日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

Modifying Ruby for Designing and Implementing DSLs

Sebastian Günther

Domain-Specific Languages (DSL) are becoming an important tool for application developers. They help developers to express the particular problems and solutions of a domain in a language which represents the structure of the domain. A special kind of DSLs are internal DSL. They are built on top of an existing programming language (the host language), by modifying the semantics and using syntactic modifications. This type of DSL is especially popular with languages like Ruby, Python and Scala.

This paper shows a case study how to use Ruby to design a DSL for the domain of IT infrastructure management. In this domain, servers and their applications are deployed to provide an infrastructure of mail, database, and web servers for a client. The case study specifically shows how common object-oriented expressions may be used to express the deploying of servers, and then stepwise implements a DSL for the same purpose, thereby illustrating the used techniques. The techniques are then separately presented and discuss to show the amount of options in DSL implementation.

1 Introduction

Domain Specific Languages are programming languages or executable specification languages that offer, through appropriate notations and abstractions, expressive power tailored for a specific problem domain or application area [32] [22]. At the implementation level, two different types of DSLs can be identified. Specially crafted external DSLs require their own parsers, interpreter or compilers, while internal DSLs are built on top of an existing programming language (called the host language), allowing them to reuse the host language's infrastructure including IDEs and compilers [26]. DSL are a common practice in software engineering. Earlier DSL examples support financial products [1], signaling installations for rails [16], and video device drivers [30]. Recent domains where DSLs have been applied successfully are healthcare systems [27], and DSLs can also be used to support software development itself, for example to model software product lines [18] or to ease feature-oriented programming [21].

Modern programming languages like Ruby [31], Python [25], and Scala [33] are used to implement DSLs. This paper focuses on Ruby, a dynamic and pure object-oriented programming language. Several interpreters for Ruby exist. The two most mature ones are the original MRI[†1] written in C and JRuby[†2] written in Java. A complete language specification is available: In the form of executable tests[†3], and as a formal specification draft[†4]. Ruby supports multiparadigm programming with a mix of imperative, functional, and object-oriented expressions. Object-orientation is the basis, as clas-

Sebastian Günther, , School of Computer Science, University of Magdeburg, Germany.

†1 http://www.ruby-lang.org/en/
†2 http://jruby.codehaus.org/
†3 http://rubyspec.org/wiki/rubyspec
†4 http://ruby-std.netlab.jp/draft_spec/draft_ruby_spec-20091201.pdf

ses and even methods are objects with (re)definable properties. Ruby provides extensive runtime modification capabilities: Module and class redefinition, method extension, saving code in the form of proc objects or strings, evaluating code in any place, and much more. A special property is open classes: Even the core classes, like `Array` or `String`, can be modified. This provides many opportunities to customize the semantics of the Ruby programming language. All these modifications are done by metaprogramming.

Our research motivation is to design and implement internal DSL that are built on top of another language. In a recent contribution we collected our insights and practical experiences to form an engineering process for internal DSL [17]. Following this research, we present a practical case study in this paper how common object-oriented expressions can be changed to a more domain-specific language look.

In this paper, the particular domain we are addressing is IT infrastructure. This domain comprises initialization, configuration, and maintenance of several servers forming the infrastructure of a client. Continuously increasing requirements with regard to flexibility and extensibility put an automatic approach to these tasks at a crucial level. We designed a set of three DSLs that support various tasks in this domain. The *Boot-DSL* identifies and installs machines, the *Software-Deployment Planning DSL* (SDP-DSL) expresses relationships between packages, and finally the *Configuration Management DSL* (CM-DSL) configures and installs packages. The currently supported operating systems are Linux-based, and the hypervisors (tools that govern the virtualization of operating systems) are Amazon EC2[†5] and VMware ESX[†6].

†5 http://aws.amazon.com/ec2/

†6 http://www.vmware.com/products/esx/

Detailed description of the DSLs is available in [20].

The paper is structured as follows. In Section 2 more background information for DSLs and IT infrastructure management is given. Section 3 explains the start and result of the case study – common object-oriented expressions and a DSL that creates a machine (part of the Boot-DSL).Then Section 4 provides step-by-step instructions how to first build the DSL's syntax and then its semantics. Section 5 presents the used (and further) techniques, and Section 6 concludes this paper.

2 Background

2.1 Domain Specific Languages

DSLs can be classified along three dimensions: appearance, origin, and their implementation.

A DSL's *appearance* determines its principle physical appearance. A graphical DSL uses abstract symbols and drawings to express the relationship between domain concepts. A textual DSL uses mostly textual characters and mathematical symbols to express its meaning [8].

The *origin* of a DSL determines if the language is developed independent and free from other languages or whether it is based on another language. External DSLs require that their interpreter or compiler is written specifically for this language. This requires to develop the basic syntax, the expressions and tokens, and the languages semantics. In contrast, internal DSLs use an existing host language and build its abstractions on top of it. Thereby, the DSLs extend the semantic or paradigmatic capabilities of its host languages, e.g. by applying metaprogramming, and use available syntactic modifications, by using alternative constructs for grouping expressions or expressions delimiters. Internal DSLs are also called embedded DSLs [14].

Finally, the *implementation* of a DSL determines its technical capabilities. According to [26], DSLs

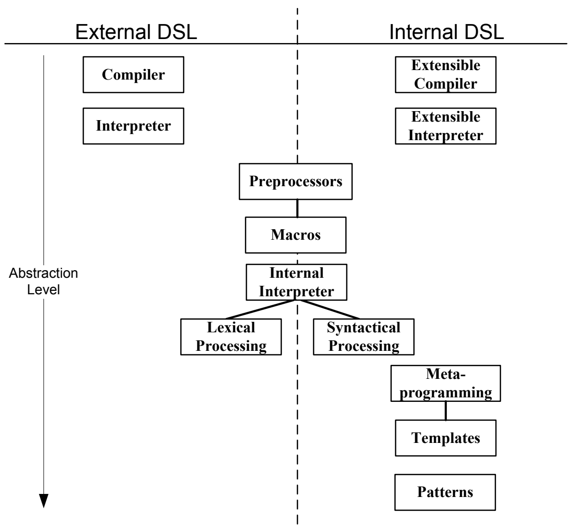

図 1 DSL-engineering mechanisms.

can be implemented by an interpreter, preprocessor, or exist as a hybrid. Another option is to use an compiled language if the DSL's runtime adaptation capabilities are not of much interest or performance considerations are of great importance. But these techniques are coarse grained, a more careful study shows that there are several ways how a DSL can be built. Combining several existing work on DSL design [2] [3] [7] [11] [12]citeSpinellis2001 [26] with our own findings, we can build the following list of mechanisms in ►Figure 1. From top to bottom, the mechanisms are ordered by increasing abstraction levels. From left to right, we show the availability of the mechanisms for external and internal DSLs – or for both. In this paper we are using the more abstract mechanisms of metaprogramming and pattern to develop the DSL.

2. 2 IT Infrastrucure Managemant

IT infrastructure management has the goal to systematically setup, maintain, and extend a clients capability to host several required applications. One of the important goals is to provide a consistent application landscape in order to to lower the cost and complexity of administration [10].

But several challenges have to be considered. First of all, the maintenance of some software and especially security critical software components, e.g. operating system updates or antivirus software, must be done by experts in a timely manner to ensure the integrity of the IT Infrastructure [4]. Second, when an infrastructure is setup for the first time, its parts and alternative applications may not be known to the user. And third, the increasing volatility of application requirement changes and the need for continuous adaptation of the infrastructure made system administration a quite complex task in the past few years [13].

Since manual configuration often results in errors [4], the need to install and configure software automatically on different machines arises. Our analysis revealed that several applications are actively engaging this problem. We shortly introduce a selection of three tools.

- **Cfengine** – A configuration management tool developed in 1993 by Burgess at the Oslo University College [5] [6]. It is an on-going research project and commercial product used by several companies. 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.
- **Puppet** – Puppet[†7] is open an 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

†7 http://reductivelabs.com/products/puppet/.

a valid state of a client system. Clients can either pull from the server, or the server pushes configurations to them.

- **Chef** – Chef[†8] is an open source configuration management tool written in Ruby. Opposing to the other examples, Chef uses an internal DSL to express configurations. Clients and server use the OpenID standard [28] for authentication, and then use a SSL-secured communication to exchange configuration information. The configuration of machines is stored as cookbooks which contain several attributes for an installation together with application-specific installation scripts.

While these tools cover a fair amount of functionality for the provision of software, they are relying on an existing infrastructure to work with. We think that the provision of the infrastructure in terms of servers, their operating systems and the particular role they play in an infrastructure is as important as the provision of software packages. That's our major motivation to bring both parts together in one DSL. The next section explains how to combine these requirements in the form of a DSL.

3 Machine Configuration – Object-Oriented Expressions vs. a DSL

Although we developed several DSL, we will focus on the Boot-DSL, which is used to express a virtual or physical machine declaration, for the following explanations. We shortly outline the most important application entities for te configuration of a machine, then show how common object-oriented expressions may be used to create machines, and finally how a DSL for the same purpose looks like.

†8 http://www.opscode.com/chef/

3.1 Application Outline

►Figure 2 shows the most commonly used entities of the application. We see the `Client` who is identified with his id and who has a private key to access its associated `Machines` via SSH. Each machine is also identified by an id, and it has among several other entities a hypervisor that is used to create the machine as well as a public IP which can be used to access the machine via the internet. Machines are generic entities for which hypervisor-specific classes exist. For example, the `EC2Machine` contains attributes like the identification of an Amazon Machine Image (AMI) id and the corresponding source file. Several `Packages`, which contain a description, version, and a set of available features (configuration options), can be installed on a machine. While the package represent a generic entity, the concrete `Installation` fills the configuration options of a package to install it for a specific machine.

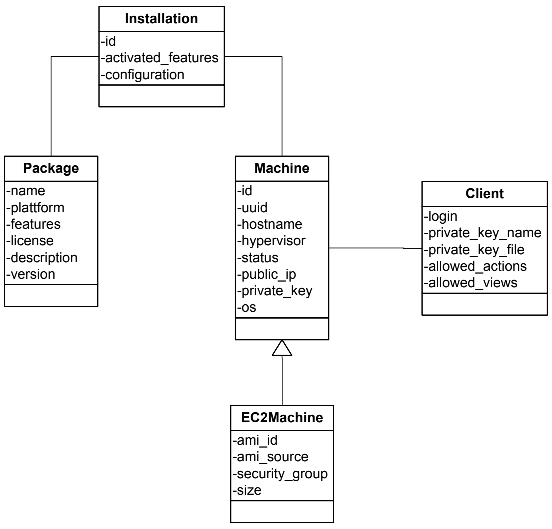

図 2 Application outline.

3.2 Object-Oriented Expressions

►Figure 3 shows an example how machines can be created using common object-oriented methods.

```
1 app_server = EC2Machine.new("Application Server")
2 app_server.set_owner("sebastian.guenther@ovgu.de")
3 app_server.set_ami("ami-dcf615b5")
4 app_server.set_ami_source("alestic/debian-5.0-len...")
5
6 hypv = EC2Hypervisor.new(app_server)
7
8 resource_bundle = Resource.Bundle.new
       (CpuResource.new(), RamResource.new())
9 app_server.set_monitored_resources(resource_bundle)
```

図 3 Common object-oriented expressions for creating and configuring an EC2 machine.

Beginning in Line 1, we see how a new machine object is instantiated. In the constructor, the name of the machine is given, and in Line 2 the owner is configured. Line 3 and 4 contain some EC2-hypervisor specific configurations. Then, Line 5 creates a new hypervisor object that receives the machine object and uses its' configured resources to create the machine. Then, Line 7 and 8 configure a set of monitored resources (CPU and RAM) which are added to the machine.

3. 3 DSL Expressions

Although these expressions are easy to read for people experienced with the Ruby programming language, the current form has some limitations. The explicit imperative form prescribes a fixed order of expressions: First the machine, then the hypervisor, and then the resources are added. The object-oriented form also explicitly ties developers to manually create the objects and associate them by handling object references. Although this example is short, it is also difficult to see that all expressions actually belong together. For example the explicit link between the resource bundle and the application server is "hidden" in Line 9.

These shortcomings can be encountered by choosing a more declarative form for expressing the same configuration, as shown in ►Figure 4.

```
app_server = machine "Application Server" do
  type EC2
  owner "sebastian.guenther@ovgu.de"
  hypervisor do
     ami "ami-dcf615b5"
     source "alestic/debian-5.0-lenny-base-2009..."
  end
  monitor :cpu, :ram
end
```

図 4 Several DSL expressions containing the same set of configuration ad in ►Figure 3.

These expressions are completly Ruby code. They use the entities of the domain and their properties mainly as methods used lik keyword. Related expressions are grouped together in blocks, and the syntax renounces parentheses or brackets to delimit expressions. Therefore, the DSL helps to declaratively express the common attributes of a machine, such as the internal hostname, public ip, the owner, and more. Attributes are checked for completeness and errors. If executed with correct and complete values, the particular-hypervisor provider is contacted automatically to create the machine and initially bootstrap the operating system for further connections, like using a SSH-based access. The effort to initially setup (boot) a machine is thus reduced to the provision of correct attributes.

We now want to see how to change the object-oriented expressions to this particular DSL format.

4 Development of the DSL

This section details the DSL development. At first we present the list of design goals stemming from common DSL design principles. Then the DSL is implemented by first designing the general syntax, then refining the syntax by removing unneeded tokens and using syntactic modifications, and finally by providing the semantics of the language using metaprogramming and open classes.

We close this section by a short review of the design goals and achievements.

4. 1 DSL Design Goals

Several works about DSL express important design principles like appropriate notations [32], compression to form for a concise language [24] [34], and absorption to express domain commonalities implicitly in the DSL [29]. With respect to these principles and forgoing object-oriented expressions, we compile the following goals for the DSL:

1. Remove unneeded tokens, such as parentheses, to provide the appropriate notation.
2. Absorb the explicit object creation.
3. Compress the relationships of objects with the help of nested blocks.
4. Use declarative expressions to describe the intent, not imperative expressions detailing the algorithm.

4. 2 Syntactic Modifications

4. 2. 1 Block Expression

Ruby's support for *closures* is used to provide the syntactical grouping of expressions and the relationships between the entities. Closures are commonly called blocks in Ruby. Through nesting they allow to visually arrange related expressions together, or in other words to compress the relationship declaration. Combining the machine and the hypervisor declaration thus can take the form shown in ►Figure 5.

```
machine {
  #...
  hypervisor {
     #...
  }
}
```

図 5 Using block expressions to group related expressions and object relationships together.

4. 2. 2 Keyword Methods

Inside the block, we can use methods to configure the properties. Ruby allows to skip the explicit definition of receivers. Expressions will be executed in the context of the receiver they are specified in. Additionally, blocks can be executed in any context, often concretely determined at runtime. Therefore, the properties can be defined like shown in ►Figure 6, using a more declarative appearance then their object-oriented counterparts, and thus describing the intent of the expression better.

```
machine {
  owner("sebastian.guenther@ovgu.de")
  hypervisor {
    ami("ami-dcf615b5")
  }
}
```

図 6 Using keyword methods for more declarative expression of object properties.

4. 2. 3 Clean Method Calls

Although the current form better groups related expressions and is more declarative then the explicit object-oriented notation, it is still complicated to read. Persons familiar with the domain will probably have a hard time to read the statements in ►Figure 6, because the parentheses have no meaning in the domain. To strengthen the usage of appropriate notation, Ruby facilitates to use words instead of the curly brackets in block expressions, allows to remove parentheses, and provides (by default) "invisible" line delimiters in the form of the newline character instead of explicit tokens such like semicolons in Java. Using these facilities, expressions as shown in ►Figure 7 can be used.

4. 3 Semantic Modifications

The usage of these expressions requires some semantic modifications on top of the existing applica-

```
1 machine do
2   owner "sebastian.guenther@ovgu.de"
3   hypervisor do
4     ami "ami-dcf615b5"
5   end
6 end
```

図 7 Clean method calls helps to eliminate parentheses and similar non-domain related symbols.

tions as well. Please note that in normal development these changes would go hand-in-hand with the syntactic changes as well, but for better explaining them, we have moved them to this place.

For the specific example we choose, the changes are moderate since the expressions are mostly used to cover object instantiation, property definition, and building relationships between the objects. The first part is to implement the methods that are used to introduce the blocks. For example the `machine` method receives an argument (as used in the original DSL-expression in ►Figure 4, but left out in the past examples) and a surrounding block. Here is an implementation stub (cf. ►Figure 8)

```
1 def machine(name, &block)
2   m = Machine.new
3   m.set_name(name)
4   m.instance_eval &block
5   retunrn m
6 end
```

図 8 Implementing the `machine` method that creates a `Machine` instance and executes all methods of the block in the context of the instance.

Block expressions, stemming from the functional programming paradigm, are an example of Ruby's support for multiparadigm-programming: Instead of imperatively describing a set of commands, the declarative expression of a desired object state suffices. The block is actually executed in Line 5 with the `instance_eval` method. It will execute the keyword methods in the context of the implicitly created machine instance. Note that the order of expressions inside the block is not relevant as ling as all required attributes are provided. The DSL users can structure the expressions according to their individual preferences.

The last change we discuss is proving the keyword methods. Back in ►Section 3, we see that the methods defined in the application are called for example `set_owner` instead of `owner`. We could implement these methods by hand – or just combine Ruby's metaprogramming support with its reflexive capabilities. We implement a method that iterates over the methods defined by `Machine` and if they start with the string `set_`, then we provide an alias. ►Figure 9 shows the implementation.

```
1 def set_alias_method(clazz)
2   methods = Object.instance_methods -
        clazz.instance_methods
3   methods.each do |method_name|
4     if method_name.match /^set_(.+)/ then
5       clazz.define_method #...
6    end
7 end
```

図 9 Defining the DSL methods on top of the objects already implemented methods.

4.4 Design Goals Review

Considering the design goals, we can see to have achieved them like follows.

- *Remove unneeded tokens, such as parentheses, to provide the appropriate notation* – This goals was achieved by using clean method calls.
- *Absorb the explicit object creation* – Using the Keyword Arguments method `machine` absorbs the object creation in the method body.

- *Compress the relationships of objects with the help of nested blocks* – The `hypervisor` method is implemented similarly to `machine`, and it is included in the middle of the machine block, thereby providing the nesting.
- Use declarative expressions to describe the intent, not imperative expressions detailing the algorithm – Inside the blocks, keyword methods are used that just declaratively determine an attribute property.

Thus the DSL's design goals are fulfilled.

5 Techniques for Implementing Internal DSLs

This section lists both the used techniques from the example and related techniques. Although shown in the context of Ruby, they are not germane to this language. Our experiences and ongoing experiments with Python and Scala show that the techniques can be used in several languages.

5. 1 Syntactic Changes

The syntax plays a vital role for a DSL. The task of syntactic changes is to provide the DSL with layout that is suitable for the domain, but it still needs to be compatible with the interpreter. Among the goals of such changes are the removal of all tokens that have no meaning in the domain (such as brackets or other delimiters) by using syntactic alternatives for existing expressions, and to combine modules, classes, and methods to represent the domain concepts. Here are the detailed techniques.

- *Block Expressions* – Are used for several purposes. The most important one is to visually group related expressions in one block. Nested blocks can also be used to represent the natural hierarchy or relationships of objects. And finally, because the block can be executed in any context, method calls don't need to specify the receiver.
- *Keyword Methods* – Ruby is a language with a low number of reserved keywords. Many methods that look like part of the language are actually method calls. This property can be used to add DSL keywords to the language which are actually methods.
- *Clean Method Calls* – The importance of methods has been explained before. In order to make them look like real language keywords, using them without brackets is crucial.
- *Method Alias* – In order to customize an existing library or even the Ruby programming language itself, provided methods can be aliased to a name better suiting the domain. Ruby even has a method for this purpose: `alias_method`.
- *Operator Expression* – Using mathematical symbols like `*, <, &&, or %` are very important for some domains. In Ruby, these are just method calls defined on the left-hand receiver. Defining them for the domain entities improves the domain-specific appearence for them.

5. 2 Semantic Modifications

Of course the objects and methods added by the syntactical extensions of a DSL need to be implemented. But not all changes are trivial or they would require an enormous manual effort (like providing the alias methods for the `Machine` entity earlier on). Deep changes can be done with one or more of the following methods.

- *Metaprogramming* – Metaprogramming refers to "programs that write [other] programs" [15]. A metaprogram uses explicit or implicit knowledge about the structure of a program to change it. Once a program has been loaded initially, changing it by adding new entities or methods is a metaprogrammatic approaches. Such changes are very common in dynamically typed languages like Ruby, providing runtime-adaptation to the execution environment.

- *Metaobject Protocol* – The origin of the term metaobject protocol is the implementation of object oriented programming in the functional language Lisp [23]. According to this reference, the protocol is defined as all the methods that govern the allocation and instantiation of objects, including the semantics such as variable declaration, visibility, method association, or namespace. Ruby implements the metaclasses `Class` and `Method` that govern the creations of these objects. They can be changed using metaprogramming and thus define a different behavior, like the replacement of object instantiation by object cloning.
- *Open Classes* – Defines the ability to change the behavior of language given classes. A good example is to add new operators and methods to built-in data structures that play a role in DSL, like Hashes or arrays. Furthermore, if a metaprotocol for the language entities has to be provided and changes can be inferred, open classes are a prerequisite.
- *Multi-Paradigm Support* – In essence multi-paradigm programming is the capability of application developers to use those paradigms for writing programs that best fits the specific purpose [9]. Object-oriented programming is the dominant form of today's programming language and is a natural way to express the complex hierarchy of objects. But other paradigms are important too, like the provision of small anonymous functions for custom sorting of collections or the sideffect-free computation of results – this is the domain of functional programming. Finally to better modularize the application, aspect-oriented programming and feature-oriented programming can be invoked too. In [19], we showed an example how these paradigms can be used together effectively.

6 Conclusion

This paper showed how to transform object-oriented expressions to a DSL. It showed how to use the facilities of the Ruby programming language to gradually change the syntax to use blocks for visually structuring or expressions and object relationships, how to use keywords for setting properties, and how syntactic alternatives and removing parentheses leads to better readable expressions. These changes were accompanied by semantic modifications too. Constructors were added to serve as constructors, blocks stemming from functional programming used to free the order of expressions but grouping them visually, and Ruby's reflective and metaprogramming capabilities were rendered to form aliases for existing methods in the objects, greatly reducing the overall manual effort to change the implementation. Ultimately the DSL overcomes much of the weaknesses of the more common object-oriented expressions: (i) No explicit object creation must occur, (ii) the relationships of objects are expressed by the nesting of blocks, (iiI) instead of providing imperative commands, declarative expressions using functional-programming concepts clearly express the intent, and (iv) the declarations can be extended or compressed to any degree. The overall results are more readable and more maintainable expressions.

Acknowledgements

We thank Salikh Zakirov for helpful comments on an earlier draft of this paper. The original work on the DSL has been done with Maximilian Haupt and Matthias Splieth as indicated in [20].

参 考 文 献

[1] Arnold, B. R. T., Deursen, A. V., and Res, M.: Algebraic Specification of a Language for describing Financial Products, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*,

IEEE, 1995, pp. 6–13.

[2] Bahlke, R. and Snelting, G.: The PSG System: From Formal Language Definitions to Interactive Programming Environments, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 8, No. 4(1986), pp. 547–576.

[3] Ballance, R. A., Graham, S. L., and De Vanter, M. L. V.: The Pan Language-Based Editing System For Integrated Development Environments, *ACM SIGSOFT Software Engineering Notes*, Vol. 15, No. 6(1990), pp. 77–93.

[4] Brown, A. B.: Oops! Coping with Human Error in IT Systems, *Queue*, Vol. 2, No. 8(2004), pp. 34–41.

[5] Burgess, M.: Cfengine: a Site Configuration Engine, *USENIX Computing systems*, Vol. 8, No. 3(1995), pp. 309–402.

[6] Burgess, M.: A tiny Overview of Cfengine: Convergent Maintenance Agent, *Proceedings of the 1st International Workshop on Multi-Agent and Robotic Systems, MARS/ICINCO*, Citeseer, 2005.

[7] Consel, C. and Marlet, R.: Architecturing Software Using A Methodology for Language Development, *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming (PLILP)*, Lecture Notes in Computer Science, Vol. 1490, Berlin, Heidelberg, New York, Springer, 1998, pp. 170–194.

[8] Cook, S., Jones, G., Kent, S., and Wills, A. C.: *Domain Specific Development with Visual Studio DSL Tools*, Addison-Wesley Professional, Amsterdam, Netherlands, 2007.

[9] Coplien, J. O.: *Multi-paradigm design for C++*, Addison-Wesley, Boston, San Francisco, et al., 1999.

[10] Couch, A., Wu, N., and Susanto, H.: Toward a Cost Model for System Administration, *Proceedings of LISA '05: Nineteenth Systems Administration Conference*, 2005, pp. 125–141.

[11] Cunningham, H. C.: A Little Language for Surveys: Constructing an Internal DSL in Ruby, *Proceedings of the 46th Annual Southeast Regional Conference (ACM-SE)*, New York, ACM, 2008, pp. 282–287.

[12] Czarnecki, K. and Eisenecker, U. W.: *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, Boston, San Franciso et al., 2000.

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

[14] Elliott, C., Finne, S., and De Moor, O.: Compiling embedded languages, *Journal of Functional Programming*, Vol. 13, No. 03(2003), pp. 455–481.

[15] Flanagan, D. and Matsumoto, Y.: *The Ruby Programming Language*, O-Reilly Media, Sebastopol, 2008.

[16] Groote, J. F., Van Vlijmen, S. F. M., and Koorn, J. W. C.: The Safety Guaranteeing System at Station Hoorn-Kersenboogerd, *Proceedings of the Tenth Annual Conference on Computer Assurance Systems Integrity, Software Safety and Process Security (COMPASS '95)*, IEEE, 1995, pp. 57–68.

[17] Günther, S.: Agile DSL-Engineering and Patterns in Ruby, Technical report (Internet) FIN-018-2009, Otto-von-Guericke-Universität Magdeburg, 2009.

[18] Günther, S.: Engineering Domain-Specific Languages with Ruby, *3. Workshop des Centers for Very Large Business Applications (CVLBA)*, Arndt, H.-K. and Krcmar, H.(eds.), Aachen, Shaker, 2009, pp. 11–21.

[19] Günther, S.: Multi-DSL Applications with Ruby, *IEEE Software*, (2010), pp. available as preprint http://doi.ieeecomputersociety.org/-10.1109/MS.2010.91, to appear in Oct. 2010.

[20] Günther, S., Haupt, M., and Splieth, M.: Utilizing Internal Domain-Specific Languages for Deployment and Maintenance of IT Infrastructures, Technical report (Internet) FIN-004-2010, Otto-von-Guericke-Universität Magdeburg, 2010.

[21] Günther, S. and Sunkle, S.: Feature-Oriented Programming with Ruby, *Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD)*, New York, ACM, 2009, pp. 11–18.

[22] Hudak, P.: Modular Domain Specific Languages and Tools, *Proceedings of the 5th International Conference on Software Reuse (ICSR)*, Devanbu, P. and Poulin, J.(eds.), 1998, pp. 134–142.

[23] Kiczales, G., Rivières, J. d., and Bobrow, D. G.: *The Art of the Metaobject Protocol*, The MIT Press, Cambridge, London, 4th edition, 1995.

[24] Ladd, D. A. and Ramming, J. C.: Two Application languages in software production, *VHLLS'94: Proceedings of the USENIX 1994 Very High Level Languages Symposium Proceedings on USENIX 1994 Very High Level Languages Symposium Proceedings*, Berkeley, CA, USA, USENIX Association, 1994, pp. 10–10.

[25] Lutz, M.: *Learning Python*, O'Reilly Media, Sebastopol, 4th edition, 2009.

[26] Mernik, M., Heering, J., and Sloane, A. M.: When and How to Develop Domain-Specific Languages, *ACM Computing Survey*, Vol. 37, No. 4(2005), pp. 316–344.

[27] Munnelly, J. and Clarke, S.: ALPH: A Domain-Specific Language for Crosscutting Pervasive Healthcare Concerns, *Proceedings of the 2nd Workshop on Domain Specific Aspect Languages (DSAL)*, New York, ACM, 2007.

[28] Recordon, D. and Reed, D.: OpenID 2.0: A Platform for User-Centric Identity Management, (2006), pp. 11–16.

[29] Tanter, É.: Contextual Values, *Proceedings of the 2008 Symposium on Dynamic Languages*

(DLS), ACM, 2008.

[30] Thibault, S., Marlet, R., and Consel, C.: A Domain-Specific Language for Video Device Drivers: from Design to Implementation, (1997), pp. 11–26.

[31] Thomas, D., Fowler, C., and Hunt, A.: *Programming Ruby 1.9 - The Pragmatic Programmers' Guide*, The Pragmatic Bookshelf, Raleigh, 2009.

[32] Van Deursen, A., Klint, P., and Visser, J.: Domain-Specific Languages: An Annotated Bibliography, *ACM SIGPLAN Notices*, Vol. 35(2000), pp. 26–36.

[33] Wampler, D. and Payne, A.: *Programming Scala*, O'Reilly Media, Sebastopol, 2009.

[34] Weinberg, G. M.: *The Philosophy of Programming Languages*, John Wiley & Sons, New York, 1971.