Reusable Self-Adaptation through Bidirectional Programming

Kevin Colson
University of Namur
Namur, Belgium
kevin.colson@student.unamur.be

Robin Dupuis
University of Namur
Namur, Belgium
robin.dupuis-1@student.unamur.be

Lionel Montrieux
National Institute of Informatics
Tokyo, Japan
lionel@nii.ac.jp

Zhenjiang Hu
National Institute of Informatics
Tokyo, Japan
hu@nii.ac.jp

Sebastián Uchitel
University of Buenos Aires - Imperial College of London
Buenos Aires, Argentina - London, UK
s.uchitel@imperial.ac.uk

Pierre-Yves Schobbens
University of Namur
Namur, Belgium
pierre-yves.schobbens@unamur.be

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

ISBN 978-1-4503-4187-5.

DOI: 10.1145/1235

1. INTRODUCTION

Self-adaptive systems are sometimes 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 [4]. 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 probes and executors, which is sometimes captured in a third layer, between the target system and the adaptation layer [10].

Research on reusability of adaptive layers has focused on providing frameworks for adaptation, allowing developers to customize each phase of the feedback loop [3, 10], without having to implement the entire layer themselves. These approaches rely on users carefully crafting pairs of monitors and effectors, in order for the adaptive layer to keep an up-to-date 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 model, which can lead to counter-productive adaptation decisions.

In this paper, we provide a synchronization mechanism that is correct by construction. Our approach focuses on the adaptation of configuration files, which are part of the target system, and describe the configuration of the system. While this focus limits the kinds of adaptations that our approach can handle, it allows us to guarantee, by construction, the correctness of the synchronization of configuration files (which we call *concrete models*) with their abstract representations. This is not a trivial task to manually carry out, especially considering common constructs found in configuration files, such as default values or context overriding.

To ensure that abstract and concrete models are consistent, we synchronize them using bidirectional transformations (BX) [7], automatically derived from bidirectional pro-

grams. BXs consist of a pair of functions: a forward transformation, and a backward transformation. The forward transformation, or *get*, takes a source as input and generates a view. The backward transformation, or *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 [8]. Bidirectional programming languages are Domain-Specific Languages (DSLs) that help developers write BXs. *Well-behaved* BXs ensure that the composition of the *get* and *put* functions, or the opposite, is the identity function [7]. In this paper, 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 paper is structured as follows: Section 2 gives a practical example of the problems our approach solves. Section 3 provides background on self-adaptive systems, and BXs. Our approach is presented in Section 4. Section 5 details our use of bidirectional programming for synchronization. In Section 6, we present our case study which consists of two scenarios: adaptation and migration. We examine threats to the case study's validity in Section 7. After discussing related work in Section 8, we conclude in Section 9.

2. PROBLEMS IN REUSE AND SYNCHRONIZATION

In this section, we use web servers as an example to 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, even for simply configuration files. We use two web server implementations throughout this paper: Apache HTTP Server[1], and Nginx[2].

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., different implementations of 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 different versions of the same implementation. 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 this example, with Nginx:

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

The *keepalive_timeout* instruction 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;
  }
}
```

[1]https://httpd.apache.org/
[2]http://nginx.org/

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

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

Default values and contexts are common in many types of configuration files. They can greatly complicate the adaptation of configuration files, as a change in one entry may have effects in multiple contexts.

3. BACKGROUND

3.1 Adaptive layer

Adaptation is triggered by changes in a self-adaptive system, its environment, and/or its goals [4]. If the data shows changes in the system, its environment, or its goals, that require adaptation, changes made by this adaptation have to be effected on the system. Those steps are realized by a layer above the system. The MAPE-K loop architecture [12] is composed of four consecutive phases, as well as a common knowledge base that helps sharing informations between those phases. 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" [12].

3.2 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. The forward transformation, called *get*, takes a source as input, and produces a view. The backward transformation, called *put*, takes both a source and an updated view as input, and produces an updated source, where changes made to the view are embedded into the source. The two transformations are defined as follows [8]:

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

A BX could, for example, synchronize a 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* [7]. They provide *well-behaved* synchronization between source and view. To be well-behaved, a BX has to satisfy two laws, *PutGet* and *GetPut*, defined as follows [6]:

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

Several programming languages exist to help developers write BXs. They usually are either *get-based* programming languages [**?**, **?**], 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[15], where the *put* function is provided by the developer, and a *get* function automatically derived, to produce a well-behaved BX. 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 [6].

3.3 BiGUL

In this paper, we use the putback-based bidirectional programming language BiGUL [15]. Below is a simple BiGUL program.

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

The source, defined on lines 1 and 2, contains two fields; the view, defined on line 3, only contains one field. The program, which specifies the *put* behavior, is defined on lines 5-14. The program's signature, on line 5, indicates that it matches a source with a view. Line 6-14 are a `rearrAndUpdate` instruction, which 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 7 to 9 indicate that the element `va` in the view will be matched to `a`. Lines 10 to 12 indicate that the element `sa` in the source will be matched to `a` as well. Finally, lines 13 and 14 indicate that the element matched with `a` in the source will be replaced by the element matched with `a` in the view. Therefore, the element `sb` in the source will be left unchanged.

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 probes, gauges, and effectors, will have to ensure that their implementation is correct. This can be difficult and time-consuming.

4. MODEL ABSTRACTION IN SELF-ADAPTATION

Adaptation performed directly on configuration files requires the customization of the adaptation logic if it is to be reused across several implementations. For example, a self-adaptive system could adapt the path of a web server's access log file. This can be done with both Apache and Nginx, but differently. With Nginx, the relevant configuration is the following:

```
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 despite the fact that it provides the same functionality. In Nginx, the adaptation mechanisms just need to change one instruction, while in Apache they need to change two.

4.1 Abstraction

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 1). Various implementations of the same system would each have their own concrete model, all feeding into an abstract model (Figure 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.

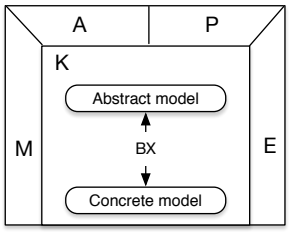

Figure 1: model abstraction in MAPE loop

In addition to facilitating the reusability of adaptive layers, our approach also provides a way to copy parts or the

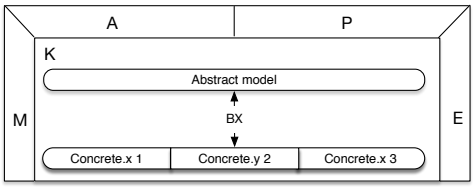

Figure 2: abstract model equivalent for all systems

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 an other 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 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 a `nofile` 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 = "nofile"
```

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.

4.1.1 Adaptive Layer Reusability

In our approach, the adaptation phases work on different models (Figure 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";
```

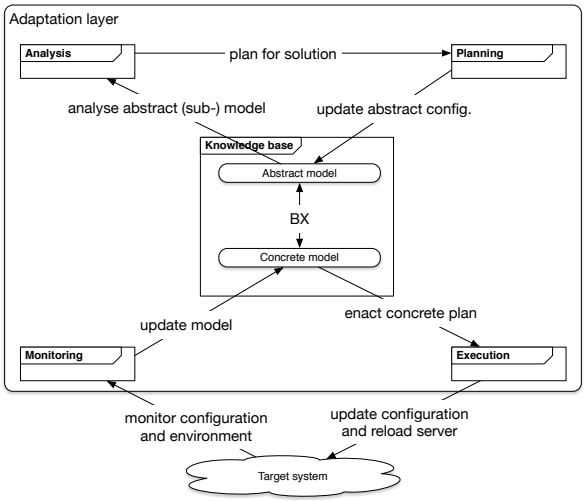

Figure 3: Architecture of the approach

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"
```

Let us assume that an adaptation rule specifies that when disk space is short on a server, no more accesses will be logged. The adaptation will consist of 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 = "nofile"
```

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
```

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 4 shows that different partial abstract models can be extracted from the same concrete model, for different adaptation concerns. Figure 5 uses our approach's composition

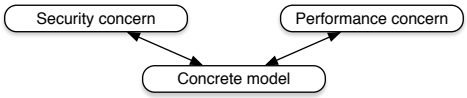

Figure 4: direct concern extraction

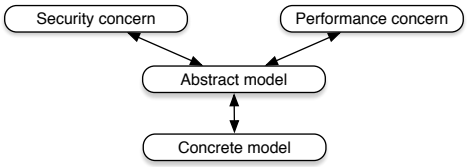

Figure 5: indirect concern extraction

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.

4.1.2 *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`.

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 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.

4.2 Synchronization

A synchronization mechanism between concrete and abstract models must be well-behaved, and must handle the typical constructs of configuration files discussed in Section 2.

Synchronization 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 bugs 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 guarnatee that bidirectional transformations will be well-behaved.

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.

5. BIDIRECTIONAL PROGRAMMING

In this section, we show how challenges caused by typical constructs in configuration files can be overcome with putback-based bidirectional programming.

5.1 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 previously empty fields 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. The implementation in BiGUL can be found on our repository[3].

```
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  }
```

This pseudo-code defines the put behavior. 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 is equal to the default value, we check the current value in the 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.

5.2 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 3.2. 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 6 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 upper 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 /textitput function that takes an extra argument: the result of a function `f(v)` that determines whether a value in the view is a default value or not. We use it to redefine `put`:

[3]https://github.com/prl-tokyo/bigul-configuration-adaptation

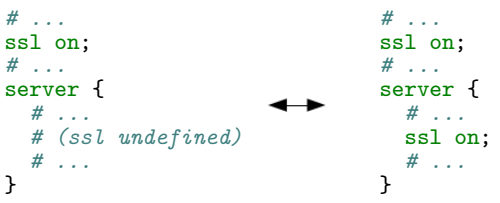

Figure 6: Context overriding

```
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'`:

```
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.

6. 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 new server's concrete configuration using our approach.

This case study focuses on BX within a self-adaptive sytem, 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.

6.1 Setup

6.1.1 *Internal representation*

We present here the sample configuration files used in our case study. Listing 1 shows the Apache configuration file and Listing 2 shows the Nginx 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.

Listing 1: 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>
```

To turn a configuration file into a source usable by BiGUL, 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 into the configuration file, which is done using a pretty printer and a set of rules.

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 3 shows a simplified version of the Apache source, while Listing 4 shows a simplified version of the Nginx source.

6.1.2 *View*

Both simplified sources in Listings 3 and 4 give the same resulting view. Listing 5 presents a simplified version of the

Listing 2: Nginx configuration file

```
pid /run/nginx.pid;
user www-data;
worker_processes 4;
events {
  worker_connections 768;
}
http {
  keepalive_timeout 65;
  keepalive_requests 100;
  access_log /var/log/nginx/access.log;
  error_log /var/log/nginx/error.log;
  ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
  tcp_nodelay on;
  tcp_nopush on;
  server_tokens on;
  gzip on;
  gzip_comp_level 2;
  server {
    listen 80;
    root /var/www/html;
    server_name example.com;
    error_log /var/log/nginx/error.log;
    keepalive_timeout 65;
    keepalive_requests 100;
  }
}
```

Listing 3: Simplified Apache source

```
apacheSource :: ApacheWebserver
apacheSource = ApacheWebserver {
  aDocumentRoot = Nothing,
  aKeepAlive = Just "On",
  aKeepAliveTimeout = Just "65",
  aMaxKeepAliveRequests = Just "100",
  aListen = Just ["80"],
  aDirectoryIndex = Nothing,
  aSSLCertificateFile = Nothing,
  aSSLCertificateKeyFile = Nothing,
  aVirtualHosts = Just [
    VirtualHost {
      sVirtualHostAddress = Just "*:80",
      sDocumentRoot = Just "/var/www/html",
      sKeepAlive = Just "On",
      sKeepAliveTimeout = Just "65",
      sMaxKeepAliveRequests = Just "100",
      sLocation = Nothing,
      sServerName = Just "example.com",
      sDirectoryIndex = Nothing,
      sSSLEngine = Nothing,
      sSSLCertificateFile = Nothing,
      sSSLCertificateKeyFile = Nothing
    }
  ]
}
```

view extracted using the *get* transformation.

6.2 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.

6.2.1 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 Nginx and Apache configuration files discussed in Section 6.1.1. They are represented by the Haskell records, portions of which are on Listing 3 and Listing 4, 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. Samples of the abstract models are in Listing 5, and the entire records are on the repository.

We then simulated the adaptation by changing the following values in the views. The changes are 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.

6.2.2 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.

Listing 4: Simplified Nginx source

```
nginxSource :: NginxWebserver
nginxSource = NginxWebserver {
  nWorkerProcesses = Just "4",
  nHttp = Just Http {
    hKeepaliveDisable = Nothing,
    hKeepaliveTimeout = Just "65",
    hKeepaliveRequests = Just "100",
    hRoot = Nothing,
    hServer = Just [
      Server {
      sKeepaliveDisable = Nothing,
      sKeepaliveTimeout = Just "65",
      sKeepaliveRequests = Just "100",
      sListen = Just ["80"],
      sLocation = Nothing,
      sRoot = Just "/var/www/html",
      sServerName = Just ["example.com"],
      sSsl = Nothing,
      sSslCertificate = Nothing,
      sSslCertificateKey = Nothing }
    ],
    hSsl = Nothing,
    hSslCertificate = Nothing,
    hSslCertificateKey = Nothing
  }
}
```

Listing 5: Simplified view

```
reducedView :: CommonWebserver
reducedView = CommonWebserver {
  vRoot = "html",
  vKeepaliveTimeout = "65",
  vSSL = "off",
  vSSLCertificate = "",
  vSSLCertificateKey = "",
  vServers = [
  VServer {
    vListen = ["80"],
    vServNames = ["example.com"],
    vServRoot = "/var/www/html",
    vServKeepaliveTimeout = "65",
    vServSSL = "off",
    vServSSLCertificate = "",
    vServSSLCertificateKey = ""
  }
  ]
}
```

6.3 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 a newly deployed web server using a different technology.

6.3.1 Experiment

First, we confirmed that the first web server is 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 source, 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.

6.3.2 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.

7. THREATS TO VALIDITY

7.1 Internal Validity

In both scenarios, we simulated the outcome of a MAPE loop, by manually modifying abstract models, rather than implementing a feedback loop. Since this paper focuses on the synchronization between concrete and abstract models, and the associated challenges, we argue that a simulated MAPE loop does not negatively impact the case study's validity. Similary, we ignored the issue of transferring updated configuration files to the servers, and reloading them.

7.2 External Validity

We assumed that no modification was done on the configuration file between parsing and 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 [23].

8. RELATED WORK

8.1 Reusability

Klein et al. introduce a new way to program for self-adaptation based on optional code that can be dynamically deactivated, and apply this technique to a web application [14]. *Garlan et al.* show the use of a framework, Rainbow, that is composed of reusable parts to which the user can hook personalized code [10]. Rainbow was also extended by *Swanson et al.* with a framework called REFRACT, which brings failure avoidance components and algorithms [18]. *Barna et al.* propose a platform for deploying self-managing web applications on cloud called Hogna [3]. Although these approaches also allow for the reuse of abstract models, they require the careful development of both monitors 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* present different patterns that can be reused for adaptation [16].

8.2 Models within self-adaptation

Vogel and Giese present a model-driven approach for adaptation that contains different types of models for specific adaptation levels [19]. This allows separation of concerns. They also present an approach to ease the development of architectural monitoring based on incremental model synchronization [21]. They demonstrate an executable modeling language for ExecUtable RuntimE MegAmodels (EUREMA), that makes the development of adaptation engines easier by following a model-driven engineering approach that uses megamodels. Megamodels are models that represent a system at runtime along with its adaptation activities [20]. *Angelopoulos et al.* use Rainbow as a comparison for their

framework, Zanshin, which is requirement-based instead of architecture-based [1]. They compare both approaches, which use different kinds of models. *Georgas et al.* use a model to record the history of a managed system's states [11]. 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. Another example of adaptation around models is the one of *Bailey et al.* They 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 [2]. None of these approaches use bidirectional programming. As BXs are often not labeled as such, they need to be manually maintained, and their *well-behavedness* needs to be manually guaranteed. *Anderson et al.* presents the computational reflection paradigm within the self-adaptive context [**?**]. The causality property from this paradigm states that two entities are causally connected if changes made in one of the entities are reflected in the other. The self-adaptation in regard to computational reflection should have its meta-models representation causally connected to the running system. In our approach, the model and its abstraction are causally connected by the BX and the model itself is causally connected to the running system by the effectors.

8.3 Bidirectional transformations

An example of synchronization between models supporting the Atlas Tranformation Language (ATL) is offered by *Xiong et al.* [22]. 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. We use putback-based programming instead, which gives user total control over the put transformation. *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 [5]. They do not mention configuration files and their specificities. Another application of BXs for synchronizing documents is presented by *Hu et al.* [13]. 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 [17]. *Foster et al.* propose a general theory of *quotient lenses* [9]. 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 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 [24]. They use a get-based bidirectional programming language.

9. CONCLUSION AND FUTURE WORK

In this paper, we presented an approach based on abstract models and bidirectional programming for self-adaptation. Our approach provides, by construction, a provably correct synchronization between concrete and abstract models, as opposed to ad-hoc approaches that rely on developers to carefully build monitors and effectors. Our approach facilitates the reuse of adaptive layers across different implementations of the 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 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 BXs. We discussed the challenges that arise from typical constructs in configuration files, and showed that they can be overcome using bidirectional programming. 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, and that the knowledge in the abstract model could easily be copied between different implementations.

In future work, we will plan a more detailed evaluation of our approach, using an actual feedback loop. We will also investigate the applicability and performance of our approach on large scale examples, and expand its scope beyond configuration files.

Acknowledgements

The authors would 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.

This work is partially supported by the Nation Basic Research Program (973 Program) of China (grant No. 2015CB352201) and by JSPS Grant-in-Aid for Scientific Research (A) No. 25240009 of Japan.

10. REFERENCES

[1] Konstantinos Angelopoulos, Vitor E. Silva Souza, and Joao Pimentel. 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, 2013.

[2] Christopher Bailey, Lionel Montrieux, Rogério de Lemos, Yijun Yu, and Michel Wermelinger. 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, 2014.

[3] Cornel Barna, Hamoun Ghanbari, Marin Litoiu, and Mark Shtern. 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, May 2015.

[4] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. 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, January 2009.

[5] Krzysztof Czarnecki, Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In Richard Paige, editor, *Theory and Practice of Model Transformations*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer Berlin Heidelberg, 2009.

[6] Sebastian Fischer, ZhenJiang Hu, and Hugo Pacheco. The essence of bidirectional programming. *Science China Information Sciences*, 58(5):1–21, May 2015.

[7] Nathan Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 2010.

[8] Nathan Foster, Benjamin Pierce, and Steve Zdancewic. Updatable Security Views. In *Proc. Computer Security Foundations Symposium (CSF'09)*, pages 60–74. IEEE, July 2009.

[9] Nathan Foster, Alexandre Pilkiewicz, and Benjamin Pierce. Quotient lenses. In *ACM Sigplan Notices*, volume 43, pages 383–396. ACM, 2008.

[10] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[11] John Georgas, André van der Hoek, and Richard Taylor. Architectural runtime configuration management in support of dependable self-adaptive software. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–6. ACM, 2005.

[12] I. B. M. Group. An architectural blueprint for autonomic computing. Technical report, IBM White paper, 2005.

[13] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. 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, 2004.

[14] Cristian Klein, Martina Maggio, Karl-Erik Arzén, and Francisco Hernández-Rodriguez. Brownout: building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 700–711. ACM, 2014.

[15] Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. 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, 2016. ACM.

[16] Andres Ramirez and Betty Cheng. 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, 2010.

[17] Hui Song, Gang Huang, Franck Chauvel, Yingfei Xiong, Zhenjiang Hu, Yanchun Sun, and Hong Mei. Supporting runtime software architecture: A bidirectional-transformation-based approach. *Journal of Systems and Software*, 84(5):711–723, 2011.

[18] Jacob Swanson, Myra Cohen, Matthew Dwyer, Brady Garvin, and Justin Firestone. 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, 2014.

[19] Thomas Vogel and Holger Giese. 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, 2010.

[20] Thomas Vogel and Holger Giese. Model-Driven Engineering of Self-Adaptive Software with EUREMA. *ACM Transactions on Autonomous and Adaptive Systems*, 8(4):1–33, January 2014.

[21] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. Incremental model synchronization for efficient run-time monitoring. In *Models in Software Engineering*, pages 124–139. Springer, 2010.

[22] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. 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, 2007.

[23] Yingfei Xiong, Hui Song, Zhenjiang Hu, and Masato Takeichi. Supporting parallel updates with bidirectional model transformations. In *Theory and Practice of Model Transformations*, pages 213–228. Springer, 2009.

[24] Yijun Yu, Yu Lin, Zhenjiang Hu, Soichiro Hidaka, Hiroyuki Kato, and Lionel Montrieux. Maintaining invariant traceability through bidirectional transformations. In *Proceedings of the 34th International Conference on Software Engineering*, pages 540–550. IEEE Press, 2012.