Reversible 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

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 simple 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:

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

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:

```plaintext
<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:

```plaintext
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:

```plaintext
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:

```plaintext
http {
    keepalive_timeout 100s;
    server {
        keepalive_timeout 100s;
    }
}
```
will be equivalent to the following, because the value of the
\texttt{keepalive\_timeout} entry for the \texttt{server} context is defined
within its \texttt{http} ancestor.

\begin{verbatim}
http {
  keepalive\_timeout 100s;
  server {
  }
}
\end{verbatim}

Default values and contexts are common in many types of
configuration files. They can greatly complicate the adap-
tation 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 sys-
tem, 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 architec-
ture [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 re-
  source”;
- “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 con-
  struct the actions needed to achieve goals and objec-
  tives. The planning mechanism uses policy informa-
  tion 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 synchro-
nize 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 \texttt{get}, takes
a source as input, and produces a view. The backward
transformation, called \texttt{put}, takes both a source and an up-
dated 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]:

\begin{verbatim}
get :: Source \rightarrow View
put :: Source \rightarrow View \rightarrow Source'
\end{verbatim}

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

\begin{verbatim}
get (x:xs) = x
put (x:xs) y = y:xs
\end{verbatim}

A subset of BX is called well-behaved bidirectional trans-
formations, sometimes called \textit{lenses} [7]. They provide well-
behaved synchronization between source and view. To be
well-behaved, a BX has to satisfy two laws, \texttt{PutGet} and
\texttt{GetPut}, defined as follows [6]:

\begin{verbatim}
get (put s v) = v --\texttt{PutGet}
put s (get s) = s --\texttt{GetPut}
\end{verbatim}

Several programming languages exist to help developers
write BXs. They usually are either \textit{get-based} programming
languages [?, ?], where the \texttt{get} function is provided by
the developer, and a \texttt{put} function automatically derived, to pro-
duce a well-behaved BX; or \textit{putback-based} programming lan-
guages[15], where the \texttt{put} function is provided by the devel-
opener, and a \texttt{get} function automatically derived, to produce a
well-behaved BX. For each \texttt{get} function, there may be many
\texttt{put} functions that form a well-behaved BX. The advantage
of putback-based languages is that, under some conditions,
given a \texttt{put} function, there is at most one \texttt{get} function that
forms a well-behaved BX [6].

3.3 BiGUL

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

\begin{verbatim}
data Src = Src {sa :: String,
  sb :: Int} deriving (Show)
data View = View {va :: String} deriving (Show)

abc :: BiGUL Src View
abc = $(rearrAndUpdate
  [p] View {
  [p] Src {
  [d] a = Replace
  [1] [1]
  va = a
  })
  [1]})
\end{verbatim}

The source, defined on lines 1 and 2, contains two fields; the
view, defined on line 3, only contains one field. The pro-
gram, which specifies the \texttt{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 \texttt{rearrAndUp-
date} instruction, which matches elements of the source with
elements of the view, and performs the specified operations
to update the source. \texttt{rearrAndUpdate} takes three argu-
ments: 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 \texttt{va} in the view will
be matched to \texttt{a}. Lines 10 to 12 indicate that the element
\texttt{sa} in the source will be matched to \texttt{a} as well. Finally, lines
13 and 14 indicate that the element matched with \texttt{a} in the
source will be replaced by the element matched with \texttt{a}
in the view. Therefore, the element \texttt{sb} in the source will be
left unchanged.

BiGUL guarantees that, if it can compile a bidirectional pro-
gram into a BX, that BX will successfully run only if it is
well-behaved. Hence, using BiGUL for synchronization be-
tween concrete and abstract models guarantees, by construc-
tion, 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:

```plaintext
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:

```plaintext
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 specifics 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](image)

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 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:

```plaintext
abstractLog :: String
```

The abstract models for both web servers could then be:

```plaintext
****** 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:

```plaintext
****** Nginx access log config ******
access_log "'/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

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

After adaptation

```plaintext
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:

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

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

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

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

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

```plaintext
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:

```bash
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:

```plaintext
get {
  if (a empty)
    then return default
  else return a
}

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 guarantee 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:

```plaintext
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 value 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\(^\dagger\), 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:\(^\dagger\)

\(^\dagger\)https://github.com/prl-tokyo/bigul-configuration-adaptation
### 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 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.

#### 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

```plaintext
User www-data
ServerRoot "/etc/apache2"
PidFile /var/run/apache2/apache2.pid
LogLevel warn
ErrorLog /var/log/apache2/error.log
IncludeOptional conf-enabled/*.conf
IncludeOptional mods-enabled/*.load
ServerSignature On
ServerTokens OS
Listen 443
Listen 80
IncludeOptional mods-enabled/*.conf
IncludeOptional conf-enabled/*.conf
ServerName www.example.com
ServerAdmin webmaster@localhost
DocumentRoot /var/www/html
ServerName www.example.com
IncludeOptional conf-enabled/*.conf
IncludeOptional mods-enabled/*.load
ServerSignature On
ServerTokens OS
Listen 443
Listen 80
IncludeOptional mods-enabled/*.conf
IncludeOptional conf-enabled/*.conf
ServerName www.example.com
ServerAdmin webmaster@localhost
DocumentRoot /var/www/html
```

Listing 2: Nginx configuration file

```plaintext
http {
    # Default server template.
    server {
        # Listen for HTTP on port 80.
        listen 80;
        # Include all configuration files from the directory.
        include conf-enabled/*.conf;
        # Include all configuration files from the directory.
        include modss-enabled/*.load;
        # Listen for HTTP on port 443.
        listen 443 ssl;
        # Include all configuration files from the directory.
        include conf-enabled/*.conf;
        # Include all configuration files from the directory.
        include modss-enabled/*.load;
    }
}
```

#### 6.1.2 View

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.

Listing 3: Apache source

```plaintext
module Apache where

import Data.Map (Map)
import Language.HsPrelude (Id, docs)

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

get s = get' _ s

PutGet

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

PutPut

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

This proves that context overriding can be handled using put-based bidirectional programming.

Listing 4: Nginx source

```plaintext
module Nginx where

import Data.Map (Map)
import Language.HsPrelude (Id, docs)

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

get s = get' _ s

PutGet

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

PutPut

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

This proves that context overriding can be handled using put-based bidirectional programming.

#### 6.2 CASE STUDY

This case study focuses on BX within a self-adaptive system, using two implementations: Apache and Nginx; we consider their Apache and Nginx configuration files, respectively.

We use the `apache2.conf` and `nginx.conf` 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 5: Apache source

```plaintext
module Apache where

import Data.Map (Map)
import Language.HsPrelude (Id, docs)

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

get s = get' _ s

PutGet

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

PutPut

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

This proves that context overriding can be handled using put-based bidirectional programming.

Listing 6: Nginx source

```plaintext
module Nginx where

import Data.Map (Map)
import Language.HsPrelude (Id, docs)

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

get s = get' _ s

PutGet

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

PutPut

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

This proves that context overriding can be handled using put-based bidirectional programming.

Figure 6: Context overriding

```plaintext
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))
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 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.
Listing 2: Nginx configuration file

```plaintext
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_nopush on;
  tcp_nodelay on;
  gzip on;
  gzip_comp_level 2;
  gzip on;
  gzip on;

  server {
    listen 80;
    root /var/www/html;
    server_name example.com;
    error_log /var/log/nginx/error.log;
    keepalive_timeout 65;
    keepalive_requests 100;
  }
}
```

view extracted using the \textit{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 \textit{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.

```haskell
Values before adaptation
vListen = "80",
vServKeepaliveTimeout = "65",
vServSSL = "off",
vServSSLCertificate = "",
vServSSLCertificateKey = ""
```

```haskell
Values after adaptation
vListen = "443",
vServKeepaliveTimeout = "75",
vServSSL = "on",
vServSSLCertificate = "/srv/ssl/cert.pem",
vServSSLCertificateKey = "/srv/ssl/cert.key"
```

#### Modifications in Nginx

```haskell
sListen = Just "443",
sKeepaliveTimeout = Just "75",
sSSLCertificate = Just "/srv/ssl/cert.pem",
sSSLCertificateKey = Just "/srv/ssl/cert.key"
```

```haskell
Values before adaptation
vListen = "80",
vServSSL = "off",
vServSSLCertificate = "",
vServSSLCertificateKey = ""
```

```haskell
Values after adaptation
vListen = "443",
vServKeepaliveTimeout = "75",
vServSSL = "on",
vServSSLCertificate = "/srv/ssl/cert.pem",
vServSSLCertificateKey = "/srv/ssl/cert.key"
```

#### Modifications in Apache

```haskell
sListen = Just "443",
sVirtualHostAddress = Just ":443",
sKeepaliveTimeOut = Just "75",
sServerName = Just "example.com",
sSSLCertificate = Just "/srv/ssl/cert.pem",
sSSLCertificateKey = Just "/srv/ssl/cert.key"
```

Here, the \texttt{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 \texttt{vServSSL} to "on" activates SSL and the next two items provide the location of the SSL certificate and its key. The modification of the \texttt{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 \textit{put} transformations generated by our bidirectional programs. The following are samples of the updated sources:

```haskell
apacheSource :: ApacheWebserver

apacheSource :: ApacheWebserver {
  sDocumentRoot = Just "/var/www/html",
  sKeepalive = Just "On",
  sMaxKeepAliveRequests = Just "100",
  sListen = Just ["80"],
  aDirectoryIndex = Nothing,
  aSSLCertificateFile = Nothing,
  aVirtualHosts = Just [VirtualHost {
    sServerName = Just "example.com",
    sLocation = Nothing,
    sSSLCertificateKeyFile = Nothing,
    sSSLCertificate = Just "/srv/ssl/cert.pem",
    sSSLEngine = Just "On",
    sVirtualHostAddress = Just ":443",
  }]
}
```

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

```haskell
nginxSource :: NginxWebserver
nginxSource = NginxWebserver { 
nWorkerProcesses = Just "4",
nHttp = Just Http { 
hKeepaliveTimeout = Just "65",
hKeepaliveRequests = Just "100",
hRoot = Nothing,
hServer = Just [ 
  Server { 
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 } , 
},
},
},
}
```

Listing 5: Simplified view

```haskell
reducedView :: CommonWebserver
reducedView = CommonWebserver { 
vRoot = "html",
vKeepaliveTimeout = "65",
vSsl = "off",
vSslCertificate = ",
vSslCertificateKey = ",
vServers = [
  VServer { 
    vListen = ["80"],
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. Similarly, 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 RE-FRACT, 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 (EU-REMA), 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 [7]. 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 Transformation 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


