



## THESIS / THÈSE

### MASTER IN COMPUTER SCIENCE

### Reusable Configuration Self-Adaptation through Bidirectional Programming

Colson, Kevin; Dupuis, Robin

*Award date:*  
2016

*Awarding institution:*  
University of Namur

[Link to publication](#)

#### General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

#### Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

University of Namur  
Computer sciences faculty  
Academic year 2015-2016



# **Reusable Configuration Self-Adaptation through Bidirectional Programming**

Kevin Colson

Robin Dupuis

Supervisor: Pierre-Yves Schobbens  
Advisors: Zhenjiang Hu, Lionel  
Montrieux

A thesis presented for the master  
degree in  
computer sciences

# Abstract

In self-adaptive systems, an adaptation strategy can apply to several implementations of a target system. Reusing this strategy requires models of the target system that are independent of its implementation. In particular, configuration files must be transformed into abstract configurations, but correctly synchronizing these two representations is not trivial. We propose an approach that uses putback-based bidirectional programming to guarantee that this synchronization is correct by construction. We demonstrate the correctness of our approach and how it handles typical features of configuration files, such as implicit default values and context overriding. We also show that our approach can be used to migrate configuration files from one implementation to another.

We illustrate our approach with a case study, where we use the same abstract model to adapt two web server implementations. For each implementation, we provide a bidirectional program that correctly synchronizes the configuration file with an abstract model of the configuration. A first scenario demonstrates that the same changes on the abstract model produce, for each implementation, a new configuration that correctly reflects the changes made to the abstract model, without side effects. A second scenario validates the migration of a configuration file from the format used by one web server implementation to another.

**Keywords:** Self-adaptation, synchronization, bidirectional programming, model abstraction

# Acknowledgements

This thesis is based on a paper written in cooperation with Pr. Zhenjiang Hu and Dr. Lionel Montrieux, from the National Institute of Informatics, Tokyo, Japan, as well as Pr. Sebastián Uchitel, from the University of Buenos Aires, Argentina and Pr. Pierre-Yves Schobbens, from the University of Namur, Belgium [Colson et al., 2016]. We would like to thank those persons for their precious advices and their time.

The authors would also like to thank Dr. Hsiang-Shang Ko and Mr. Li Liu, from the National Institute of Informatics, Tokyo, Japan, as well as Mr. Jorge Cunha Mendes, from INESC Technology and Science, Porto, Portugal, for their help with BiGUL development.

Any errors are the authors' own.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>State of the art</b>	<b>8</b>
2.1	Self-adaptive systems . . . . .	8
2.1.1	Self-* Properties . . . . .	10
2.1.2	Adaptation loop . . . . .	12
2.1.3	Reusability . . . . .	17
2.1.4	Models within self-adaptation . . . . .	18
2.2	Bidirectional transformations . . . . .	19
<b>3</b>	<b>Configuration files abstraction</b>	<b>23</b>
3.1	Definition . . . . .	24
3.2	Model Abstraction in Self-Adaptive systems . . . . .	28
3.2.1	Adaptive Layer Reusability . . . . .	29
3.2.2	Configuration Reusability . . . . .	32
<b>4</b>	<b>Bidirectional Programming</b>	<b>34</b>
4.1	Model Synchronization . . . . .	35
4.2	BiGUL . . . . .	37
4.3	Challenges . . . . .	40
4.3.1	Default values . . . . .	40
4.3.2	Context overriding . . . . .	42
<b>5</b>	<b>Case Study</b>	<b>44</b>
5.1	Setup . . . . .	45
5.1.1	Internal representation . . . . .	45
5.1.2	View . . . . .	46

5.1.3	Parsers and pretty printers . . . . .	47
5.2	Scenario 1: Adaptation . . . . .	58
5.2.1	Experiment . . . . .	58
5.2.2	Results . . . . .	59
5.3	Scenario 2: Migration . . . . .	60
5.3.1	Experiment . . . . .	60
5.3.2	Results . . . . .	60
5.4	Evaluation . . . . .	61
5.4.1	Reusability . . . . .	61
5.4.2	Models within self-adaptation . . . . .	64
5.4.3	Threats to Validity . . . . .	65
<b>6</b>	<b>Conclusion</b>	<b>67</b>
	<b>Appendices</b>	<b>75</b>
<b>A</b>	<b>Configuration files</b>	<b>76</b>
<b>B</b>	<b>BiGUL codes</b>	<b>79</b>

# Chapter 1

## Introduction

Self-adaptive systems are often represented in two layers: a target system, and an adaptation layer. Systems modeled around the MAPE-K loop, for example, frequently adopt this distinction, where the adaptation layer is implemented using a feedback loop whose stages are Monitor, Analyze, Plan, and Execute, using a Knowledge base where data about the target system and its environment is stored [Cheng et al., 2009]. This design allows for a clear separation between the system itself (the target system) and the adaptation logic, which is confined to the adaptive layer. Communication between the target system and the adaptive layer is typically implemented using sensors and effectors, which is sometimes captured in a third layer, between the target system and the adaptation layer [Garlan et al., 2004].

A particular type of systems, when implemented using such an architecture, imply that (some of) the artefacts monitored by the adaptive layer are (some of) those that are modified by the effectors. Self-adaptive systems that adapt configuration files are a prime example: changes made to the configuration of the target system are monitored, along with other data sources, and the adaptation layer is able to modify the same configuration files, before applying them to the target system.

Adaptation can be performed on systems that have similar functionalities although they have a different implementation by customizing the adaptive layer for each specific case. For example, a web server may be concretely implemented with Apache, Nginx or an other technology. The adaptation of an aspect of these web servers configuration, like security, will require a

customized adaptive layer for each technology. Even though the analysis and adaptation logic is the same regardless of the technology used, the format of the configuration files changes. This implies a loss of generality and reusability.

Current research on reusability of adaptive layers has focused on providing frameworks for adaptation, allowing developers to customize each phase of the feedback loop [Barna et al., 2015, Garlan et al., 2004], without having to implement the entire adaptation layer themselves. It is then possible for developers to write adaptation logic that relies on abstract models of the system, ignoring implementation details. The adaptation logic can then be reused for several implementations of the same system. However, these approaches rely on users carefully crafting pairs of sensors and effectors, in order for the adaptive layer to keep an up-to-date abstract model of the target system and its environment. The correctness of that synchronization between models and target system is not trivial to prove. A bug in the synchronization will lead to a progressive drift between the target system and the abstract model, which can lead to counter-productive adaptation decisions.

While adaptation mechanisms on web servers usually use the log files, in this thesis, we provide a correct by construction synchronization mechanism that is focused on the adaptation of configuration files. Those configuration files are part of the target system, and describe the configuration of the system. This focus limits the kinds of adaptations that our approach can handle. However, it allows us to guarantee, by construction, the correctness of the synchronization of configuration files (which we call *concrete models*) with their abstract representations.

Abstract and concrete models must be consistent, i.e. changes made to one should be reflected on the other. Typically, this is implemented in an ad-hoc fashion, with back and forth transformations that may be hidden in the adaptation layer’s implementation, or the translation layer in the case of frameworks such as Rainbow. Ad-hoc implementations do not provide any guarantees over the correctness of the synchronization mechanism, and it can be very difficult to reason about it in a systematic way. This is especially the case in models with complex structures, like configuration files. To ensure that abstract and concrete models are consistent, we synchronize them us-

ing bidirectional transformations (BX) [Foster, 2010], automatically derived from bidirectional programs. BXs consist of a pair of functions: a forward transformation (or *get*), and a backward transformation (or *put*). *Get* takes a source as input and generates a view, while *put* takes the original source and the new view as input, and outputs a source where the view has been embedded in the original source [Foster et al., 2009]. *Well-behaved* BXs ensure that the composition of the *get* and *put* functions, or the opposite, is the identity function [Foster, 2010]. Bidirectional programming languages are Domain-Specific Languages (DSLs) that help developers write BXs. In this thesis, we use BiGUL, a putback-based bidirectional programming language and compiler. The behavior of *put* is described with the BiGUL language, and the compiler generates a pair of *get* and *put* functions that are guaranteed to form a well-behaved BX. Our solution guarantees the correctness of the synchronization by construction, ensuring that the models will not drift apart due to errors in an ad-hoc implementation.

Abstract models can also be used to migrate configurations from an implementation to another. We demonstrate the applicability of bidirectional programming to guarantee the correctness of the synchronization, and show how typical constructs in configurations are dealt with. We also report on a case study that illustrates our approach with two web server implementations.

The rest of this thesis is structured as follows: Chapter 2 provides background on self-adaptive systems, and bidirectional programming. Our approach and its uses is presented in Chapter 3. Chapter 4 details our use of bidirectional programming for synchronization. Chapter 5 reports on a case study, made of two scenarios: adaptation and migration, show their implementation and evaluate the outcome. We then conclude in Chapter 6.

# Chapter 2

## State of the art

Our thesis focuses on the potential use of bidirectional programming to allow some reusability in self-adaptive systems. This section introduces the concepts behind those self-adaptive systems and the principles of bidirectional programming and transformations.

### 2.1 Self-adaptive systems

A large and important part of software design consists in predicting and analyzing the cases that the software will have to deal with and making sure the requirements are met in all of those cases. It is generally not easy to find a satisfying coverage of cases, and ensuring that none has been left out is impossible [Laddaga et al., 2006]. Moreover, today's software are often developed and run in an ever changing environment. Their complexity is also increasing over time. This means that such systems need constant adaptation and maintenance according to the changes their context suffers. If the behavior of the software is pre-determined, the exact inputs and environment conditions at runtime don't modify the way the software behave. The adaptation of the system to new conditions is often performed via off-line maintenance, with human intervention delays that may vary greatly. Due to this ever increasing complexity, the cost of such changes can be high. This need for change should be detected, and the modification effected, while the system is running. Consequently, efforts have been spent on developing softwares that can automatically adapt when changes occur. They are named *Self-adaptive systems* [Salehie and Tahvildari, 2009]. Those systems aim to

be able to understand, monitor and modify themselves when a change in the requirements, goals, environment or inputs of the system calls for it.

A DARPA Broad Agency Announcement on self-adaptive software (BAA-98-12) gave a definition of those systems in December of 1997: “Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.” [Laddaga, 1997]

This means that the system can achieve its goals in several ways and has sufficient understanding of its implementation to make effective modifications at runtime. It should then include the functionalities needed to evaluate its own behavior, plan the changes to perform and effect those changes.

The adaptation that self-adaptive systems are able to perform, can cover a large amount of concerns such as security, performance or fault management. Research has been made on applications in areas such as information survivability, control systems, perceptual applications and communication protocols. More specifically, two applications have created self-adaptive systems that survive malicious attacks [Doyle and McGeachie, 2001, Shrobe, 2001]. In addition, several papers describes application of self-adaptive software to communication protocol definition and testing [Adamis and Tarnay, 2001, Harangozó and Tarnay, 2001, Tarnay, 2001]. *Goldman et al.* also presents a paper on the use of self-adaptive software in a hard real-time controller [Goldman et al., 2001] while *Bakay* presents a soft real-time controller [Bakay, 2001].

### 2.1.1 Self-\* Properties

Self-adaptation covers a set of adaptivity properties often called self-\* properties. IBM usually cite 4 of them: self-configuration, self-optimization, self-healing and self-protection [IBM, 2005]. Depending on the considered target system and its goals and policy, the developers may want to treat these aspects as distinct and address each one separately, or even only addressing some of them instead of implementing a whole self-managing system [Kephart and Chess, 2003].

#### Self-configuration

Deploying, configuring, and integrating large, complex systems can be difficult, time-consuming, and error-prone, even for IT professionals. Self-configuring components should automatically adapt to changes in the environment, based on policies provided by the developer. Those policies might represent business-level objectives for example, specifying what is to be achieved but not how to achieve it.

The considered changes could include the introduction of new components, the suppression of existing ones, or heavy modification to the characteristics of the system. For example, when a component is introduced, it would incorporate itself perfectly, and the rest of the system would adapt to its presence. It would learn and consider the configuration of the other components and would notify them of its presence and capabilities so that they could automatically use it appropriately. Dynamic adaptation helps ensure continuous strength and productivity of the IT infrastructure, resulting in business growth and flexibility.

#### Self-optimizing

Complex systems may have a large amount of adjustable parameters that must be set correctly for the system to perform optimally, but, most of the time, only a few people know how to tune them. Self-optimizing components can tune themselves to meet user or business needs. The modifications could mean the reallocating resources to improve overall utilization or ensuring that specific transactions can be completed in a given time-limit. Autonomic systems will continually try to improve their operation by looking for new

opportunities to make themselves more efficient in performance or cost.

For example, without self-optimizing functions, assigning the excess server capacity to some low priority task when an application does not use all of its assigned computing resources would be rather difficult. In such cases, businesses should buy and maintain a distinct infrastructure for each application to meet the peaks of resources needs of that application.

### **Self-healing**

Serious problems in complex systems can sometimes take teams of programmers several weeks to diagnose and fix. Self-healing components are able to detect system malfunctions resulting from software or hardware failure or just bugs, diagnose their causes and initiate policy-based repairs without disrupting the environment. The corrections could mean a component altering its own state or making changes in other components in the environment. Then, the system as a whole becomes more resilient because simple operations are less expected to fail.

For example, using knowledge about the system configuration, such a component could analyze information from log files, possibly interrogating additional monitors. The system would then match the diagnosis with known patches, install an appropriate one, and retry.

### **Self-protection**

Despite firewalls and intrusion detection tools, developers must protect systems from malicious attacks. Self-protecting components can detect hostile behaviors as they occur, or even anticipate problems based on reports, and modify themselves to become less vulnerable. The hostile behaviors can include unauthorized access and use, virus infection and proliferation or denial-of-service attacks. Self-protecting capabilities allow businesses to consistently and continuously enforce their security and privacy policies.

### 2.1.2 Adaptation loop

Adaptation is triggered by changes in a self-adaptive system, its environment, and/or its goals [Cheng et al., 2009]. In this case, changes made by this adaptation have to be effected on the system. The adaptation logic can be added to the system either internally or externally [Salehie and Tahvildari, 2009]. In the internal approach, the adaptation logic is entangled with the core application and is therefore system dependent. This makes it difficult to maintain, evolve, and reuse. In the external approach, the adaptation logic is clearly separated from the core application. Most of the existing approaches adopt the external approach since it allows the realization of some important software qualities such as the reusability and modifiability. This external adaptive layer can be designed in several ways, however, the MAPE-K loop architecture [IBM, 2005], pictured in Figure 2.1, is the most common. It is composed of four consecutive phases, *Monitor*, *Analyse*, *Plan* and *Execute*, as well as a common knowledge base that helps sharing informations between those phases and contains information about the system state and its adaptation policies. *Cheng et al.* define the four phases as follows:

- “The monitor function provides the mechanisms that collect, aggregate, filter and report details (such as metrics and topologies) collected from a managed resource”;
- “The analyze function provides the mechanisms that correlate and model complex situations (for example, time-series forecasting and queuing models). These mechanisms allow the autonomic manager to learn about the IT environment and help predict future situations”;
- “The plan function provides the mechanisms that construct the actions needed to achieve goals and objectives. The planning mechanism uses policy information to guide its work”;
- “The execute function provides the mechanisms that control the execution of a plan with considerations for dynamic updates” [IBM, 2005].

#### Sensors and effectors

Sensors and effectors are very important parts of a self-adaptive system. A set of sensors is used by the **monitor** function to collect the relevant data for the adaptation process. Once the changes to perform have been planned,

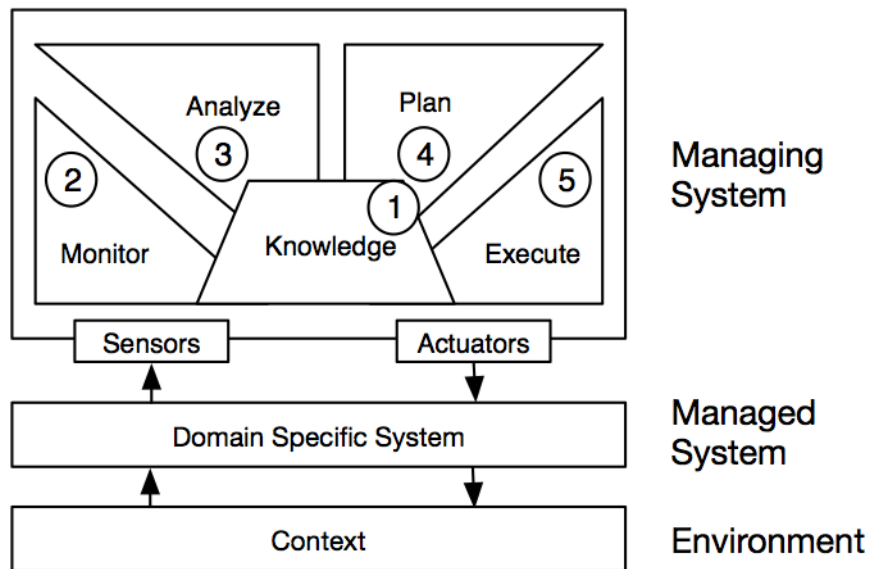


Figure 2.1: The MAPE loop  
<http://homepage.lnu.se/staff/digmsi/MFT/>

the `execute` function uses a set of effectors, sometimes called **actuators**, to apply them to the system. Implementing those two sets can be done during the development of the system, if the developers aim to make it self-adaptive in the first place. They can also be added later, on an already existing system, in order to attach an adaptive layer to it, like discussed by Parekh et al. [Parekh et al., 2006] for example. In any case, implementing the sensors and effectors is often the first step when making self-adaptive software.

Figure 2.2, taken from [Salehie and Tahvildari, 2009], shows the most common techniques for realizing sets of sensors and effectors.

“Logging is likely to be the simplest technique for capturing information from software. The logs need to be filtered, processed, and analyzed to mine significant information. The IBM Generic Log Adapter (GLA) and the Log Trace Analyzer (LTA) [IBM, 2004] are examples of tools for this purpose. Sensing and monitoring techniques from other areas can also be used. For instance, some of the protocols, standards, and formats that have been utilized are: CBE (Common Base Events) [IBM, 2004], WBEM (Web-Based Enterprise Management)<sup>1</sup> (containing CIMCommon Information Model<sup>2</sup>), and SIENA (Scalable Internet Event Notification Architectures) [Carzaniga et al., 2001]. Another noteworthy standard for sensing is ARM (Application Response Measurement)<sup>3</sup>, which enables developers to create a comprehensive end-to-end management system with the capability of measuring the applications availability, performance, usage, and end-to-end response time.

Profiling tools and techniques can also help in defining desirable sensors. The Java environment provides JVMTI (Java Virtual Machine Tool Interface) for this purpose<sup>4</sup>. Software management frameworks, such as JMX (Java Management eXtensions)<sup>5</sup> provide powerful facilities for both sensing and effecting.

Some of the effectors are based on a set of design patterns that al-

---

<sup>1</sup><http://www.dmtf.org/standards/wbem/>

<sup>2</sup><http://www.dmtf.org/standards/cim/>

<sup>3</sup><http://www.opengroup.org/tech/management/arm/>

<sup>4</sup><http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/>

<sup>5</sup><http://jcp.org/en/jsr/detail?id=3>

Entity	Technique	Example
Sensors	Logging	Generic Log Adapter (GLA), Log Trace Analyzer (LTA) [IBM 2005]
	Monitoring & events information models	Common Information Model (CIM), Common Base Events (CBE)
	Management protocols and standards	Simple Network Management Protocol (SNMP), Web-Based Enterprise Management (WBEM), Application Response Measurement (ARM), Siena [Carzaniga et al. 2001]
	Profiling	JVM Tool Interface (JVMTI)
	Management frameworks	Java Management eXtension (JMX)
	Aspect-oriented programming	Build to Manage [IBM BtM], Java Runtime Analysis Toolkit [ShiftOne JRat]
	Signal monitoring	Heartbeat and pulse monitoring [Hinchey and Sterritt 2006]
	Effectors	Design patterns
Architectural patterns		Microkernel pattern, Reflection pattern, Interception pattern [Buschmann et al. 1996; Alur et al. 2001]
Autonomic patterns		Goal-driven self-assembly, self-healing clusters and utility-function driven resource allocation [Kephart 2005]
Middleware-based effectors		Integrated middleware, Middleware interception [Popovici et al. 2002], Virtual component pattern [Schmidt and Cleeland 1999]
Metaobject protocol		TRAP/J [Sadjadi et al. 2004]
Dynamic aspect weaving		JAC [Pawlak et al. 2001], TRAP/J [Sadjadi et al. 2004]
Function pointers		Callback in CASA [Mukhija and Glinz 2005]

Figure 2.2: Techniques for Sensors and Effectors

low the software system to change some artifacts during runtime. For instance, wrapper (adapter), proxy, and strategy are well-known design patterns for this purpose [Gamma et al., 1995]. *Landauer and Bellman* utilize the wrapping idea at the architecture level of adaptive systems [Landauer and Bellman, 2001]. Moreover, microkernel, reflection, and interception are architectural patterns suitable for enabling adaptability in a software system [Alur et al., 2001, Buschmann et al., 1996]. Furthermore, *Kephart* mentions several design patterns, namely goal-driven self-assembly, self-healing clusters, and utility-function-driven resource allocation for self-configuring, self-healing, and self-optimizing, respectively [Kephart, 2005].

An important class of techniques for effectors is based on middleware. In these solutions, system developers realize effectors at the middleware layer by intercepting the software flow [Popovici et al., 2002], or by using design patterns [Schmidt and Cleeland, 2000]. Other solutions have been proposed for implementing effectors using dynamic aspect weaving (e.g., in JAC [Pawlak et al., 2001]), metaobject protocol (e.g., in TRAP/J [Sadjadi et al., 2004]), and the function pointers technique (e.g., in CASA [Mukhija and Glinz, 2005] for realizing callback). Some middlewares provide support for dynamic aspect weaving. For example, JBoss has an AOP module, with the capability of dynamic weaving.” [Salehie and Tahvildari, 2009]

### 2.1.3 Reusability

Other works have aimed to provide some form of reusability of the adaptive layer on different target systems. For example, *Klein et al.* introduce optional code, which can be deactivated dynamically, to self-adaptive systems [Klein et al., 2014]. *Garlan et al.* show the use of a framework, Rainbow, composed of reusable parts to which the user can hook personalized code [Garlan et al., 2004]. Rainbow was extended by *Swanson et al.* with REFRACT, which brings failure avoidance components and algorithms [Swanson et al., 2014]. *Barna et al.* propose Hognu, a platform for deploying self-managing web applications on cloud infrastructures [Barna et al., 2015]. *Ramirez and Cheng* introduce adaptation patterns [Ramirez and Cheng, 2010]. Reuse in self-adaptive systems can also be achieved using *Dynamic Software Product Lines*. *Abbas* uses variability in product lines to identify assets that can be reused across self-adaptive systems [Abbas, 2011]. *Abbas et al.* later study different types of variability, and their consequences on reuse [Abbas and Andersson, 2015]. We compare those approaches with the results given by ours in Section 5.4.

### 2.1.4 Models within self-adaptation

Other works have also researched around the use of models in self-adaptation. *Vogel and Giese*, for example, present a model-driven approach for adaptation that contains different types of models for specific adaptation levels [Vogel and Giese, 2010]. They also present an approach to ease the development of architectural monitoring based on incremental model synchronization [Vogel et al., 2010]. They demonstrate an executable modeling language for ExecUtable RuntimeE MegAmodels (EUREMA), using megamodels. Megamodels are models that represent a system at runtime along with its adaptation activities [Vogel and Giese, 2014]. *Angelopoulos et al.* compare Rainbow with their framework, Zanshin, which is requirement-based instead of architecture-based [Angelopoulos et al., 2013]. *Georgas et al.* use a model to record the history of a managed system's states [Georgas et al., 2005]. This model can be used by a developer to reconfigure a system in another state if he thinks that the current state can lead to a dangerous situation. *Bailey et al.* perform adaptation on Role-Based Access Control (RBAC) models at run-time by changing the access control policies, while ensuring that adapted policies satisfy some security constraints [Bailey et al., 2014].

## 2.2 Bidirectional transformations

Bidirectional transformations (BX) are used to synchronize the contents of two related documents. A BX is a pair of transformations between a source document and a view document. As depicted in Figure 2.3, the forward transformation, called *get*, takes a source as input, and produces a view. The backward transformation, called *put*, takes two arguments as input, an updated view and the original source, and produces an updated source, where changes made to the view are embedded into the source. The original source is given as input for the function to consult to restore any information not present in the view. The two transformations are defined as follows in Haskell [Foster et al., 2009]:

```
get :: Source -> View
put :: Source -> View -> Source
```

A BX could, for example, synchronize a non-empty list of elements (the source) with the first element of the list (the view):

```
get (x:xs) = x
put (x:xs) y = (y:xs)
```

A subset of BX is called *well-behaved* bidirectional transformations, sometimes called *lenses* [Foster, 2010]. They provide *well-behaved* synchronization between source and view, meaning that the combination of the put and get functions and the opposite is the identity function. The need for such synchronization originates from the "view update" problem: given a source document representing data and a view document representing a subset of this data, not necessarily in the same formalism, translate changes made on the view into equivalent changes in the source. In practice, the extraction of the view from the source intentionally bypasses some information, such as comments, formatting details or unnecessary parts of data, because the purpose of the view often is to present this data more concisely. The opposite transformation, from the updated view to the source, must then restore these missing parts. As such, most of the time, well-behaved bidirectional transformations are not bijective functions, as multiple sources, namely the ones that only differs by those left out details, will map to the same view. A bijective BX would imply a very strict condition: the two models must contain identical information, only presented differently. In particular, it is

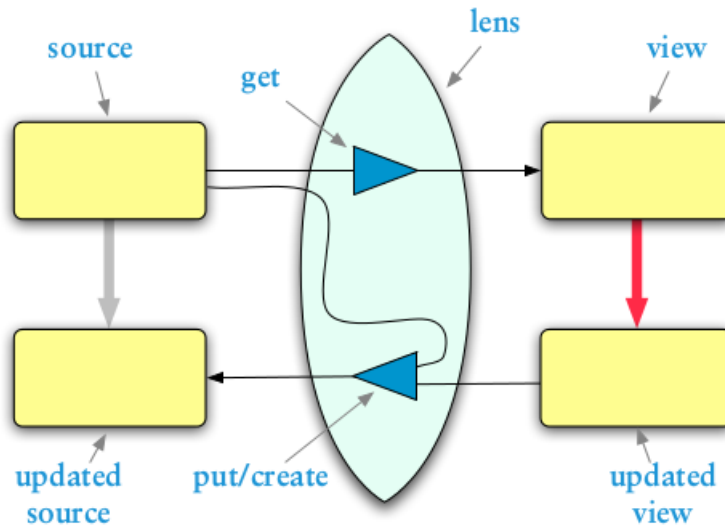


Figure 2.3: The put-get lens

impossible to define a bijective BX between two sets of models if they have different cardinalities [Stevens, 2008].

To be considered well-behaved, a BX has to satisfy two laws, *GetPut* and *PutGet*, defined as follows [Fischer et al., 2015]:

```

put s (get s) = s --GetPut
get (put s v) = v --PutGet

```

In other words, the *GetPut* law states that extracting a view from a source, then putting the unmodified view back into the source should yield the original source. It ensures that the *put* direction correctly restores any information present in the source but not taken in the view. The *PutGet* law states that any view put back in a given source then extracted again, without modification, should give the exact same view. It limits the use that *put* can make of its inputted source, as it can only use it for the parts of information that are not present in the view. These laws are effectively weaker than forcing *get* and *put* to be the inverse of each other. This would mean that both would be bijective functions and would keep them from doing what makes well-behaved BX so useful in practice: abstracting unimportant details and

unnecessary data.

We can show that our previous example, synchronizing a list with its first element, is a well-behaved BX, as it satisfies these two laws.

```
get (put (x:xs) y) = get (y:xs) = y
put (x:xs) (get (x:xs)) = put (x:xs) x = (x:xs)
```

In this thesis, when talking about synchronization mechanisms, the term ‘correct’ is to be understood as the verification of the PutGet and GetPut laws, ensuring the source and the target will not drift apart from each other. This term does not imply that the directives abstracted from different sources to the same view have the same effects on the web server, as it is the developer’s job to ensure this.

Several programming languages exist to help developers write BXs. They usually are either *get-based* programming languages [Hidaka et al., 2011, Bohannon et al., 2008], where the *get* function is provided by the developer, and a *put* function automatically derived, to produce a well-behaved BX; or *putback-based* programming languages [Ko et al., 2016], where the *put* function is provided by the developer, and a *get* function automatically derived, to produce a well-behaved BX. *Get-based* languages are more frequently used, as it often is easier to specify how the desired information should be extracted from the source than specifying how the updated information should be embedded back into the source. However, for each *get* function, there may be many *put* functions that form a well-behaved BX. The advantage of *putback-based* languages is that, under some conditions, given a *put* function, there is *at most* one *get* function that forms a well-behaved BX [Fischer et al., 2015].

Examples and applications of bidirectional transformations can be highlighted through the literature. An example of synchronization between models supporting the Atlas Transformation Language (ATL) is offered by *Xiong et al.* [Xiong et al., 2007]. They propose an automatic approach to synchronizing models which are conform to their respective metamodels. Metamodels are related by a unidirectional model transformation. They are able to generate a synchronization infrastructure from that transformation, a process very similar to get-based bidirectional programming languages. This

means that one of potentially many possible putback transformations will be chosen for the user. *Czarnecki et al.* present notes from the GRACE International Meeting on Bidirectional Transformations where the multidisciplinary aspects of bidirectional transformations are presented, including model and graph transformations [Czarnecki et al., 2009]. Another application of BXs for synchronizing documents is presented by *Hu et al.* [Hu et al., 2004]. This application focuses on a XML editor that supports dynamic refinements of a structured document. *Song et al.* present an algorithm that wraps any BX into a synchronizer, to allow for both the source and the view to be updated simultaneously [Song et al., 2011]. *Foster et al.* propose a general theory of *quotient lenses* [Foster et al., 2008]. They are bidirectional transformations that are well-behaved modulo a set of equivalence relations defined by developers. This would allow the implementation of BXs that would not be well-behaved due to some inessential details such as whitespace. It also is a get-based bidirectional programming approach. Another example where BXs are applied is *Yu et al.*'s synchronization between models and generated code by recording manual changes made to the code in a BX, and replaying them when the manually edited code is overwritten by the code generator [Yu et al., 2012]. They use a get-based bidirectional programming language.

# Chapter 3

## Configuration files abstraction

In this Chapter, we introduce configuration files, and illustrate how abstract models can facilitate the development of reusable self-adaptation mechanisms that are not tied to a particular implementation of a system. We also highlight the difficulty of manually developing correct synchronization mechanisms. We use two web server implementations as an example: Apache HTTP Server<sup>1</sup>, and Nginx<sup>2</sup>.

We then describe our solution for abstracting configuration files into abstract models; we show how our solution allows for the reuse of adaptation across various implementations of the same system, and how it allows for the translation of a configuration file from one implementation to another, while conserving the configured behavior.

---

<sup>1</sup><https://httpd.apache.org/>

<sup>2</sup><http://nginx.org/>

## 3.1 Definition

Configuration files allow users to specify how they want an application to behave. They generally follow a tree structure. Different implementations of a same service, e.g., a web server, will likely have similar configuration options, but the syntax of the configuration files, as well as the entries available, may change between implementations, or between versions. For example, both Nginx and Apache allow for log configuration, but in different ways:

---

```
access log in Nginx
```

```
access_log "/var/logs/access.log";
```

---

```
access log in Apache
```

```
LogFormat "%v:%p %h %l %u %t \"%r\" %>s %O \"%{Referer}i\"  
→ \"%{User-Agent}i\" vhost_combined  
CustomLog "/var/logs/access.log" vhost_combined
```

Both Nginx and Apache allow users to set the path of the access log file, but Apache also provides additional options to specify the format of the log file.

Moreover, a web server can behave differently for some of the websites it serves, e.g., by serving some of its content on a secure connection only, generating error pages, or requiring authentication. Configuration files reflect that ability through contexts. For instance, in Nginx, a `server` context may contain the default behavior that will display a web page, while another `server` context will contain the entries to handle secure connections. Those functionalities are configured in Apache using the `VirtualHost` context, which defines a virtual server and its associated behavior:

---

```
Contexts in Nginx
```

```
http {  
    server { # http server  
        listen 80;  
        # ...  
    }  
    server { # ssl server  
        listen 443;  
        # ...  
    }  
}
```

```
}  
}
```

---

## Contexts in Apache

---

```
<VirtualHost *:80>  
    # http server  
</VirtualHost>  
<VirtualHost *:443>  
    # ssl server  
</VirtualHost>
```

In addition to entries and contexts, configuration files can support other features, such as default values or context overriding. A default value is a value assumed for an entry that does not appear in a configuration file. For example, with Nginx:

```
http {  
    keepalive_timeout 75s;  
}
```

the *keepalive\_timeout* entry has a default value of *75s*. Therefore, this configuration displays the same behavior as the following, where *keepalive\_timeout* has been omitted:

```
http {}
```

Context overriding infers the value of a missing entry in a certain context by looking at the value for this entry in the closest ancestor context that defines it, or the default value if no ancestor defines it. We again use Nginx for this example:

```
http {  
    keepalive_timeout 100s;  
    server {  
        keepalive_timeout 100s;  
    }  
}
```

will be equivalent to the following, since the value of the `keepalive_timeout` entry for the `server` context is defined within its `http` ancestor.

```

http {
    keepalive_timeout 100s;
    server {
    }
}

```

Adaptation logic that directly uses the configuration files cannot easily be reused with different implementations of the target system, or sometimes even with different versions of the same implementation. For example, the path to access logs is represented differently in Apache and Nginx:

```

access_log "/var/logs/access.log";

```

With Apache, we need to define the format of the data that will be written in the file, then the path to the file with the format of the log entries, in two separated entries:

```

LogFormat "%v:%p %h %l %u %t \"%r\" %>s %O \"%{Referer}i\"
↪ \"%{User-Agent}i\"" vhost_combined
CustomLog "/var/logs/access.log" vhost_combined

```

The adaptation mechanisms would be different for Apache and Nginx and wouldn't be reusable, even though they provide the same functionality. In Nginx, the adaptation mechanism only needs to change one instruction, while in Apache it needs to change two.

However, abstracting the implementation details into an abstract model would allow for the reuse of the same adaptation logic to adapt the log files' path, both on Apache and Nginx servers. This is the first problem we address.

This abstract model must be synchronized with the configuration file. Typically, this is implemented in an ad-hoc fashion, with back and forth transformations that may be hidden in the adaptation layer's implementation, or the translation layer in the case of frameworks such as Rainbow [Garlan et al., 2004]. Ad-hoc implementations do not provide any guarantees over the correctness of the synchronization mechanism. Yet, a buggy implementation will lead to the abstract model drifting away from the configuration files it is supposed to represent.

In the log file example, if the access log format, used in the Apache implementation, is not part of the abstract model, any changes to the abstract model will have to be reflected to the configuration file, without modifying the log format. Default values and context overriding will make it difficult to prove that the synchronization mechanism is correct. A buggy synchronization mechanism can lead to unproductive adaptation decisions, based on incorrect data. Hence, the second problem we address is the generation of a synchronization mechanism that is correct *by construction*.

## 3.2 Model Abstraction in Self-Adaptive systems

By abstracting the specificities of each model into a common abstract model, we are able to reuse the analyze and the plan phases. This abstract model must be synchronized with a concrete (i.e., implementation-dependent) model of the configuration (Figure 3.1). Various implementations of the same system would each have their own concrete model, all feeding into an abstract model (Figure 3.2). This abstract model is extracted according to the data that the adaptation layer requires. It can contain the shared information of the concrete models, or only the data for an aspect developers want to focus on, such as security or performance. Our approach guarantees, by construction, that the synchronization between concrete and abstract model is well-behaved, and allows developers to reuse general analyze and plan phases for each system by deploying them on the abstract models.

In addition to facilitating the reusability of adaptive layers, our approach also provides a way to copy parts or the entirety of the information between systems that can have a common abstraction. The system knowledge represented in the abstract model can be replaced by the data contained in another abstract model and this new configuration can then be copied in the new system. In the web server example, a concrete model of an Apache configuration could be abstracted. Then, this information could be copied into the abstract model of another web server, and a *put* transformation, using an empty source, could translate it into the desired concrete configuration file. This second server doesn't need to be implemented in Apache like the

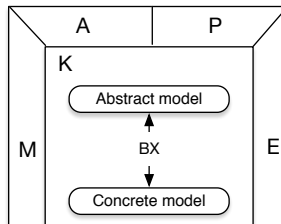


Figure 3.1: model abstraction in MAPE loop

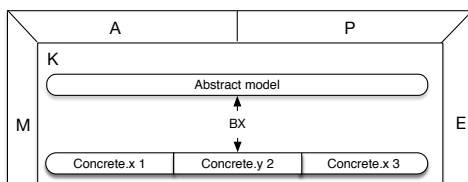


Figure 3.2: abstract model equivalent for all systems

first one. This would effectively copy the configuration from an implementation to another.

Making those adaptation mechanisms reusable means that each part that was specific to an implementation has to be more generic to suit all implementations, or has to be lost if it cannot be generalized. For example, the abstract model for access log files in web servers could have this format, where "" as value would mean that there is no log file:

```
abstractLog :: String
```

The abstract models for both web servers could then be:

```
_____ Abstract for Nginx (with log file) _____
```

```
abstractLog = "/var/logs/access.log"
```

```
_____ Abstract for Apache (without log file) _____
```

```
abstractLog = ""
```

This example shows that an abstract model can contain less data than a concrete one. Here, the log format, available in the Apache configuration, is omitted in the abstract model. It was necessary in order to construct a common type for Apache and Nginx, because of the absence of the ability to change the format in Nginx.

### 3.2.1 Adaptive Layer Reusability

In our approach, the adaptation phases work on different models (Figure 3.3). The monitor phase and the execute phase work on the concrete models of

the systems and therefore need to be customized for each implementation, since the concrete models keep their specificities. In contrast, the analyze and plan phases can both work on the abstract models, and can therefore be reused across several implementations of the target system. For example, an `abstractLog` instruction on the abstract model could represent the following Nginx and Apache configurations:

```
_____ Nginx access log config _____  
access_log "/var/logs/access.log";
```

```
_____ Apache access log config _____  
LogFormat "%v:%p %h %l %u %t \"%r\" %>s %O \"%{Referer}i\"  
↳ \"%{User-Agent}i\" vhost_combined  
CustomLog "/var/logs/access.log" vhost_combined
```

Both would produce the same abstract model:

```
abstractLog = "/var/logs/access.log"
```

An adaptation rule may specify that, when disk space is short on a server, no more accesses will be logged. The adaptation will consist in stopping the collection of access logs. The analyze and plan phases will reflect this in the abstracted models.

```
_____ Before adaptation _____  
abstractLog = "/var/logs/access.log"
```

```
_____ After adaptation _____  
abstractLog = ""
```

As the *execute* phase uses concrete models, changes to the abstract model must be reflected to the concrete model. If the server uses Apache, we get the following, as the `LogFormat` must not be removed in case it is used in some other entry:

```
LogFormat "%v:%p %h %l %u %t \"%r\" %>s %O \"%{Referer}i\"  
↳ \"%{User-Agent}i\" vhost_combined
```

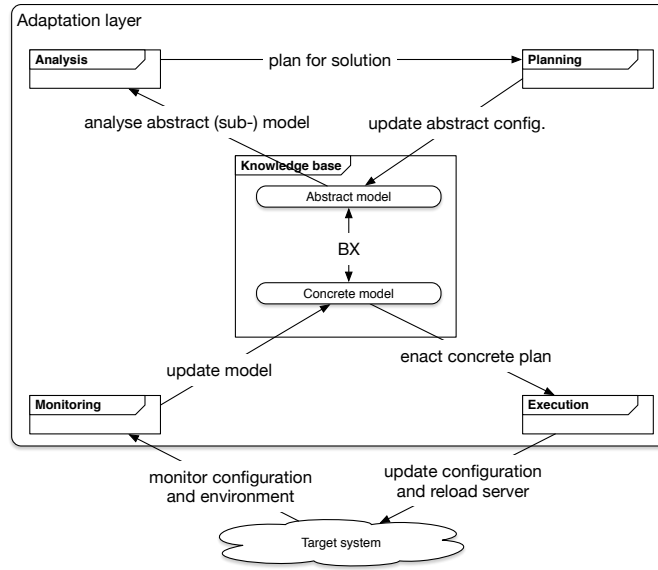


Figure 3.3: Architecture of the approach

If the server uses Nginx, no more instructions about the log file for accesses will appear.

In addition, our approach allows for the extraction of a subset of a model, for the adaptation of a particular concern. For example, a MAPE loop for security adaptation would only need a model containing web servers' security features. Our approach achieves this with two alternatives. Figure 3.4 shows that different partial abstract models can be extracted from the same concrete model, for different adaptation concerns. Figure 3.5 uses our approach's composition capabilities. First, a complete abstract model is extracted from the concrete model. Then, partial models are extracted from the abstract model. Because our approach to synchronization can be composed, we can guarantee that changes made to a partial abstract model will be reflected to the complete abstract model, and then to the concrete model.

The biggest concern with this approach is to write this "mapping" between the abstract configuration and the concrete configuration. Those models need to be synchronized, since the execute and monitor phases will work only on the concrete configuration.

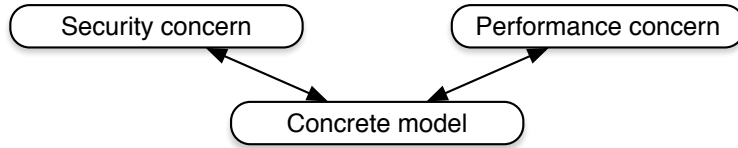


Figure 3.4: direct concern extraction

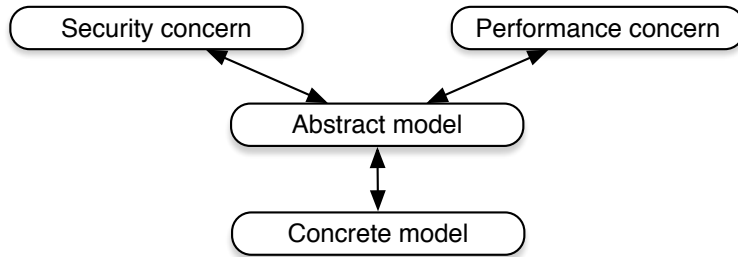


Figure 3.5: indirect concern extraction

### 3.2.2 Configuration Reusability

Using our approach, the user can also reuse parts or all of the abstract configuration from one system to another. This allows for the duplication of a configuration, as well as for migration, which consists in modifying the technology used for a system while preserving its behavior.

Developers can reuse the view of a system A for another system, B. The information in the view will then be reflected in system B and it will behave like system A, as pictured in Figure 3.6.

For example, a user might want to migrate an Apache server to Nginx, without losing the configuration. In this case, by only replacing the current web server with Nginx, and replacing the synchronization mechanism in the adaptive layer with the one related to Nginx, the configuration in the abstract model will be copied in the new server.

The user might also want to add a new web server to the set and have it behave like an existing one. By copying the information in the abstract model of the existing one into the abstract model of the new one, and using

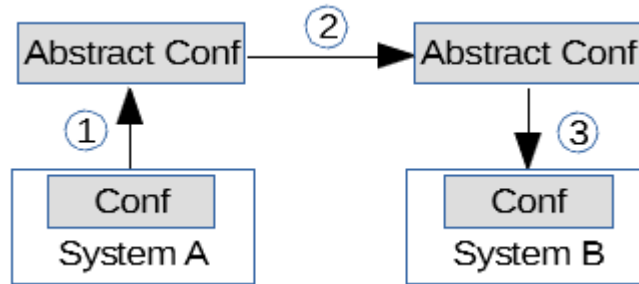


Figure 3.6: Configuration reusability

the correct synchronization mechanism for the chosen implementation, the behavior will be correctly replicated.

In our access log example, the configuration of the first system, implemented with Apache and containing the following:

```

LogFormat "%v:%p %h %l %u %t \"%r\" %>s %O \"%{Referer}i\"
  ↪ \"%{User-Agent}i\" vhost_combined
CustomLog "/var/logs/access.log" vhost_combined
  
```

would be transformed into the abstract model, which would then contain

```
accessLogPath = "/var/logs/access.log"
```

assuming we only abstract the path. This model could be reused in another system that would be implemented with Nginx. After the abstract configuration is reflected in the new system, the Nginx configuration file would contain:

```
access_log "/var/logs/access.log"
```

and this part of the configuration would have been successfully copied.

## Chapter 4

# Bidirectional Programming

In this section, we further explain our use of bidirectional transformations. We show that the synchronization mechanism needed between our concrete and abstract model is difficult to implement correctly, before making the case for a synchronization mechanism that is correct by construction using bidirectional programming. We then present the language used to implement this solution: BiGUL. Finally, we show how challenges caused by typical constructs in configuration files can be overcome with putback-based bidirectional programming.

## 4.1 Model Synchronization

A synchronization mechanism between the concrete and abstract models considered in our solution must be well-behaved, and must handle the typical constructs of configuration files discussed in Section 3.1, meaning default values and context overriding.

Model synchronization usually consists of a pair of transformation functions, one for each direction. Usually, the user writes both but has no guarantee of their well-behavedness. For example:

```
_____ Example get behavior _____  
get {  
  if (a empty)  
    then return default  
    else return a  
}
```

```
_____ Example put behavior _____  
put {  
  if (b == default)  
    then return empty  
    else return b  
}
```

This pair of functions may seem well-behaved. However, if `a` contains the value `default`, the `get` function will return the `default` value too. Since the `put` function returns `empty` for the `default` input, the value of the first model will be modified from `default` to `empty` after a combination of `get` and `put`. These bug cases are sometimes hard to find and fix. Proving that a pair of functions is well-behaved can be difficult and time-consuming.

This problem can be solved by using bidirectional programming, since languages like BiGUL guarantee that the computed bidirectional transformations will be well-behaved. The kind of functions we defined as example can't be written using BXs as it is not well-behaved. However, if the confusion between `empty` and `default` was intentional, for any relevant reason the developer might have, *Foster et al.* propose a general theory of *quotient*

*lenses* [Foster et al., 2008]. They are bidirectional transformations that are well-behaved modulo a set of equivalence relations defined by developers. This would allow the implementation of BXs that would not be well-behaved due to some inessential details, like in our example.

Each BX is specific to a pair of models. Therefore, a new bidirectional program has to be defined for each implementation used in the system. The same BX can be reused if a new system is deployed with the same implementation as an existing one. Since adaptation made on the abstract model remains the same, only the BX would need to be replaced for different implementations of the system.

## 4.2 BiGUL

To implement the bidirectional transformations, we use the putback-based bidirectional programming language called BiGUL [Ko et al., 2016]. Below is a simple BiGUL program showing the type definition of both source and view and a basic BX extracting an element from the source to construct the view.

```
1 data Src = Src {sa :: Int,
2               sb :: String,
3               sc :: Int} deriving (Show)
4 data View = View {va :: String} deriving (Show)
5
6 abc :: BiGUL Src View
7 abc = $(rearrAndUpdate
8       [p| View {
9           va = a
10          }|]
11      [p| Src {
12          sb = a
13          }|]
14      [d| a = Replace
15          |])
```

In this example, the source model, defined on lines 1 to 3, contains three fields; the view model, defined on line 4, only contains one field. The program, defined on lines 6-15, specifies the *put* behavior. The program's signature, on line 6, indicates that it matches a source of type `Src` with a view of type `View`. Line 7-15 are a `rearrAndUpdate` instruction. This matches elements of the source with elements of the view, and performs the specified operations to update the source. `rearrAndUpdate` takes three arguments: a pattern for the view, a pattern for the source, and a pattern for the operations to perform on elements of the source that were matched with elements of the view. Lines 8 to 10 indicate that the element `va` in the view will be matched to `a`. Lines 11 to 13 indicate that the element `sb` in the source will be matched to `a` as well. Finally, lines 14 and 15 indicate that the element matched with `a` in the source will be replaced by the element matched with `a` in the view. Therefore, the other elements, `sa` and `sc` in the source will be left unchanged.

This kind of construct works well as long as we are working with single sources and views. However, we need other ones when working with more complex structures, such as lists. A BX synchronizing lists of records, for example, cannot match the different sources with the views based on their position, since it is not static. To solve this kind of problem, the BiGUL construct `Align` has been implemented. The following is a small BiGUL program showing its basic usage with a BX performing the same synchronization than in the previous example on each element of a list. We use the same source and view type definition as above.

```

1 alignDemo :: BiGUL [Src] [View]
2 alignDemo = Align
3   --source condition
4   (\ _ -> return True)
5   --match
6   (\ (Src {sb = S}) (View {va = V}) -> return (S == V))
7   --transformation
8   ($ (rearrAndUpdate
9     [p| View {
10       va = a
11     } |]
12     [p| Src {
13       sb = a
14     } |]
15     [d| a = Replace;
16       |] ))
17   --create
18   (\ View {
19     va = a
20   } -> return Src {
21     sa = 1,
22     sb = a,
23     sc = 5
24   })
25   --conceal
26   (\ _ -> return Nothing)

```

In this example, the signature at line 1 indicates that the program matches a list of sources of type `Src`, with a list of views of type `View`. The rest of

the program, from line 2 to line 26, constitutes an `Align` instruction. First, a condition on the source must be given. Any source not matching it won't be considered. All sources are considered here so line 4 ensures the condition always returns `True`. Line 6 defines a matching condition between the source and the view. `Align` finds for each view the first matching source that has not been matched with previous views, and updates the source using the program defined on lines 8-16, which is the same as the previous example. If there is no matching source, one is created using the creation argument, as shown on lines 18-24. After creation, the created source should match with the view as determined by the matching condition. Finally, for a source not matched with any view, the concealment argument is applied. Here, line 26 deletes those sources.

Many other constructs and instructions exist in BiGUL. However, at the time of writing, it was still a young and evolving language so most of those were constantly changing and new ones appearing, making presenting an exhaustive list of them difficult and of little use. The interested reader can learn more about it in the BiGUL paper [Ko et al., 2016] or by visiting the BiGUL development repository<sup>1</sup>.

BiGUL guarantees that, if it can compile a bidirectional program into a BX, that BX will successfully run only if it is well-behaved. Hence, using BiGUL for synchronization between concrete and abstract models guarantees, by construction, the correctness of the synchronization. By contrast, developers writing couples of unidirectional transformations will have to ensure that their implementation is correct. This can be tedious and time-consuming.

---

<sup>1</sup>[https://bitbucket.org/prl\\_tokyo/bigul/src](https://bitbucket.org/prl_tokyo/bigul/src)

## 4.3 Challenges

### 4.3.1 Default values

Default values can vary depending on the implementation (Apache, Nginx, ...), or even the version of an implementation (e.g., Apache 1.4 vs. Apache 2.0).

The abstract model needs to be independent from the implementation of the target system. The adaptation layer cannot infer the value of an empty field in the abstract model by using default values, since it doesn't know which technology is used. Therefore, the bidirectional transformation between the two models must replace any field that would be empty in the abstract model by the correct default value.

Each BX is passed the default values specific to the version of the implementation considered. The challenge is to add this knowledge to the transformation while maintaining the guarantee that it is well-behaved. When reflecting changes from the abstract model to the concrete one, the fields that were empty in the original concrete model must stay empty, unless their value was modified and is now different from the default value.

The following pseudo-code shows how we solve this issue.

```
1  addDefault def {
2    if (viewValue = def)
3      then if (oldSourceValue empty)
4          then newSourceValue = empty
5          else newSourceValue = viewValue
6      else newSourceValue = viewValue
7  }
```

Keep in mind that this pseudo-code only defines the put behavior, while the get behavior is automatically inferred. For example, the *ssl* instruction, if not defined, will be empty in the source. The get behavior inferred from the put will write this instruction to its default value *off* in the view. When reflecting changes to the source, if the value in the view is equal to the default value, we check the value in the original source. If it is empty, we know that the default value in the view was inferred and we putback *empty*. If it is not,

then the user might want this instruction to appear in the file despite being set to the default, and we write it in the updated source.

The following shows the BiGUL code implementing this behavior. Some variants of this function have also been added to handle some specific cases. The complete implementation can be found on our repository<sup>2</sup> or in the appendices B.

```

1  addDefault :: String -> BiGUL m (Maybe String) String
2  addDefault def = CaseV [
3    ((return . (== def)),
4      (CaseS [
5        $(normal [p| Nothing |]) $ Rearr (RConst def) (EIn
6          ↪ (ELeft (EConst ()))) Replace,
7        $(normal [p| (Just _) |]) $ $(rearr [| \ x -> (Just x)
8          ↪ |]) Replace
9      ])),
10   ((return . (/= def)),
11     ($(rearr [| \ x -> (Just x) |]) Replace) )
12 ]

```

Line 1 gives the signature of the function. It takes the default value for the given directive as an argument and maps a `Maybe String`, the source, with a `String`, the view. On line 2, the default value is assigned to `def` and we test a condition on the view using the `CaseV` construct. If the value from the view is equal to the default one (line 3), we then test a condition on the original source using the `CaseS` construct at line 4. Line 5 defines the case where the source value was empty, thus assigning empty to the updated source as well, and line 6 defines the case where the original source contained a value, thus replacing it with the updated value from the view. Line 8 is the other branch of the `CaseV` construct from line 3, where the value from the view differs from the default one. In this case, line 9 states that the value from the original source should be replaced by the one from the view.

---

<sup>2</sup><https://github.com/pr1-tokyo/bigul-configuration-adaptation>

### 4.3.2 Context overriding

Adding the knowledge of the default values to the transformation brings a new challenge. Many configuration files use context overriding, as discussed in Section 3.1. Therefore, an undefined directive in a context should not always be considered to represent its default value. If the directive is defined in an ancestor, its value is inherited in the nested contexts, unless it is redefined. Figure 4.1 shows an example. The default value for `ssl` is `off`. While `ssl` is not defined in the `server` context on the left hand side, it *is* defined in the parent, and hence applies to the `server` context as well, instead of the default value. On the right hand side, the value of `ssl` has been explicitly set to `on` in the `server` context. Therefore, both sides are equivalent.

In bidirectional programs, when getting the abstract model from the concrete one, empty fields can't be automatically replaced by their default values. The BX must check the ancestor contexts for any value that would override it.

While not implemented in our case study, we can prove that it is possible to write a bidirectional program that handles context overriding properly, and produces a well-behaved BX.

The `put` function will update elements one by one. The challenge consists in providing to our function some knowledge about the fields related to the element it is currently updating. We define `put'`, a `put` function that takes an extra argument: the result of a function  $f(v)$  that provides the necessary information about the ancestors of the currently updating element. We use it to redefine `put`:

```
put s v = put' (f v) s v
```

Because of how `put'` can be implemented in BiGUL, we know that the result of  $f(v)$  will not be used in the generated `get'`, which we can use to redefine `get`:

```
get s = get' _ s
```

BiGUL guarantees a well-behaved bidirectional transformation, therefore we know that the `PutGet` and `GetPut` laws hold for `put'` and `get'`:

```

# ...
ssl on;
# ...
server {
  # ...
  # (ssl undefined)
  # ...
}

```

↔

```

# ...
ssl on;
# ...
server {
  # ...
  ssl on;
  # ...
}

```

Figure 4.1: Context overriding

```

get' _ (put' (f v) s v) = v
put' (f v) s (get' _ s) = s

```

We can then show that `get` and `put` satisfy the PutGet and GetPut laws, and therefore form a well-behaved BX:

```

get (put s v) = get' _ (put' (f v) s (get' _ s))
               = get' _ s = v
put s (get s) = put' (f v) s (get' _ s) = s

```

This proves that context overriding can be handled using put-based bidirectional programming, while, as previously said, not implemented in our case study.

# Chapter 5

## Case Study

This case study is based on web server configuration files. We use two implementations: Apache and Nginx; we consider their `apache2.conf` and `nginx.conf` configuration files, respectively.

We designed two scenarios. The first one simulates an adaptation layer that results in a switch from insecure client connections to only secure ones using *SSL*. The second scenario simulates an adaptation layer that adds a new web server to the server pool it is handling. This can be used to reduce the system load or improve overall system quality. This new server needs to be configured. We suppose that its configuration has to be the same as another web server in the pool. However, those two servers aren't implemented with the same technology. We show that the copied abstract configuration from a server using implementation A to a server using implementation B is correctly reflected in the concrete configuration of the new server, using our approach.

This case study focuses on BX within a self-adaptive system, and hence some of the operations that would be performed on a live system are simulated or ignored. For example, in the first scenario, a self-adaptive system would have to update the configuration file on the target system, as well as reload the web server, for the new configuration to be taken into account; in the second scenario, a new server would need to be commissioned, before the configuration file can be transferred, and the web server started.

## 5.1 Setup

### 5.1.1 Internal representation

We present here the sample configuration files used in our case study. Annex A.1 shows the Nginx configuration file and Annex A.2 shows the Apache configuration file. They are simple configuration files, each defining log file locations, a single context where simple HTML files are served, and a few other configuration items for the servers to run correctly.

The BiGUL program can't handle configuration files directly. We use a parser to translate the data from the configuration file format to the source format (a Haskell record).

The source format is static, so it must contain everything that is possible to write in a configuration file. The entries that are not in a particular configuration file cannot be ignored, and hence the source contains all possible entries. We use the `Maybe` monad in Haskell to denote configuration items that are not present in the configuration file.

Once adaptation has been made, we obtain a new source that has to be translated back into the configuration file, which is done using a pretty printer and a set of rules.

The parsers and pretty printers are presented in details later in this section.

We present here a simplified example of the sources extracted from the configuration files using parsers. The full sources are available on our repository. Listing 5.1 shows a simplified version of the Apache source, while Listing 5.2 shows a simplified version of the Nginx source.

Listing 5.1: Simplified Apache source

```
1 apacheSource :: ApacheWebserver
2 apacheSource = ApacheWebserver {
3   aDocumentRoot = Nothing,
4   aKeepAlive = Just "On",
5   aKeepAliveTimeout = Just "65",
6   aMaxKeepAliveRequests = Just "100",
7   aListen = Just ["80"],
8   aDirectoryIndex = Nothing,
9   aSSLCertificateFile = Nothing,
10  aSSLCertificateKeyFile = Nothing,
11  aVirtualHosts = Just [
12    VirtualHost {
13      sVirtualHostAddress = Just " *:80",
14      sDocumentRoot = Just "/var/www/html",
15      sKeepAlive = Just "On",
16      sKeepAliveTimeout = Just "65",
17      sMaxKeepAliveRequests = Just "100",
18      sLocation = Nothing,
19      sServerName = Just "example.com",
20      sDirectoryIndex = Nothing,
21      sSSLEngine = Nothing,
22      sSSLCertificateFile = Nothing,
23      sSSLCertificateKeyFile = Nothing
24    }
25  ]
26 }
```

## 5.1.2 View

Both simplified sources in Listings 5.1 and 5.2 give the same resulting view. Listing 5.3 presents a simplified version of the view extracted using the *get* transformation.

Listing 5.2: Simplified Nginx source

```
1  nginxSource :: NginxWebserver
2  nginxSource = NginxWebserver {
3    nWorkerProcesses = Just "4",
4    nHttp = Just Http {
5      hKeepaliveDisable = Nothing,
6      hKeepaliveTimeout = Just "65",
7      hKeepaliveRequests = Just "100",
8      hRoot = Nothing,
9      hServer = Just [
10     Server {
11       sKeepaliveDisable = Nothing,
12       sKeepaliveTimeout = Just "65",
13       sKeepaliveRequests = Just "100",
14       sListen = Just ["80"],
15       sLocation = Nothing,
16       sRoot = Just "/var/www/html",
17       sServerName = Just ["example.com"],
18       sSsl = Nothing,
19       sSslCertificate = Nothing,
20       sSslCertificateKey = Nothing }
21     ],
22     hSsl = Nothing,
23     hSslCertificate = Nothing,
24     hSslCertificateKey = Nothing
25   }
26 }
```

### 5.1.3 Parsers and pretty printers

As previously said, parsers are needed to extract readable sources for the bidirectional transformation from the configuration files since our BXs can't use those configuration files directly. While parsers provide a formatted source from configuration files, the pretty printers do the opposite: they take the updated source as input to write the new configuration file. It's important to note that both cases only modify the formatting of the data.

This subsection explains how the parsers and pretty printers are formed and provides examples for the ease of comprehension. The examples are presented with Nginx only since they are similar for Apache or any other

Listing 5.3: Simplified view

```
1 reducedView :: CommonWebserver
2 reducedView = CommonWebserver {
3   vRoot = "html",
4   vKeepaliveTimeout = "65",
5   vSSL = "off",
6   vSSLCertificate = "",
7   vSSLCertificateKey = "",
8   vServers = [
9     VServer {
10      vListen = ["80"],
11      vServNames = ["example.com"],
12      vServRoot = "/var/www/html",
13      vServKeepaliveTimeout = "65",
14      vServSSL = "off",
15      vServSSLCertificate = "",
16      vServSSLCertificateKey = ""
17    }
18  ]
19 }
```

web server technology. However, since the examples remain minimal, some specificities that the reader can find in the full code won't be discussed here. Those specificities are mostly due to implementation details and are not really important for the reader to understand how the parsing works.

## Parsers

The parsing process can be divided in two steps. The first one consists of extracting the data from the configuration file into a tree which contains all the information. The second step translates this tree into a suitable source file for the BX to use. Those steps greatly facilitate the parsing work since a tree is a simple structure, thus easy to form and manipulate. Moreover, because of the simple structure of the two functions, modifications are also easy to make.

The first step is realized by using a tool called *Peggy*<sup>1</sup>. This tool provides a parser generator which use a Parsing Expression Grammar (PEG). The

---

<sup>1</sup><https://hackage.haskell.org/package/peggy>

parsing of the sample configuration file represented in Listing 5.4 to a tree is made thanks to the Peggy code shown in Listing 5.5.

Listing 5.4: Nginx configuration file example

```
1  pid /run/nginx.pid;
2  user www-data;
3  events {
4      worker_connections 768;
5  }
6  http {
7      gzip on;
8      keepalive_timeout 65;
9      access_log /var/log/nginx/access.log;
10     error_log /var/log/nginx/error.log;
11     server {
12         listen 80;
13         root /var/www/html;
14         server_name example.com;
15     }
16 }
```

Lines 2 and 3 state that the configuration file is going to be translated into a tree (TreeFile type) and that the configuration file is composed of directives. Between curly brackets is the code that specifies that the root element of the tree is called "root" and that the directives are separated in 2 children in the tree depending on their types. Line 4 to 7 describe what a directive is. In our case, 3 different possibilities exist: We can have an instruction, a block (which is the name we used for the contexts in the configuration files) or a comment. Depending on what the directive is, it will be placed in the tree at the right place. The tree structure used is as follows: each node represents a context, the root node being the main context of the configuration file, then, the instructions contained in this context are placed in the left child of the node while its sub-contexts are placed in the right child. The comments are ignored and will not be found in our tree since they are not relevant data for adaptation. The two following lines (8 and 9) specify what an instruction is. It's composed of an undetermined suite of letters, number or specific symbols (0-9 a-z A-Z \_ / ' : = ! { } [ ] \* \$ ") represented by the *symbols* word. As previously seen, in a configuration file,

## Listing 5.5: parsing code using Peggy

```

1  [peggy|
2  configFileNginx :: TreeFile
3      = directive* {let ds = catMaybes $1 in TreeFile "root" (lefts ds)
4      ↪ (rights ds)}
5  directive :: Maybe (Either Instruction TreeFile)
6      = instruction { Just (Left $1) }
7      / block { Just (Right $1) }
8      / comment { Nothing }
9  instruction :: Instruction
10     = [symbols]+ ';' { removeFirstSpace (break (==' ') $1) }
11  block ::: TreeFile
12     = [a-z /]+ "{" directive* "}" { let ds = catMaybes $2 in TreeFile
13     ↪ (fst (break (==' ') $1)) (lefts ds) (rights ds) }
14  comment ::: ()
15     = "#" [^\n]* { () }
16  ]

```

instructions often have a value. This value is included in the definition of an instruction in our code. An annex function called `removeFirstSpace`, which is not showed here, is tasked to remove the first space character from the second argument of a pair of *String*. This function combined with the `break` function allows to separate an instruction from its value. Both of them are then placed in the tree under the branch that gathers entities of the *Instruction* type. The definition of a block can be found at lines 10 to 11. A block is generally a composition of letters followed by curly brackets that contains an undetermined number of directives. As it was said, the block represents a sub-context and therefore has the tree structure. Its root will take the name of the found block (the letters before the curly brackets) and the directives will be separated in the same way than explained before, recursively. There exists blocks with a more complicated syntax than the one presented in the example, which often have a value before the curly brackets, but we consider that as an implementation detail since the principle is still the same. The corresponding code is therefore not presented. The value before the curly bracket of such block is inserted in the set of the block's values with the name of the block as instruction name. The last lines of code explain what a comment is. In an Nginx configuration file like our example uses, a comment

begins with the “#” symbol and can be followed by anything. Even if we have to consider them in the parsing to avoid causing errors, we ignore them in the creation of the tree.

The tree resulting from the parsing of the sample configuration file in Listing 5.4 is given in Figure 5.1. It clearly shows the intended structure and the repartition of the different directives and their values.

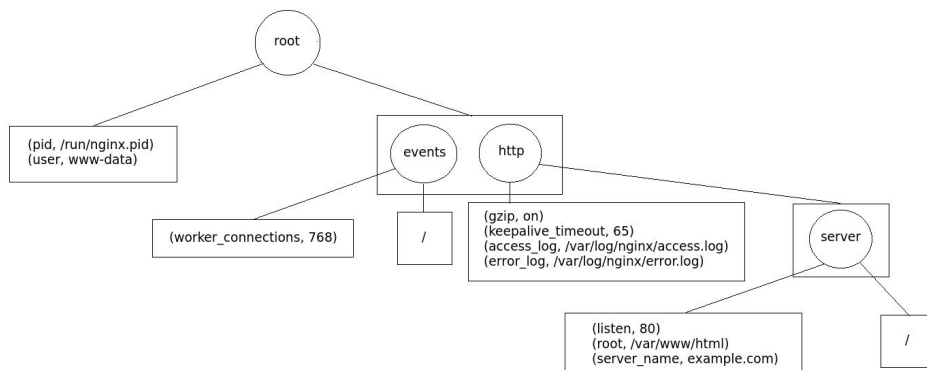


Figure 5.1: Resulting tree after parsing

Once the tree is filled with all the information from the configuration file, we can pass to the second step, which consists of creating the source from the tree. The general approach is simple. We create a source type thanks to a function that takes the tree as input. We then just fill the source type by taking the right values in the tree. In regard to the contexts, we provide another function that creates the context type with the rest of the tree as input. This new function is then called in the main one. The code sample in Listing 5.6 shows what the source creation looks like for the tree given as example.

Lines 1 to 7 is the definition of the main function, which creates the source file from the tree. Its signature is shown at line 2. We can see that values

Listing 5.6: Source generation from the tree generated by parsing

```

1  --Function that will put the Tree into the source type
2  createSourceNginx :: TreeFile -> NginxWebserver
3  createSourceNginx tree = NginxWebserver {
4      nPid = getValue tree "pid",
5      nUser = getValue tree "user"
6      nEvents = getEvents tree,
7      nHttp = getHttp tree
8  }
9
10 --Function that will contract an nEvents type
11 getEvents :: TreeFile -> Maybe Events
12 getEvents tree = if (isInChildren tree "events") then
13     Just Events {
14         eWorkerConnections = getValue (head (giveTree [tree] ["root","events"]))
15         ↪ "worker_connections"
16     } else
17     Nothing
18
19 --Function that will contract an nHttp type
20 getHttp :: TreeFile -> Maybe Http
21 getHttp tree = if (isInChildren tree "http") then
22     Just Http {
23         hGzip = getValue (head (giveTree [tree] ["root","http"])) "gzip",
24         hKeepaliveTimeout = getValue (head (giveTree [tree] ["root","http"]))
25         ↪ "keepalive_timeout",
26         hAccessLog = getValue (head (giveTree [tree] ["root","http"])) "access_log",
27         hErrorLog = getValue (head (giveTree [tree] ["root","http"])) "error_log",
28         hServer = getServers tree
29     } else
30     Nothing
31
32 --Function that will contract a hServer type
33 getServers :: TreeFile -> Maybe [Server]
34 getServers tree = if (isInChildren (head (giveTree [tree] ["root","http"])) "server")
35     ↪ then
36     Just (map getServerValues (giveTree [tree] ["root","http","server"]))
37     else
38     Nothing
39
40 --Function that will construct a Server type
41 getServerValues :: TreeFile -> Server
42 getServerValues tree = Server {
43     sListen = getValueList tree "listen",
44     sRoot = getValue tree "root",
45     sServerName = getValueList tree "server_name"
46 }

```

taken in the tree, by the `GetValue` function that is not shown here since retrieving a value from a tree is quite trivial, are assigned to the different fields of the source on lines 4 and 5. However, while the returned values at lines 4 and 5 must be a *String* type, the ones at lines 6 and 7 must be a specific type representing the contexts in the configuration file. New functions are then created that return this specific type from the tree. Those functions are defined from line 10 to the end of the code. Lines 10 to 16 provide the definition of a function that returns a type that is assimilated to the *events* context in the configuration file. If this context is present, the function looks for the instructions and their values in that context to create the source. If it's not defined in the configuration file, the function returns *Nothing*. The same principle is again applied for lines 18 to 28 which define the *http* context. This context can also contain another which is the *server* one. A new function has to be defined. This function is a bit more special than the others since the *http* context can have more than one server. At first, it gathers all the *server* contexts defined in *http*, then it gathers those server values by calling a function that collects the values in the same way used above (e.g. the *events* context). The function that gathers all the *server* contexts is defined at lines 30 to 35 and the function that looks for the values inside the context is defined from line 37 to line 43.

Once this second step is complete, the source file has been created. A reduced source file example was already presented in this section in Listing 5.1 for the reader to see what it looks like when the parsing is finished.

It's important to note that all the possible configuration files are not handled by our parsers. Since, in most web servers implementations, several hundreds of different directives are available to tune the configuration, often with the addition of extensions too, writing parsers for all of them would have taken far too much time while not having a great impact on the actual case study. Some instructions and extensions were thus ignored.

The parsers and their internal functions were tested thanks to a set of unit tests made with the Haskell package `Test.HUnit`. They were also tested with multiple sample configuration files. Some of those files were created by hand to carefully test extreme cases while the others were gathered from different sources on the net after checking that they did not contain non-handled instructions. They also were tested themselves on web servers to

prove their correctness. An example of unit test which shows one test of the `getValue` function used in the parsers is presented to the reader.

Listing 5.7: Example of unit test

```
1 treeFileFunctions1 = TestCase (assertEqual "getValue in TreeFile -  
  ↪ value found" (Just "18") (getValue (TreeFile "tree1"  
  ↪ [("test", "18"), ("test2", "24")] []) "test"))  
2  
3 treeFileFunctionsTests = TestList [TestLabel "Right getValue test"  
  ↪ treeFileFunctions1]
```

In this test, we create a tree by hand to avoid bias of other functions' errors. This tree contains 2 instructions with their values. The first one is the instruction `"test"` which has a value of `"18"` and the second instruction is `"test2"` with a value of `"24"`. We create a test case that is supposed to find the value `"18"` when applying the function `getValue` to the tree with `"test"` as argument to know which value the function has to take. If the `getValue` function works well, the test will pass. If not, an error will be prompted when running the test set.

## Pretty printers

The tool used to pretty print the source back to a configuration file is a simple package called *Text.PrettyPrint*<sup>2</sup>. The approach looks like the second step of the parsing process in reverse except that we don't obtain a tree as result. Indeed, to pretty print, a function that takes the source as input is created. It gathers the values in the source to format the information like a configuration file. This function then returns a *Doc* type which is printable in a file. The printed *Doc* type gives us the configuration file. A sample of pretty print code is provided in listing 5.8. It recreates the initial configuration file given as example in listing 5.4.

Lines 1 to 6 define the main function that will serve to print the configuration file. As previously said, line 2 shows that it takes the source as input

---

<sup>2</sup><https://hackage.haskell.org/package/pretty-1.1.3.4/docs/Text-PrettyPrint.html>

## Listing 5.8: Example of pretty print code

```

1  --Function that will pretty print the NginxWebserver
2  printNginx :: NginxWebserver -> Doc
3  printNginx source = printMaybe (printSimpleInstruction "pid") (mPid source)
4    $$ printMaybe (printSimpleInstruction "user") (nUser source)
5    $$ printMaybe printEvents (nEvents source)
6    $$ printMaybe printHttp (nHttp source)
7
8  --Function that will print the events context
9  printEvents :: Events -> Doc
10 printEvents source = text "events {"
11   $$ nest 5 (printMaybe (printSimpleInstruction "worker_connections")
12     ↪ (eWorkerConnections source))
13   $$ text "}"
14
15 --Function that will print the http context
16 printHttp :: Http -> Doc
17 printHttp source = text "http {"
18   $$ nest 5 (printMaybe (printSimpleInstruction "gzip") (hGzip source))
19   $$ nest 5 (printMaybe (printSimpleInstruction "keepalive_timeout") (hKeepaliveTimeout
20     ↪ source))
21   $$ nest 5 (printMaybe (printSimpleInstruction "access_log") (hAccessLog source))
22   $$ nest 5 (printMaybe (printSimpleInstruction "error_log") (hErrorLog source))
23   $$ nest 5 (printMaybe printServers (hServer source))
24   $$ text "}"
25
26 --Function that will print a list of servers context
27 printServers :: [Server] -> Doc
28 printServers [] = empty
29 printServers (x:xs) = printServer x $$ printServers xs
30
31 --Function that will print a server context
32 printServer :: Server -> Doc
33 printServer source = text "server {"
34   $$ nest 5 (printMaybe (printListInstruction "listen") (sListen source))
35   $$ nest 5 (printMaybe (printSimpleInstruction "root") (sRoot source))
36   $$ nest 5 (printMaybe (printListInstruction "server_name") (sServerName source))
37   $$ text "}"

```

and provides a *Doc*. From line 3 to line 6, we transform the source into the *Doc* type. The `printMaybe` function is useful to change the *Maybe* type used in the source to the *Doc* type. However, as we consider this as implementation detail, it's not showed in the code sample. As for `printMaybe`, the `printSimpleInstruction` function is not showed. It associates the following *String* to the value found in the source to allow the `printMaybe` to change it to a *Doc* type. Like the second step in the parsing process, the contexts (or the blocks) need special functions to handle their specificities. They are

defined further in the code and are called at lines 5 and 6. The function that handles the *events* context is defined from line 8 to line 12. By the same principle, it changes the `worker_connection` instruction to a *Doc*. However, there is an addition to that code. The `nest 5` instruction is used to indent the code of the block. Indeed, we tried to construct the configuration file as clearly as possible. This includes a clear format and a good indentation. The following part of code defines the transformation of the *http* context using, again, the same approach (lines 15 to 22). In that context, we can have the *server* block. We therefore need to create a new function which, like in the parser, is a bit more specific. Since we can have more than one *server* block in the *http* context, a function that handles this problem is provided at lines 24 to 27. It basically applies a function to change the *Server* type to the *Doc* type to each server found in the list taken from the source. The transformation of the *Server* type is given at lines 29 to 35. It uses the same approach than previously.

As discussed in the parsers section about specific blocks, those were taken into account to pretty print since the specific directives are known but not showed in the example code as it is considered an implementation detail. For example, concerning the specific blocks which often have a value before the curly brackets, the pretty print process looks for that value and puts it at the right place while printing the text.

The fact that the order of the instruction is handmade need to be emphasized. Indeed, the creation of the *Doc* type fixes the position of the instructions when the new configuration file will be printed. In that concern, some knowledge about the format of a configuration file to be able to position the instructions and the blocks correctly is needed. It also explains that the newly created configuration file can present differences in comparison to the old one. However, those differences are not semantic as the intended behavior is preserved.

Once this process is finished, we obtain a new well formatted configuration file like the one given as sample in Listing 5.4.

As for the parsers, the pretty print process and the functions it uses were tested with multiple samples of configuration files either created by hand or found on the net. The tested files for the pretty print are the same than

for the parsers. The principle here was to parse the testing configuration file (the parsers were supposed to work correctly since they were also tested) to create a source and then printing that source into a new configuration file. We then compare both configuration files to see if they have the same behavior. The comparison is either made by hand which consists in looking at them simultaneously to see if there is any differences or either by running them on a web server. The hand made comparison was possible only if the configuration files were small enough to be treated without a high risk of human mistakes. We have chosen to use both comparison to avoid the time consuming process of running the configuration file on a server then test its behavior when it was possible. No example of unit test is provided here since it is very similar to the one presented for the parsers.

## 5.2 Scenario 1: Adaptation

In this scenario, we show that two web servers using a different technology, but with the same behavior for a specific concern, can be adapted using our approach. Both behaviors should be adapted in the same way. The MAPE loop updates the SSL configuration. The initial configuration does not use SSL, and the adaptation will activate and configure it.

### 5.2.1 Experiment

We first ran both servers to confirm that they serve pages over HTTP, but not over HTTPS.

We then extracted the concrete model from the Apache and Nginx configuration files discussed in Section 5.1.1. They are represented by the Haskell records, portions of which are on Listing 5.1 and Listing 5.2, respectively. The whole sources are available on our repository.

The abstract models were then built using the `get` transformations generated by our Nginx and Apache bidirectional programs, also available on our repository and given in the annexes B. Samples of the abstract models are in Listing 5.3, and the entire records are on the repository.

We then simulated the adaptation by changing the following values in the views. The changes were made on the same items and with the same values in both models, as if both had been modified by the same adaptation rules.

```
_____ Values before adaptation _____  
vListen = ["80"],  
vServKeepaliveTimeout = "65",  
vServSSL = "off",  
vServSSLCertificate = "",  
vServSSLCertificateKey = ""  
  
_____ Values after adaptation _____  
vListen = ["443"],  
vServKeepaliveTimeout = "75",  
vServSSL = "on",
```

```
vServSSLCertificate = "/srv/ssl/cert.pem",  
vServSSLCertificateKey = "/srv/ssl/cert.key"
```

Here, the *vListen* item represents the port on which the server listens. Its value is set from 80, the default HTTP port, to 443, the default HTTPS port. Setting *vServSSL* to "on" activates SSL and the next two items provide the location of the SSL certificate and its key. The modification of the *vServKeepaliveTimeout* item is not mandatory for a working SSL configuration, but was added to extend the example.

Those changes were then reflected to the sources using the put transformations generated by our bidirectional programs. The following are samples of the updated sources:

---

Modifications in Nginx

---

```
sKeepaliveTimeout = Just "75",  
sListen = Just ["443"],  
sSsl = Just "on",  
sSslCertificate = Just "/srv/ssl/cert.pem",  
sSslCertificateKey = Just "/srv/ssl/cert.key"
```

---

Modifications in Apache

---

```
aListen = Just ["443"],  
sVirtualHostAddress = Just "*:443",  
sKeepAliveTimeout = Just "75",  
sSSLEngine = Just "On",  
sSSLCertificateFile = Just "/srv/ssl/cert.pem",  
sSSLCertificateKeyFile = Just "/srv/ssl/cert.key"
```

The new sources were then pretty printed in new configuration files. The servers were reloaded to use their new configuration, with the expectation that both would then serve pages over HTTPS, but not HTTP.

## 5.2.2 Results

The two web servers, that did not use SSL initially, ran with SSL activated after the simulated adaptation. The changes in the view were correctly reflected to the source, without manual modification of the configuration files.

## 5.3 Scenario 2: Migration

For this scenario, we show that our approach allows to copy an abstract model of a web server technology, and use this copy to replicate the server's behavior on an other web server, newly deployed or not, using a different technology.

### 5.3.1 Experiment

First, we confirmed that the first web server was running properly, and behaved as expected. It used Nginx.

We then used the `get` transformation for Nginx to generate the abstract model of the server configuration. The `put` transformation for Apache was then used, with an empty original source, simulating the fact that the server was newly deployed and not yet configured, to produce a concrete Apache model, that represent an equivalent configuration to the original Nginx configuration.

We pretty printed the configuration file for Apache and then ran an Apache web server with this configuration file. We verified that the behavior of the Apache server was identical to the behavior of the Nginx server.

### 5.3.2 Results

Both servers ran correctly after the migration. The configuration of the Nginx server was unchanged, and the Apache server exhibited the same behavior as the Nginx server.

## 5.4 Evaluation

We unfortunately could not perform a sound and robust evaluation of our approach in terms of performance. However, we can still compare it to other works aiming to provide reusability for adaptation logic or revolving around the use of models for self-adaptation. We can show that our solution, while not being thoroughly tested yet, follows the ideas of other studies in these fields while bringing its own improvements, mainly the use of a synchronization mechanism correct by construction. We use the work specified in the state of the art (sections 2.1.3 and 2.1.4) to compare our method. The related papers are consequently explained in more details in this section.

### 5.4.1 Reusability

Other works have aimed to provide some form of reusability of the adaptive layer on different target systems.

*Klein et al* presents a method which allows to deactivate optional features of a software by automatically performing an adaptation on the said software [Klein et al., 2014]. This method can only be applied to specific softwares that they call *brownout-compliant*. Those are software in which some features can be stopped if needed without altering the whole functionality. We can take a publicity banner on an auction site as example. The banner is not vital for the site to run but is a great addition to boost the sales. They validate their approach on 2 web application prototypes (RUBIS and RUBBoS) which consists in modifying those web applications in 3 steps to isolate optional software code in order to be able to deactivate them when it's needed. They also introduce a value that is manipulated and checked by the adaptation engine which helps to know when to perform the adaptation. The reusability is found across the part of code which manipulates the value and the code that triggers and perform the adaptation. However, isolating the features has to be done manually and depends on the software.

The first thing that is worth mentioning as comparison between our approach and this method is that the brownout methods can only be applied to a subset of self-adaptive software while our approach is potentially usable on any self-adaptive system. In both method, code has to be modified. Indeed, in their work, manual code has to be written in order to isolate the optional

code and to link that code to the self-adaptivity. We also have to write specific parts of code to reuse the adaptive layer like the monitors and the effectors as well as the bidirectional transformations for new technologies. Our method aims more to reuse already written adaptive layer on specific softwares while their approach allows to reuse parts of code on any new software.

Next, we compare our approach with the *Rainbow* framework. *Garlan et al* provide a framework to perform adaptation on a software that can be reused across a wide range of systems [Garlan et al., 2004]. However, while the framework provides reusable parts to perform adaptation, the user still has to hook personalized codes to it: the actual rules for data analysis and fault state detection along with the adaptation policies and the internal model representation of the target system. Moreover, the monitors and effectors also need to be manually written, except if similar codes have been written already and can be reused. Indeed, the effectors and monitors need to be adapted to each specific software but several systems may require the same data to be monitored or the same parts to be effected, depending on the desired adaptation. The interfaces for those effectors are provided by *Rainbow*. It can also share some (*K*)*nowledge* across similar software. *Rainbow* was later extended by *Swanson et al.* This new framework called *REFRACT* brings failure avoidance components and algorithms [Swanson et al., 2014]. *Angelopoulos et al.* also compared *Rainbow* with their framework, *Zanshin*, which is requirement-based instead of architecture-based [Angelopoulos et al., 2013].

Our approach is somewhat similar to the *Rainbow* framework in the way that our approach also requires specific sensors and effectors to be written for the target system and an internal model representation of it. *Rainbow* is probably more generic in terms of reusability. It can be reused even on systems that are not closely related while our approach aims to reuse parts of the adaptive layer for systems that presents similar adaptation needs. By extension, this comparison is also applicable to *REFRACT*.

*Barna et al* present a platform used to develop self-managing systems on cloud applications [Barna et al., 2015]. The reusable parts of the platform include the monitoring, a base to construct the logic to analyze the metrics and to plan the changes, a performance model, a mechanism to provide custom cloud managers. Each of those points are also customizable to a certain

degree.

While *Hogna* aimed at cloud application, and consequently applicable to that specific type of systems, our approach, which is potentially applicable to every kind of systems, seems less customizable and more rigid. With *Hogna*, the monitors can be reused and adapted if needed. Our method suggests that the monitors need to be written specifically for the system. *Hogna* provides many tools to customize the platform accordingly to the system (cloud managers, the modification of the logic) while our approach is the reutilization of parts that can be reused because of the similarities between the systems. If those parts can't be reused, they need to be entirely written and no tool is provided to help the work.

Although these approaches also allow for the reuse of abstract models, they generally require the careful development of both sensors and effectors, forming a BX that need to be shown to keep the abstract model in sync with the target system's configuration. Our contribution is different in that only one direction of the transformation needs to be written, and the other one is automatically derived, in such a way that guarantees that the BX is well-behaved, and hence the abstract model correctly synchronized with the target system's configuration.

[Ramirez and Cheng, 2010] present some adaptation patterns. They do so by abstracting the adaptation expertise from existing self-adaptive systems. They find similarities between those systems and construct a systematic pattern to use in order to build such system with common features. The features have a wide range going from the monitoring needs to the adaptation politics.

Our work stands in the same spirit about abstracting raw knowledge in the adaptation mechanism in order to systematically reuse parts of code. However, we aim to reuse parts of code, their goal is a bit different by focusing on the reuse of an approach to build a specific type of self-adaptive system. By using the same approach, some parts of code can eventually be reused but it's the consequence of their approach and not the purpose. In addition, our approach could probably fit in one of these patterns or be a pattern on its own since it can be reused for adapting systems that presents the same adaptation rules.

*Abbas et al* present autonomic software product lines [Abbas, 2011, Abbas and Andersson, 2015]. They describe how we can reuse parts of code to create self-adaptive systems as a product line. This includes the notion of variability and how it influences the building process of the system. One of their component uses an abstract model, which is very generic to be able to build a wide range of systems, that can be specified to represent the desired system. Once this model is specified, the self-adaptive system can be created by choosing the right code according to that model.

We can emphasize here a big similarity with our approach : the abstract model. We can see that abstracting a model helps to reuse parts of code and this is exactly the purpose of our approach. In their work, the model is then specified to link the appropriate parts of code. This is also similar to our approach since the source of our BX is tied to the technology the system uses. The difference is that they use this abstraction to reuse parts of code in the creation of self-adaptive system while we use the abstraction to be able to reuse parts of code across systems that present the same adaptation logic.

#### **5.4.2 Models within self-adaptation**

Other works also researched around the use of models in self-adaptation.

*Vogel and Giese*, for example, present a model-driven approach for adaptation that provides multiple architectural runtime models as a basis for adaptation [Vogel and Giese, 2010]. These models abstract from the underlying system and can be at different levels of abstraction to ease the work of the adaptation layer. In addition, each model focuses on a specific concern, instead of a single model covering all the adaptation concerns. This allows each model to revolve around one specific self-management capability, hence simplifying the work of developers writing the adaptation logic. This approach also support reusability of the adaptive layer since those models could be reused across several managed systems. In another paper, the authors explained in more depth their model transformation engine: it is based on incremental model synchronization and allows for monitoring at runtime [Vogel et al., 2010]. It also showed a significantly better ratio between development costs and performance than manually developed solutions.

Their approach is similar to ours in many ways. First, a source model, extracted from the target system using sensors and applying modifications using effectors, is used. Then, the idea of abstracting this source model into one or several abstract target models also is a common point. Indeed, our solution allows for specific concerns to be extracted from the abstract model, using bidirectional transformations composition, hence keeping all the guarantees given by their usage.

In addition to those works, *Vogel and Giese* demonstrate an executable modeling language for ExecUtable Runtime Megamodels (EUREMA), using megamodels. Megamodels are models that represent a system at runtime along with its adaptation activities and support adaptation engines with multiple feedback loops [Vogel and Giese, 2014]. *Georgas et al.* use a model to record the history of a managed system’s states [Georgas et al., 2005]. This model can be used by a developer to reconfigure a system in another state if he thinks that the current state can lead to a dangerous situation. *Bailey et al.* perform adaptation on Role-Based Access Control (RBAC) models at run-time by changing the access control policies, while ensuring that adapted policies satisfy some security constraints [Bailey et al., 2014].

None of these approaches use bidirectional programming. *Anderson et al.* pose computational reflection as a central concept of self-adaptive systems [Andersson et al., 2009]. While computational reflection is often studied in programming languages, they argue that it is also useful to reason about self-adaptation. They identify key reflection properties in self-adaptive systems. One of them is reflective computation, which includes causal relationships between a system and its self-representation. Our synchronization allows for that causal relationship to be maintained.

### 5.4.3 Threats to Validity

#### Internal Validity

In both scenarios, we simulated the outcome of a MAPE-K loop, by manually modifying abstract models, rather than implementing a feedback loop. Since this paper focuses on the synchronization mechanism between concrete and abstract models, and the associated challenges, we argue that a simulated

MAPE-K loop does not negatively impact the validity of the case study. Similarly, we ignored some operations that would need to be performed on a live system such as transferring updated configuration files to the servers, and reloading them.

As stated in section 5.1.3, the parsers and pretty printers only handle a subset of all possible directives and were tested with configuration files created by hand or found on the net. This implies, although we tried to be as exhaustive as possible, that any input file containing an un-handled directive can lead to an error of the parsers or the pretty printers (semantic or syntactic). However, those currently un-handled directives can potentially be implemented in the future. Meanwhile, the incompleteness of the unit tests should not be a problem since the extreme cases were all tested and a very varied set of configuration files were found and created. The possible errors are due to specific combinations of directives that may cause unexpected behavior.

### **External Validity**

We assumed that no manual or automatic modification was done on the configuration file between the moment we parse it and its rewriting. In a production system, a synchronization mechanism able to cope with concurrent modifications of view and source would be required, such as the one described by Xiong et al [Xiong et al., 2009].

# Chapter 6

## Conclusion

In this thesis, we presented an approach based on abstract models and bidirectional programming for self-adaptation. Our approach provides, by construction, a provably correct synchronization between two sets of models: concrete, implementation dependent, models and abstract models. It is opposed to ad-hoc approaches that rely on developers to carefully build pairs of unidirectional transformations and ensuring that the two models won't drift apart due to a synchronization error. Our approach facilitates the reuse of adaptation logic across different implementations or versions of a system. In contrast with a concrete model closely related to the target system, an abstract model aims to capture the similarities shared by several implementations of a target system. This allows any adaptation logic using the abstract model to analyze the current state of the system to be reused for each implementation, and eases the work of developers.

We demonstrated the use of bidirectional programming to solve the synchronization problem between concrete and abstract models, with proven guarantees on the well-behavedness of the generated bidirectional transformations. We then discussed the specific case of configuration files. We described the challenges that arose from typical constructs in this kind of files, and showed that they can be overcome using bidirectional programming. Finally, the approach has been implemented and its application demonstrated in a web server case study. The results showed that adaptation was performed correctly for both implementations using different technologies but a common abstract model, and that the knowledge in the abstract model could easily be copied between different implementations.

# Bibliography

- [Abbas, 2011] Abbas, N. (2011). Towards autonomic software product lines. In *Proceedings of the 15th International Software Product Line Conference, Volume 2, SPLC '11*, pages 44:1–44:8, New York, NY, USA. ACM.
- [Abbas and Andersson, 2015] Abbas, N. and Andersson, J. (2015). Harnessing variability in product-lines of self-adaptive software systems. pages 191–200. ACM Press.
- [Adamis and Tarnay, 2001] Adamis, G. and Tarnay, K. (2001). Frame-based self-adaptive test case selection. In *International Workshop on Self-Adaptive Software*, pages 129–141. Springer.
- [Alur et al., 2001] Alur, D., Crupi, J., and Malks, D. (2001). Core j2ee patterns: Best practices and design strategies.
- [Andersson et al., 2009] Andersson, J., De Lemos, R., Malek, S., and Weyns, D. (2009). Reflecting on self-adaptive software systems.
- [Angelopoulos et al., 2013] Angelopoulos, K., Souza, V. E. S., and Pimentel, J. (2013). Requirements and architectural approaches to adaptive software systems: A comparative study. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 23–32. IEEE Press.
- [Bailey et al., 2014] Bailey, C., Montrieux, L., de Lemos, R., Yu, Y., and Wermelinger, M. (2014). Run-time generation, transformation, and verification of access control models for self-protection. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 135–144. ACM.

- [Bakay, 2001] Bakay, Á. (2001). Model-based adaptivity in real-time scheduling. In *International Workshop on Self-Adaptive Software*, pages 52–65. Springer.
- [Barna et al., 2015] Barna, C., Ghanbari, H., Litoiu, M., and Shtern, M. (2015). Hogna: A Platform for Self-Adaptive Applications in Cloud Environments. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 83–87. IEEE.
- [Bohannon et al., 2008] Bohannon, A., Foster, J. N., Pierce, B. C., Pilkiewicz, A., and Schmitt, A. (2008). Boomerang: Resourceful Lenses for String Data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '08*, pages 407–419, New York, NY, USA. ACM.
- [Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). A system of patterns: Pattern-oriented software architecture.
- [Carzaniga et al., 2001] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. (2001). Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383.
- [Cheng et al., 2009] Cheng, B. H. C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Serugendo, G. D. M., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H. M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H. A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., and Whittle, J. (2009). Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems*, number 5525 in Lecture Notes in Computer Science, pages 1–26. Springer Berlin Heidelberg.
- [Colson et al., 2016] Colson, K., Dupuis, R., Montrieux, L., Hu, Z., Uchitel, S., and Schobbens, P.-Y. (2016). Reusable self-adaptation through bidirectional programming. In *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '16*, pages 4–15, New York, NY, USA. ACM.

- [Czarnecki et al., 2009] Czarnecki, K., Foster, N., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. (2009). Bidirectional Transformations: A Cross-Discipline Perspective. In Paige, R., editor, *Theory and Practice of Model Transformations*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer Berlin Heidelberg.
- [Doyle and McGeachie, 2001] Doyle, J. and McGeachie, M. (2001). Exercising qualitative control in autonomous adaptive survivable systems. In *International Workshop on Self-Adaptive Software*, pages 158–170. Springer.
- [Fischer et al., 2015] Fischer, S., Hu, Z., and Pacheco, H. (2015). The essence of bidirectional programming. *Science China Information Sciences*, 58(5):1–21.
- [Foster, 2010] Foster, N. (2010). *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, Department of Computer and Information Science.
- [Foster et al., 2009] Foster, N., Pierce, B., and Zdancewic, S. (2009). Updatable Security Views. In *Proc. Computer Security Foundations Symposium (CSF’09)*, pages 60–74. IEEE.
- [Foster et al., 2008] Foster, N., Pilkiewicz, A., and Pierce, B. (2008). Quotient lenses. In *ACM Sigplan Notices*, volume 43, pages 383–396. ACM.
- [Gamma et al., 1995] Gamma, E., Johnson, R., Helm, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.
- [Garlan et al., 2004] Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., and Steenkiste, P. (2004). Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54.
- [Georgas et al., 2005] Georgas, J., van der Hoek, A., and Taylor, R. (2005). Architectural runtime configuration management in support of dependable self-adaptive software. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–6. ACM.
- [Goldman et al., 2001] Goldman, R. P., Musliner, D. J., and Krebsbach, K. D. (2001). Managing online self-adaptation in real-time environments. In *International Workshop on Self-Adaptive Software*, pages 6–23. Springer.

- [Harangozó and Tarnay, 2001] Harangozó, Z. and Tarnay, K. (2001). Fdts in self-adaptive protocol specification. In *International Workshop on Self-Adaptive Software*, pages 113–128. Springer.
- [Hidaka et al., 2011] Hidaka, S., Hu, Z., Inaba, K., Kato, H., and Nakano, K. (2011). GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 480–483.
- [Hu et al., 2004] Hu, Z., Mu, S.-C., and Takeichi, M. (2004). A programmable editor for developing structured documents based on bidirectional transformations. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 178–189. ACM.
- [IBM, 2004] IBM (2004). Autonomic computing toolkit: Developers guide. tech. rep. sc30-4083-03.
- [IBM, 2005] IBM (2005). An architectural blueprint for autonomic computing. Technical report, IBM White paper.
- [Kephart, 2005] Kephart, J. O. (2005). Research challenges of autonomic computing. In *Proceedings of the 27th international conference on Software engineering*, pages 15–22. ACM.
- [Kephart and Chess, 2003] Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.
- [Klein et al., 2014] Klein, C., Maggio, M., Arzén, K.-E., and Hernández-Rodríguez, F. (2014). Brownout: building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 700–711. ACM.
- [Ko et al., 2016] Ko, H.-S., Zan, T., and Hu, Z. (2016). BiGUL: A Formally Verified Core Language for Putback-based Bidirectional Programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016*, pages 61–72, New York, NY, USA. ACM.
- [Laddaga, 1997] Laddaga, R. (1997). Self-adaptive software darpa baa 98-12.

- [Laddaga et al., 2006] Laddaga, R., Robertson, P., and Shrobe, H. (2006). Self-adaptive software: Internalized feedback. *SOFTWARE EVOLUTION AND FEEDBACK*, page 507.
- [Landauer and Bellman, 2001] Landauer, C. and Bellman, K. L. (2001). New architectures for constructed complex systems. *Applied Mathematics and Computation*, 120(1):149–163.
- [Mukhija and Glinz, 2005] Mukhija, A. and Glinz, M. (2005). Runtime adaptation of applications through dynamic recomposition of components. In *International Conference on Architecture of Computing Systems*, pages 124–138. Springer.
- [Parekh et al., 2006] Parekh, J., Kaiser, G., Gross, P., and Valetto, G. (2006). Retrofitting autonomic capabilities onto legacy systems. *Cluster Computing*, 9(2):141–159.
- [Pawlak et al., 2001] Pawlak, R., Duchien, L., Florin, G., and Seinturier, L. (2001). Jac: A flexible solution for aspect-oriented programming in java. In *International Conference on Metalevel Architectures and Reflection*, pages 1–24. Springer.
- [Popovici et al., 2002] Popovici, A., Gross, T., and Alonso, G. (2002). Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147. ACM.
- [Ramirez and Cheng, 2010] Ramirez, A. and Cheng, B. (2010). Design patterns for developing dynamically adaptive systems. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS’10)*, pages 49–58. ACM Press.
- [Sadjadi et al., 2004] Sadjadi, S. M., McKinley, P. K., Cheng, B. H., and Stirewalt, R. K. (2004). Trap/j: Transparent generation of adaptable java programs. In *OTM Confederated International Conferences” On the Move to Meaningful Internet Systems”*, pages 1243–1261. Springer.
- [Salehie and Tahvildari, 2009] Salehie, M. and Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2).

- [Schmidt and Cleeland, 2000] Schmidt, D. C. and Cleeland, C. (2000). Applying a pattern language to develop extensible orb middleware.
- [Shrobe, 2001] Shrobe, H. (2001). Model-based diagnosis for information survivability. In *International Workshop on Self-Adaptive Software*, pages 142–157. Springer.
- [Song et al., 2011] Song, H., Huang, G., Chauvel, F., Xiong, Y., Hu, Z., Sun, Y., and Mei, H. (2011). Supporting runtime software architecture: A bidirectional-transformation-based approach. *Journal of Systems and Software*, 84(5):711–723.
- [Stevens, 2008] Stevens, P. (2008). *A Landscape of Bidirectional Model Transformations*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424. Springer-Verlag GmbH.
- [Swanson et al., 2014] Swanson, J., Cohen, M., Dwyer, M., Garvin, B., and Firestone, J. (2014). Beyond the rainbow: self-adaptive failure avoidance in configurable systems. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China*, pages 377–388.
- [Tarnay, 2001] Tarnay, K. (2001). Self-adaptive protocols. In *International Workshop on Self-Adaptive Software*, pages 106–112. Springer.
- [Vogel and Giese, 2010] Vogel, T. and Giese, H. (2010). Adaptation and abstract runtime models. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 39–48. ACM.
- [Vogel and Giese, 2014] Vogel, T. and Giese, H. (2014). Model-Driven Engineering of Self-Adaptive Software with EUREMA. *ACM Transactions on Autonomous and Adaptive Systems*, 8(4):1–33.
- [Vogel et al., 2010] Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., and Becker, B. (2010). Incremental model synchronization for efficient run-time monitoring. In *Models in Software Engineering*, pages 124–139. Springer.
- [Xiong et al., 2007] Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., and Mei, H. (2007). Towards automatic model synchronization from model

transformations. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 164–173. ACM.

[Xiong et al., 2009] Xiong, Y., Song, H., Hu, Z., and Takeichi, M. (2009). Supporting parallel updates with bidirectional model transformations. In *Theory and Practice of Model Transformations*, pages 213–228. Springer.

[Yu et al., 2012] Yu, Y., Lin, Y., Hu, Z., Hidaka, S., Kato, H., and Montrieux, L. (2012). Maintaining invariant traceability through bidirectional transformations. In *Proceedings of the 34th International Conference on Software Engineering*, pages 540–550. IEEE Press.

# Appendices

# Appendix A

## Configuration files

Listing A.1: Nginx configuration file

```
1 pid /run/nginx.pid;
2 user www-data;
3 worker_processes 4;
4 events {
5     worker_connections 768;
6 }
7 http {
8     keepalive_timeout 65;
9     keepalive_requests 100;
10    access_log /var/log/nginx/access.log;
11    error_log /var/log/nginx/error.log;
12    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
13    tcp_nodelay on;
14    tcp_nopush on;
15    server_tokens on;
16    gzip on;
17    gzip_comp_level 2;
18    server {
19        listen 80;
20        root /var/www/html;
21        server_name example.com;
22        error_log /var/log/nginx/error.log;
23        keepalive_timeout 65;
24        keepalive_requests 100;
25    }
26 }
```

## Listing A.2: Apache configuration file

```
1 User www-data
2 ServerRoot "/etc/apache2"
3 PidFile /var/run/apache2/apache2.pid
4 KeepAlive On
5 MaxKeepAliveRequests 100
6 KeepAliveTimeout 65
7 ErrorLog /var/log/apache2/error.log
8 LogLevel warn
9 DocumentRoot html
10 Listen 80
11 Listen 443
12 ServerTokens OS
13 ServerSignature On
14 IncludeOptional mods-enabled/*.load
15 IncludeOptional mods-enabled/*.conf
16 IncludeOptional conf-enabled/*.conf
17 <VirtualHost *:80>
18     ServerName www.example.com
19     ServerAdmin webmaster@localhost
20     DocumentRoot /var/www/html
21     ErrorLog /var/log/apache2/error.log
22     KeepAlive On
23     MaxKeepAliveRequests 100
24     KeepAliveTimeout 65
25     <Directory />
26         Options FollowSymLinks
27         AllowOverride None
28         Require all denied
29     </Directory>
30     <Directory /var/www/>
31         Options Indexes FollowSymLinks
32         AllowOverride None
33         Require all granted
34     </Directory>
35 </VirtualHost>
```

# Appendix B

## BiGUL codes

Listing B.1: Apache bidirectional transformation

```

1  {-# LANGUAGE TypeOperators, TypeFamilies, FlexibleContexts,
   ↪   DeriveGeneric #-}
2
3  -----
4  --IMPORTS
5  -----
6  ----BiGUL imports
7  import Generics.BiGUL
8  import Generics.BiGUL.AST
9  import Generics.BiGUL.TH
10 import Control.Monad
11 import GHC.Generics
12 --parser-printer imports
13 import ApachePrettyPrinter
14 import ApacheSourceCreator
15 import TreeConfigApacheFiller
16 ----apache imports
17 import Apache_output
18 import TypeFiles.ApacheTypes
19 import ApacheDefaultValues
20 import TypeFiles.Common
21
22 --source and view records defined as BiGUL types
23 deriveBiGULGeneric 'ApacheWebserver
24 deriveBiGULGeneric 'VirtualHost
25 deriveBiGULGeneric 'Directory
26 deriveBiGULGeneric 'Location
27 deriveBiGULGeneric 'Files
28 deriveBiGULGeneric 'DirDirectives
29
30 deriveBiGULGeneric 'CommonWebserver
31 deriveBiGULGeneric 'VServer
32 deriveBiGULGeneric 'VLocation
33
34 --importing view
35 apacheView' :: CommonWebserver
36 apacheView' = apacheOutput
37
38 -----
39 --TRANSFORMATIONS
40 -----
41 --global transformation

```

```

42 transApache :: MonadError' e m => DefaultValues -> BiGUL m
   ↳ ApacheWebserver CommonWebserver
43 transApache defaults = $(rearrAndUpdate [p| CommonWebserver {
44     vRoot = root,
45     vIndex = index,
46     vKeepaliveTimeout = kaTimeout,
47     vKeepaliveMaxRequests = kaMaxRequests,
48     vSendfile = sendfile,
49     vSSL = ssl,
50     vSSLCACertificate = caCertif,
51     vSSLCARevocationFile = caRevocFile,
52     vSSLCertificate = certif,
53     vSSLCertificateKey = certifKey,
54     vSSLCiphers = ciphers,
55     vSSLPreferServerCiphers = preferServCiphers,
56     vSSLProtocols = protocols,
57     vSSLVerifyClient = verifyClient,
58     vSSLVerifyDepth = verifyDepth,
59     vServers = servers
60 }
61 [| [p| ApacheWebserver {
62     aDocumentRoot = root,
63     aDirectoryIndex = index,
64     aEnableSendfile = sendfile,
65     aKeepAliveTimeout = kaTimeout,
66     aMaxKeepAliveRequests = kaMaxRequests,
67     aSSLEngine = ssl,
68     aSSLCACertificateFile = caCertif,
69     aSSLCARevocationFile = caRevocFile,
70     aSSLCertificateFile = certif,
71     aSSLCertificateKeyFile = certifKey,
72     aSSLCipherSuite = ciphers,
73     aSSLHonorCipherOrder = preferServCiphers,
74     aSSLProtocol = protocols,
75     aSSLVerifyClient = verifyClient,
76     aSSLVerifyDepth = verifyDepth,
77     aVirtualHosts = servers
78 }
79 [| [d| root = addDefault (d_DocumentRoot defaults);
80     index = addDefault (d_DirectoryIndex defaults);
81     kaTimeout = addDefault (d_KeepAliveTimeout defaults);
82     kaMaxRequests = addDefault (d_MaxKeepAliveRequests defaults);
83     sendfile = addDefault (d_EnableSendfile defaults);
84     ssl = addDefault (d_SSLEngine defaults);
85     caCertif = addDefault (d_SSLCACertificateFile defaults);

```

```

86     caRevocFile = addDefault (d_SSLLCARevocationFile defaults);
87     certif = addDefault (d_SSLLCertificateFile defaults);
88     certifKey = addDefault (d_SSLLCertificateKeyFile defaults);
89     ciphers = addDefault (d_SSLLCipherSuite defaults);
90     preferServCiphers = addDefault (d_SSLLHonorCipherOrder
91     ↪ defaults);
92     protocols = addDefault (d_SSLLProtocol defaults);
93     verifyClient = verifyClientDefault (d_SSLLVerifyClient
94     ↪ defaults);
95     verifyDepth = addDefault (d_SSLLVerifyDepth defaults)
96     servers = CaseV [
97         $(branch [p| [] |]) $ $(rearr [| \ [] -> Nothing |])
98         ↪ Replace,
99         $(branch [p| (_:_)|]) $ (Compose ($(rearr [| \ x -> (Just
100         ↪ x) |]) Replace) (transServer defaults))
101     ]
102     [])
103
104 --servers transformation
105 transServer :: MonadError' e m => DefaultValues -> BiGUL m
106 ↪ [VirtualHost] [VServer]
107 transServer defaults = Align
108     --source condition
109     (\ _ -> return True)
110     --match
111     --defines on which field the records will match between source and
112     ↪ view
113     (\ (VirtualHost {sServerName = (Just sName)}) (VServer {vServNames
114     ↪ = vNames}) -> return ((head sName) == (head vNames)))
115     --trans
116     ($ (rearrAndUpdate [p| VServer {
117         vListen = listen,
118         vServNames = servNames,
119         vServRoot = servRoot,
120         vServIndex = servIndex,
121         vServKeepaliveTimeout = servKaTimeout,
122         vServKeepaliveMaxRequests = servKaMaxRequests,
123         vServSendfile = servSendfile,
124         vServSSL = servSSL,
125         vServSSLLCACertificate = servCaCertif,
126         vServSSLLCARevocationFile = servCaRevocFile,
127         vServSSLCertificate = servCertif,
128         vServSSLCertificateKey = servCertifKey,
129         vServSSLCiphers = servCiphers,
130         vServSSLPreferServerCiphers = servPreferServCiphers,

```

```

124     vServSSLProtocols = servProtocols,
125     vServSSLVerifyClient = servVerifyClient,
126     vServSSLVerifyDepth = servVerifyDepth,
127     vLocations = servLocations
128 } |] [p| VirtualHost {
129     sVirtualHostAddress = listen,
130     sDocumentRoot = servRoot,
131     sEnableSendfile = servSendfile,
132     sKeepAliveTimeout = servKaTimeout,
133     sLocation = servLocations,
134     sMaxKeepAliveRequests = servKaMaxRequests,
135     sServerName = servNames,
136     sDirectoryIndex = servIndex,
137     sSSLEngine = servSSL,
138     sSSLCACertificateFile = servCaCertif,
139     sSSLCARevocationFile = servCaRevocFile,
140     sSSLCertificateFile = servCertif,
141     sSSLCertificateKeyFile = servCertifKey,
142     sSSLCipherSuite = servCiphers,
143     sSSLHonorCipherOrder = servPreferServCiphers,
144     sSSLProtocol = servProtocols,
145     sSSLVerifyClient = servVerifyClient,
146     sSSLVerifyDepth = servVerifyDepth
147 } |] [d| servRoot = addDefault (d_DocumentRoot defaults);
148     servIndex = addDefault (d_DirectoryIndex defaults);
149     servKaTimeout = addDefault (d_KeepAliveTimeout defaults);
150     servKaMaxRequests = addDefault (d_MaxKeepAliveRequests
151     ↪ defaults);
152     servSendfile = addDefault (d_EnableSendfile defaults);
153     servNames = addDefaultList (d_ServerName defaults);
154     listen = addDefaultList (d_Listen defaults);
155     servSSL = addDefault (d_SSLEngine defaults);
156     servCaCertif = addDefault (d_SSLCACertificateFile
157     ↪ defaults);
158     servCaRevocFile = addDefault (d_SSLCARevocationFile
159     ↪ defaults);
160     servCertif = addDefault (d_SSLCertificateFile defaults);
161     servCertifKey = addDefault (d_SSLCertificateKeyFile
162     ↪ defaults);
163     servCiphers = addDefault (d_SSLCipherSuite defaults);
164     servPreferServCiphers = addDefault (d_SSLHonorCipherOrder
165     ↪ defaults);
166     servProtocols = addDefault (d_SSLProtocol defaults);
167     servVerifyClient = verifyClientDefault (d_SSLVerifyClient
168     ↪ defaults);

```

```

163         servVerifyDepth = addDefault (d_SSLVerifyDepth defaults);
164         servLocations = CaseV [
165             $(branch [p| [] |]) $ $(rearr [| \ [] -> Nothing |])
                ↳ Replace,
166             $(branch [p| (:_:_) |]) $ (Compose ($(rearr [| \ x ->
                ↳ (Just x) |]) Replace) (transLocation defaults))
                ]
167     ] )
168 ] )
169 --create
170 --adds a new server to the source if a new one was added to the
    ↳ view
171 (\ VServer {
172     vListen = listen,
173     vServNames = servNames,
174     vServRoot = servRoot,
175     vServIndex = servIndex,
176     vServKeepaliveTimeout = servKaTimeout,
177     vServKeepaliveMaxRequests = servKaMaxRequests,
178     vServSendfile = servSendfile,
179     vServSSL = servSSL,
180     vServSSLCACertificate = servCaCertif,
181     vServSSLCARevocationFile = servCaRevocFile,
182     vServSSLCertificate = servCertif,
183     vServSSLCertificateKey = servCertifKey,
184     vServSSLCiphers = servCiphers,
185     vServSSLPreferServerCiphers = servPreferServCiphers,
186     vServSSLProtocols = servProtocols,
187     vServSSLVerifyClient = servVerifyClient,
188     vServSSLVerifyDepth = servVerifyDepth,
189     vLocations = locations
190 } -> return VirtualHost {
191     sVirtualHostAddress = emptyListCheck listen,
192     sDirectory = Nothing,
193     sDocumentRoot = emptyCheck servRoot,
194     sEnableSendfile = emptyCheck servSendfile,
195     sErrorDocument = Nothing,
196     sErrorLog = Nothing,
197     sFiles = Nothing,
198     sKeepAlive = Nothing,
199     sKeepAliveTimeout = emptyCheck servKaTimeout,
200     sLocation = Nothing,
201     sMaxKeepAliveRequests = emptyCheck servKaMaxRequests,
202     sServerName = emptyListCheck servNames,
203     sDirectoryIndex = emptyCheck servIndex,
204     sCustomLog = Nothing,

```

```
205     sSSLEngine = emptyCheck servSSL,
206     sSSLCACertificateFile = emptyCheck servCaCertif,
207     sSSLCARevocationFile = emptyCheck servCaRevocFile,
208     sSSLCertificateFile = emptyCheck servCertif,
209     sSSLCertificateKeyFile = emptyCheck servCertifKey,
210     sSSLCipherSuite = emptyCheck servCiphers,
211     sSSLHonorCipherOrder = emptyCheck servPreferServCiphers,
212     sSSLProtocol = emptyCheck servProtocols,
213     sSSLSessionCacheTimeout = Nothing,
214     sSSLVerifyClient = emptyCheck servVerifyClient,
215     sSSLVerifyDepth = emptyCheck servVerifyDepth,
216     sAcceptPathInfo = Nothing,
217     sAccessFileName = Nothing,
218     sAddDefaultCharset = Nothing,
219     sAllowEncodedSlashes = Nothing,
220     sContentDigest = Nothing,
221     sDefine = Nothing,
222     sEnableMMAP = Nothing,
223     sError = Nothing,
224     sErrorLogFormat = Nothing,
225     sFileETag = Nothing,
226     sHostnameLookups = Nothing,
227     sInclude = Nothing,
228     sIncludeOptional = Nothing,
229     sLimitInternalRecursion = Nothing,
230     sLimitRequestBody = Nothing,
231     sLimitRequestFields = Nothing,
232     sLimitRequestFieldSize = Nothing,
233     sLimitRequestLine = Nothing,
234     sLimitXMLRequestBody = Nothing,
235     sLogLevel = Nothing,
236     sMaxRangesOverlaps = Nothing,
237     sMaxRangesReversals = Nothing,
238     sMaxRanges = Nothing,
239     sOptions = Nothing,
240     sProtocol = Nothing,
241     sRLimitCPU = Nothing,
242     sRLimitMEM = Nothing,
243     sRLimitNPROC = Nothing,
244     sServerAdmin = Nothing,
245     sServerPath = Nothing,
246     sServerSignature = Nothing,
247     sSetHandler = Nothing,
248     sSetInputFilter = Nothing,
249     sSetOutputFilter = Nothing,
```

```
250     sTimeOut = Nothing,
251     sTraceEnable = Nothing,
252     sUseCanonicalName = Nothing,
253     sUseCanonicalPhysicalPort = Nothing,
254     sAlias = Nothing,
255     sAliasMatch = Nothing,
256     sRedirect = Nothing,
257     sRedirectMatch = Nothing,
258     sRedirectPermanent = Nothing,
259     sRedirectTemp = Nothing,
260     sScriptAlias = Nothing,
261     sScriptAliasMatch = Nothing,
262     sAddAlt = Nothing,
263     sAddAltByEncoding = Nothing,
264     sAddAltByType = Nothing,
265     sAddDescription = Nothing,
266     sAddIcon = Nothing,
267     sAddIconByEncoding = Nothing,
268     sAddIconByType = Nothing,
269     sDefaultIcon = Nothing,
270     sHeaderName = Nothing,
271     sIndexHeadInsert = Nothing,
272     sIndexIgnore = Nothing,
273     sIndexIgnoreReset = Nothing,
274     sIndexOptions = Nothing,
275     sIndexOrderDefault = Nothing,
276     sIndexStyleSheet = Nothing,
277     sReadmeName = Nothing,
278     sScriptLog = Nothing,
279     sScriptLogBuffer = Nothing,
280     sScriptLogLength = Nothing,
281     sCGIDScriptTimeout = Nothing,
282     sDirectoryCheckHandler = Nothing,
283     sIndexRedirect = Nothing,
284     sDirectorySlash = Nothing,
285     sFallbackResource = Nothing,
286     sPassEnv = Nothing,
287     sSetEnv = Nothing,
288     sUnsetEnv = Nothing,
289     sAddOutputFilterByType = Nothing,
290     sFilterChain = Nothing,
291     sFilterDeclare = Nothing,
292     sFilterProtocol = Nothing,
293     sFilterProvider = Nothing,
294     sFilterTrace = Nothing,
```

```
295     sImapBase = Nothing,
296     sImapDefault = Nothing,
297     sImapMenu = Nothing,
298     sLogFormat = Nothing,
299     sTransferLog = Nothing,
300     sAddCharset = Nothing,
301     sAddEncoding = Nothing,
302     sAddHandler = Nothing,
303     sAddInputFilter = Nothing,
304     sAddLanguage = Nothing,
305     sAddOutputFilter = Nothing,
306     sAddType = Nothing,
307     sDefaultLanguage = Nothing,
308     sMultiviewsMatch = Nothing,
309     sRemoveCharset = Nothing,
310     sRemoveEncoding = Nothing,
311     sRemoveHandler = Nothing,
312     sRemoveInputFilter = Nothing,
313     sRemoveLanguage = Nothing,
314     sRemoveOutputFilter = Nothing,
315     sRemoveType = Nothing,
316     sCacheNegotiatedDocs = Nothing,
317     sForceLanguagePriority = Nothing,
318     sLanguagePriority = Nothing,
319     sReflectorHeader = Nothing,
320     sBrowserMatch = Nothing,
321     sBrowserMatchNoCase = Nothing,
322     sSetEnvIf = Nothing,
323     sSetEnvIfExpr = Nothing,
324     sSetEnvIfNoCase = Nothing,
325     sUserDir = Nothing,
326     sSSLCACertificatePath = Nothing,
327     sSSLCADNRequestFile = Nothing,
328     sSSLCADNRequestPath = Nothing,
329     sSSLCARevocationCheck = Nothing,
330     sSSLCARevocationPath = Nothing,
331     sSSLCertificateChainFile = Nothing,
332     sSSLCompression = Nothing,
333     sSSLInsecureRenegotiation = Nothing,
334     sSSLOCSPPDefaultResponder = Nothing,
335     sSSLOCSPEnable = Nothing,
336     sSSLOCSPOverrideResponder = Nothing,
337     sSSLOCSPResponderTimeout = Nothing,
338     sSSLOCSPResponseMaxAge = Nothing,
339     sSSLOCSPResponseTimeSkew = Nothing,
```

```

340     sSSLOCSPUseRequestNonce = Nothing,
341     sSSLOpenSSLConfCmd = Nothing,
342     sSSLOptions = Nothing,
343     sSSLSessionTicketKeyFile = Nothing,
344     sSSLSessionTickets = Nothing,
345     sSSLSRPUnknownUserSeed = Nothing,
346     sSSLSRPVerifierFile = Nothing,
347     sSSLStaplingErrorCacheTimeout = Nothing,
348     sSSLStaplingFakeTryLater = Nothing,
349     sSSLStaplingForceURL = Nothing,
350     sSSLStaplingResponderTimeout = Nothing,
351     sSSLStaplingResponseMaxAge = Nothing,
352     sSSLStaplingResponseTimeSkew = Nothing,
353     sSSLStaplingReturnResponderErrors = Nothing,
354     sSSLStaplingStandardCacheTimeout = Nothing,
355     sSSLStrictSNIVHostCheck = Nothing,
356     sSSLUseStapling = Nothing
357   })
358   --conceal
359   (\ _ -> return Nothing)
360
361
362   --locations transformation
363   transLocation :: MonadError' e m => DefaultValues -> BiGUL m
364   ↪ [Location] [VLocation]
365   transLocation defaults = Align
366     --source condition
367     (\ _ -> return True)
368     --match
369     --defines on which field the records will match between source and
370     ↪ view
371     (\ (Location { lPath = (Just sPath) } ) (VLocation { vLocationPath
372     ↪ = vPath } ) -> return (sPath == vPath))
373     --trans
374     $($rearrAndUpdate [p| VLocation {
375     vLocationPath = locPath,
376     vLocIndex = locIndex,
377     vLocSendfile = locSendfile
378   } |] [p| Location {
379     lPath = locPath,
380     lDirDirectives = Just DirDirectives {
381     dDirectoryIndex = locIndex,
382     dEnableSendFile = locSendfile
383   }
384   } |] [d| locPath = $($rearr [| \ x -> (Just x) |]) Replace;

```

```

382         locIndex = addDefault (d_DirectoryIndex defaults);
383         locSendfile = addDefault (d_EnableSendfile defaults)
384     |] ))
385     --create
386     --adds a new location to the source if a new one was added to the
387     → view
388     (\ VLocation {
389         vLocationPath = locPath,
390         vLocIndex = locIndex,
391         vLocSendfile = locSendfile
392     } -> return Location {
393         lMatch = (Just False),
394         lPath = (Just locPath),
395         lDirDirectives = Just DirDirectives {
396             dAcceptPathInfo = Nothing,
397             dAddDefaultCharset = Nothing,
398             dContentDigest = Nothing,
399             dDefine = Nothing,
400             dEnableMMAP = Nothing,
401             dEnableSendFile = emptyCheck locSendfile,
402             dError = Nothing,
403             dErrorDocument = Nothing,
404             dFileETag = Nothing,
405             dForceType = Nothing,
406             dHostnameLookups = Nothing,
407             dInclude = Nothing,
408             dIncludeOptional = Nothing,
409             dLimitRequestBody = Nothing,
410             dLimitXMLRequestBody = Nothing,
411             dLogLevel = Nothing,
412             dMaxRangeOverlaps = Nothing,
413             dMaxRangeReversals = Nothing,
414             dMaxRanges = Nothing,
415             dRLimitCPU = Nothing,
416             dRLimitMEM = Nothing,
417             dRLimitNPROC = Nothing,
418             dServerSignature = Nothing,
419             dSetHandler = Nothing,
420             dSetInputFilter = Nothing,
421             dSetOutputFilter = Nothing,
422             dUseCanonicalName = Nothing,
423             dUseCanonicalPhysicalPort = Nothing,
424             dRedirect = Nothing,
425             dRedirectMatch = Nothing,
426             dRedirectPermanent = Nothing,

```

```
426         dRedirectTemp = Nothing,
427         dAuthBasicAuthoritative = Nothing,
428         dAuthBasicFake = Nothing,
429         dAuthBasicProvider = Nothing,
430         dAuthBasicUseDigestAlgorithm = Nothing,
431         dAuthName = Nothing,
432         dAuthType = Nothing,
433         dAuthUserFile = Nothing,
434         dAuthMerging = Nothing,
435         dAuthzSendForbiddenOnFailure = Nothing,
436         dRequire = Nothing,
437         dAuthGroupFile = Nothing,
438         dAddAlt = Nothing,
439         dAddAltByEncoding = Nothing,
440         dAddAltByType = Nothing,
441         dAddDescription = Nothing,
442         dAddIcon = Nothing,
443         dAddIconByEncoding = Nothing,
444         dAddIconByType = Nothing,
445         dDefaultIcon = Nothing,
446         dHeaderName = Nothing,
447         dIndexHeadInsert = Nothing,
448         dIndexIgnore = Nothing,
449         dIndexIgnoreReset = Nothing,
450         dIndexOptions = Nothing,
451         dIndexOrderDefault = Nothing,
452         dIndexStyleSheet = Nothing,
453         dReadmeName = Nothing,
454         dCGIDScriptTimeout = Nothing,
455         dDirectoryCheckHandler = Nothing,
456         dDirectoryIndex = emptyCheck locIndex,
457         dIndexRedirect = Nothing,
458         dDirectorySlash = Nothing,
459         dFallbackResource = Nothing,
460         dPassEnv = Nothing,
461         dSetEnv = Nothing,
462         dUnsetEnv = Nothing,
463         dAddOutputFilterByType = Nothing,
464         dFilterChain = Nothing,
465         dFilterDeclare = Nothing,
466         dFilterProtocol = Nothing,
467         dFilterProvider = Nothing,
468         dFilterTrace = Nothing,
469         dImapBase = Nothing,
470         dImapDefault = Nothing,
```

```

471         dImapMenu = Nothing,
472         dCustomLog = Nothing,
473         dLogFormat = Nothing,
474         dTransferLog = Nothing,
475         dAddCharset = Nothing,
476         dAddEncoding = Nothing,
477         dAddHandler = Nothing,
478         dAddInputFilter = Nothing,
479         dAddLanguage = Nothing,
480         dAddOutputFilter = Nothing,
481         dAddType = Nothing,
482         dDefaultLanguage = Nothing,
483         dModMimeUsePathInfo = Nothing,
484         dMultiviewsMatch = Nothing,
485         dRemoveCharset = Nothing,
486         dRemoveEncoding = Nothing,
487         dRemoveHandler = Nothing,
488         dRemoveInputFilter = Nothing,
489         dRemoveLanguage = Nothing,
490         dRemoveOutputFilter = Nothing,
491         dRemoveType = Nothing,
492         dForceLanguagePriority = Nothing,
493         dLanguagePriority = Nothing,
494         dReflectorHeader = Nothing,
495         dKeptBodySize = Nothing,
496         dBrowserMatch = Nothing,
497         dBrowserMatchNoCase = Nothing,
498         dSetEnvIf = Nothing,
499         dSetEnvIfExpr = Nothing,
500         dSetEnvIfNoCase = Nothing,
501         dSSLCipherSuite = Nothing,
502         dSSLOptions = Nothing,
503         dSSLRenegBufferSize = Nothing,
504         dSSLRequireSSL = Nothing,
505         dSSLUserName = Nothing,
506         dSSLVerifyClient = Nothing,
507         dSSLVerifyDepth = Nothing
508     },
509     lRequireCons = Nothing,
510     lOptions = Nothing
511 })
512 -- conceal
513 (\ _ -> return Nothing)
514
515 -----

```

```

516 --DEFAULT VALUES GESTION
517 -----
518 --defaults gestion for simple fields
519 addDefault :: MonadError' e m => String -> BiGUL m (Maybe String)
520   ↪ String
521 addDefault def = CaseV [ ((return . (== def)),
522   (CaseS [ $(normal [p| Nothing |]) $ Rearr (RConst def) (EIn (ELeft
523   ↪ (EConst ()))) Replace,
524   $(normal [p| (Just _) |]) $ $(rearr [| \ x -> (Just x)
525   ↪ |]) Replace
526   ])) ,
527   ((return . (/= def)),
528   ($ (rearr [| \ x -> (Just x) |]) Replace) )
529   ]
530
531 --defaults gestion for list fields
532 addDefaultList :: MonadError' e m => String -> BiGUL m (Maybe
533   ↪ [String]) [String]
534 addDefaultList def = CaseV [ ((return . (== [def])),
535   (CaseS [ $(normal [p| Nothing |]) $ Rearr (RConst [def]) (EIn
536   ↪ (ELeft (EConst ()))) Replace,
537   $(normal [p| (Just _) |]) $ $(rearr [| \ x -> (Just x)
538   ↪ |]) Replace
539   ])) ,
540   ((return . (/= [def])),
541   ($ (rearr [| \ x -> (Just x) |]) Replace) )
542   ]
543
544 --defaults gestion for the VerifyClient directive
545 verifyClientDefault :: MonadError' e m => String -> BiGUL m (Maybe
546   ↪ String) String
547 verifyClientDefault def = CaseV [ ((return . (liftM2 (&&) (== def) (==
548   ↪ "yes"))),
549   (CaseS [ $(normal [p| Nothing |]) $ Rearr (RConst def) (EIn (ELeft
550   ↪ (EConst ()))) Replace,
551   $(normal [p| (Just _) |]) $ $(rearr [| \ "yes" -> (Just
552   ↪ "require") |]) Replace
553   ])) ,
554   ((return . (liftM2 (&&) (== def) (==
555   ↪ "no"))),
556   (CaseS [ $(normal [p| Nothing |]) $ Rearr (RConst def) (EIn (ELeft
557   ↪ (EConst ()))) Replace,
558   $(normal [p| (Just _) |]) $ $(rearr [| \ "no" -> (Just
559   ↪ "none") |]) Replace
560   ])) ,

```

```

548             ((return . (== def)),
549 (CaseS [ $(normal [p| Nothing |]) $ Rearr (RConst def) (EIn (ELeft
    ↳ (EConst ()))) Replace,
550     $(normal [p| (Just _) |]) $ $(rearr [| \ def -> (Just
    ↳ def) |]) Replace
551     ])) ,
552             ((return . (liftM2 (&&) (/= def) (==
    ↳ "yes"))),
553 ($ (rearr [| \ "yes" -> (Just "require") |]) Replace) ),
554             ((return . (liftM2 (&&) (/= def) (==
    ↳ "no"))),
555 ($ (rearr [| \ "no" -> (Just "none") |]) Replace) ),
556             ((return . (/= def)),
557 ($ (rearr [| \ x -> (Just x) |]) Replace) )
558         ]
559
560 -----
561 --OPERATIONS
562 -----
563 --performs get and show extracted view in console
564 --does not override existing view
565 getApache1 x = (catchBind (get (transApache defaults) x) (\v -> Right
    ↳ (show v)) (\e -> Left e))
566 extractConfig = parseTreeApache "apache.conf" >>= \ (Right tree) ->
    ↳ return (createSourceApache tree) >>= return . getApache1
567
568 --performs get and rewrites view file
569 extractConfigToFile = do
570     content <- extractConfig
571     case content of
572     Left e -> putStrLn ("some error: " ++ show e)
573     Right r -> writeFile "Apache_output.hs" ("module Apache_output
    ↳ where"++"\n"++"import TypeFiles.Common"++"\n"++"apacheOutput
    ↳ :: CommonWebserver"++"\n"++"apacheOutput = "++r)
574
575
576 --performs putback and show new source in console
577 --does not override existing source config file
578 putApache1 x = catchBind (put (transApache defaults) x apacheView')
    ↳ (Right . printApache) Left
579 putbackConfig = parseTreeApache "apache.conf" >>= \ (Right tree) ->
    ↳ return (createSourceApache tree) >>= return . putApache1
580
581 --performs putback and rewrites source config file
582 putbackConfigToFile = do

```

```

583     content <- putbackConfig
584     case content of
585         Left e -> putStrLn ("some error: " ++ show e)
586         Right r -> writeFile "apache.conf" ((show r))
587
588
589 -----
590 --OTHER FUNCTIONS FOR TESTING PURPOSE
591 -----
592 testPut :: BiGUL (Either ErrorInfo) s v -> s -> v -> Either ErrorInfo
593         ↪ s
594 testPut u s v = catchBind (put u s v) (\s' -> Right s') (\e -> Left e)
595
596 testGet :: BiGUL (Either ErrorInfo) s v -> s -> Either ErrorInfo v
597 testGet u s = catchBind (get u s) (\v' -> Right v') (\e -> Left e)
598
599 --demo function for showing source
600 showSource = parseTreeApache "apache.conf" >>= \(Right tree) -> return
601         ↪ (createSourceApache tree) >>= return . show
602
603 -----
604 --OTHER ANNEX FUNCTIONS
605 -----
606 --replaces an empty field by Nothing
607 --used when creating a new element in the source
608 emptyCheck :: String -> Maybe String
609 emptyCheck input = if (input == "") then Nothing else (Just input)
610
611 --replaces an empty list by Nothing
612 --used when creating a new element in the source
613 emptyListCheck :: [String] -> Maybe [String]
614 emptyListCheck input = if (input == []) then Nothing else (Just input)

```

Listing B.2: Nginx bidirectional transformation

```

1  {-# LANGUAGE TypeOperators, TypeFamilies, FlexibleContexts,
   ↪   DeriveGeneric #-}
2
3  -----
4  --IMPORTS
5  -----
6  ----BiGUL imports
7  import Generics.BiGUL
8  import Generics.BiGUL.AST
9  import Generics.BiGUL.TH
10 import Control.Monad
11 import GHC.Generics
12 --parser-printer imports
13 import NginxPrettyPrinter
14 import NginxSourceCreator
15 import TreeConfigNginxFiller
16 ----nginx imports
17 import Nginx_output
18 import TypeFiles.NginxTypes
19 import NginxDefaultValues
20 import TypeFiles.Common
21
22 --source and view records defined as BiGUL types
23 deriveBiGULGeneric 'NginxWebserver
24 deriveBiGULGeneric 'Events
25 deriveBiGULGeneric 'Http
26 deriveBiGULGeneric 'Server
27 deriveBiGULGeneric 'Location
28
29 deriveBiGULGeneric 'CommonWebserver
30 deriveBiGULGeneric 'VServer
31 deriveBiGULGeneric 'VLocation
32
33 --importing view
34 nginxView' :: CommonWebserver
35 nginxView' = nginxOutput
36
37
38 -----
39 --TRANSFORMATIONS
40 -----
41 --global transformation

```

```

42 transNginx :: MonadError' e m => DefaultValues -> BiGUL m
   ↳ NginxWebserver CommonWebserver
43 transNginx defaults = $(rearrAndUpdate [p| CommonWebserver {
44     vRoot = root,
45     vIndex = index,
46     vKeepaliveTimeout = kaTimeout,
47     vKeepaliveMaxRequests = kaMaxRequests,
48     vSendfile = sendfile,
49     vSSL = ssl,
50     vSSLCACertificate = caCertif,
51     vSSLCARevocationFile = caRevocFile,
52     vSSLCertificate = certif,
53     vSSLCertificateKey = certifKey,
54     vSSLCiphers = ciphers,
55     vSSLPreferServerCiphers = preferServCiphers,
56     vSSLProtocols = protocols,
57     vSSLVerifyClient = verifyClient,
58     vSSLVerifyDepth = verifyDepth,
59     vServers = servers
60 }
61 [| [p| NginxWebserver {
62     nHttp = Just Http {
63         hRoot = root,
64         hIndex = index,
65         hKeepaliveTimeout = kaTimeout,
66         hKeepaliveRequests = kaMaxRequests,
67         hSendFile = sendfile,
68         hSsl = ssl,
69         hSslCertificate = certif,
70         hSslCertificateKey = certifKey,
71         hSslCiphers = ciphers,
72         hSslClientCertificate = caCertif,
73         hSslCrl = caRevocFile,
74         hSslPreferServerCiphers = preferServCiphers,
75         hSslProtocols = protocols,
76         hSslVerifyClient = verifyClient,
77         hSslVerifyDepth = verifyDepth,
78         hServer = servers
79     }
80 }
81 [| [d| root = addDefault (d_root defaults);
82     index = addDefault (d_index defaults);
83     kaTimeout = addDefault (d_keepalive_timeout defaults);
84     kaMaxRequests = addDefault (d_keepalive_requests defaults);
85     sendfile = addDefault (d_send_file defaults);

```

```

86     ssl = addDefault (d_ssl defaults);
87     caCertif = addDefault (d_ssl_client_certificate defaults);
88     caRevocFile = addDefault (d_ssl_crl defaults);
89     certif = addDefault (d_ssl_certificate defaults);
90     certifKey = addDefault (d_ssl_certificate_key defaults);
91     ciphers = addDefault (d_ssl_ciphers defaults);
92     preferServCiphers = addDefault (d_ssl_prefered_server_ciphers
93     ↪ defaults);
94     protocols = addDefault (d_ssl_protocols defaults);
95     verifyClient = verifyClientDefault (d_ssl_verify_client
96     ↪ defaults);
97     verifyDepth = addDefault (d_ssl_verify_depth defaults)
98     servers = CaseV [
99     ↪ $(branch [p| [] |]) $ $(rearr [| \ [] -> Nothing |])
100     ↪ Replace,
101     ↪ $(branch [p| (_:_ |)] $ (Compose ($(rearr [| \ x ->
102     ↪ (Just x) |]) Replace) (transServer defaults))
103     ]
104     |])
105
106 --servers transformation
107 transServer :: MonadError' e m => DefaultValues -> BiGUL m [Server]
108 ↪ [VServer]
109 transServer defaults = Align
110 ↪ --source condition
111 ↪ (\ _ -> return True)
112 ↪ --match
113 ↪ --defines on which field the records will match between source and
114 ↪ view
115 ↪ (\ (Server {sServerName = (Just sName)}) (VServer {vServNames =
116 ↪ vName} ) -> return ((head sName) == (head vName)))
117
118 --trans
119 ($ (rearrAndUpdate [p| VServer {
120     vListen = listen,
121     vServNames = servNames,
122     vServRoot = servRoot,
123     vServIndex = servIndex,
124     vServKeepaliveTimeout = servKaTimeout,
125     vServKeepaliveMaxRequests = servKaMaxRequests,
126     vServSendfile = servSendfile,
127     vServSSL = servSSL,
128     vServSSLCACertificate = servCaCertif,
129     vServSSLCARevocationFile = servCaRevocFile,
130     vServSSLCertificate = servCertif,

```

```

124     vServSSLCertificateKey = servCertifKey,
125     vServSSLCiphers = servCiphers,
126     vServSSLPreferServerCiphers = servPreferServCiphers,
127     vServSSLProtocols = servProtocols,
128     vServSSLVerifyClient = servVerifyClient,
129     vServSSLVerifyDepth = servVerifyDepth,
130     vLocations = locations
131 } |] [p| Server {
132     sListen = listen,
133     sIndex = servIndex,
134     sRoot = servRoot,
135     sKeepaliveTimeout = servKaTimeout,
136     sKeepaliveRequests = servKaMaxRequests,
137     sSendFile = servSendfile,
138     sServerName = servNames,
139     sSsl = servSSL,
140     sSslCertificate = servCertif,
141     sSslCertificateKey = servCertifKey,
142     sSslCiphers = servCiphers,
143     sSslClientCertificate = servCaCertif,
144     sSslCrl = servCaRevocFile,
145     sSslPreferServerCiphers = servPreferServCiphers,
146     sSslProtocols = servProtocols,
147     sSslVerifyClient = servVerifyClient,
148     sSslVerifyDepth = servVerifyDepth,
149     sLocation = locations
150 } |] [d| servRoot = addDefault (d_root defaults);
151     servIndex = addDefault (d_index defaults);
152     servKaTimeout = addDefault (d_keepalive_timeout
153     ↪ defaults);
154     servKaMaxRequests = addDefault (d_keepalive_requests
155     ↪ defaults);
156     servSendfile = addDefault (d_send_file defaults);
157     servNames = addDefaultList (d_server_name defaults);
158     listen = addDefaultList (d_listen defaults);
159     servSSL = addDefault (d_ssl defaults);
160     servCaCertif = addDefault (d_ssl_client_certificate
161     ↪ defaults);
162     servCaRevocFile = addDefault (d_ssl_crl defaults);
163     servCertif = addDefault (d_ssl_certificate defaults);
164     servCertifKey = addDefault (d_ssl_certificate_key
165     ↪ defaults);
166     servCiphers = addDefault (d_ssl_ciphers defaults);
167     servPreferServCiphers = addDefault
168     ↪ (d_ssl_prefered_server_ciphers defaults);

```

```

164         servProtocols = addDefault (d_ssl_protocols defaults);
165         servVerifyClient = verifyClientDefault
           ↪ (d_ssl_verify_client defaults);
166         servVerifyDepth = addDefault (d_ssl_verify_depth
           ↪ defaults);
167         locations = CaseV [
168             $(branch [p| [] |]) $ $(rearr [| \ [] -> Nothing |])
           ↪ Replace,
169             $(branch [p| (_:_) |]) $ (Compose ($(rearr [| \ x ->
           ↪ (Just x) |]) Replace) (transLocation defaults))
           ]
170     ] ))
171 --create
172 --adds a new server to the source if a new one was added to the
173 ↪ view
174 (\ VServer {
175     vListen = listen,
176     vServNames = servNames,
177     vServRoot = servRoot,
178     vServIndex = servIndex,
179     vServKeepaliveTimeout = servKaTimeout,
180     vServKeepaliveMaxRequests = servKaMaxRequests,
181     vServSendfile = servSendfile,
182     vServSSL = servSSL,
183     vServSSLCACertificate = servCaCertif,
184     vServSSLCARevocationFile = servCaRevocFile,
185     vServSSLCertificate = servCertif,
186     vServSSLCertificateKey = servCertifKey,
187     vServSSLCiphers = servCiphers,
188     vServSSLPreferServerCiphers = servPreferServCiphers,
189     vServSSLProtocols = servProtocols,
190     vServSSLVerifyClient = servVerifyClient,
191     vServSSLVerifyDepth = servVerifyDepth,
192     vLocations = locations
193 } -> return Server {
194     sListen = emptyListCheck listen,
195     sIndex = emptyCheck servIndex,
196     sRoot = emptyCheck servRoot,
197     sErrorPage = Nothing,
198     sKeepaliveDisable = Nothing,
199     sKeepaliveTimeout = emptyCheck servKaTimeout,
200     sKeepaliveRequests = emptyCheck servKaMaxRequests,
201     sSendFile = emptyCheck servSendfile,
202     sServerName = emptyListCheck servNames,
203     sAccessLog = Nothing,

```

```
204     sErrorLog = Nothing,
205     sGzip = Nothing,
206     sGzipCompLevel = Nothing,
207     sSsl = emptyCheck servSSL,
208     sSslCertificate = emptyCheck servCertif,
209     sSslCertificateKey = emptyCheck servCertifKey,
210     sSslCiphers = emptyCheck servCiphers,
211     sSslClientCertificate = emptyCheck servCaCertif,
212     sSslCrl = (emptyCheck servCaRevocFile),
213     sSslPreferServerCiphers = emptyCheck servPreferServCiphers,
214     sSslProtocols = emptyCheck servProtocols,
215     sSslSessionTimeout = Nothing,
216     sSslVerifyClient = emptyCheck servVerifyClient,
217     sSslVerifyDepth = emptyCheck servVerifyDepth,
218     sLocation = Nothing,
219     sAccessRule = Nothing,
220     sAddHeader = Nothing,
221     sAio = Nothing,
222     sAuthBasic = Nothing,
223     sAuthBasicUserFile = Nothing,
224     sAutoindex = Nothing,
225     sAiExactSize = Nothing,
226     sAiFormat = Nothing,
227     sAiLocaltime = Nothing,
228     sAncientBrowser = Nothing,
229     sAncientBrowserValue = Nothing,
230     sModernBrowser = Nothing,
231     sModernBrowserValue = Nothing,
232     sCharset = Nothing,
233     sOverrideCharset = Nothing,
234     sSourceCharset = Nothing,
235     sCharsetType = Nothing,
236     sChunkedTransferEncoding = Nothing,
237     sClientHeaderBufferSize = Nothing,
238     sClientBodyBufferSize = Nothing,
239     sClientBodyInFileOnly = Nothing,
240     sClientBodyInSingleBuffer = Nothing,
241     sClientBodyTempPath = Nothing,
242     sClientBodyTimeout = Nothing,
243     sClientMaxBodySize = Nothing,
244     sClientHeaderTimeout = Nothing,
245     sConnectionPoolSize = Nothing,
246     sDefaultType = Nothing,
247     sDirectio = Nothing,
248     sDirectioAlignment = Nothing,
```

```
249     sDisableSymlinks = Nothing,
250     sEtag = Nothing,
251     sExpires = Nothing,
252     sGzipBuffers = Nothing,
253     sGzipDisable = Nothing,
254     sGzipMinLength = Nothing,
255     sGzipHttpVersion = Nothing,
256     sGzipProxied = Nothing,
257     sGzipTypes = Nothing,
258     sGzipVary = Nothing,
259     sIfModifiedSince = Nothing,
260     sIgnoreInvalidHeaders = Nothing,
261     sInclude = Nothing,
262     sLargeClientHeaderBuffers = Nothing,
263     sLimitConn = Nothing,
264     sLimitConnLogLevel = Nothing,
265     sLimitConnStatus = Nothing,
266     sLimitRate = Nothing,
267     sLimitRateAfter = Nothing,
268     sLimitReq = Nothing,
269     sLimitReqLogLevel = Nothing,
270     sLimitReqStatus = Nothing,
271     sLingingClose = Nothing,
272     sLingingTime = Nothing,
273     sLingingTimeout = Nothing,
274     sLogNotFound = Nothing,
275     sLogSubrequest = Nothing,
276     sOpenLogFileCache = Nothing,
277     sMaxRanges = Nothing,
278     sMemcachedBind = Nothing,
279     sMemCachedBufferSize = Nothing,
280     sMemcachedConnectTimeout = Nothing,
281     sMemcachedForceRanges = Nothing,
282     sMemcachedGzipFlag = Nothing,
283     sMemcachedNextUpstream = Nothing,
284     sMemcachedNextUpstreamTimeout = Nothing,
285     sMemcachedNextUpstreamTries = Nothing,
286     sMemcachedReadTimeout = Nothing,
287     sMemcachedSendTimeout = Nothing,
288     sMergeSlashes = Nothing,
289     sMsiePadding = Nothing,
290     sMsieRefresh = Nothing,
291     sOpenFileCache = Nothing,
292     sOpenFileCacheErrors = Nothing,
293     sOpenFileCacheMinUses = Nothing,
```

```

294     sOpenFileCacheValid = Nothing,
295     sOutputBuffers = Nothing,
296     sPostponeOutput = Nothing,
297     sPortInRedirect = Nothing,
298     sReadAhead = Nothing,
299     sRecursiveErrorPages = Nothing,
300     sValidReferer = Nothing,
301     sRefererHashBucketSize = Nothing,
302     sRefererHashMaxSize = Nothing,
303     sRequestPoolSize = Nothing,
304     sResetTimeoutConnection = Nothing,
305     sResolver = Nothing,
306     sResolverTimeout = Nothing,
307     sSatisfy = Nothing,
308     sSendLowat = Nothing,
309     sSendTimeout = Nothing,
310     sSendFileMaxChunks = Nothing,
311     sServerNameInRedirect = Nothing,
312     sServerTokens = Nothing,
313     sSslBufferSize = Nothing,
314     sSslDhparam = Nothing,
315     sSslEcdhCurve = Nothing,
316     sSslPasswordFile = Nothing,
317     sSslSessionCache = Nothing,
318     sSslSessionTicketKey = Nothing,
319     sSslSessionTickets = Nothing,
320     sSslStapling = Nothing,
321     sSslStaplingFile = Nothing,
322     sSslStaplingResponder = Nothing,
323     sSslStaplingVerify = Nothing,
324     sSslTrustedCertificate = Nothing,
325     sTcpNodelay = Nothing,
326     sTcpNopush = Nothing,
327     sTryFiles = Nothing,
328     sTypes = Nothing,
329     sTypesHashBucketSize = Nothing,
330     sTypesHashMaxSize = Nothing,
331     sUnderscoresInHeaders = Nothing
332   })
333   -- conceal
334   (\ _ -> return Nothing)
335
336 --locations transformation
337 transLocation :: MonadError' e m => DefaultValues -> BiGUL m
338   ↪ [Location] [VLocation]

```

```

338 transLocation defaults = Align
339   --source condition
340   (\ _ -> return True)
341   --match
342   --defines on which field the records will match between source and
   ↪ view
343   (\ (Location { lLocationPath = (Just sPath) } ) (VLocation {
   ↪ vLocationPath = vPath } ) -> return (sPath == vPath))
344   --trans
345   $($rearrAndUpdate [p| VLocation {
346     vLocationPath = locPath,
347     vLocIndex = locIndex,
348     vLocSendfile = locSendfile
349   } |] [p| Location {
350     lLocationPath = locPath,
351     lIndex = locIndex,
352     lSendFile = locSendfile
353   } |] [d| locPath = $(rearr [| \ x -> (Just x) |]) Replace;
354     locIndex = addDefault (d_index defaults);
355     locSendfile = addDefault (d_send_file defaults)
356   |]))
357   --create
358   --adds a new location to the source if a new one was added to the
   ↪ view
359   (\ VLocation {
360     vLocationPath = locPath,
361     vLocIndex = locIndex,
362     vLocSendfile = locSendfile
363   } -> return Location {
364     lLocationPath = emptyCheck locPath,
365     lRoot = Nothing,
366     lIndex = emptyCheck locIndex,
367     lErrorPage = Nothing,
368     lKeepaliveDisable = Nothing,
369     lKeepaliveTimeout = Nothing,
370     lKeepaliveRequests = Nothing,
371     lSendFile = emptyCheck locSendfile,
372     lAccessLog = Nothing,
373     lErrorLog = Nothing,
374     lLocation = Nothing,
375     lAccessRule = Nothing,
376     lAddHeader = Nothing,
377     lAio = Nothing,
378     lAlias = Nothing,
379     lAuthBasic = Nothing,

```

```
380     lAuthBasicUserFile = Nothing,
381     lAutoindex = Nothing,
382     lAiExactSize = Nothing,
383     lAiFormat = Nothing,
384     lAiLocaltime = Nothing,
385     lAncientBrowser = Nothing,
386     lAncientBrowserValue = Nothing,
387     lModernBrowser = Nothing,
388     lModernBrowserValue = Nothing,
389     lCharset = Nothing,
390     lOverrideCharset = Nothing,
391     lSourceCharset = Nothing,
392     lCharsetType = Nothing,
393     lChunkedTransferEncoding = Nothing,
394     lClientBodyBufferSize = Nothing,
395     lClientBodyInFileOnly = Nothing,
396     lClientBodyInSingleBuffer = Nothing,
397     lClientBodyTempPath = Nothing,
398     lClientBodyTimeout = Nothing,
399     lClientMaxBodySize = Nothing,
400     lDefaultType = Nothing,
401     lDirectio = Nothing,
402     lDirectioAlignment = Nothing,
403     lDisableSymlinks = Nothing,
404     lEtag = Nothing,
405     lExpires = Nothing,
406     lGzip = Nothing,
407     lGzipBuffers = Nothing,
408     lGzipCompLevel = Nothing,
409     lGzipDisable = Nothing,
410     lGzipMinLength = Nothing,
411     lGzipHttpVersion = Nothing,
412     lGzipProxied = Nothing,
413     lGzipTypes = Nothing,
414     lGzipVary = Nothing,
415     lIfModifiedSince = Nothing,
416     lInclude = Nothing,
417     lInternal = Nothing,
418     lLimitConn = Nothing,
419     lLimitConnLogLevel = Nothing,
420     lLimitConnStatus = Nothing,
421     lLimitRate = Nothing,
422     lLimitRateAfter = Nothing,
423     lLimitReq = Nothing,
424     lLimitReqLogLevel = Nothing,
```

```
425     lLimitReqStatus = Nothing,
426     lLingeringClose = Nothing,
427     lLingeringTime = Nothing,
428     lLingeringTimeout = Nothing,
429     lLogNotFound = Nothing,
430     lLogSubrequest = Nothing,
431     lOpenLogFileCache = Nothing,
432     lMaxRanges = Nothing,
433     lMemcachedBind = Nothing,
434     lMemCachedBufferSize = Nothing,
435     lMemcachedConnectTimeout = Nothing,
436     lMemcachedForceRanges = Nothing,
437     lMemcachedGzipFlag = Nothing,
438     lMemcachedNextUpstream = Nothing,
439     lMemcachedNextUpstreamTimeout = Nothing,
440     lMemcachedNextUpstreamTries = Nothing,
441     lMemcachedReadTimeout = Nothing,
442     lMemcachedSendTimeout = Nothing,
443     lMemcachedPass = Nothing,
444     lMsiePadding = Nothing,
445     lMsieRefresh = Nothing,
446     lOpenFileCache = Nothing,
447     lOpenFileCacheErrors = Nothing,
448     lOpenFileCacheMinUses = Nothing,
449     lOpenFileCacheValid = Nothing,
450     lOutputBuffers = Nothing,
451     lPostponeOutput = Nothing,
452     lPortInRedirect = Nothing,
453     lReadAhead = Nothing,
454     lRecursiveErrorPages = Nothing,
455     lValidReferer = Nothing,
456     lRefererHashBucketSize = Nothing,
457     lRefererHashMaxSize = Nothing,
458     lResetTimedoutConnection = Nothing,
459     lResolver = Nothing,
460     lResolverTimeout = Nothing,
461     lSatisfy = Nothing,
462     lSendLowat = Nothing,
463     lSendTimeout = Nothing,
464     lSendFileMaxChunks = Nothing,
465     lServerNameInRedirect = Nothing,
466     lServerTokens = Nothing,
467     lTcpNodelay = Nothing,
468     lTcpNopush = Nothing,
469     lTryFiles = Nothing,
```

```

470         lTypes = Nothing,
471         lTypesHashBucketSize = Nothing,
472         lTypesHashMaxSize = Nothing
473     })
474     -- conceal
475     (\ _ -> return Nothing)
476
477
478     -----
479     --DEFAULT VALUES GESTION
480     -----
481     --defaults gestion for simple fields
482     addDefault :: MonadError' e m => String -> BiGUL m (Maybe String)
483     ↪ String
484     addDefault def = CaseV [ ((return . (== def)),
485         (CaseS [ $(normal [p| Nothing |]) $ Rearr (RConst def) (EIn (ELeft
486             ↪ (EConst ()))) Replace,
487             $(normal [p| (Just _) |]) $ $(rearr [| \ x -> (Just x)
488                 ↪ |]) Replace
489         ])) ,
490         ((return . (/= def)),
491         ($ (rearr [| \ x -> (Just x) |]) Rearr) )
492     ]
493
494     --defaults gestion for list fields
495     addDefaultList :: MonadError' e m => String -> BiGUL m (Maybe
496     ↪ [String]) [String]
497     addDefaultList def = CaseV [ ((return . (== [def])),
498         (CaseS [ $(normal [p| Nothing |]) $ Rearr (RConst [def]) (EIn
499             ↪ (ELeft (EConst ()))) Replace,
500             $(normal [p| (Just _) |]) $ $(rearr [| \ x -> (Just x)
501                 ↪ |]) Replace
502         ])) ,
503         ((return . (/= [def])),
504         ($ (rearr [| \ x -> (Just x) |]) Rearr) )
505     ]
506
507     --defaults gestion for the VerifyClient directive
508     verifyClientDefault :: MonadError' e m => String -> BiGUL m (Maybe
509     ↪ String) String
510     verifyClientDefault def = CaseV [ ((return . (liftM2 (&&) (== def) (==
511         ↪ "yes"))),
512         (CaseS [ $(normal [p| Nothing |]) $ Rearr (RConst def) (EIn (ELeft
513             ↪ (EConst ()))) Replace,

```

```

505     $(normal [p| (Just _) |]) $ $(rearr [| \ "yes" -> (Just
506     ↪ "on") |]) Replace
507   )),
508     ((return . (liftM2 (&&) (== def) (==
509     ↪ "no"))),
510   (CaseS [ $(normal [p| Nothing |]) $ Rearr (RConst def) (EIn (ELeft
511     ↪ (EConst ()))) Replace,
512     $(normal [p| (Just _) |]) $ $(rearr [| \ "no" -> (Just
513     ↪ "off") |]) Replace
514   ])) ,
515     ((return . (liftM2 (&&) (/= def) (==
516     ↪ "yes"))),
517   (($rearr [| \ "yes" -> (Just "on") |]) Replace) ),
518     ((return . (liftM2 (&&) (/= def) (==
519     ↪ "no"))),
520   (($rearr [| \ "no" -> (Just "off") |]) Replace) ),
521     ((return . (/= def)),
522   (($rearr [| \ x -> (Just x) |]) Replace) )
523 ]
524
525 -----
526 --OPERATIONS
527 -----
528 --performs get and show extracted view in console
529 --does not override existing view
530 getNginx1 x = (catchBind (get (transNginx defaults) x) (\v -> Right
531     ↪ (show v)) (\e -> Left e))
532 extractConfig = parseTreeNginx "nginx.conf" >>= \(Right tree) ->
533     ↪ return (createSourceNginx tree) >>= return . getNginx1
534
535 --performs get and rewrites view file
536 extractConfigToFile = do
537   content <- extractConfig
538   case content of
539     Left e -> putStrLn ("some error: " ++ show e)
540     Right r -> writeFile "Nginx_output.hs" ("module Nginx_output
541     ↪ where"++"\n"++"import TypeFiles.Common"++"\n"++"nginxOutput ::
542     ↪ CommonWebserver"++"\n"++"nginxOutput = "++r)

```

```

538
539
540 --performs putback and show new source in console
541 --does not override existing source config file
542 putNginx1 x = catchBind (put (transNginx defaults) x nginxView')
    ↳ (Right . printNginx) Left
543 putbackConfig = parseTreeNginx "nginx.conf" >>= \(Right tree) ->
    ↳ return (createSourceNginx tree) >>= return . putNginx1
544
545 --performs putback and rewrites source config file
546 putbackConfigToFile = do
547     content <- putbackConfig
548     case content of
549     Left e -> putStrLn ("some error: " ++ show e)
550     Right r -> writeFile "nginx.conf" ((show r))
551
552
553 -----
554 --OTHER FUNCTIONS FOR TESTING PURPOSE
555 -----
556 testPut :: BiGUL (Either ErrorInfo) s v -> s -> v -> Either ErrorInfo
    ↳ s
557 testPut u s v = catchBind (put u s v) (\s' -> Right s') (\e -> Left e)
558
559 testGet :: BiGUL (Either ErrorInfo) s v -> s -> Either ErrorInfo v
560 testGet u s = catchBind (get u s) (\v' -> Right v') (\e -> Left e)
561
562 --demo function for showing source
563 showSource = parseTreeNginx "nginx.conf" >>= \(Right tree) -> return
    ↳ (createSourceNginx tree) >>= return . show
564
565
566 -----
567 --OTHER ANNEX FUNCTIONS
568 -----
569 --replaces an empty field by Nothing
570 --used when creating a new element in the source
571 emptyCheck :: String -> Maybe String
572 emptyCheck input = if (input == "") then Nothing else (Just input)
573
574 --replaces an empty list by Nothing
575 --used when creating a new element in the source
576 emptyListCheck :: [String] -> Maybe [String]
577 emptyListCheck input = if (input == []) then Nothing else (Just input)

```