Chapter 4
Bidirectional Transformations for Self-Adaptive Systems

Lionel Montrieux, Naoyasu Ubayashi, Tianqi Zhao, Zhi Jin, and Zhenjiang Hu

Abstract Bidirectional transformations are a synchronisation mechanism between documents, a source, and a view. A bidirectional transformation is a pair of functions, one that extracts a view from a source and the other that updates a source according to changes made to the view. Bidirectional programming is a recent technique that helps developers to easily write bidirectional transformations and ensure that they satisfy properties of interest. In this chapter, we argue that bidirectional transformations and bidirectional programming are useful techniques in the context of self-adaptive systems. We present four applications of bidirectional transformation for construction of adaptive systems: abstraction, separation of concerns, rule-based adaptation, and uncertainty-aware programming.

4.1 Introduction

Bidirectional transformations [6, 11, 18] have been the focus of a lot of attention lately, both in the programming language community [2, 10, 12, 16, 26] and in the software engineering community [28, 29]. They are a recent way of solving the old view-update problem, defined decades ago in the database community. As bidirectional programming languages are growing more mature, they are getting easier to use for software engineers and more efficient and more reliable. Perhaps the strongest argument in favour of bidirectional programming is its ability to provide a synchronisation mechanism between a source and a view that is guaranteed to be correct *by construction*.

L. Montrieux (✉) · Z. Hu
National Institute of Informatics, Tokyo, Japan
e-mail: lionel.montrieux@zalando.de

N. Ubayashi
Kyushu University, Fukuoka, Japan

T. Zhao · Z. Jin
Peking University, Beijing, China

© Springer Nature Singapore Pte Ltd. 2019

Y. Yu et al. (eds.), *Engineering Adaptive Software Systems*,
https://doi.org/10.1007/978-981-13-2185-6_4

In this chapter, we present four different ways in which bidirectional programming and bidirectional transformations can improve the state of the art in engineering self-adaptive systems. In particular, we focus on self-adaptive systems developed around the MAPE-K feedback loop model [19].

First, bidirectional programming can be used to synchronise concrete and abstract models in the knowledge base, allowing developers to write their adaptation layer independently of the implementation of the target system. This facilitates the reuse of MAPE-K components and allows developers to easily swap the implementation of the target system, without having to rewrite the adaptation layer.

Second, bidirectional programming is useful to achieve separation of concerns, and hence reuse of components, in the adaptation layer. By extracting from the knowledge base small models that are tailored to a particular aspect being analysed, bidirectional transformations simplify the development of small, focused analysis and planning components and may even improve performance, due to the reduced size of the models to consider.

Third, bidirectional programming is used in the context of view-based adaptation rules for the analysis of the target system and its environment. Bidirectional programming comes as a natural approach to implement the νRule approach, where adaptation rules are applied depending on the state of the environment, captured in views.

Fourth, bidirectional programming is applied in the field of uncertainty-aware software development, a software development approach that makes uncertainty a first-class citizen. Bidirectional transformations can extract partial models from code and uncertainty-aware artefacts and reflect changes made to the partial models, back to the code.

The rest of this chapter is organised as follows: In Sect. 4.2, we introduce bidirectional transformations and bidirectional programming. In Sect. 4.3, we discuss how bidirectional programming can synchronise concrete and abstract models, and in Sect. 4.4, we focus on separation of concerns. Section 4.5 discusses the use of bidirectional programming with νRule -based adaptation. Then, Sect. 4.6 considers the role of bidirectional programming in the context of uncertainty-aware development. Section 4.7 concludes this chapter.

4.2 Bidirectional Programming

A bidirectional transformation is a pair of functions, a forward transformation `get` and a backward transformation `put`, used to synchronise two documents [11]. The forward transformation takes a source as input and produces a view; the backward transformation takes a source and a view as inputs and uses the view to update the source, producing an updated source.

In this paper, we will use Haskell, a functional language, to specify bidirectional transformation. One big reason for us to choose Haskell is that a set of bidirectional languages (libraries) have been developed in Haskell.

In Haskell, the types for `get` and `put` are the following for a source of type `Source` and a view of type `View`:

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

For example, the following code defines a bidirectional transformation between a list of integers (the source) and a single element (the view).

```
1 get :: [Int] -> Int
2 get (x:xs) = x
3
4 put :: [Int] -> Int -> [Int]
5 put (x:xs) y = y:xs
```

The `get` function extracts the head of the list, while the `put` function updates the head of the source list with the value in the view, as illustrated by the following example:

```
> get [1,2,3]
1

> put [1,2,3] 9
[9,2,3]
```

A particularly interesting class of bidirectional transformations are *well-behaved* bidirectional transformations [11, 12]. Intuitively, a well-behaved bidirectional transformation provides a "correct" synchronisation between source and view. More formally, a bidirectional transformation is well behaved if it satisfies two properties: `GetPut` and `PutGet`.

`GetPut` is the identity law. If a view is extracted from a source and used and unchanged to update the source, the source should not change:

```
put s (get s) = s
```

`PutGet` is the change conservation law. If a view has been updated, then using it to update the source and then extracting a view from that updated source should produce the same updated view:

```
get (put s v) = v
```

The example we used above is a well-behaved bidirectional transformation. Using the same source as above, both laws of well-behaved bidirectional transformations are satisfied:

```
> put [1,2,3] (get [1,2,3]) = put [1,2,3] 1 = [1,2,3]

> get (put [1,2,3] 99) = get [99,2,3] = 99
```

Our example is trivial, but it can be very difficult to write a complex bidirectional transformation, let alone proving it well behaved. Still, bidirectional transformations

have been used in a variety of applications [6], including spreadsheets [5], graph transformations [16], and many more.

Bidirectional programming languages are domain-specific languages that aim to simplify the development of bidirectional transformations [11]. Developers write one direction of the transformation, and the bidirectional programming language's compiler automatically derives the other direction, to form a well-behaved bidirectional transformation, if it exists.

We can classify these bidirectional programming languages in two categories: *get-based* languages and *put-based* (also called *putback-based*) languages.

Get-based languages let developers write a `get` function and automatically derive a `put` function, forming a well-behaved bidirectional transformation. GroundTRam [17] is one of these languages. The advantage of get-based languages is that the `get` function is relatively easy to write. The inconvenience is that for a given `get` function, there may be many `put` functions that form a well-behaved bidirectional transformation. For example, the following code snippet shows three different `put` functions for a single get. They all form well-behaved bidirectional transformations:

```
1  get :: [Int] -> Int
2  get (x:xs) = x
3
4  put1 :: [Int] -> Int -> [Int]
5  put1 (x:xs) y = y:xs
6
7  put2 :: [Int] -> Int -> [Int]
8  put2 (x:xs) y = if x==y then y:xs else y:[]
9
10 put3 :: [Int] -> Int -> [Int]
11 put3 x y = if x>=y then y:xs else y:[]
```

Put-based languages, on the other hand, let developers write a `put` function and automatically derive a `get` function to form a well-behaved bidirectional transformation. While the `put` function is often more difficult to write than the `get` function, we know that for a given `put`, there exists *at most one* `get` that forms a well-behaved bidirectional transformation [9]. Hence, put-based languages give developers more control over their bidirectional transformations. Examples of put-based languages include BiFluX [25], BiGUL [21], or Brul [30].

Bidirectional programming, like bidirectional transformations, has been applied to a variety of areas in software engineering, such as access control [13, 24], model-code synchronisation [29], or self-adaptive systems [4].

4.3 Bidirectional Programming and Abstraction

In self-adaptive systems built around the MAPE-K loop [19], the adaptation layer monitors the target system and its environment, analyses data, plans changes, and executes changes on the target system. Often, the self-adaptive layer is decoupled from the target system. The monitoring of the system is done through probes and gauges and the execution through effectors. This decoupling makes it easy to reuse the same self-adaptive layer for multiple target systems. In this section, we show how bidirectional programming can facilitate the reuse of a self-adaptive layer for multiple target systems *independently* of the implementation of the target system [4]. In particular, we focus on the adaptation of configuration files, which is a common way of controlling the behaviour of the target system.

Configuration files are trees. They contain key-value pairs, each used to configure a particular aspect of the software. In some configuration files, these pairs can be placed inside blocks, and blocks can contain other blocks. Depending on the configuration file, the order of pairs and/or blocks may matter. Keys that do not appear in the configuration file are given a default value when parsed by the software. The use of blocks also allows for context overriding, where the value of a key in a block takes, for that block, precedence over other values of the same key defined in ancestor blocks or over the default value.

An example of software that uses configuration files is a web server. There are many implementations of web servers, such as Apache, Nginx, or Microsoft IIS. Each implementation defines its own configuration format, syntax, and semantics. Yet there are a lot of similarities between each implementation's configuration files. After all, they all describe the behaviour of web servers.

Using bidirectional programming, it is possible to synchronise a concrete configuration file with an abstract web server configuration that is independent of the implementation used. For each implementation, a bidirectional transformation synchronises the concrete configuration file (the source of the transformation) with the abstract configuration (the view), as depicted on Fig. 4.1. Since the transformations

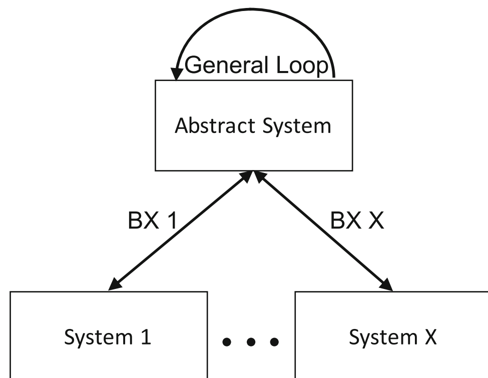


Fig. 4.1 Abstraction with bidirectional programs

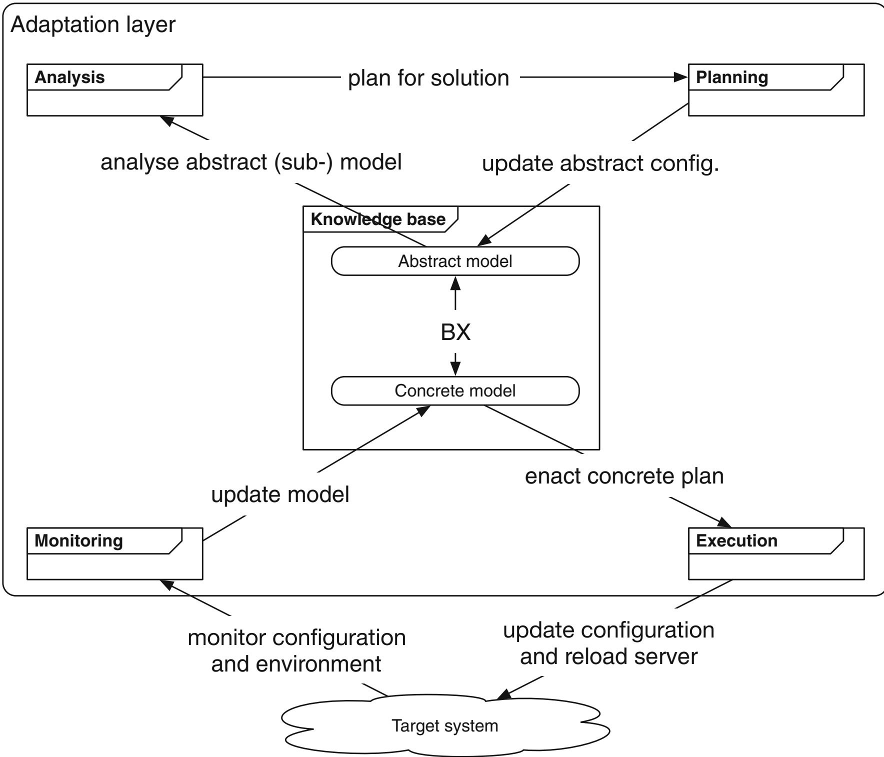


Fig. 4.2 MAPE-K loop over abstract configuration

for all implementations use the same type for the view, it is then possible to reuse the abstract implementation when implementing the MAPE-K loop, as illustrated on Fig. 4.2. The monitoring stage keeps track of the environment and of changes in the software's concrete configuration. Changes in the configuration trigger a new `get` transformation that updates the abstract configuration. Both the analysis and the planning stages use the abstract configuration and are therefore reusable across any implementation. The planning stage can directly modify the abstract configuration to enact adaptation. In the execution phase, a `put` transformation synchronises the abstract configuration with the concrete configuration, before the configuration file is transferred to the target system, and the target system restarted or reloaded, if necessary.

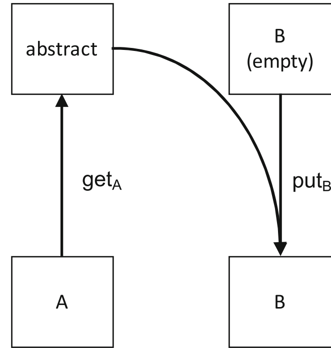


Fig. 4.3 Migration with bidirectional programs

4.3.1 Migration

An additional benefit of using bidirectional transformations to synchronise concrete and abstract configurations is the possibility to migrate the target system from one implementation to another while conserving the same configuration. Figure 4.3 illustrates such a scenario. Let A and B be two implementations of the target system, each with its own configuration format. Two bidirectional programs are written to synchronise the concrete configurations with a common abstract configuration. To migrate the target system from implementation A to implementation B, an abstract configuration is first extracted from the concrete configuration of A, using the `get` transformation provided by the bidirectional program for A. Then, the abstract configuration is used, together with an empty template configuration for B, to produce, through the `put` transformation provided by the bidirectional program for B, a concrete configuration for B that captures the same behaviour as the concrete configuration for A.

4.4 Bidirectional Programming and Separation of Concerns

In the previous section, we discussed how to apply bidirectional programming to easily develop synchronisation between abstract and concrete models in the knowledge base. This synchronisation, correct by construction, allows for the reused of parts of the adaptation layer, regardless of the implementation of the target system. This is not the only way in which bidirectional programming can be helpful in the knowledge base. It can be also used to facilitate separation of concerns in adaptation layers that adapt a target system according to multiple concerns.

For example, the adaptation of a web application deployed on an IaaS cloud service could take into account the usage of the cloud instances, security concerns, and service availability requirements. While it is possible to consider these concerns together as a multi-objective optimisation problem, we argue that considering them separately has benefits and that bidirectional programming helps to develop such systems.

4.4.1 Extracting Submodels

In self-adaptive systems, the analysis of a particular concern may only require a subset of the entire model(s)[1] in the knowledge base. Using a bidirectional program, it is possible to extract the exact model subset necessary to perform adaptation according to a given concern and to keep it synchronised with the whole model. In this situation, the complete model is the source, while a submodel for a given concern is a view. The submodel can be analysed and then passed on to the planning phase, which can directly modify the submodel. A `put` transformation will ensure that the source model is updated accordingly. Figure 4.4 illustrates this process.

There are several advantages in extracting a submodel for each concern:

- The phases using the submodel do not need to change if unrelated changes happen in the complete model. It is therefore easier to implement reusable analysis and planning phases;
- If the analysis and/or planning phases use techniques whose performance degrades with large inputs, such as model checking, a comparatively small submodel can speed up the adaptation;
- Once views are extracted, it is easy to parallelise the execution of the analysis and planning phases of each concern.

One inconvenient of this approach is that two separate concerns may cause conflicting modifications of the source model. When evaluating the entire source at once, mitigation strategies could easily be employed. Our approach would make this more complicated. Ordering can be used to favour the more important concerns

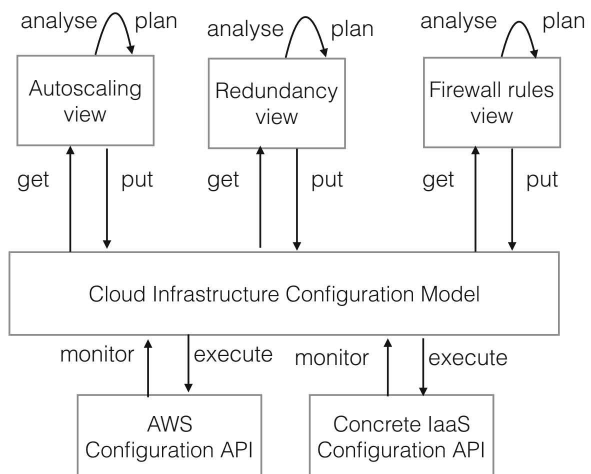


Fig. 4.4 Extracting a submodel

[1]Since multiple models can easily be merged into a single one using a virtual root element, we assume from now on that the knowledge base contains only one model.

over the less important ones. In the case of several, equally important and potentially conflicting concerns, separation may cause difficulties. However, it may still be possible to group these concerns into a single view and perform the analysis for these concerns simultaneously on a model that is still smaller than the source.

Using bidirectional programs provides guarantees, by construction, of the correctness of the synchronisation mechanism between the source and each of the views. In our approach, only the analysis and planning phases use a view. The monitoring phase directly updates the source, and the execution phase uses data from the source, after it has been modified by all the planning phases. In the spirit of Weyns et al.'s MAPE-K patterns for distributed systems [27], our solution follows a M(AP) + E pattern.

4.4.2 Current vs. Desired State of the Model

In addition to using views to achieve separation of concerns, bidirectional programming offers a solution to the issue of the model's state. The solution is to use two views, one for the current state of the model and one for the desired state of the model. Both views are derived from the same source. Components can use the most appropriate view.

In the source, a new status field is added to each element in the model. The status indicates whether the element should not change (`0`), be created (`1`), or be deleted (`2`) from the current model. Modifications to an element are treated as both a deletion and a creation. The `put` transformation for the current model always sets that value to `0` in the source. However, the `put` transformation for the desired model sets the status to `1` for all elements created in the source and sets it to `2` for all deletions, without actually deleting the element from the source. The `get` transformations derived are simple: for the current model, it selects all elements whose status is `0` or `2`; for the desired model, it selects all elements whose status is `0` or `1`.

4.4.3 Evaluation Order and Concurrent Evaluation

The concurrent evaluation of multiple views can cause consistency issues. Let two views, V_0^1 and V_0^2, extracted from the same version of the source S_0 (i.e. there has been no `put` transformation run between the extraction of the two sources), using `get` transformations BX^1 and BX^2, respectively. The respective analysis and planning phases for V_0^1 and V_0^2 could make any changes to the views, resulting in V_1^1 and V_1^2, respectively. Propagating these changes to the source must be done sequentially, as languages like BiGUL do not support the simultaneous update of a source using multiple views. We assume that the transformation using V_1^1 is performed first, but the argument is valid the other way around. Using BX^1, a `put` transformation updates S_0, which becomes S_1. If a `get` transformation with BX^2

still produces V_0^2, then the other view has not been affected, and the second put transformation can be performed. Otherwise, the second view is based on outdated data, and it is possible that, should the analysis and planning be run again, another result would be reached.

There are several ways to solve this issue. Perhaps the simplest approach is to define a partial order between the different concerns and deal with them sequentially according to the ordering (i.e. for each concern, start with get and then analyse and plan and finish with put). Another strategy is, at design time, to manually inspect the views produced and run those views that are produced from entirely distinct subsets of the source in parallel, as they are completely separate. Finally, a more sophisticated approach would be to deal with several concerns in parallel, keeping track of the view produced by each get transformation (i.e. before it is modified). After every put, all get transformations would be run again. If they produce the same view as the previous get transformation, then the view has not been affected by the changes introduced by put, and the updated view is still valid. Otherwise, the analysis and planning must be run again. In the worst case, this will be equivalent to the sequential scenario, with the addition of the comparison between different versions of the views.

4.5 Declarative Description of Adaptation Logic

Adaption logic plays an important role in self-adaptive systems, specifying when and how to update the behaviour and/or structure of the system in response to changes of the environment and the system itself. So far, it has been implemented mostly by hardwired code for analysing and planning in the MAPE-K loop. In this section, we show that adaptation logic can be declaratively specified in putback-based bidirectional languages such as BiFluX and BiGUL, which would allow developers to utilise bidirectional programming to systematically construct robust self-adaptive systems.

4.5.1 Adding Views to Adaptation Rules

Rule-based adaptation [1, 7, 22] provides a powerful mechanism to develop self-adaptive systems, enabling systems to modify their behaviour, reconfigure their structure, and evolve over time, reacting to changes in the operating context [3]. In a rule-based adaptation system, a set of adaptation rules are used to specify adaptation logic of what particular action should be performed in reaction to monitored events.

Typically, an adaptation rule takes the form of *condition* $\Rightarrow$ *action* where *condition* specifies the trigger of the rule, which is often fired as a result of a set of monitoring operations, and *action* specifies an operation sequence to perform

in response to the trigger. For instance, in a smart room system, we may have the following rule:

$$\begin{aligned} &\mathit{Light.Power} = \mathit{off} \ \wedge \ \mathit{Time} = \mathit{daytime} \\ &\quad \Rightarrow \mathit{Blind.State} := \mathit{open}; \ \mathit{Window.State} := \mathit{open} \end{aligned}$$

which declares that if the light is powered off in daytime, then the blind and the window must be opened. Rule-based adaptation has advantages of readability and elegance of each individual rule, the efficiency of plan process and the ease of rule modification.

In spite of these advantages, adaptation rules pay attention only to local transformations. However, such local transformations can be structured to ease the satisfaction of the system's global goals. In many cases, some environment features may hint at different system goals, and different system goals imply different adaptation policies.

We introduce another concern, *situation*, for capturing such a hint [20]. With this concern, the adaptation logics are of two layers. The first layer intends to capture the requirement changes when the situation changes. The second layer intends to capture, for a certain system goal, the situation changes or changes in some of the entities in the environment that require actions to continue satisfying the system goal.

Then, to enable the dynamic decision on the system adaptation, three types of νRules can be identified:

- situation $\rightarrow$ goal setting: It captures the phenomena that the user may have different desires in different situations.
- goal setting: situation $\rightarrow$ behaviour pattern: This means that, given a goal setting, the system should behave according to different behaviour patterns when situated in different situations. A behaviour pattern consists of a set of environment features.
- goal setting: environment features $\rightarrow$ system features: This means that, given a goal setting, some of the system features need to be enabled by current emerged environment features to better satisfice the goal setting.

The first type of rules is meaningful in decision-making about adaptation. For example, *everybody is sleeping* is a situation that represents "everybody has been in bed". When in this situation, the system normally switches to the goal setting of the "sleeping mode" without taking into account other environment features.

When a goal setting needs to be continuously satisfied, different situations may also indicate different system behaviour modes. This is represented by the second type of rules. In fact, situation is a concept that has received much attention from philosophers and logicians. The earlier formal notion of situation was introduced as a means to give a more realistic formal semantics for speech acts than what was then available. In contrast with a *world* which determines the value of every proposition, a situation corresponds to the limited parts of reality we perceive, reason about, and live in. With limited information, a situation can provide answers to some, but

not all, questions about the world. The advantage of including situation is then to decrease the sensing cost as, normally, only parts of the environmental setting need to be detected when making a decision in a particular situation. This is important when the number of environment entities is large. The other advantage could be fitting to the human recognition. In many cases, only a few features are required to identify a situation, while others are less important.

4.5.2 *νRule: View-Based Adaptation Rule*

We assume that the environment states, the goal settings and the system behaviours are represented by feature bindings and propose to structure adaptation rules into νRules. There are two types of rules. The first type of νRule is the behaviour rule. It is made of three parts: an observable view (v) that could be a goal configuration, a conjunction of conditions (C) that could be a situation of the environment (a group of significant environment features that indicates the situation) or a set of environment features (that does not indicate a situation but captures the environment states) and a sequence of actions (A). The second type of view-deciding rule is made of two parts: an observable situation of the environment (C) and a system goal configuration (A). For unification, we assume the view part of the second type of rule is $true$.

The concrete syntax of νRule is shown in Fig. 4.5.

A νRule

$$v \vdash C \Rightarrow A$$

can be read as "if v holds, action A should be taken under condition C and preserve state v". The view v and the condition C are defined over feature bindings, where a feature is a goal setting or an environment attributes or a system component. A feature binding fb has two alternatives: *feature* can be either bound with a *value* or a *value interval*. For example, $Light.Intensity = 2$' means the intensity of the light is 2, $Light.Intensity = (1, 3]$ equals to $1 < Light.Intensity \leq 3$.

A is a set of asked system component settings a, and each setting a binds a constant *value* to a *feature*. For example, in the following νRule

Fig. 4.5 Syntax of νRule

view-based rule	ν*Rule*	$::= v \vdash C \Rightarrow A$
view	v	$::= fb$
feature binding	fb	$::= feature = value$
		$\mid\ feature = value\ interval$
conditions	C	$::= c_1 \wedge c_2 \wedge \ldots \wedge c_m$
condition	c	$::= fb$
action sequence	A	$::= a_1; a_2; \ldots ; a_n$
action	a	$::= feature := value$

$$(r_1)\ Light.Power = off$$
$$\vdash Time = daytime\ \wedge Blind.State = close$$
$$\Rightarrow Blind.State := open;\ \ Window.State := open$$

r_1 declares that when the current goal is to keep the light powered off, if it is in daytime and the blind is closed, then the system components, the blind, and the window will be opened.

Generally, the νRule implies that the *view* needs to be kept after adaptation. That is the reason of calling the rule the *view-based* adaptation rule. *The key is the use of the idea of "view" in the rule specification. Rather than showing how to propagate changes (out), the* view-based *rules specify how a view can be kept through changes of necessary system components for responding the environment changes.*

In the condition of a νRule, we do not support the *or* operation. This does not weaken the expressiveness of νRules, as a νRule with a c_1 *or* c_2 condition is equivalent to two νRules, with conditions c_1 and c_2, respectively.

4.5.3 Implementation of νRule in BiGUL

The proposed νRule s can be implemented in BiGUL. The implementation of νRule by BiGUL includes two parts: (1) representation of the view and the source models and (2) translation of νRule s as BiGUL updates.

4.5.3.1 Representation

For a specific νRule, the view model only specifies the binding state of one feature, and the source model includes the binding states of all features monitored from the environment and system. Therefore, we represent the view as a feature binding, which is a tuple consisting of the feature name and its value, and represent the source as a set of feature bindings. Figure 4.6 gives an example of the source and the view for νRule $r1$, in the sense that the view is a projection of the source, where only the value of feature "Light.power" is considered.

4.5.3.2 Translation

With the source and the view represented, a νRule can be translated into updates in BiGUL. We describe the translation from νRule to BiGUL by using the νRule $r1$ as an example. The translated BiGUL program is as follows (Fig. 4.7):

This program means updating the source using the view: if the view feature "Light.power" takes the value "off", then the source feature model will be updated with the value of feature "Blind.state" set to "on", and the value of feature "Window.state" set to "open".

```
s :: [(String, String)]
s = [
      ("aircon_power","off"), ("light_power","off"), ("light_intensity","off"),
      ("curtain","open"), ("window_state","open"), ("setpoint","no"),
      ("somebody_home","true"), ("family_sleep","true"), ("weather","sunny"),
      ("time","daytime"), ("brightness","10"), ("temperatue","10"),
      ("room_temperatue","10"), ("room_brightness","10"), ("blind_state","open"),
      ("cdplayer_power","on"), ("light_style","gentle")

    ]

v :: [(String, String)]
v = [
      ("aircon_power","on"), ("light_power","on")
    ]
```

Fig. 4.6 An example of the source and the view

```
rule1 = Case [
          $(adaptive [| \s v -> v == ("Light_power","off") &&
                                s !! ("Time","daytime") &&
                                s !! ("Blind_state", "close") &&
                \s v -> set' s [("Blind_state","on"), ("Window_state", "open")]],
          $(normal [| \s v -> True |]) $
                rep_i (list_id "Blind_state")
           ]
```

Fig. 4.7 The BiGUL program of νRule r1

Fig. 4.8 Consistency check

```
checkEqual r1 r2 s v =
  case (put2rules [r1, r2] s v, put2rules [r2, r1] s v)
    (Right leftS, Right rightS) ->
      if leftS == rightS then True
      else False
```

A νRule is regarded as valid when it satisfies the view preservation property: if the *view* holds, it should still hold after the execution of the action. Implementing a νRule as a BiGUL program can facilitate the check of its validity. A BiGUL program is guaranteed to produce a well-behaved bidirectional transformation if one exists, i.e. a BiGUL program satisfies the `GetPut` and `PutGet` laws (see Sect. 4.2). Therefore, a successfully compiled BiGUL program is guaranteed to be view preserved, and in this case, the corresponding νRule is guaranteed to be a view-preserved rule.

While a νRule is valid when it is view preserved, a νRule set is regarded as well behaved only if (1) each νRule in this set is valid and (2) every two rules in this set are order independent. While the validity of a single νRule can be automatically checked, the order independence between two rules can be checked through the "`checkEqual`" function in Fig. 4.8. For two νRule s $r1$ and $r2$, if executing $r2$ after $r1$ and executing $r1$ after $r2$ lead to the exactly same adaptation results, $r1$ and $r2$ will be considered as order independent.

4.6 Bidirectional Transformations for Uncertainty-Aware Software Development

4.6.1 Uncertainty in Software Development

Recently, uncertainty has attracted a growing interest among researchers. Research themes spread over uncertainty of goal modelling, UML modelling, model transformations, and testing. Garlan D. argues that software systems such as self-adaptive systems must embrace uncertainty within the engineering discipline of software engineering [15]. As a representative work, a method for expressing uncertainty using a partial model is proposed in [8]. A partial model can represent a specific type of uncertainty in which there are uncertain issues known and shared among the stakeholders including developers and customers. For example, there are alternative user requirements although it is uncertain which alternative should be selected. A partial model is a single model containing all possible alternative designs of a system and is encoded in propositional logic. We can check whether or not a model satisfies some interesting properties even if there are uncertain concerns.

4.6.2 Modular Programming for Uncertainty

Modularity is one of the important principles in software engineering. Unfortunately, the state-of-the-art module mechanisms do not regard an uncertain concern as a first-class software module. If uncertainty can be dealt with modularly, we can add or delete uncertain concerns to/from code whenever these concerns arise or are fixed to certain concerns.

To deal with this problem, a new programming style supporting *modularity for uncertainty* is proposed in [14]. This approach consists of three key ideas: (1) a pluggable interface for describing uncertainty, (2) interface-based modular reasoning for uncertainty, and (3) management support for tracing when and why uncertain concerns arise or are resolved. This interface called *Archface-U*, which supports *component-and-connector* architecture, consists of two kinds of interfaces, *component* and *connector*.

Figure 4.9 (Printer-scanner system), a well-known parallel system that falls into a deadlock [23], is an example of *Archface-U* descriptions. Two processes P and Q acquire the lock from each of the shared resources, the printer and the scanner, and then release the locks. The symbols {} and [] represent *alternative* and *optional*, respectively. A component is the same with ordinary Java interface. A connector, which is specified using the notation similar to FSP (finite state processes), defines the message interactions among components. FSP is based on process algebra and generates finite LTS (labelled transition systems). An arrow in FSP indicates a sequence of actions. For example, `GET` (List 1, line 22) shows that the action `scanner.get` is executed after the action `printer.get` is executed.

```
[List 1]
01: interface component cPrinter {
02:   public void get();
03:   public void put();
04:   public void print();
05:  [public void utility();]
06: }
07:
08: interface component cScanner {
09:   public void get();
10:   public void put();
11:   public void scan();
12:  [public void utility();]
13: }
14:
15: interface component cCopyMachine {
16:   public void copy();
17: }
```

```
18: interface connector cSystem (
19:   cCopyMachine P, cCopyMachine Q,
20:   cPrinter printer, cScanner scanner) {
21:
22:   GET  = (printer.get -> scanner.get);
23:   PUT  = (printer.put -> scanner.put);
24:   COPY = (scanner.scan -> printer.print);
25:
26:   P.copy = (GET -> COPY -> PUT -> P.copy);
27:   Q.copy = (GET -> COPY -> PUT -> Q.copy);
28: }
```

```
[List 2]
01: interface connector uSystem
02:   extends cSystem (
03:   cPrinter printer, cScanner scanner) {
04:
05:   GET = ({printer.get -> scanner.get,
06:           scanner.get -> printer.get});
07: }
```

Fig. 4.9 Archface-U description (Printer-scanner system)

In *Archface-U*, uncertain concerns are defined as a subinterface as shown in List 2. By extending the existing interface, we can introduce uncertainty modularly. In List 2, it is uncertain how to acquire printer and scanner resources in two processes, P and Q.

As shown in Fig. 4.9, we can explicitly represent uncertainty using *alternative* and *optional* language constructs. If a developer is writing a program and he or she becomes aware of the existence of uncertainty, the developer only has to modify *Archface-U* as shown in List 2. The developer does not have to modify the original code, because the essential information containing uncertain concerns is expressed in the *Archface-U* and the behavioural properties can be checked using only this information as explained below. If an uncertain concern is fixed to certain, a developer only has to delete the corresponding inheritance (List 2) and modify the original *Archface-U* (List 1) if needed.

4.6.3 *Modular Reasoning Based on Partial Model*

We can use the verification power provided by a partial model as illustrated in Fig. 4.10. A partial model is generated from *Archface-U* definitions including uncertainty represented by *alternative* and *optional*. Uncertainty is a target of compilation. The type checker verifies whether code is a subset of the partial model. From the theoretical aspect, type checking is passed when each code is a refinement of *Archface-U*. Our compiler is based on the refinement calculus focusing on simulation.

Behavioural properties represented by LTL (linear temporal logic) can be automatically verified using model checkers such as LTSA (LTS analyser) supporting FSP. If a property is verified by a model checker and the type check is successfully passed, the program satisfies important properties such as deadlock free. We show a verification process in details. In case of the printer-scanner system, there are four

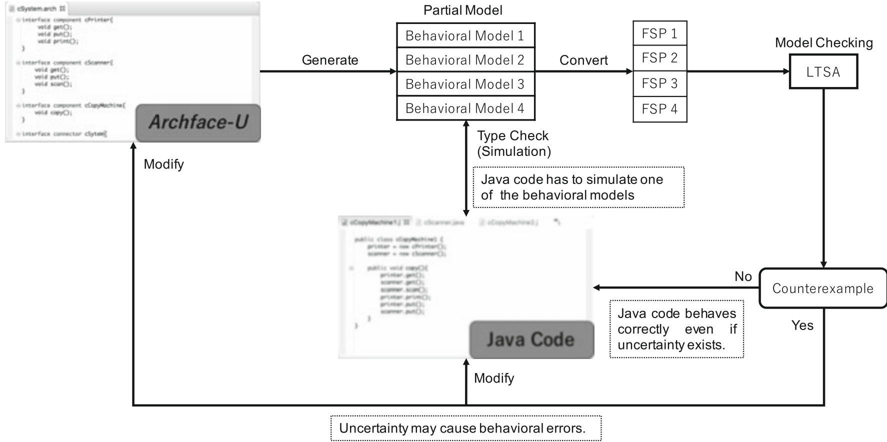
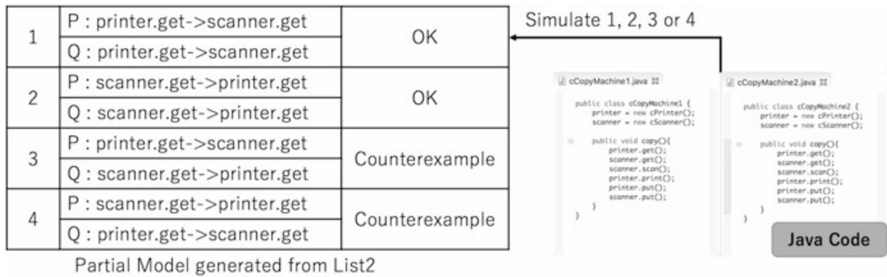


Fig. 4.10 Uncertainty-aware modular reasoning



1	P : printer.get->scanner.get	OK
	Q : printer.get->scanner.get	
2	P : scanner.get->printer.get	OK
	Q : scanner.get->printer.get	
3	P : printer.get->scanner.get	Counterexample
	Q : scanner.get->printer.get	
4	P : scanner.get->printer.get	Counterexample
	Q : printer.get->scanner.get	

Fig. 4.11 Partial model and Java program

possible resource acquisition sequences. Type check is passed if Java code simulates one of these sequences. In Fig. 4.11, Java code simulates the sequence 1, and the type check is passed. If counterexamples are not generated by a model checker, we can select any sequence (either of 1, 2, 3, or 4 is okay). We can proceed the development even if uncertain concerns exist, because the code simulating sequence 1 is correct. Unfortunately, counterexamples are generated in case of Fig. 4.11, and these counterexamples show that the acquisition order must be the same. In this case, uncertainty may cause a deadlock although the Java code in Fig. 4.11 is correct. A developer can confirm whether or not he or she can embrace this uncertainty before modifying the code. In this case, the developer should not modify the code. As another situation, assume that a developer makes the code simulating the sequence 3 or 4. Although the type check is passed, the code is not correct because counterexamples are generated. In this case, a developer has to change the code to a new version simulating the sequence 1 or 2. In this case, a developer can resolve uncertain concerns and make a correct program before debugging and testing.

State explosion is a crucial problem when applying model checking to source code. In our approach, model checking is performed in terms of only FSP descriptions in *Archface-U*. Code is not the direct target of model checking. As a result, the number of states is reduced. Nevertheless, code can be indirectly verified by the model checker if the code conforms to its *Archface-U* via type checker. Our approach mitigates the problem of state explosion by integrating type checking with model checking.

4.6.4 Bidirectional Transformation for Uncertainty

Our approach can be regarded as an application of a bidirectional transformation. *Get* uses *Archface-U* and code to produce a partial model as a view. *Put* uses *Archface-U*, code, and a partial model to reflect changes made to the partial model into the code.

4.7 Conclusion

In this chapter, we have introduced bidirectional transformation, as well as bidirectional programming. We have shown how bidirectional programming is a technique that can be applied to various aspects of the engineering of self-adaptive systems. We targeted four areas in particular: abstraction, separation of concerns, νRule - based adaptation, and uncertainty-aware software development.

References

1. Acher, M., Collet, P., Fleurey, F., Lahire, P., Moisan, S., Rigault, J.P., et al.: Modeling context and dynamic adaptations with feature models. In: Proceedings of the 4th International Workshop on Models@run.time, Denver (2009)
2. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’08, San Francisco, pp. 407–419. ACM, New York (2008)
3. Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software engineering for self-adaptive systems. In: Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26. Springer, Berlin/Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_1

4. Colson, K., Dupuis, R., Montrieux, L., Hu, Z., Uchitel, S., Schobbens, P.Y.: Reusable self-adaptation through bidirectional programming. In: SEAMS'16: 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. ACM, Austin (2016)
5. Cunha, J., Fernandes, J.P., Mendes, J., Pacheco, H., Saraiva, J.: Bidirectional transformation of model-driven spreadsheets. In: Hu, Z., de Lara, J. (eds.) Theory and Practice of Model Transformations. Lecture Notes in Computer Science, no. 7307, pp. 105–120. Springer, Berlin/Heidelberg (2012)
6. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: a cross-discipline perspective. In: Paige, R.F. (ed.) Theory and Practice of Model Transformations. Lecture Notes in Computer Science, no. 5563, pp. 260–283. Springer, Berlin/Heidelberg (2009)
7. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: A language for specifying security and management policies for distributed systems. Department of Computing, Imperial College, Technical Report, London (2000)
8. Famelis, M., Salay, R., Chechik, M.: Partial models: towards modeling and reasoning with uncertainty. In: 2012 34th International Conference on Software Engineering (ICSE), Zurich, pp. 573–583 (2012)
9. Fischer, S., Hu, Z., Pacheco, H.: "Putback" is the Essence of Bidirectional Programming. Technical Report GRACE-TR 2012-08, National Institute of Informatics (2012)
10. Fischer, S., Hu, Z., Pacheco, H.: The essence of bidirectional programming. Sci. China Inf. Sci. **58**(5), 1–21 (2015)
11. Foster, J.N.: Bidirectional programming languages. Ph.D. thesis, University of Pensylvania (2009)
12. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3), 17 (2007)
13. Foster, J., Pierce, B., Zdancewic, S.: Updatable security views. In: 22nd IEEE Computer Security Foundations Symposium, CSF'09, Port Jefferson, pp. 60–74 (2009)
14. Fukamachi, T., Ubayashi, N., Hosoai, S., Kamei, Y.: Conquering uncertainty in Java programming. In: Proceedings of the 37th International Conference on Software Engineering – ICSE'15, Florence, vol. 2, pp. 823–824. IEEE Press, Piscataway (2015)
15. Garlan, D.: Software engineering in an uncertain world. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER'10, Santa Fe, pp. 125–128. ACM, New York (2010)
16. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K.: Bidirectionalizing Graph Transformations. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP'10, Baltimore, pp. 205–216. ACM, New York (2010)
17. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Nakano, K.: GRoundTram: an integrated framework for developing well-behaved bidirectional model transformations. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lawrence, pp. 480–483 (2011)
18. Hu, Z., Schürr, A., Stevens, P., Terwilliger, J.F.: Dagstuhl seminar on bidirectional transformations (BX). SIGMOD Rec. **40**(1), 35–39 (2011)
19. IBM Corp.: An architectural blueprint for autonomic computing. Technical report, 3rd edn. (2005)
20. Jin, Z.: Environment Modeling Based Requirements Engineering for Software Intensive Systems. Elsevier/Morgan Kaufmann/HZ Books, Cambridge (2018)
21. Ko, H.S., Zan, T., Hu, Z.: 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, St. Petersburg, pp. 61–72. ACM, New York (2016)
22. Lanese, I., Bucchiarone, A., Montesi, F.: A framework for rule-based dynamic adaptation. In: Proceedings of the 5th International Conference on Trustworthy Global Computing, TGC'10, Munich, pp. 284–300. Springer (2010)

23. Magee, J., Kramer, J.: Concurrency: State Models & Java Programs, 2nd edn. Wiley, Hoboken (2006)
24. Montrieux, L., Hu, Z.: Towards Attribute-Based Authorisation for Bidirectional Programming, pp. 185–196. ACM, Vienna (2015)
25. Pacheco, H., Zan, T., Hu, Z.: BiFluX: a bidirectional functional update language for XML. In: 6th International Symposium on Principles and Practice of Declarative Programming (PPDP 2014), Canterbury (2014)
26. Voigtländer, J.: Bidirectionalization for free! (pearl). In: POPL 2009, Savannah, pp. 165–176. ACM (2009)
27. Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M.: On patterns for decentralized control in self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. Lecture Notes in Computer Science, no. 7475, pp. 76–107. Springer, Berlin/Heidelberg (2013)
28. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), Atlanta, pp. 164–173. ACM (2007)
29. Yu, Y., Lin, Y., Hu, Z., Hidaka, S., Kato, H., Montrieux, L.: Maintaining invariant traceability through bidirectional transformations. In: 2012 34th International Conference on Software Engineering (ICSE), Zurich, pp. 540–550 (2012)
30. Zan, T., Liu, L., Ko, H.S., Hu, Z.: Brul: a putback-based bidirectional transformation library for updatable views. In: Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016. CEUR Workshop Proceedings, vol. 1571, pp. 77–89. CEUR-WS.org, Eindhoven (2016)