1. DSL Engineering Process

A graphical summary of the process is shown in ►Figure 1. From top to bottom, we see the three process phases as boxes, and the tasks mentioned above. Arrows between steps and tasks show the possible paths iterations may take.

This concludes our process description. More details can be found in [22]. We will now explain how we developed three DSL for the management of IT Infrastructures, where we start with a complete domain design in ►Section 3, and explain the DSL's language designs and implementations in ►Subsections 4.2, 4.3, and 4.4.

3 Domain Design

The first step in designing a domain specific language is to understand the domain. We collected several articles, books, tools and existing languages. Here is a selection of the most influential parts – the material that formed our process, vocabulary, and language design.

3.1 Configuration Management

Changing requirements of software often results in new releases that made system administration a quite complex task in the past few years [15]. Software often needs to be installed on various machines and with different configurations. Furthermore there is a need to keep the installed applications up to date. But manual configuration often results in errors [5]. Consequentially, the need to install and configure software automatically on different machines arises. In the literature, this is called *configuration management* [15].

Our analysis revealed several tools supporting this domain: Cfengine, Puppet, and Chef. From a historical perspective, they are all built on top of each other. The developers working on Puppet took their inspiration from Cfengine, and later the developers working on Chef were inspired by the usage of Puppet. We explain all tools by outlining two parts. The first part is about the *components* used by the tool. Components are the static parts and entities that the tool implements, we explain them to gain hints about possible domain entities that are used as objects in our DSLs. The components are combined in the *architecture* and dynamically utilized for infrastructure setup and maintenance. The architecture again is analyzed because it provides the general process of deploying and maintaining software.

3.1.1 Cfengine

Cfengine is a configuration management tool developed in 1993 by Burgess at the Oslo University College [7]. It is an on-going research project and commercial product. Today, Cfengine is widely-used by several companies [8].

Cfengine assures valid system states that are expressed as *policies*. A policy can be applied to a single system or to all systems that are managed by Cfengine. Furthermore, a system can operate autonomously from the centralized policies. The tool uses an external DSL to define centralized specifications. In the following, we will introduce the components and architecture of Cfengine according to [6].

Components

- **Policies** – Used to describe a host's configuration.
- **Operators** – Primitive commands used to carry out maintenance checks and repairs.
- **Classes** – Used to structure the configuration into discrete units.
- **States** – Describe concrete system states as a configuration of global parameters.

Architecture

– **cfagent** – Agent that manipulates system resources.
– **cfservd** – Server that is able to share files and receive requests to execute an existing policy on an individual system.
– **cfexecd** – Scheduling daemon that can either supplement or replace `cron` (task scheduler).
– **cfrun** – Executes policy on a system.
– **cfkey** – Key generator for securing the server.

Example The example provided in ►Figure 2 illustrates a part of a policy for an Apache webserver. The Lines 6 to 9 define several packages that will be installed when applying the policy. Afterwards, in Line 13 and 20, operating system package manager is specified.

```
 1 bundle agent packages
 2 {
 3 vars:
 4
 5  "match_package" slist => {
 6                              "apache2",
 7                              "apache2-mod_php5",
 8                              "apache2-prefork",
 9                              "php5"
10                              };
11 packages:
12
13  solaris::
14
15   "$(match_package)"
16
17      package_policy => "add",
18      package_method => solaris;
19
20  redhat|SuSE::
21
22   "$(match_package)"
23
24      package_policy => "add",
25      package_method => yum;
26
27 }
```

Fig. 2. Cfengine: Policy for an Apache 2 Webserver

3.1.2 Puppet

Puppet is open source and a more recent approach of configuration management implemented in Ruby. Puppet is implemented following the client-server architecture: A central server provides dynamic configurations to its clients. Those configurations define a valid state of a client system. Clients can either pull from the server, or the server pushes configurations to them. Consecutively we will present the components and architecture of Puppet as introduced in [28].

Components

– **Manifest** – Describes a configuration of a system.
– **Resource** – A general system property like a package manager or a cron job.
– **Provider** – A specific system property (like the package manager `apt` for Debian-Linux).
– **Node** – An individual system.
– **Templates** Used to generate configuration files for systems.

Architecture

– **puppet** – Stand-alone Manifest evaluator.
– **puppetmasterd** – Daemon providing configurations to nodes.
– **puppetd** – Applies a Manifest to a node.
– **puppetca** – SSL server used for receiving certification requests from clients.
– **puppetrun** – Command line tool for manually triggering configuration runs.

Example As before, we will give a short introduction to Puppet by installing the Apache webserver. The source code is shown in ►Figure 3. It shows how several modules are configured using templates that are completed with parameters given to the manifest.

```
class apache2::basic inherits apache2 {
  apache2::config { "base":
    order => "000",
    ensure => present,
    content => template("apache2/base.conf.erb"),
  }
  apache2::module { "dir": ensure => present } # provides DirectoryIndex

  apache2::config { "mpm":
    order => "010",
    ensure => present,
    content => template("apache2/mpm-$real_mpm.conf.erb"),
  }
  # ...
}
```

Fig. 3. Puppet: Manifest for Installing Apache

3.1.3 Chef

Like Puppet, Chef is an open source configuration management tool written in Ruby [34]. Opposing to the other examples, Chef uses an internal DSL to express configurations. Clients and server use the OpenID standard [36] for authentication, and then use a SSL-secured communication to exchange configuration information.

Since Chef was chosen as part of our implementation, we will explain Chef in more detail than in the preceding two examples. Since Chef is lacking scientific publications up to now, we will use the information provided in the Chef wiki [35] for the following explanation.

Components

Chef has a fairly complex model consisting of several components. We separate them into hardware, software, and maintenance.

Hardware

- **Nodes** – Represents a physical or virtual system embedded in the infrastructure, like a server or a router.

Software

- **Cookbooks** – General container for sets of configuration options.
- **Recipes** – Recipes are attached to cookbooks and contain a list of resources and the action that should be applied to them.
- **Resources** – General system property (like a package ma
- **Attributes** – Used to set various properties within a Recipe. Attributes can be used to enable cross-platform configurations for an application.
- **Libraries** – Additional libraries integrate arbitrary Ruby code within a cookbook.
- **Templates** – Text files using special inline constructs to insert concrete configuration options.
- **Providers** – Specific system property, like the package manager "apt" for Debian-Linux.

Maintenance

- **Roles** – Aggregates different recipes to form a specific appearance of nodes.

Architecture

Chef's architecture basically consists of four components: Chef-Server, Chef-Client, Chef-Indexer, and Chef-Solo.

- **Chef-Server** – Central management program providing authentication and communication with nodes. Sends recipes and files to the nodes. The Chef-server functionality is provided by a web-frontend and a REST-API[1]
- **Chef-Client** – Each node attached to the server is a client. To change a clients packages we configure its *RunList* by adding Recipes. These Recipes will be applied to the Node by starting a client-run.
- **Chef-Indexer** – Enables full-text search within the entire infrastructure.
- **Chef-Solo** – Stand-alone tool for individual nodes that can be used in the absence of a Chef-Server.

[1] REST is an acronym for Representational-State-Transfer. The mechanism tries to overcome the statelessness of HTTP by using all in the HTTP standard available method for a "language" preserving state. See [17] for more details.

3.1.4 Summary

Seen from the current perspective, Chef is the roundup of the previous presented configuration management approaches. A client-server architecture, reusable cookbooks and recipes, many customization options, and secured communication – features that have been applied with a different degree to the other approaches too. Since Chef is open-source and provides an internal DSL usable for further modification, we will use it in the center of our current implementation.

3.2 Software Deployment

Companies focus on a consistent application landscape to lower the cost and complexity of administration [10]. The installation of security critical software components, e.g. operating system updates or antivirus software, must be done by experts to ensure the integrity of the IT Infrastructures [5]. Additionally, most users do not have the knowledge to install the needed software.

In this paper, we use the following concept for organizing software packages. *Packages* represent simple software artifacts. The target of software deployment is to integrate new packages into an application infrastructure already consisting of different packages. Because they depend on other functionality, we also introduce RESOURCES for grouping packages. The relationship between both is visualized in ►Figure 4. The root node of this *Dependency-Tree* is the software product that will be deployed. Every level of the tree consists either of packages or resources. It starts with a package that depends on some resources which can be provided by packages that again can depend on resources.

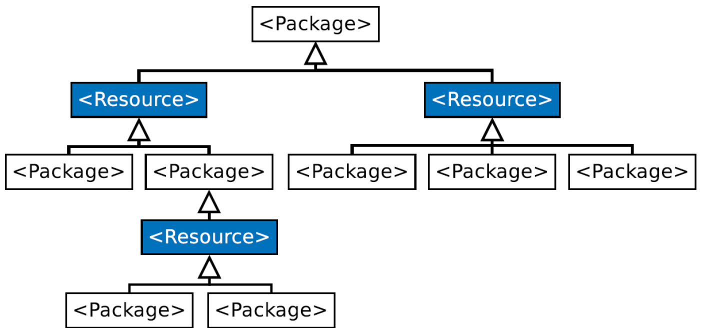

Fig. 4. Relationship between Packages and Resources

HEYDARNOORI developed the following software deployment process that describes the integration of software into an existing application infrastructures [25].

1. In the **Acquiring**-Phase, information about the infrastructure and the new software is collected. This consists of their metadata, dependencies, available servers, and existing data structures.

2. The **Planning**-Phase analyses that software is already installed and could be used for the deployment. Based on this information, the described dependency-tree will be created. With the help of a software deployment planner, the deployment plan will be created. Its task is to choose an appropriate package for every resource in the dependency-tree and to define the order in which the packages will be installed.
3. In the **Installation**-Phase, all elements of the deployment-plan will be installed.
4. Afterwards, the **Configuration**-Phase configures the installed packages.
5. When all packages are configured, they get activated in the **Execution**-Phase.

The Planning-Phase is covered by the SDP-DSL and Phases 3 to 5, Installation, Configuration and Execution respectively, are covered by the CM-DSL.

Our investigation on software deployment tools that deal with textual notations of packages focused on two further tools beside Chef. Both tools are not yet discussed in scientific literature, hence we reference their websites.

Sprinkle
The domain covered by the software deployment tool Sprinkle [11] is similar to Chef's domain. In contrast to Chef, Sprinkle does not implement a client-server architecture. With Sprinkle, the user has to define scripts that include operating system specific commands for the installation of software packages. Furthermore, the DSL defines policies for combining packages and deployment blocks for describing the delivery of applications. But Sprinkle is not able to configure and maintain the installed applications.

Poolparty
Poolparty [30] provides a DSL whose domain covers more the provision of dynamically scalable virtual servers instead of pure software deployment. But Poolparty changes the way of handling server installations by making servers "touchable" entities. This opens up the possibility to manage servers and operating systems with expressions similar to those that handle packages in Chef or Sprinkle.

Thus, through the same level of abstraction for deploying servers, operating systems, and software packages, we are able to see the deployment in a more integrated way.

Summary
The Sprinkle and Poolparty DSL showed which entities are most important for software deployment: Packages, resources, and the machines they are installed on. We also adopted the style how attributes are defined and the way how block-level hierarchies express structured relationships.

4 DSL-Support for IT Infrastructures

The former section presented many details in the domain of IT Infrastructure. When we analyze the typical required tasks, the whole complexity shrinks down to the identification and installation of machines and installation of packages. Supporting these tasks is our basic motivation for the DSLs. In the following, we will first introduce the architecture that is the common base of our DSLs. Afterwards we will explain the *lifecycle* of infrastructures, give a broad overview about the DSLs, and subsequently detail each DSL.

We use the Ruby programming language [18, 40] to implement DSLs within a tool called CIN. CIN uses Chef as a library to access machines and install software. The most important domain entities are cookbooks, packages, resources, installations, and machines (cf. ►Figure 5). A cookbook is a reference to a Chef cookbook and hence contains the knowledge about how to install software. Software is represented by packages that can be installed or can furthermore be provided by resources – in combination with resources they form a dependency-tree that we introduced in the further section. An installation represents a package that is completed with attributes for configuring this package and is finally deployed on a machine (physical or virtual). All entities are stored in a database.

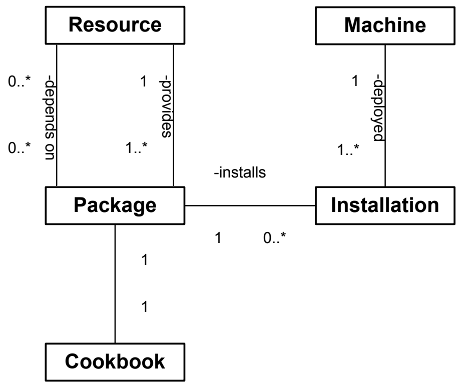

Fig. 5. CIN: Entity Model

DSL expressions create or manipulate objects of the database, and calling methods on these entities manipulates machines and packages in the infrastructure. In the following explanation, we will separate these DSL tasks in a section about *configuration* and one about *utilization.* Since this paper's focus is the DSLs and their interaction, we refrain from giving a more detailed explanation of CIN.

4.1 Infrastructure Lifecycle

IT Infrastructures consist of several physical or virtual servers and software packages. We identified three lifecycle phases:

- **Initialization** – In this phase, existing machines and novel installed machines are identified and several software packages are deployed on them.
- **Evolution** – The infrastructure is subject to frequent changes, like software updates or downgrades, installing additional packages, and consolidating, migrating, or extending the machines.
- **Teardown** – If the infrastructure needs to be replaced completely, all machines are removed.

We developed three DSLs that support these phases. The basic machine initialization is the task of the *Boot-DSL*. It allows either to add existing machines or to install novel ones. For new installations, we currently support the Amazon EC2[2] and VMWare ESX[3] hypervisor. The Boot-DSL configures a SSH-access to the machine and also installs the Chef-client that was explained in ►Section 3.1.3.

Once the machines are operable, the next task is to configure and install the relevant software packages with the help of the *Configuration Management DSL* (CM-DSL). Configuration determines basic attributes and metadata for the packages. For the deployment, two options exist. The first one is to stick to the CM-DSL and use the package declaration to also define on which machine the application runs. This requires resolving package dependencies manually, including the correct order of package installation commands and the manual modification of cookbooks. The other option is to use the *Software Deployment Planning DSL* (SDP-DSL). After a package dependency graph has been created, the DSL provides automatic dependency resolving and the option to express where packages are deployed in a more concise way.

From here on, the infrastructure maintenance is supported with a combination of these DSLs. To update packages or install novel ones, CM-DSL and SDP-DSL can be used together. And for consolidation or extension of the infrastructure, Boot-DSL and SDP-DSL also work together.

We will now explain each DSL in detail.

4.2 Boot-DSL

4.2.1 Configuration

The task of the Boot-DSL is to identify and additionally install machines. The DSL has the following three types of attributes:

- **Identification** – Define the owner, the operating system, and the optional hostname (as some machines are only accessible via private IPs, defined by the hypervisor).

[2] http://aws.amazon.com/ec2/
[3] http://www.vmware.com/products/esx/

- **Hypervisor** – A hypervisor is the tool with which a physical operating system communicates with a virtual hosted one. Using a `hypervisor` expression inside Boot-DSL indicates that the machine has to be installed. Options identify the used image, the machine size and more.
- **Operations** – The last type defines additional operations for the machine. For example, `bootstrap!` expresses to additionally install the Chef-client on the machine.

Let us take a look at two Boot-DSL expressions. The first one expresses how to add a local computer to the network, and the second expresses how to setup a new virtual machine with Amazon EC2.

The configuration of a local machine is shown in ►Figure 6. The expression begins with the machines name which is followed by an expression block. Inside the block, we configure the machine's owner, the operating system, and the hostname. The owner also identifies a SSH key that is used to communicate with the machine and execute further communications. Line 5 contains the mentioned `bootstrap!` command.

```
1 machine "Application Server" do
2   owner "sebastian.guenther@ovgu.de"
3   os :ubuntu
4   hostname "admantium.com"
5   bootstrap!
6 end
```

Fig. 6. Boot-DSL: Identify an Existing Machine

The installation of a virtual machine is shown in ►Figure 7. As before, line 2 and 3 define the owner and the operating system. In line 4, a special `hypervisor` block is declared. It configures the required properties for the installation of an Amazon EC2 virtual machine. We declare the Amazon Machine Image (AMI), the size of the machine, the security group (determines which ports are open), and the name of the private key file used to access this machine via SSH. Afterwards, in line 11, we configure the resources that should be monitored and recorded in log files. The last line calls the mentioned `bootstrap!` method again.

4.2.2 Utilization

When Boot-DSL expressions are executed, they trigger further processes if either `bootstrap!` or `hypervisor` is used. For `bootstrap`, CIN connects to the machine with the SSH keys of the owner, and automatically installs the CIN-proxy and all dependencies. It also calls the Chef-client run and registers the machine with the Chef-server. For the `hypervisor` method, CIN reads the machine's owners Amazon EC2 access credentials, and interacts with the EC2 Web-Service. The service returns the EC2 internal id and public IP of the new machine, which are saved

```
machine "Auxiliary Server" do
  owner "sebastian.guenther@ovgu.de"
  os :debian
  hypervisor :ec2 do
    ami "ami-dcf615b5"
    source 'alestic/debian-5.0-lenny-base-2009...'
    size :m1_small
    securitygroup "default"
    private_key "ec2-us-east"
  end
  hostname "admantium.com"
  monitor :cpu, :ram
  bootstrap!
end
```

Fig. 7. Boot-DSL: Identify and Install a Virtual Machine with EC2

along with the other information. Existing machines can be maintained by calling the `reboot` or `destroy!` methods.

4.3 Configuration Management DSL

4.3.1 Configuration

After a machine has been initialized successfully, we need to install and configure software packages. Manual configuration is error-prone and requires a heavy time investment [15], but carefully designed automatic support makes the deployment and maintenance of infrastructures much easier. We explained the Chef infrastructure in Section 3. The community around Chef provides many cookbooks for configuring servers. But they have a flaw: Different configurations require different fully-specified cookbooks. The motivation for the CM-DSL is to use generic cookbooks that are customized by attributes for specific installations.

We now want to give an example of how our CM-DSL works. This will be done by showing how a *package* can be installed. The first step is to define a cookbook, which is illustrated in ►Figure 8. The expression in line 1 defines the cookbook's name. This is followed by a code block with a `do ... end` notation. Within this block, we will set some metadata describing the cookbook.

```
1 cookbook :apache2_cookbook do
2   author      "Guenther, Haupt, Splieth"
3   description "Apache Cookbook"
4 end
```

Fig. 8. CM-DSL: Defining a Cookbook

The second step is to define the package (cf. ►Figure 9). The expression in line 1 defines the package's name. Afterwards, we define the supported platforms, the features that are enabled in the package and the software license. In addition, we define some tags, provide a description and define the supported versions. Line 8 associates the package with a cookbook.

```
1 package :apache2 do
2   platforms   :debian, :ubuntu
3   features    "mod_ssl"
4   license     "Apache 2 License"
5   tags        "Webserver, OpenSource"
6   description "Package for Apache-HTTP-Server"
7   versions    "2.0 - 2.2"
8   cookbook    :apache2_cookbook
9 end
```

Fig. 9. CM-DSL: Defining a New Package

The last step is to define an installation object (cf. ►Figure 10). In line 2, we see how the installation is associated with the package. Lines 4 and 5 set specific installation attributes for the cookbook.

```
1 installation.configure do
2   package   :apache2
3   machine   "www.admantium.com"
4   attribute :contact, "admin@admantium.com"
5   attribute :listen_ports, "80, 443, 4000"
6 end
```

Fig. 10. CM-DSL: Defining a New Installation

4.3.2 Utilization

To deploy the package, we execute the `installation.install!` method. This triggers a complete Chef-Client run – illustrated in ►Figure 11.

The first thing is to synchronize the node's current configuration with the one of the Chef server. Afterwards, cookbooks are synchronized. The last step is to apply all recipes in the nodes run list. Thereby, the node's information is applied to customize the relevant cookbooks, the recipes are applied by using the cookbooks templates and files, and the node is modified accordingly.

Maintenance

After the successful installation of a package, it can be removed by using the method `uninstall!`. This method removes a package only if a removal recipe exists. The subsequent changing of configurations is realized by the method `update!`. a example for using this method is given in ►Figure 12. Concerning ►Figure 10, the "'listen_port"' and "'timeout"' attributes are changed.

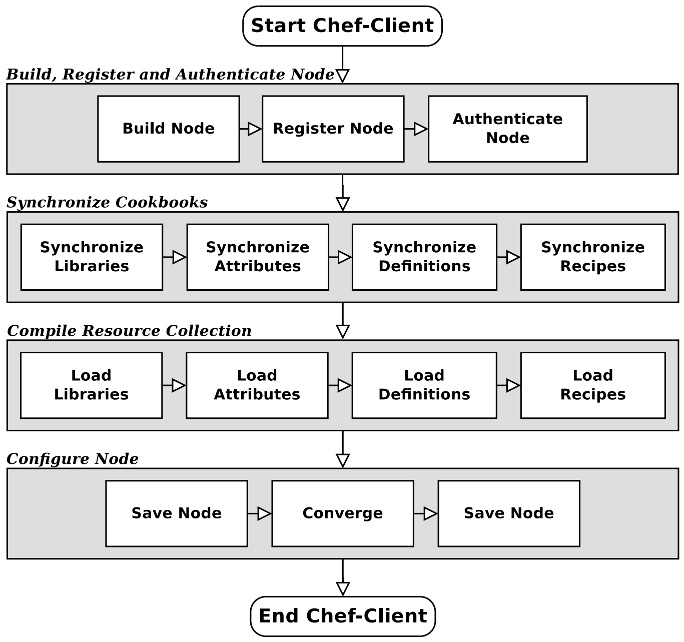

Fig. 11. Steps of a Chef Run (redrawing of [35])

```
installation.configure do
  attribute :listen_ports, "8080"
  attribute :timeout, "500"
end
```

Fig. 12. CM-DSL: Defining a New Installation

The method `upgrade!` can be used to upgrade software in case there is a new release or security patch. In effect, the installation will be reapplied using the already configured attributes.

4.4 Software Deployment Planning DSL

While the CM-DSL requires manual package dependency resolving, the SDP-DSL resolves them automatically. We first build a dependency tree consisting of packages and resources. Then, the user either explicitly chooses a package for each dependency until the whole dependency-tree is satisfied, or heuristics help him by preselecting packages.

This section uses the Redmine project management application from ►Section 5, and the corresponding dependency graph in ►Figure 14 for the background example.

4.4.1 Configuration

As described in 3.2, the relationship between packages and resources is represented by the dependency-tree. This information is static and must be entered into the database before using the SDP-DSL. For example, to add the database resource to Redmine, we just need to execute `redmine.add_resource "Database"`. And to express that MySQL supports this package, we execute `mysql.provides "Database"`

4.4.2 Utilization

Utilization of the SDP-DSL is separated into configuring dependency resolving in the graph manually or to use the given meta-information about packages to compose a default selection.

At first, we demonstrate manual dependency resolving in ►Figure 13. In Line 2, we start setting the dependencies of *Redmine*. We use XPath-like[4] expressions to select nodes in the dependency-tree. This known way of selecting elements in a tree makes it easy to handle the dependencies.

```
1 redmine.set_resources do
2   set "Redmine/Database", 'SQLite'
3   set "Redmine/Rails Server", 'Passenger Phusion' do
4     set "./Webserver", 'Apache'
5   end
6   set "//Mail Server", 'Remote SMTP'
7 end
```

Fig. 13. SDP-DSL: Manual Selection of Package Dependencies

The Redmine package represents the root node of the dependency-tree (c.f. ►Figure 14). The resources *Database* and *Rails Server* are accessible as direct children through *Redmine/Database* and *Redmine/Rails Server*, respectively. The method `set` receives a XPath expressions and a package-name. Line 4 expresses that Apache will be set as the package fulfilling the *Webserver* dependency of *PassengerPhusion*. But not only single elements can be selected: Line 6 searches through the dependencies and sets all dependencies matching *Mail Server* to *Remote SMTP*.

The interaction between CM-DSL and SDP-DSL is shown in ►Figure 15. We configure the RemoteSMTP by setting the attributes *user*, *password*, *host* and *port* of the package using inline CM-DSL expressions.

Second, we see how a package's meta-data can be used to provide a default selection. Therefore, we developed several heuristics that are divided into the following two categories:

– **Dependencies** –Regarding the global dependency graph, following rules are checked.

[4] http://www.w3.org/TR/XPath/

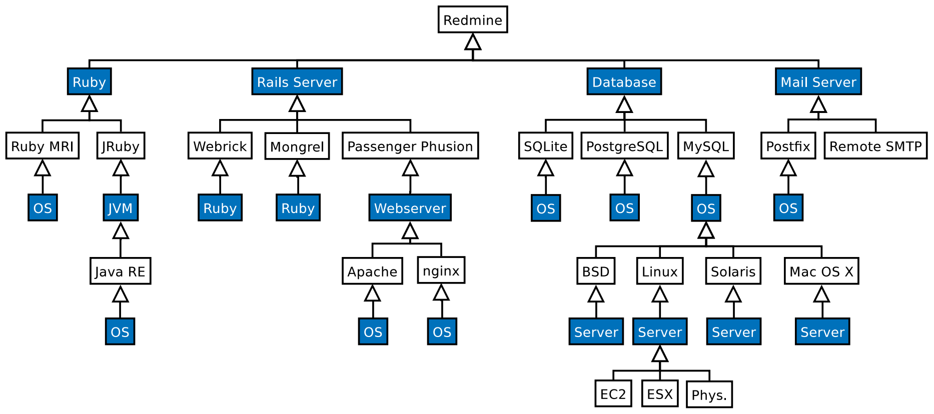

Fig. 14. The Complete Dependency Graph of the Application

```
set "//MailServer",:RemoteSMTP do
  attribute :user, 'someuser'
  attribute :password, 'my_secret'
  attribute :host, 'mail.google.com'
  attribute :port, 465
end
```

Fig. 15. SDP-DSL: Interaction with the Configuration Management DSL

  - Packages will not be preselected if they cause conflicts on the server they will be installed on or in the infrastructure at all.
  - Use already installed packages, e.g. do not install two DHCP servers, but instead apply both configurations on the installed one, if possible.
  - Respect constellations already established between packages that proved to be stable in the past.
- **Manual preferences** – The user is able to *prefer* or *force* the selection of packages. The first option selects the given package if it does not cause any conflict. The second option selects the package even if it causes conflicts. SDP-DSL tries to resolve the conflict by changing other already selected packages of the dependency tree.

►Figure 16 demonstrates how metrics can be applied. This code snippet combines the stored meta information of the packages with the flexible selection of dependencies through the XPath-like syntax. Subsequently, Boot-DSL and CM-DSL commands result.

Lines 2 and 3 state the task of installing MySQL on server1, and lines 5 and 6 install Phusion Passenger. Line 8 lists a set of metrics: To use an open-source license. This metric is applied in Line 9 to the webserver. And in Line 10, we force Ubuntu as the operating system for all database packages.

```
deploy 'Redmine' do
  enroll "Redmine/Database", on => ['Server1'] do
    prefer :package => 'MySQL'
  end
  enroll "Redmine/RailsServer", on => ['Server2', 'Server3'] do
    force :package => 'Phusion Passenger'
  end
  metrics = {:licence => 'opensource'}
  prefer :resources => "Redmine/*/WebServer", :metrics => metrics
  force :resources => "//Database/*/OS", :os => 'Ubuntu'
end
```

Fig. 16. SDP-DSL: Metrics-Based Package Preselection

5 Infrastructure Case-Study

In this section, we will present an infrastructure case-study that is deployed entirely by CIN and the presented DSLs. In the following, we use the DSLs to "speak" for themselves, and only occasionally explain expression details.

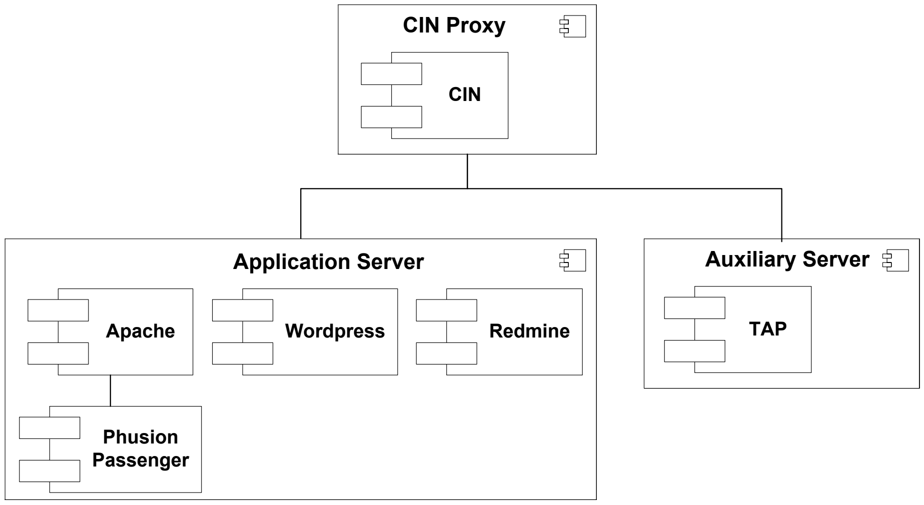

Fig. 17. Infrastructure Case-Study: Machines and Software

The infrastructure consists of three machines and five applications as shown in ►Figure 17. From these three machines, the CIN proxy and the application server are already installed, while we use Amazon EC2 to setup the auxiliary server. Additionally to the CIN application, the following applications are hosted:

– **Apache** – Apache is by far the most often used web server. It supports several roles: Serving web applications or static files, proxy for other web servers, or as a caching server. It is designed to handle several hundred requests per second in a very efficient manner. Applications written in different programming languages are supported by extending Apache with modules (http://apache.org).

- **Phusion Passenger** – Extends Apache to support Rails and Rack-compatible[5] web applications (http://modrails.com).
- **Redmine** – An open source web application written in Ruby on Rails[6]. Redmine is a combination of project management, wiki, issue tracking, and source code browsing (http://www.redmine.org).
- **TAP** – TAP is a web application written in the lightweight Sinatra[7] framework. It extends the known Twitter microblogging-platform with the option to store Tweets for a much longer time than the original application does.
- **Wordpress** – The most often used blog application written in PHP. Wordpress supports multiple editor workflows, its behavior can be extended with various plug-ins, and new releases appear regularly (http://wordpress.com).

In order to provide a step-by-step explanation, we separate the tasks of configuring and deploying the infrastructure in the following steps:
- Machine declaration and installation,
- Cookbook and package declaration,
- Dependency graph declaration,
- Package configuration and installation.

5.1 Machine Declaration and Installation

The first step is to provide the machines with the help of the Boot-DSL. The *application server* is already installed, we just need to identify and bootstrap it with the code show in ►Figure 18.

```
machine "Application Server" do
  owner "sebastian.guenther@ovgu.de"
  hostname "admantium.com"
  os :debian
  bootstrap!
end
```

Fig. 18. Infrastructure Case-Study: Setup of the Application Server

The *auxillary server* should run on Amazon EC2. The Boot-DSL is used to identify the machine and install it with the help of the EC2 API (cf. ►Figure 19). We use the smallest available machine size and the Gentoo operating system.

[5] Rack is a common interface to most Ruby web frameworks, see http://rack.rubyforge.org for more details.

[6] http://rubyonrails.org/

[7] http://sinatrarb.com

Note that the customized owner not only identifies to whom the machine belongs, but also provides the necessary keys to authenticate with the Amazon service.

```
machine "Auxiliary Server" do
  owner "sebastian.guenther@ovgu.de"
  os :gentoo
  hypervisor :ec2 do
    ami "ami-f691749f"
    source "edoceo.ec2-ami/edoceo.gentoo#...
    size :m1_small
    securitygroup "default"
    private_key "ec2-us-east"
  end
  bootstrap!
end
```

Fig. 19. Infrastructure Case-Study: Setup of the Auxiliary Server

These expressions add the application server and the auxiliary server to the CIN proxy and its running Chef-server. Using the CM-DSL, we can now install packages on the servers.

5.2 Cookbook and Package Declaration

For each application, we need a Chef cookbook, a cookbook object, and a package object, which are important for both the CM-DSL and SDP-DSL. In order to show some details regarding the configuration, we will focus on explaining the Redmine application from now on.

Cookbook
Redmine is a Rails application that means that we only have to download the application, unzip it, configure basic properties, setup the database, extend the Apache configuration file, and start the web application. All these steps are executed through a cookbook taken from [35]. We only show a small extract here that downloads and extracts an archived version of the application (cf. ►Figure 20).

Cookbook Object
A cookbook object is a simple entity providing the name of a Chef cookbook. CM-DSL requires this object to configure a dependency run, and the SDP-DSL influences cookbook attributes. ►Figure 21 shows the relevant source code.

Package Object
Finally we define the package object. This object is equally important in both CM-DSL and SDP-DSL. A package contains several attributes that define various metadata about the package – the complete specification of Redmine is shown in ►Figure 22.

```
include_recipe "rails"

bash "install_redmine" do
  cwd "/srv"
  user "root"
  code <<-EOH
    wget http://rubyforge.org/frs/download.php/#...
    tar xf redmine-#{node[:redmine][:version]}.tar.gz
    chown -R #{node[:apache][:user]} redmine-#
  EOH
  not_if { File.exists?("/srv/redmine-#...
end
```

Fig. 20. Infrastructure Case-Study: Redmine Cookbook (extract)

```
cookbook :redmine do
  author      "Matthias Splieth"
  description "Redmine Project Managemant"
  package     :redmine
end
```

Fig. 21. Infrastructure Case-Study: Redmine Cookbook Object

```
package :redmine do
  platforms   :debian, :ubuntu, :gentoo # ...
  license     "GNU General Public License v2 (GPL)"
  tags        "Webapplication, Wiki, Bugtracking, SVN, Git"
  description "Complete development project infrastructure"
  versions    "0.9.3"
  cookbook    :redmine
end
```

Fig. 22. Infrastructure Case-Study: Redmine Package Object

5.3 Interlude: Direct Package Installation with Manual Dependency Resolution

At this point of configuration, we have all required information to manually install the packages. The only requirement is to define an installation object and process it. This object determines which package should be applied on which machine. Let's take a closer look at the example in ►Figure 23.

```
installation.configure do
  package :redmine
  machine "Application Server"
  install!
end
```

Fig. 23. Infrastructure Case-Study: Definition and Execution of an Installation Object

Redmine has several dependencies: It requires a Ruby interpreter, Rails, and a database. Without the option to configure the cookbooks, we would manually need to implement a different Chef cookbook for all installation options and their combinations. Given two interpreters, five rails version, and three database connections, we end up with 30 combinations. The motivation of the CM-DSL is to configure the cookbooks within the DSL.

But still, the need to either provide a full default cookbook or to remember the remaining dependencies. With the help of the SDP-DSL, we can define a dependency graph. This helps in automatically resolving requirements and suggesting alternate packages. We suppose to use the SDP-DSL, and continue to declare a dependency graph.

5.4 Dependency Graph Declaration

The dependency graph is a structure consisting of an interwoven set of packages and resources. A graphical representation is shown in ►Figure 14. The root element is the Redmine package. At the second level, we express which resources are required: Ruby, a Rails Application Sever, Database, and Mail Server. We say that packages provide resources, and therefore list packages at the third level. For example, we can use Webrick, Mongrel, or Apache to satisfy the Rails Server dependency. We can choose which dependencies are modeled in the graph freely.

Now to the SDP-DSL. Packages are already defined, so declaring resources and the dependencies between resources and packages are left. This is expressed in ►Figure 24. For Redmine, we add two dependencies on the Rails Server and the Ruby resources (lines 2 and 3), and Lines 6 to 7 define the Ruby MRI (the common Ruby interpreter written in C) and Passenger Phusion package to provide this resource.

```
1 redmine.configure do
2   dependency "Ruby"
3   dependency "Rails Server"
4 end
5
6 rubymri.provides "Ruby"
7 passenger.provides "Rails Server"
```

Fig. 24. Infrastructure Case-Study: Declaring Resources for the Dependency Graph

5.5 Package Configuration and Installation

With the dependency graph in place, there are only two steps left for the final infrastructure rollout. First, we satisfy all package dependencies of Redmine by using the `set` method. This method is called with an XPath-like [37] query for selecting specific resources inside the dependency graph and for setting the specific package that provides this resource. In ►Figure 25, we see how the database resource is provided by the MySQL package, the web server configured with Apache, and more.

```
redmine = Package.get :name => "Redmine"
redmine.set_resources do
  set "Redmine/Database", :MySQL
  set "Redmine/Rails Server", :apache do
    set "./SeverModule", "PassengerPhusion"
  end
  set "Redmine/Mail Server", :RemoteSMTP
  set "*/OS", :Debian
end
```

Fig. 25. Infrastructure Case-Study: Configuring the Infrastructures Dependencies

The second step is the final rollout of the infrastructure. The default case is to just execute `deploy` with the package name and another parameter specifying the machine. This works only if all dependencies are defined. All packages are installed on the same machine. Before installing anything, SDP-DSL checks various conditions: all requirements are met, the machine has the correct operating system, and so on.

In some circumstances, it may be better to install some packages on different machines, like providing a separate database server. Also, the default configuration may be changed because users prefer other packages. We express this concern with the code shown in ►Figure 26. Let us explain top-down.

– Line 1 selects the Redmine package and executes a block in its context, thus gaining access to all dependencies.
– Line 2 shows the `enroll` method with three parameters: a) Selecting the package that implements a resource, b) defining the machine on which this package is installed, and c) a block with a modification of the pre-configured package selection.
– Line 2 and Line 5 install the Rails Server and the Database.

```
1 deploy :Redmine, :on => ["admantium.com"] do
2   enroll "Redmine/Database", :on => ["Application Server"] do
3     prefer :package => "MySQL"
4   end
5   enroll "Redmine/Rails Server", :on => ["Application Server"] do
6     force :package => Apache2"
7   end
8   force :os => :Debian, :resources => "//Database/*/OS"
9 end
```

Fig. 26. Infrastructure Case-Study: Installing the Infrastructure

5.6 Summary

Most effort of using the DSL to deploy and maintain an infrastructure is the task to provide cookbooks and the configuration objects (package, resource). Once this initial setup occurred, the expressions to setup machines and install packages are small. If we assume the existence of cookbooks and configuration options to already exist, we only need around 30 lines of code to setup the explained infrastructure. And furthermore, the architecture is highly agile. Adding packages or new machines, updates, and – in the future – migrations require the same amount of code. With the help of DSLs, infrastructures become a lot more manageable.

6 Summary and Future Work

This paper presented the utilization of domain-specific languages for the deployment and maintenance of IT Infrastructures. We started to explain the used DSL engineering process. Afterwards, we introduced several tools and their DSLs: Cfengine, Puppet and Chef. From this domain material, we identified three important tasks in infrastructure management: (1) To initially setup physical or virtual machines, (2) to arbitrarily configure and install software packages on the machines, and (3) to guide the user in selecting appropriate software packages. In order to integrate all these tasks within one tool, we designed three integrated DSL for these tasks. A case study showed how the DSLs were used to setup an infrastructure consisting of three machines and five applications.

This approach has several advantages in comparison to others. At first, easy to write and to read declarative expressions are all that is needed to setup complex infrastructures. The DSLs allow a high degree of customization and once written can be reused to implement similar infrastructures. Second, the DSL are integrated form the initial setup of machines using hypervisors down to a SSH-connection and the local system's package manager. This facilitates to have one common data model for the infrastructure. Third, this system is open for concerns such as user and identity management, backup, security and more. We just need other DSLs for these concerns, and can then freely mix the concerns via the DSL in declarative expressions. In total, this flexibility makes our approach a well-grounded alternative to other approaches.

References

1. B. R. T. Arnold, A. V. Deursen, and M. Res. Algebraic Specification of a Language for describing Financial Products. In *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE, 1995.
2. R. Bahlke and G. Snelting. The PSG System: From Formal Language Definitions to Interactive Programming Environments. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(4):547–576, 1986.

3. R. A. Ballance, S. L. Graham, and M. L. V. De Vanter. The Pan Language-Based Editing System For Integrated Development Environments. *ACM SIGSOFT Software Engineering Notes*, 15(6):77–93, 1990.
4. J. Bentley. Programming Pearls: Little Languages. *Communications of the ACM*, 29(8):711–721, 1986.
5. A. B. Brown. Oops! coping with human error in it systems. *Queue*, 2(8):34–41, 2004.
6. M. Burgess. A tiny overview of cfengine: Convergent maintenance agent. In *Proceedings of the 1st International Workshop on Multi-Agent and Robotic Systems, MARS/ICINCO*. Citeseer, 2005.
7. M. Burgess et al. Cfengine: a site configuration engine. *USENIX Computing systems*, 8(3):309–402, 1995.
8. Cfengine. Companies using cfengine, March 2010. http://cfengine.com/pages/companies.
9. C. Consel and R. Marlet. Architecturing Software Using A Methodology for Language Development. In *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, volume 1490 of *Lecture Notes in Computer Science*, pages 170–194, Berlin, Heidelberg, New York, 1998. Springer.
10. A. Couch, N. Wu, and H. Susanto. Toward a cost model for system administration. In *Proceedings of LISA '05: Nineteenth Systems Administration Conference*, pages 125–141, 2005.
11. M. Crafter. Sprinkle. http://www.redartisan.com/2008/5/27/sprinkle-intro, May 2008.
12. M. J. E. Cuaresma and N. Koch. Requirements Engineering for Web Applications - A Comparative Study. *Journal of Web Engineering*, 2(3):193–212, 2004.
13. H. C. Cunningham. A Little Language for Surveys: Constructing an Internal DSL in Ruby. In *Proceedings of the 46th Annual Southeast Regional Conference (ACM-SE)*, pages 282–287, New York, 2008. ACM.
14. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, San Franciso et al., 2000.
15. 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*, 2007.
16. T. Dinkelaker and M. Mezini. Dynamically Linked Domain-Specific Extensions for Advice Languages. In *Proceedings of the 2008 AOSD Workshop on Domain-Specific Aspect Languages (DSAL)*, pages 1–7, New York, 2008. ACM.
17. T. R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
18. D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O-Reilly Media, Sebastopol, 2008.
19. M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, San Francisco et al., 2003.
20. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Harlow et al., 10th edition, 1997.
21. J. F. Groote, S. F. M. Van Vlijmen, and J. W. C. Koorn. The Safety Guaranteeing System at Station Hoorn-Kersenboogerd. In *Proceedings of the Tenth Annual Conference on Computer Assurance Systems Integrity, Software Safety and Process Security (COMPASS '95)*, pages 57–68. IEEE, 1995.

22. S. Günther. Agile DSL-Engineering and Patterns in Ruby. Technical report (Internet) FIN-018-2009, Otto-von-Guericke-Universität Magdeburg, 2009.
23. S. Günther. Engineering Domain-Specific Languages with Ruby. In H.-K. Arndt and H. Krcmar, editors, *3. Workshop des Centers for Very Large Business Applications (CVLBA)*, pages 11–21, Aachen, 2009. Shaker.
24. S. Günther and S. Sunkle. Feature-Oriented Programming with Ruby. In *Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD)*, pages 11–18, New York, 2009. ACM.
25. A. Heydarnoori, F. Mavaddat, and F. Arbab. Towards an automated deployment planner for composition of web services as software components. *Formal Aspects of Component Software*, 2:279–293, 2005.
26. P. Hudak. Modular Domain Specific Languages and Tools. In P. Devanbu and J. Poulin, editors, *Proceedings of the 5th International Conference on Software Reuse (ICSR)*, pages 134–142, 1998.
27. F. v. Kutschera. *Sprachphilosophie*. Wilhelm Fink Verlag, München, 2nd edition, 1975.
28. R. Labs. Puppet, March 2010.
http://projects.reductivelabs.com/projects/puppet/wiki.
29. P. Lehmann. *Meta-Datenmanagement in Data-Warehouse-Systemen - Rekonstruierte Fachbegriffe als Grundlage einer konstruktiven, konzeptionellen Modellierung*. Dissertation, Otto-von-Guericke-Universität Magdeburg, 2001.
30. A. Lerner. Poolparty. http://auser.github.com/poolparty/, March 2010.
31. R. C. Martin. *Clean Code - A Handbook of Agile Software Craftsmanship*. Prentice Hall, Upper Saddle River, Boston, Indianapolis et al., 2009.
32. M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Survey*, 37(4):316–344, 2005.
33. J. Munnelly and S. Clarke. ALPH: A Domain-Specific Language for Crosscutting Pervasive Healthcare Concerns. In *Proceedings of the 2nd Workshop on Domain Specific Aspect Languages (DSAL)*, New York, 2007. ACM.
34. Opscode. Chef webpage, March 2010. http://www.opscode.com/chef/.
35. Opscode. Chef wiki, March 2010. http://wiki.opscode.com/display/chef/.
36. D. Recordon and D. Reed. OpenID 2.0: A Platform for User-Centric Identity Management. pages 11–16, 2006.
37. J. Simpson. *XPath and XPointer - Locating Content in XML Documents*. O'Reilly Media, Sebastopol, 2002.
38. D. Spinellis. Notable Design Patterns for Domain-Specific Languages. *Journal of Systems and Software*, 56(1):91–99, 2001.
39. S. Thibault, R. Marlet, and C. Consel. A Domain-Specic Language for Video Device Drivers: from Design to Implementation. pages 11–26, 1997.
40. D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby 1.9 - The Pragmatic Programmers' Guide*. The Pragmatic Bookshelf, Raleigh, 2009.
41. A. Van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35:26–36, 2000.
42. E. Visser. WebDSL: A Case Study in Domain-Specic Language Engineering. In R. Lammel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 4143 of *Lecture Notes in Computer Science*. Springer. Tutorial for International Summer School GTTSE 2007, 2008.
43. D. S. Wile. Supporting the DSL Spectrum. *Journal of Computing and Information Technology*, 9(4):263–287, 2001.