Master Thesis
Software Science

Radboud University

A Lens-Based Formalisation of View-Based Editing for Variant-Rich Software Systems

Author:
Bob Ruiken
s4721306

First supervisor/assessor:
Daniel Strüber
`d.strueber@cs.ru.nl`

Second assessor:
Thorsten Berger
`thorsten.berger@chalmers.se`

November 26, 2022

Abstract

Software systems typically become increasingly customizable over time. For instance, organizations need to tailor their systems towards specific customer requirements, including runtime environments, hardware, or performance and energy consumption. Organizations then typically start maintaining different cloned variants of the system—a process called clone & own—which is cheap and agile, but does not scale with the number of cloned variants and causes substantial maintenance overheads. Then, organizations often integrate the variants into a configurable ('variational') platform. Using variational code, for instance using `#ifdef` preprocessor directives, optional features can be implemented in the same codebase. However, configurable platforms are complex systems, since developers need to work on many different variants at the same time. `#ifdef` statements easily clutter the source code, challenging program comprehension and making development error-prone. We advocate view-based editing of variational source code. Ideally, a developer can choose the subset of features to work on. Previous work introduced a first conceptual investigation of view-based editing of variational code, but, in absence of a more rigorous description of supported operations and underlying assumptions, fell short to give any soundness guarantees.

We address this gap by formalizing and implementing view-based editing for variational source code in a framework we call *Variational Lenses*. We map out conditions under which soundness can be guaranteed, and give a soundness proof. Our formalisation relies on lenses—a solution to the view-update problem typically considered for bidirectional transformations. We formalise two functions, one to obtain a restricted view from the source code set, and one to update the source code given changes made to this restricted view.

We evaluate the framework by showing how it can be applied to common edit operations, which were previously identified from a real-world system. With this evaluation, we show that the new method can do at least as much as the previous system, plus adding more flexibility in the meanwhile. This flexibility is added by enabling developers to create and edit independent features in a view where other features are hidden.

Contents

List of Figures

Chapter 1

Introduction

Almost any software needs to exist in different variants. Organizations build software variants to address variable customer requirements. Two main strategies exist for building such software systems. First, with *clone & own* [7], a complete system is *cloned* and altered to customer-specific needs. This strategy is cheap, flexible and well supported by tools such as *GIT*. Second, an opposing strategy, which scales well with many variants, is to integrate all variants into a *configurable and integrated platform* [30], also known as a software product line. A common example of such a platform is the Linux kernel [27], which has over 15000 features nowadays and can be customized to run on small embedded devices or large supercomputer clusters. However, creating a configurable and integrated platform is costly, while it is a complex software system that can be difficult to maintain.

Integrated platforms rely on variability mechanisms—implementation techniques to realize variation points. The most common mechanisms are static compile-time statements such as `#ifdef` preprocessor directives in C or C++. While these mechanisms are popular, they negatively influence the ease of writing, maintaining and evolving of code [9, 28], since the preprocessor directives clutter source code, and the developer needs to work on many different variants at the same time. To reduce the complexity of developing an integrated platform, the developer could work on a *view* (or *projection*) of the system. Based on a *configuration* certain features are activated or deactivated, leading to irrelevant parts of the code being removed. This king of code editing has been proposed as *view-based editing* or *projectional editing of variational software* before [33, 20, 18].

A more recent example is a method created by Stănciulescu et al. [29], which, together with previous methods it is based on, make use of so-called *lenses*. Lenses present a solution for the view-update problem, which originates from databases and are a form of bidirectional programming [11]. The idea of lenses is to define two functions that map between two sets of data. We usually talk about these sets being the Source (S) and the View (V): a lens then consists of two functions $get : S \to V$ and $put : V \to S$. Together with these functions, a lens has to fulfil a number of lens laws to guarantee soundness in the context of a round-trip: of retrieving a particular view, modifying it and feeding it back into the source. One can see how lenses are applicable here, as we have a source set of variational software and a view containing a subset of the code. However the work of Stănciulescu has two limitations. First, the lens laws have not been proven for the method described in it. As we will show, even one of the lens laws does not hold for it. The second limitation is in one of their functions of the lens: it is not possible to feed the changes back into the source that should not influence the retrieved projection. Intuitively, when a developer creates a view based on a configuration and then modifies the view (i.e., evolves the source code), it is not possible to let the change apply to more variants than are in the view.

In the present work, we aim to overcome these limitations by presenting *Variational Lenses*, a framework for view-based editing of variational code. We provide definitions for the *get* and a *put* functions with as few limitations as possible. As we will demonstrate with a counterexample, support for completely unrestricted editing is not possible without losing soundness guarantees. Furthermore, we will give a specific and realistic condition to enable soundness guarantees, which we will formally prove. We will also see that while the previous method does not have this restriction, it should have it in order to adhere to the same law.

To provide our definition and proofs, we rely on a variability management framework called Virtual Platform. The Virtual Platform is a development system that aims to bridge the gap between the opposing strategies *clone & own* and an integrated platform [22]. We chose to build our definitions on the Virtual Platform, since it provides a suitable abstraction of variational code, with conceptual structures and operations supporting several different programming languages. To formalise the operators, we first needed to formalise the relevant parts of the Virtual Platform. After formalising the operators, we developed a definition of *Variational Lenses*. This type of lens is able to carry configuration data with it to support the functions we created. This new type of lens is needed, as existing lenses are not able to carry configuration information with them. The functions we created are then proved to comply with the lens laws of this new lens. Lastly, we evaluate our new formalisation and implementation of view-based editing based on a real-world version history from [29]. In this evaluation, we will see that the new method can do at least as much as the previous method while complying with the lens laws.

Note here that we are differentiating between an “old” and a “new” method. It should be noted that this is not exactly the case. We do not principally create a new method, we offer the notion of view-based editing with choices and ambitions in the same way as the previous work. Our “new” method differs from the previous work in the formalisation, implementation and the proofs that we supply for our formalisation.

In summary, we contribute:

1. A formalisation of a part of the Virtual Platform, as a foundation for generically describing variational lenses.
2. A formalisation of view-based editing in terms of the formalisation of the Virtual Platform that allows for more flexibility than the previous method.
3. A formalisation of Variational Lenses together with the relevant lens laws.
4. A proof that the view-based editing adheres to the lens laws.
5. An evaluation of the method using real-world data extracted from, and a comparison to the method by, Stanciulescu et al. [29].
6. An implementation of the operators in the Virtual Platform[1].

[1]`https://bitbucket.org/easelab/vp/src/master/`

Chapter 2

Background

2.1 Software Product Lines

We will first give a brief introduction to *Software Product Lines*. More background details can be found in for example [31, 2, 5]. Software Product Lines use the same idea as age-old industrial product lines. The idea behind them is that variants of products can be easily created when multiple products use the same general parts. If these parts can be created in the same factory, we only need to combine these in different ways to create multiple products. This idea can be read in the context of industrial manufacturing, where we can for example create different aeroplanes that have great overlap in their parts. We can also read this in the context of software, this is where we talk about Software Product Lines.

Concepts used in software product lines include features, configurations, variants and products. Since we will also use these concepts throughout this work, let us look at them in more detail. *Features* are "*a logical unit of behaviour that is specified by a set of functional and quality requirements*", according to [5]. This means that features implement the requirements of a system, an example would be sending messages in an e-mail system. These features can be toggled on and off, they are binary. In source code, features can be implemented for example using `#ifdef` statements. *Configurations* can be seen as a list of all features with all of them either enabled or disabled. Configurations can be deemed valid with the use of *Feature Models*, which describe relations between features. One could see how features such as *Linux* and *Windows* should not occur simultaneously. *Variants* or *products* are configurations applied to full systems. By applying them, certain parts in the source code can be removed, whilst others should stay, decided by the variability mechanism (i.e. the `#ifdef` statements).

2.2 Virtual Platform

In this section, we will go more in-depth into some specifics of the Virtual Platform [22]. We do this because we want to formally define the framework. Besides that, we want to extend the framework with two accompanying operators, which will of course be formalised in the created formalisation.

The Virtual Platform consists of several conceptual structures on which the operators within it work. These structures are especially important to us as we will formalise them in Chapter 3. The most important structure is the assets, making up the *Asset Tree* (AT), an abstraction of software repositories with their included assets. Assets make up the complete structure of the software product line. They can be anything from folders to methods. The asset tree is a hierarchical, non-cyclic tree where the nodes are all assets. Each asset has a specific asset type, a version, a presence condition and a number of possible child assets. Each asset may also have a *Feature Model* (FM) attached. Certain asset types may only be in children of certain other asset types, for example, we do not want a folder asset type to reside in a file asset type. Features have names and two parameters, being *optional* and *incomplete*. Optionality describes whether or not the feature is mandatory, incompleteness describes whether the feature is fully implemented or not. Since we want to have feature models, features can also have subfeatures. A feature model can then consist of just a root feature and a special feature called "unassigned". This last feature can be used to mount features resulting from clone operations which were previously not mounted in the model at all. The Virtual Platform uses a *Trace Database*, which keeps track of clones of assets through the asset tree, this together with the versions of assets and features can be used to propagate changes between clones.

The Virtual Platform already contains numerous operators to work on the internal structures. The two operators we will add, work alongside the existing operators. This means that we do not have to keep in mind how they might interact with the internal structures. Our new operators are special in the way that they have a necessary order in which they can be used, the *put* must be used after the *get*, and the *get* operator can only be used while there is no 'active' clone in progress.

For this work, the most important structures are the assets and the features. We do not need to formalise the notion of the trace database or the versions as our new operators will not use these. In our formalisation, we also abstract the feature models such that they are described using an expression. This simplifies the formalisation of features as they can then consist of just a name.

$$\begin{tikzcd}
s \arrow[r, dashed, "\text{Derived edit}"] \arrow[d, bend right, "get\,(s)"'] & s' \arrow[d, dashed, bend right, "get\,(s')"'] \\
v \arrow[u, dashed, bend right, "put\,(v, s)"'] \arrow[r, "\text{Manual edit}"'] & v' \arrow[u, bend right, "put\,(v', s)"']
\end{tikzcd}$$

Figure 2.1: Overview of a standard lens.

2.3 Lenses

A lens l is a mapping between a set C of "concrete" structures (also called *sources*) and a set A of "abstract" structures (also called *views*), consisting of three functions *get*, *put* and *create* [11]. The *get* function takes a concrete structure and yields an abstract structure. The *put* function does the reverse: it takes an abstract structure combined with the original concrete structure and gives back a new concrete structure. Finally, the *create* function is much like the *put* function, but lacks the original concrete part. The *create* function can thus create a concrete structure from just an abstract structure. Formally, we get:

$$\begin{aligned}
&get &&: C \to A \\
&put &&: A \times C \to C \\
&create &&: A \to C
\end{aligned}$$

For a set of these three functions to be called a lens, they have to satisfy some lens laws. These lens laws ensure so-called acceptability (PutGet) and stability (GetPut):

$$\begin{aligned}
put\,(get\ c)\ \ c &= c && \textsc{GetPut} \\
get\,(put\ a\ c) &= a && \textsc{PutGet} \\
get\,(create\ a) &= a && \textsc{CreateGet}
\end{aligned}$$

How all of these definitions work together can best be seen in the form of the schema shown in Figure 2.1. Here we can see two concrete structures s and s', connected by the *get* and *put* functions. The abstract structure v is the result of applying *get* to the source s and v' is the view that results from the edits made to v. We see a dashed line back to s from v, this represents the GetPut law. A similar dashed line can be seen from s' to v', this represents the PutGet law. We of course also have an arrow from v' to s', showing how to finish the loop towards the new concrete structure.

```
#ifdef F
  a
#else
  b
#endif
```

(a) Source code

$F\langle a, b\rangle$

(b) Choice calculus

Figure 2.2: Different representations of variational source code.

2.4 Current State of Research

The most closely related work to ours is by Stănciulescu et al., which is a view-based editing method [29] as well. Their system is defined in the formalisation of variational software called *Choice Calculus* [33] and also uses the concept of lenses. Choice calculus abstracts away the notation of `#ifdef` statements into a more formal syntax to simplify the definitions. Our work will not be based on choice calculus, but rather on the conceptual structures of the Virtual Platform. We will still take a closer look at how this method works because we are going to compare to this method in the evaluation (Section 6).

Choice Calculus is a way to abstract variational source code into a more consise format [8]. Figures 2.2a and 2.2b show how a piece of variational source code is represented using choice calculus. In the first subfigure, we see the source code, where `a` is guarded by the feature `F`. If `F` does not hold, then we want `b` to be active. In the second subfigure we see this represented in the choice calculus. We again see the feature F and the codes a and b. The representation means that if F is true, we pick a, otherwise we pick b. If an `#ifdef` does not contain an `#else` statement, the second argument of the choice can be replaced by an empty line of code. For example, removing the *else* clause from our example would give us $F\langle a, \iota\rangle$. The b line is now removed and replaced by ι.

In the previous work which was also based on lenses, the *get* operator requires a *choice*. With this *choice*, the programmer determines the set of variants included in the view (expressed in terms of activated and deactivated features). In our example of Figure 2.2b, a choice of F would yield just a, instead of the full choice. The negation of F ($\neg F$) would give just b. If F is not decided on by the choice, the choice block remains in full.

The *put* operator requires another expression, called the *ambition*. The ambition is a way for the programmer to determine the set of variants to which the changes should be applied. The way the *put* operator deals with this ambition, is by creating a new top-level choice (a representation of an

`#ifdef`) in the form of $(c \wedge a)\langle v', s\rangle$. Here, c and a are the chosen choice and ambition in the *get* and *put* operators, v' is the edited view and s is the source from which the view originated. This top-level choice is then minimised such that the common parts are extracted from both branches. Let us take for example the choice calculus expression from Figure 2.2b and apply the *get* operator with as choice F. We already know this results in just a as the source code. We as the programmer now decide to change this to c. To push these changes back into the original source code, we use the *put* operator. As the ambition, we again choose F, resulting in the choice calculus expression of $(F \wedge F)\langle c, F\langle a, b\rangle\rangle$. We can simplify this expression in two ways: firstly, we can simplify the first choice expression by replacing $F \wedge F$ to just F. Secondly, we can replace the $F\langle a, b\rangle$ by b, since we know that F does not hold in this part of the first choice level.

The limitations of the previous work, as mentioned before, are twofold. Firstly, the method is not proven to be correct with the lens laws. This means that we cannot strictly say that the method uses lenses. More importantly, we cannot ensure the round-tripping properties that lenses want to ensure. Moreover, we will later give a counterexample that disproves one of the lens laws for the previous work. The second limitation of the previous work is the way the *put* operator works. It does not allow changes to be propagated to a scope beyond the *choice* expression from the *get* operator. This was a deliberate design choice as it holds to the *Edit Isolation Principle*, where changes can only be made to what is visible in the view. To see how this works, let us look at Figure 2.2a. If we create a view with only feature `F`, we result in a view with just `a`. Now the *Edit Isolation Principle* says that we cannot make any edits such that any of the left out code can be changed. In this case this means that we cannot make changes to the code in `b`. This work aims to lift the above two limitations, we want to create a method in which the *Edit Isolation Principle* is lifted and for which we can prove the lens laws.

2.5 Running Example

In this work, we will generally use the sample example system to show the workings of the formalisations. This is a simple application that initially has three classes embedded in one file. Our system has two features, namely an optional *CLI* and *logging*. We have that *logging* can only be enabled when *CLI* is enabled, guarded by a feature model. In the end, we want to be able for a developer to work only on a part of the file, and then be able to also push these changes back into the full source file.

```
public class Base
{
  ...
}

#ifdef CLI
public class CLI
{
  ...
}
#endif

#ifdef logging
public class logging
{
  ...
}
#endif
```

(a) Original example source code.

```
public class Base
{
  ...
}

public class CLI
{
  ...
}
```

(b) Example view of example source code.

Figure 2.3: Running example source code.

An example of how this source code might look can be found in Figure 2.3. We first what the entire source code looks like in Figure 2.3a. As explained before, we have one base class that is not locked behind a feature and two classes that are locked behind the *CLI* and *logging* feature, respectively. To show why projectional editing can be beneficial, see Figure 2.3b. In this view we have left out the *logging* feature and have selected the *CLI* feature. We can see this, as the *CLI* feature is not surrounded by `#ifdef` statements, and the *logging* feature is left out altogether. The view is easier to edit for a developer, as there is fewer code. The benefits seem marginal here, but one can imagine that within the classes `Base` and `CLI`, there might be specific code for the other features as well, which would also become less complex in the view.

Chapter 3

Formalisation

In this chapter, we formally define the conceptual structures that comprise the Virtual Platform. After we have formally defined the structures, we define the lens functions on top of this formalisation.

3.1 Main Data Structures

We have a set of features $\mathcal{F}$, which contain the names of the features. We also define a set $\mathcal{C}$, containing all possible contents of assets. In our case, the contents are strings, so $\mathcal{C}$ can be seen as the set of all possible strings. Then we have a set $\mathcal{I}$ to denote all identities of assets (we will use these identifiers to identify assets.)

We create a notion of *Optional types*, this notation is used to extend a set with the "no value" ($\perp$) value:

$$\lceil t \rceil = \{\perp\} \cup t$$

$Exp\,(\mathcal{F})$ are expressions over features defined using the following grammar:

$$e ::= f \mid \ \neg e \mid \mathsf{True} \mid \mathsf{False} \mid e \wedge e \mid e \vee e \mid e \Rightarrow e$$

Here we have that $f \in \mathcal{F}$. The operators are logical conjunctions, logical disjunctions and logical implications.

Asset types are needed to correctly specify different types of assets in the Virtual Platform:

$$\mathsf{Types} = \{\mathsf{Root}, \mathsf{Repository}, \mathsf{Folder}, \mathsf{File}, \mathsf{Class}, \mathsf{Method}, \mathsf{Field}, \mathsf{Block}\}$$

For now, we define feature models or feature trees as a single expression of type $Exp\left(\mathcal{F}\right)$. Actual feature models are usually defined using trees to visualise the hierarchy in them. In the virtual platform, every asset is allowed to have its own feature model.

The set of all assets is denoted using $\mathcal{A}$. Each element $a \in \mathcal{A}$ is a tuple of the following type:

$$a \in \mathcal{I} \times \mathsf{Types} \times \mathcal{P}\left(\mathcal{A}\right) \times \lceil Exp\left(\mathcal{F}\right) \rceil \times Exp\left(\mathcal{F}\right) \times \mathcal{C}$$

If $(n, t, ch, fm, pc, c) \in \mathcal{A}$, then n is the name, t is the type of asset, ch are the children of the asset, fm is the optional feature model, pc is the presence condition binding the asset to a feature model and c is the content of the asset.

Note that the children of assets are defined using a set of assets. The reason for putting the actual assets in them instead of the identities of the assets is solely to ease the definitions we will create later.

Another more important point is that we need some way to order the children of assets. For this, we use a partial ordering on the assets. If we were to not have any ordering, the children of an asset could appear in any order. This means that the contents of those assets (code) will not have a certain order. We do not want an asset that uses some variable to appear before the asset that defines that variable. For this reason, we need an ordering of the set of assets.

If we were to put a full ordering on all assets, we would order too much. For instance, we do not need to compare children of two distinct assets. Instead, we could introduce a full ordering, but only between the children of assets. That is, every set of children is ordered. But even this is too strong. We can restrict the ordering to just children of assets with certain asset types. The reason for this is that we do not need ordering of children of for example folder types. It does not matter to us if some file or folder precedes another file or folder. Following this logic, we only need to place an ordering on the children of the following asset types: File, Class, Method and Block. We define the ordering relation on the assets as an irreflexive, asymmetric and transitive strict partial order $<_{\mathcal{A}}$.

To more easily access the properties of assets we define the following

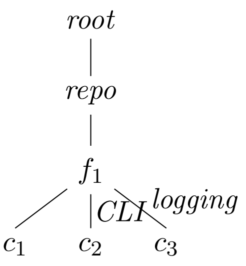


Figure 3.1: The example system in the form of an asset tree.

functions:

$\mathsf{Name} : \mathcal{A} \rightarrow \mathcal{I}$ $\mathsf{Name}\left((n, \_, \_, \_, \_, \_)\right) = n$	$\mathsf{Type} : \mathcal{A} \rightarrow \mathsf{Types}$ $\mathsf{Type}\left((\_, t, \_, \_, \_, \_)\right) = t$
$\mathsf{Children} : \mathcal{A} \rightarrow \mathcal{P}(\mathcal{A})$ $\mathsf{Children}\left((\_, \_, ch, \_, \_, \_)\right) = ch$	$\mathsf{FM} : \mathcal{A} \rightarrow \lceil Exp(\mathcal{F}) \rceil$ $\mathsf{FM}\left((\_, \_, \_, fm, \_, \_)\right) = fm$
$\mathsf{PC} : \mathcal{A} \rightarrow Exp(\mathcal{F})$ $\mathsf{PC}\left((\_, \_, \_, \_, pc, \_)\right) = pc$	$\mathsf{Content} : \mathcal{A} \rightarrow \mathcal{C}$ $\mathsf{Content}\left((\_, \_, \_, \_, \_, c)\right) = c$

A full system in the Virtual Platform is defined as a tuple:

$$\mathcal{V} = \langle \mathcal{A}, \mathcal{F}, \mathcal{I}, \mathcal{C}, <_{\mathcal{A}} \rangle$$

This tuple thus contains the assets, the features, the identities and the contents in the platform, together with a partial order on the assets.

Example 3.1. In this example, we will create the running example as described in the background (Section 2.5) in terms of the formalisation described above. To this end, we need to fill out the tuple that describes a system ($\mathcal{V}$). This tuple consists of the assets, the features, the identities and the contents.

First, we define the set of features, identities and contents. Our codebase will consist of six assets, so we get six identities and six contents. We only have two features in our system. For the features, we get $\mathcal{F} = \{\mathit{logging}, \mathit{CLI}\}$, for the identities, we have $\mathcal{I} = \{t_1, r_1, f_1, c_1, c_2, c_3\}$ and finally for the asset contents we have $\mathcal{C} = \{s_1, s_2, \ldots, s_6\}$. It can be imagined that these contents are made up off lines of code. Contents may also be empty in case of for example root, repository, folder of file assets.

Remember that we want the *logging* feature to only be available when the *CLI* feature is activated, so we create a feature model, defined by $\mathit{fm} =$

logging $\Rightarrow$ *CLI*. With this last piece of information we can create the actual assets of the system:

$$\begin{aligned}
a_1 &= (c_1, \mathsf{Class}, \emptyset, \bot, \mathsf{True}, s_1)\\
a_2 &= (c_2, \mathsf{Class}, \emptyset, \bot, \mathit{CLI}, s_2)\\
a_3 &= (c_3, \mathsf{Class}, \emptyset, \bot, \mathit{logging}, s_3)\\
a_4 &= (f_1, \mathsf{File}, \{a_1, a_2, a_3\}, \bot, \mathsf{True}, s_4)\\
a_5 &= (r_1, \mathsf{Repository}, \{a_4\}, \bot, \mathsf{True}, s_5)\\
a_6 &= (t_1, \mathsf{Root}, \{a_5\}, \mathit{fm}, \mathsf{True}, s_6)
\end{aligned}$$

We can see that there are only two non-trivial presence conditions, namely in a_2 and a_3, which are for the *CLI* and *logging* features, respectively. We can also see that the feature model (*fm*) is located in the root of the tree (a_6). The last part we need to define to complete the system is the ordering of the assets. There is only one asset whose children need to be ordered, that is a_4. We define the classes to be ordered such that $a_1 <_{\mathcal{A}} a_2 <_{\mathcal{A}} a_3$.

The asset tree can also be seen in Figure 3.1. This format is what we will use throughout this work. We have the identities of the assets as nodes, and the children of these nodes are connected using edges. The edges are labelled with the presence conditions of the children, but only if there is a non-trivial presence condition (if it is not True). We can easily see the hierarchy in this format, we first have the root, then the repository, followed by the file, which contains three classes. We see the two non-trivial presence conditions in the labels of the edges from f_1 to c_2 and f_1 to c_3. One thing to note is that feature models are not present in this representation.

3.2 Parent Functions

Next, we will define a number of functions to find the parent of an asset. We will need these definitions later to define the restrictions of the platform and in the definitions of the new operators. To create this function, we first define a function to retrieve all the parents of an asset and then define that this function may give back at most one element. So we first define a function to find the parents of an asset:

$$\mathsf{FindParents} : \mathcal{A} \to \mathcal{P}(\mathcal{A})$$

$$\mathsf{FindParents}(a) = \left\{a' \;\middle|\; a \in \mathsf{Children}(a') \wedge a' \in \mathcal{A}\right\}$$

With this definition, we can create the parent function that actually returns one parent (or none):

$$\mathsf{Parent} : \mathcal{A} \to \lceil \mathcal{A} \rceil$$

$$\mathsf{Parent}\,(a) = \begin{cases} a' & \text{if } \mathsf{FindParents}(a) = \{a'\} \\ \bot & \text{otherwise} \end{cases}$$

Example 3.2 (Continutation of Example 3.1)**.** Let us take a look at how the parent functions work in terms of the example system we defined before.

$$\begin{array}{ll} \mathsf{FindParents}\,(a_1) = \{a_4\} & \mathsf{Parent}\,(a_1) = a_4 \\ \mathsf{FindParents}\,(a_4) = \{a_5\} & \mathsf{Parent}\,(a_4) = a_5 \\ \mathsf{FindParents}\,(a_6) = \emptyset & \mathsf{Parent}\,(a_6) = \bot \end{array}$$

As we can see, the FindParents should always give back either an empty set, or a singleton set. Otherwise, the Parent cannot give a sensible singular asset. We still create these two function separately, as we can handily use the first function in the restrictions we will define next.

3.3 Restrictions

In this section, we introduce several restrictions to a Virtual Platform system such that we see it as correct. These restrictions include but are not limited to, setting up a correct (connected) asset tree.

For a system $\mathcal{V} = \langle \mathcal{A}, \mathcal{F}, \mathcal{I}, \mathcal{C} \rangle$, we first limit how trees are supposed to be structured. This means that we want any asset to have at most one parent:

$$\forall_{a \in \mathcal{A}}\, |\mathsf{FindParents}\,(a)| \leq 1$$

Note that we say that the number of parents has to be less than or equal to one because an asset might have no parent at all (the root asset). We can now also define that we have only one root asset (the asset tree is connected):

$$|\{a \mid |\mathsf{FindParents}\,(a)| = 0 \wedge a \in \mathcal{A}\}| = 1$$

Finally, we also want to limit the type of assets, some asset types can only be children of certain other asset types. We start with the root type, which can only be found at the top of a tree:

$$\forall_{a \in \mathcal{A}} \mathsf{Type}(a) = \mathsf{Root} \iff \mathsf{Parent}(a) = \bot$$

Note that we do not specify that the root of the tree must be an asset with the Root type. This is because the tree may be cut up by the get operator that we define later.

The other assets are ordered as follows:

$$\begin{array}{lcl}
\forall_{a \in \mathcal{A}} \mathsf{Type}(a) = \mathsf{Root} & \iff & \forall_{c \in \mathsf{Children}(a)} \mathsf{Type}(c) = \mathsf{Repository} \\
\forall_{a \in \mathcal{A}} \mathsf{Type}(a) = \mathsf{Repository} & \implies & \forall_{c \in \mathsf{Children}(a)} \mathsf{Type}(c) \in \{\mathsf{Folder}, \mathsf{File}\} \\
\forall_{a \in \mathcal{A}} \mathsf{Type}(a) = \mathsf{Folder} & \implies & \forall_{c \in \mathsf{Children}(a)} \mathsf{Type}(c) \in \{\mathsf{Folder}, \mathsf{File}\} \\
\forall_{a \in \mathcal{A}} \mathsf{Type}(a) = \mathsf{File} & \implies & \forall_{c \in \mathsf{Children}(a)} \mathsf{Type}(c) \in \{\mathsf{Class}, \mathsf{Field}, \mathsf{Method}, \mathsf{Block}\} \\
\forall_{a \in \mathcal{A}} \mathsf{Type}(a) = \mathsf{Class} & \implies & \forall_{c \in \mathsf{Children}(a)} \mathsf{Type}(c) \in \{\mathsf{Field}, \mathsf{Method}, \mathsf{Block}\} \\
\forall_{a \in \mathcal{A}} \mathsf{Type}(a) = \mathsf{Method} & \implies & \forall_{c \in \mathsf{Children}(a)} \mathsf{Type}(c) = \mathsf{Block} \\
\forall_{a \in \mathcal{A}} \mathsf{Type}(a) = \mathsf{Field} & \implies & \mathsf{Children}(a) = \emptyset
\end{array}$$

3.4 Get Operator

We are now ready to define the *get* operator in terms of the formalisation of the Virtual Platform. Before that let us first look at what the operator wants to achieve.

The *get* operator wants to create a new asset tree that is a projection of an input asset tree, the resulting asset tree is created by a *choice* expression. This choice should be seen as some combination of features that the user wants to edit. By choosing some combination of features, some code (read: assets) may not be relevant anymore and thus will not be in the resulting asset tree.

We first give the type of the function as follows:

$$\mathsf{Get} : \mathcal{A} \times Exp\,(\mathcal{F}) \rightarrow \lceil \mathcal{A} \rceil$$

The first argument is the asset on which we want to apply the operator, note that this does not necessarily have to be the root asset. The second argument of this function is the so-called *choice*. With this expression, we "choose" which features we want to see in the result. Note that the resulting asset is optional (can be $\perp$). This is because the target asset might not satisfy the given choice, we then result in no asset at all.

The high-level workings of this operator in the Virtual Platform are as follows. We want to check for each asset if it is relevant for our *choice*. We do this by checking if the conjunction between an asset its presence condition and feature model together with the choice is satisfiable. We can do this satisfiability check using a SAT solver. If this satisfiability check results as unsatisfiable, the asset is not relevant in our result. We then do not have to check its children as they are gated by the conditions of their parents. If it is satisfiable, we include it into our result and recursively apply the same strategy to its children.

Because the asset we apply the operator on might not be the root node of the tree, we have to be careful of the feature models and presence conditions higher up in the tree. These conditions might have an impact on the branch our asset is on. To account for this, we concatenate all feature models and presence conditions higher up in the tree. We only have to go through this process for the asset we apply the operator on, since this impacts all assets further down in the tree. In the end, the asset we apply the operator on has four parts in the feature model: the original feature model, the conjunction of all parent feature models, the conjunction of all the presence ancestral presence conditions and finally the *choice*.

The recipe described above is for one asset. To make it work for an asset tree, we apply this function to the children of an asset as well. We only do this when the satisfiability check was positive. That is, we do not want to check the children of an asset that we do not want in the first place.

First, we create two functions to "fold" all ancestor feature models and presence conditions. We need these to store them in the target asset its feature model.

$$\mathsf{FoldFMs} : \mathcal{A} \to \lceil Exp\,(\mathcal{F}) \rceil$$

$$\mathsf{FoldFMs}\,(a) = \begin{cases} \mathsf{FM}(a) & \text{if } \mathsf{Parent}(a) = \bot \\ \mathsf{FM}(a) \lceil\wedge\rceil \mathsf{FoldFMs}\,(\mathsf{Parent}(a)) & \text{otherwise} \end{cases}$$

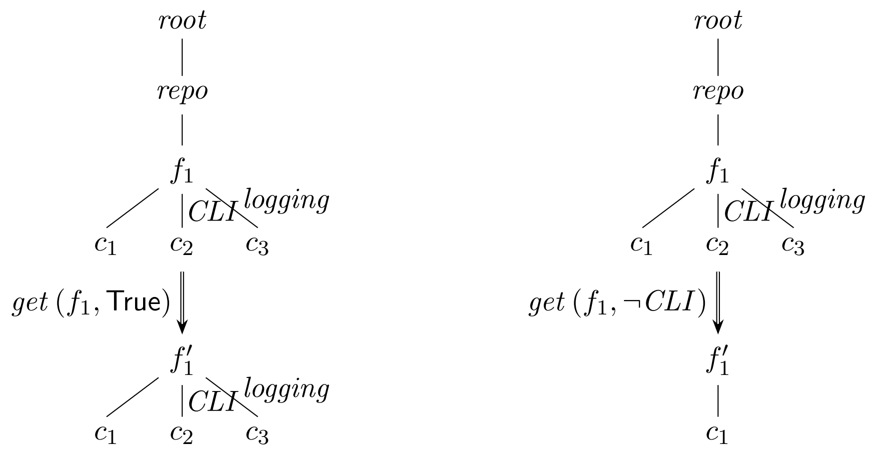
$$\mathsf{FoldPCs} : \mathcal{A} \to Exp\,(\mathcal{F})$$

$$\mathsf{FoldPCs}\,(a) = \begin{cases} \mathsf{PC}(a) & \text{if } \mathsf{Parent}(a) = \bot \\ \mathsf{PC}(a) \wedge \mathsf{FoldPCs}\,(\mathsf{Parent}(a)) & \text{otherwise} \end{cases}$$

As we can see the above two definitions are quite similar, the only difference is that presence conditions are not optional where feature models are. So in the case of feature models we use a "optional logical conjunction" ($\lceil\wedge\rceil$), which is a simple structure which with we can conjunct optional expressions:

$$e_1 \lceil\wedge\rceil e_2 = \begin{cases} e_2 & \text{if } e_1 = \bot \\ e_1 & \text{if } e_2 = \bot \\ e_1 \wedge e_2 & \text{otherwise} \end{cases}$$

With these definitions, we can create the full definition for the *get* operator. We create two definitions, the first of which describes the full operator. The second definition (Prune) is used as a helper function that applies the same general idea but it works on a set of assets.



(a) Choice = True (b) Choice = $\neg CLI$

Figure 3.2: Applications of the *get* operator on the running example.

$$\mathsf{Get}\,(a, e) = \begin{cases} \big(\mathsf{Name}(a), \mathsf{Type}(a), \mathsf{Prune}(a, e \wedge fm'), fm', \mathsf{PC}(a), \mathsf{Content}(a)\big) & \text{if } e \wedge fm' \in SAT \\ \bot & \text{otherwise} \end{cases}$$
$$\text{where } fm' = \mathsf{FoldPCs}\,(a)\,\lceil\wedge\rceil\,\mathsf{FoldFMs}\,(a)$$

$$\mathsf{Prune} : \mathcal{A} \times Exp\,(\mathcal{F}) \to \mathcal{P}\,(\mathcal{A})$$
$$\mathsf{Prune}(a, e) = \{(n, t, \mathsf{Prune}(c', check), fm, pc, c) \mid check \in SAT \wedge c' \in \mathsf{Children}(a)\}$$
$$\text{where } check = \mathsf{PC}\,(c') \wedge e \wedge \mathsf{FM}\,(c'),$$
$$\text{and } a = (n, t, \_, fm, pc, c)$$

We see that, on the asset that the operator is applied, we change the feature model to include all of the ancestor presence conditions and feature models. We do not have to do this in the Prune because we know that the assets we are working with here are not root assets in the resulting asset tree.

Another observation is that we include the feature model and presence condition of an asset in the recursive calls. That is, we include the feature models and the presence conditions in the second argument in calls of the Prune. We do this because we know when going into recursive calls, that these presence conditions and feature models must be satisfiable, otherwise the current asset would not be in the result.

Example 3.3 (Continutation of Example 3.1). We keep working on the previous example, now we will apply the *get* operator on the then defined asset tree. We will look at two applications, where our choice are firstly the trivial choice, and secondly the negation of the *CLI* feature. For simplicity, this *get* operations will not be called on the root of the tree (a_6) but on the file asset (a_4). Asset a_5 and a_6 are not as interesting, as they both only have one child.

We start with the application where we use the trivial choice. This example can be seen in Figure 3.2a. As expected, we see the result is equal to the tree that we apply the operator on, except that our file asset is now called f_1' instead of f_1. The reason for this is that this asset now contains its previously ancestral feature models and presence conditions. These are added by the operator using the FoldFMs and FoldPCs functions. Since there are no presence conditions in the tree above f_1, $\mathsf{FoldPCs}\,(a_4) = \mathsf{True}$. There is a feature model in the tree, however. This means that we get that $\mathsf{FoldFMs}\,(a_4) = \mathit{fm}$. This means that we already know that f_1' differs from f_1 by the feature model *fm*. Let us now look into what checks are done by the *get* operator to decide which assets should be in the result:

- For a_4 (labelled by f_1): $\mathit{fm} \in \mathit{SAT}$. The check should contain the choice conjucted with the feature model and presence condition of the asset. The choice is trivial and can thus be omitted, the feature model is equal to *fm*.
- For a_1 (labelled by c_1): $\mathit{fm} \in \mathit{SAT}$. The same holds here as for a_4, as a_1 does not have any presence condition or feature model attached to it.
- For a_2 (labelled by c_2): $\mathit{CLI} \wedge \mathit{fm} \in \mathit{SAT}$. The inclusion of the feature model has the same reason as above. This time we also have included the *CLI* feature, as it is the presence condition of a_2.
- For a_3 (labelled by c_3): $\mathit{logging} \wedge \mathit{fm} \in \mathit{SAT}$. Similarly to a_2, except that we have a different presence condition (*logging* instead of *CLI*).

Since all of these checks succeed, all of the assets from a_4 downwards are included in the result.

For our second example, we apply the *get* operator with a non-trivial choice, it being the negation of the *CLI* feature ($\neg \mathit{CLI}$). We again apply it on a_4, the result can be seen in Figure 3.2b. Since we apply the operator on the same asset as before, we can copy over the results of the FoldFMs and FoldPCs functions. This time, we see that we are missing both c_2 and c_3 in the result. The first one seems obvious: its presence condition is *CLI*, and we have the negation of that as our choice. The second one is actually missing because of our feature model, which says that *logging* can only co-exist with the *CLI* feature. The checks done by the *get* operator then look as follows (remember that *fm* is defined by $logging \Rightarrow CLI$):

- For a_4 (labelled by f_1): $\neg CLI \wedge fm \in SAT$. The check should contain the choice conjucted with the feature model and presence condition of the asset. The choice is the negation of the *CLI* feature, and the feature model is equal to *fm*.
- For a_1 (labelled by c_1): $\neg CLI \wedge fm \in SAT$. The same holds here as for a_4, as a_1 does not have any presence condition or feature model attached to it.
- For a_2 (labelled by c_2): $\neg CLI \wedge CLI \wedge fm \notin SAT$. The inclusion of the feature model has the same reason as above. This time we also have included the *CLI* feature, as it is the presence condition of a_2. This leads to a problem as *CLI* and $\neg CLI$ cannot be satisfied.
- For a_3 (labelled by c_3): $\neg CLI \wedge logging \wedge fm \notin SAT$. Similarly to a_2, except that we have a different presence condition (*logging* instead of *CLI*). This also leads to an unsatisfiable situation, as our feature model says that *CLI* is implied by *logging*. In this case, *logging* holds yet *CLI* does not.

Because of the unsatisfiability of the presence conditions of assets a_2 and a_3, they are not present in the result.

3.5 Put Operator

In this section, we will introduce the semantics for the put operator for the Virtual Platform. We first introduce the idea of the operator, then some new notation, and finally give the formalisation of the *put* operator. The formalisation of the final operator will be given in terms of pseudocode, where the helper functions will be defined as well have seen previously.

The *put* operator, in a way, does the opposite of the *get* operator. We want this operator to include the retrieved asset tree of the *get* operator back into the original tree on which we applied the *get* operator. Of course, this tree may have been changed between applying the two operators, we will even see that if there are no changes, the *put* operator is trivial.

3.5.1 Distinguishing Assets

We first create a way to distinguish between the assets in the "original" asset tree and the asset tree created after the *get* operator (we will also refer to this as the "new" tree.) This new and old tree have assets in common, but we know that the *get* operator might have hidden assets, and the programmer might have added or deleted assets after applying the *get* operator as well. When we talk about the assets which were there just before applying the *get* operator, we denote is using $\mathcal{A}_{old}$, knowing $\mathcal{A}_{old} \subseteq \mathcal{A}$. To denote the assets that we want to put back using the *put* operator, we use $\mathcal{A}_{new}$, again with $\mathcal{A}_{new} \subseteq \mathcal{A}$. These two sets make up all the assets in the system: $\mathcal{A}_{old} \cup \mathcal{A}_{new} = \mathcal{A}$.

Uniquely identifying assets should be done using their identifier. However, when we clone (part of) the asset tree using the *get* operator, the identifiers do not change. In the *put* operator we can handily use this to our advantage. With this identifier, we can match "old" assets with their "new" new counterpart. Thus far we have not put limitations on the identities of assets, since we did not use them. Now we want to limit the identities such that they are unique in the old and new asset sets:

$$|\mathcal{A}_{new}| = |\{\mathsf{Name}(a) \mid a \in \mathcal{A}_{new}\}|$$

$$|\mathcal{A}_{old}| = |\{\mathsf{Name}(a) \mid a \in \mathcal{A}_{old}\}|$$

This limitation ensures that any existing asset in the new set will map to exactly one asset in the old set.

3.5.2 Put By Diff

Applying the put operator means that we apply the changed assets to the existing asset tree. One way to do this is by applying a "diff" of the tree much like we have diffs for code in versioning contexts. We identify three actions that we can apply to assets: we can add, edit and remove them. For each of these actions we have to make changes to the presence conditions to correctly apply them when applying the *put* operator:

- **Adding** an asset: Add the ambition to the PC of the new asset.
- **Editing** an asset: Add the ambition to the PC of the new asset, and add the negation of the ambition to the old asset.
- **Removing** an asset: Add the negation of the ambition to the old asset.

	Added	Edited	Removed
New asset	PC $\wedge$ a	PC $\wedge$ a	
Old asset		PC $\wedge\neg a$	PC $\wedge\neg a$

Table 3.1: The difference between the result of different actions on assets, a being the ambition

At this point, we should realise that there is no difference between the actions, this is easily visualised in table form, see Table 3.1. In this table, we can see that the action does not have an effect on what we do with the presence condition of assets. The only difference is whether we talk about old or new assets. We do note the "holes" in the table, but these raise no problems. The first hole is when we remove an asset, we have nothing to do with the new asset. This makes sense as the new asset does not exist anymore. The way to solve this is by simply "trying" to add the ambition to the presence condition, if this cannot be done, it means that the asset was removed. Following the same logic, we can simply "try" to add the negation of the ambition to the old asset (we look it up using the unique identifier.) If this works, we know that the asset was either edited or removed, if this does not work, the asset was added. If we can work with this, we only need a set of changed assets for a diff. The combination of this set and the asset trees makes it possible to correctly apply the *put* operator. This does mean that we need to record actions taken on the asset tree after the *get* operator is applied. This is the least amount of work we can do, however. We need at least some form of diff. Another option would be to calculate the diff using the old and new asset trees. This would cost a lot of computing power as we would need to compare every asset and its contents.

3.5.3 Formalisation

Now that we have the idea of the *put* laid down, we can create the actual formalisation.

In Algorithm 1 we can see how the operator works in the form of an algorithm. We should note the types of the input: first, we have the new information, that being the new asset tree ($newAssets$) and the set of changed assets $changedAssets$, these have the types $\lceil\mathcal{A}\rceil$ and $\mathcal{P}(\mathcal{I})$ respectively. Then we see the arguments we also had for the *get* operator, being the original asset tree $oldAssets$, being an element of $\mathcal{A}$ and the choice expression *choice*, having type $Exp(\mathcal{F})$. The new expression we have is the *ambition*, having the same type as the first expression. Finally, as the output of the algorithm, we have the new asset tree, which we will see, is an altered version of the old asset tree. The broad idea of the algorithm is as follows. We first find out which assets we actually need to apply to the old tree, then for each of these assets, we add the negation of the ambition to the old asset and the ambition to the new asset. Finally, if there is a new asset (it is not a remove action), we add it to the old tree.

To complete these actions we need a number of functions. First up, we need a function to filter the assets that we actually need to apply, this is the FILTERASSETS function. To find the old and new assets using its identity, we have the FindOriginal and FindNew functions respectively. Finally, we have the INSERTNEWASSET function to insert the new asset into the old tree.

The FILTERASSETS function is a function that filters all assets from a set, which are already children of other assets in that same set. The reason we want to filter these assets is that we do not need to do anything with them in the old asset tree. If we have two assets a_1 and a_2, where $a_2 \in \mathsf{Children}(a_1)$ and both of these assets are in the changed assets, then adding the negation of the ambition to the old version of a_1 also impacts the old version of a_2 since it is a child of that asset. The implementation of the function can be found in Algorithm 2. The function takes two arguments, one being the asset we are currently looking at (the identity of it, so its type is $\mathcal{I}$.) The other argument is of the same type as the argument we have seen in the *put* operator: the list of changed assets is a set of identities ($\mathcal{P}(\mathcal{I})$). The function results in a simplified set of the same type. The function itself works by checking if the current asset is in the changed assets, if this is the case, all the children are removed from the changed assets. If it is not the case, the function is recursively applied to all children of this asset.

The FindOriginal and FindNew functions are very similar, the goal of both of these functions is to find the asset with the given identity. The difference is that, as the names suggest, the FindOriginal function looks in the original (old) set of assets ($\mathcal{A}_{old}$) and the FindNew function looks in the new set of assets ($\mathcal{A}_{new}$). The formalisation of these functions is as follows:

$$\mathsf{FindAssets}_c(i) = \{a \mid \mathsf{Name}(a) = i \wedge a \in c\}$$

$$\mathsf{FindOriginal} : \mathcal{I} \to \lceil \mathcal{A} \rceil$$

$$\mathsf{FindOriginal}(i) = \begin{cases} a & \text{if } \mathsf{FindAssets}_{\mathcal{A}_{old}}(i) = \{a\} \\ \perp & \text{otherwise} \end{cases}$$

$$\mathsf{FindNew} : \mathcal{I} \to \lceil \mathcal{A} \rceil$$

$$\mathsf{FindNew}(i) = \begin{cases} a & \text{if } \mathsf{FindAssets}_{\mathcal{A}_{new}}(i) = \{a\} \\ \perp & \text{otherwise} \end{cases}$$

Lastly, we need one more helper function, FlattenChildren does what its name suggests, it takes an asset and flattens all children within it into a set. The function is recursively called on the children of the children as well, such that the result consists of all the assets until the leaves of the tree.

$$\mathsf{FlattenChildren} : \mathcal{A} \to \mathcal{P}(\mathcal{A})$$
$$\mathsf{FlattenChildren}(a) = \bigcup \{\{c\} \cup \mathsf{FlattenChildren}(c) \mid c \in \mathsf{Children}(a)\}$$

Algorithm 1 PUT

```
input $newAssets$: New asset tree ($\lceil\mathcal{A}\rceil$)
input $changedAssets$: Set of changed assets ($\mathcal{P}(\mathcal{I})$)
input $oldAssets$: Original asset tree ($\mathcal{A}$)
input $choice$: Choice expression ($Exp(\mathcal{F})$)
input $ambition$: Ambition expression ($Exp(\mathcal{F})$)
output New asset tree ($\mathcal{A}$)

1: $toApply \leftarrow$ FILTERASSETS($changedAssets$, Name($asset$))
2: for all $a \in toApply$ do
3:    $origAsset \leftarrow$ FindOriginal($a$)
4:    if $origAsset \neq \perp$ then
5:       $origAsset$.PC $\leftarrow$ $origAsset$.PC $\wedge \neg ambition$
6:    end if
7:    $newAsset \leftarrow$ FindNew($a$)
8:    if $newAsset \neq \perp$ then
9:       $newAsset$.PC $\leftarrow$ $newAsset$.PC $\wedge$ $ambition$
10:      INSERTNEWASSET($newAsset$, $origAsset$)
11:   end if
12: end for
13: return $oldAssets$
```

The last function we have yet to define is the INSERTNEWASSET function. This function is only called when we have either added or edited some asset. In case of a deletion, we do not have a new asset. Adding an asset can lead to problems. For example, in Figure 3.3, we start with an asset tree containing two child assets, of which one is hidden by applying the *get* operator with $\neg C$ as the *choice*. The tree is augmented by the programmer by adding a new asset a_3, after a_1. Now when the *put* operator is applied on this new tree, together with the original tree and B as the *ambition*, which leads to an *Alignment Conflict*. The reason is that we cannot know whether we want to add the new asset a_3 either before or after a_2, since a_2 was hidden when we added the new asset. The problem in this case would not be relevant if the new asset and the hidden asset could not occur simultaneously: if the final presence condition of the new asset combined with the presence condition of the hidden asset is not satisfiable, it does not matter whether we add the new

Algorithm 2 FilterAssets

```
    input $changedAssets$: Set of changed assets ($\mathcal{P}(\mathcal{I})$)
    input $asset$: Root asset ($\mathcal{I}$)
    output Set of assets ($\mathcal{P}(\mathcal{I})$)

 1: $result \leftarrow changedAssets$
 2: if $asset \in changedAssets$ then
 3:     $newAsset \leftarrow \mathsf{FindNew}(asset)$
 4:     if $newAsset = \perp$ then
 5:         return $result$
 6:     end if
 7:     for all $c \in \mathsf{FlattenChildren}(newAsset)$ do
 8:         $result \leftarrow result \setminus \mathsf{Name}(c)$
 9:     end for
10: else
11:     for all $c \in \mathsf{Children}(\mathsf{FindNew}(asset))$ do
12:         $result \leftarrow \mathsf{Prune}(result, c)$
13:     end for
14: end if
15: return $result$
```

asset before or after the hidden asset. We also do not have to worry about alignment conflicts when we are working with changes to existing assets. When we edit an asset, we always want the "new" asset to be right next to the old asset, since that is the exact location where it was before it was edited. The "old" and the "new" assets in case of an edit can never occur at the same time, this is by definition of their presence conditions.

The positioning of children in assets is decided by the partially ordered set $<_{\mathcal{A}}$. In tree forms such as in Figure 3.3, we can visually see the ordering. So we know that $a_1 <_{\mathcal{A}} a_2$ and later that $a_1 <_{\mathcal{A}} a_3$. The problem is that these two combined can lead to two different full orders. We can get $a_1 <_{\mathcal{A}} a_3 <_{\mathcal{A}} a_2$, as in the left variant, or $a_1 <_{\mathcal{A}} a_2 <_{\mathcal{A}} a_3$, as seen in the right version. In our algorithm, the programmer must make sure that this full order ultimately is defined. As hinted before, this ordering is only relevant for a subset of the asset types, we do not want to order assets such as files.

In the end, we want our function to first check the type of the parent asset. If the ordering of that parent is not relevant, we can simply add the asset to the "new" tree without a problem. Otherwise, we need to check if the action performed on the given asset is an edit of an asset or an addition of a completely new asset. In the case of an edit, we can automatically restore the ordering of the children (we simply add the new asset right next

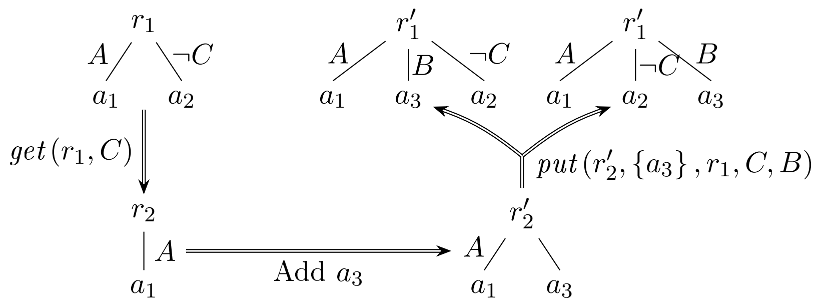


Figure 3.3: Example of how an alignment conflict might arise.

to the "old" variant.) In the other case, the programmer manually needs to restore the ordering of the children. This procedure is shown in the form of an algorithm in Algorithm 3.

Algorithm 3 InsertNewAsset

```
    input newAsset: The asset to insert ($\mathcal{A}$)
    input oldAsset: Optional "old" asset ($\lceil\mathcal{A}\rceil$)

 1: $newParent \leftarrow \mathsf{FindOriginal}(\mathsf{Parent}(newAsset))$
 2: $\mathsf{Children}(newParent) \leftarrow \mathsf{Children}(newParent) \cup \{newAsset\}$
 3: if $\mathsf{Type}(\mathsf{Parent}(newAsset)) \in \{\mathsf{Class}, \mathsf{Method}, \mathsf{Block}, \mathsf{File}\}$ then
 4:   if $oldAsset = \bot$ then
 5:     Manually repair full ordering of children of newParent
 6:   else
 7:     for all $a \in \mathsf{Children}(newParent)$ do
 8:       if $oldAsset <_{\mathcal{A}} a$ then
 9:         define $newAsset <_{\mathcal{A}} a$
10:       end if
11:     end for
12:   end if
13: end if
```

In the algorithm we first see the new asset getting added to the "old" asset tree (lines 1 and 2). In case the ordering is irrelevant, the function ends here. Otherwise, we need to manually repair the full ordering in case of a completely new asset (line 5), or we programmatically set the new ordering in case of an edit action (lines 7 until 11).

A final remark on the process of the *put* operator is that we do not do anything with the folded feature models and presence conditions that were put in the feature model of the resulting root asset of the *get* operator. We do not process this change after applying the *put* operator because these

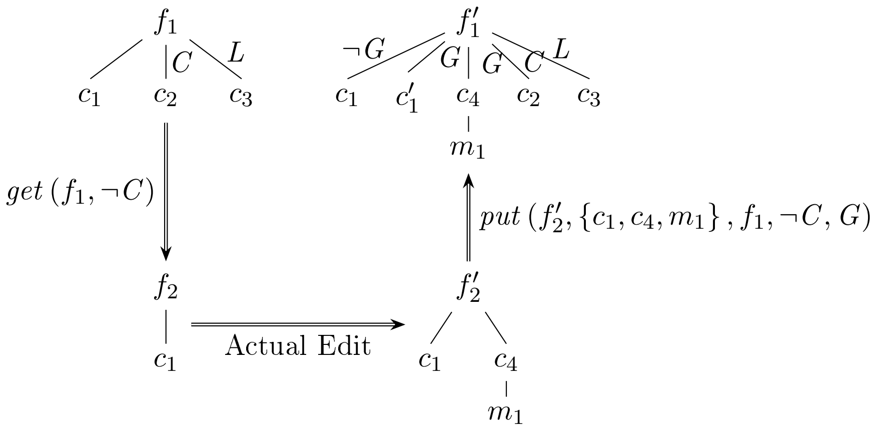


Figure 3.4: Example application of the *put* operator.

changes are not fundamentally incorrect, even when they are copied over to the original tree. Note that this copying over can only happen when the root of the new tree is changed, which will most definitely be an uncommon occurrence. The copying over is not fundamentally wrong since, at this point of the asset tree, we already know that the conditions in this feature model must hold (they came from ancestors.) It is valid to simplify a feature model at some point in an asset tree if (part of) it can already be concluded from feature models and presence conditions higher up in the tree. If this simplification is always attempted at all assets in the tree, the "not cleaning up" of the feature models is irrelevant.

Example 3.4 (Continuation of Example 3.3)**.** We continue the example in which we applied the *get* operator to the system we introduced even before that. Specifically, we will continue the second example where we applied the *get* operator with as choice expression $\neg CLI$. An overview can be seen in Figure 3.4. Note that we shortened the names of the features to their first letters (C stands for CLI and L stands for *logging*). The actual edit this time is to add a new feature called GUI (in the figure shortened to just G). To do this, we have to edit a_1 (identified by c_1) and we have to add two new assets identified by c_4 and m_1 respectively. The latter addition is a child of the first one. Now when we apply the *put* operator, we apply it with as ambition G, since we want these changes to apply to the GUI feature. The operator first filters the changed assets set to only contain the relevant assets. This filter removes m_1 from the set since it is a child of c_4. We can do this becauase this child asset will be automatically applied when applying its parent (c_4). We first apply c_1, which is simple as it is just an edit action. The old asset from the original tree used to have the trivial presence condition, this becomes the negation of the ambition. The new version of c_1, in the result identified as c_1', gets as presence condition the ambition. Lastly, we need to apply c_4, this leads to an alignment conflict as we do not know if we should add it before c_2, after c_2 or after c_3. We do not know this because c_2 and c_3 were hidden in the view. The programmer did not see these classes when they created the new class. Since this process now requires a human in the loop, we magically decide to put it right after c'_1. This new asset also gets the ambition as its presence condition. Note that the Figure resembles the same shape as the figure showing the lens overview before (Figure 2.1). In that figure, we also displayed the arrows in reverse and showed that they corresponded with the lens laws. How the lens laws work for our work is explained in the following section.

Chapter 4

Variational Lenses

In this chapter we present a new lens definition building on the formalisations of the *put* and *get* operators we have created in the previous section. We will expand on the lenses we have introduced in Section 2.3.

4.1 Configuration

If we want to constrain our *get* and *put* operators from Sections 3.4 and 3.5 into a lens, we need to create a new kind of lens. The reason is that the typing of the standard lenses constrain us too much in terms of the types they have. Our new arguments cannot fit into this old definition.

To allow for this new lens to be applicable beyond the scope of the Virtual Platform, we extract more general types from our formalisations of the operators:

$$\begin{array}{lcl} \mathit{get} & : & C \times E \to A \\ \mathit{put} & : & D(A) \times C \times E \times E \to C \\ \mathit{create} & : & A \times E \to C \end{array}$$

The *get* function now takes two arguments, first we see the concrete structure as we have seen it before, but now it comes with an E: this part carries the configuration information (in our case, the expression or choice.) The result of the *get* function still is the abstract structure as before. Note that we *will* show the definitions of the *create* functions here, but we will not go into the implementation of this function. This implementation would involve the creation of a new system in the Virtual Platform starting from an existing codebase.

$$\begin{tikzcd}[column sep=huge, row sep=large]
s \arrow[rrr, dashed, "\text{Derived edit}"] \arrow[d, bend right=50, swap, "get\,(s,c)"] & & & s' \arrow[d, dashed, bend right=20, swap, "get\,(s', c \otimes a)"] \\
v \arrow[u, dashed, bend right=20, swap, "put\,(empty\,(v)\,, s, c, c)"] \arrow[rrr, swap, "\text{Actual edit, giving } d(\cdot)"] & & & v' \arrow[u, bend right=50, swap, "put\,(d(v'), s, c, a)"]
\end{tikzcd}$$

Figure 4.1: Overview of operator relationships.

The *put* function changes similarly, we still see the concrete argument, and we have again added configuration arguments. We both have the configuration argument that was applied to the *get* operator (the *choice*) and the configuration used in the *put* operator, which for us is the *ambition*. The most important change here is that the abstract argument has changed into $D(A)$. This D stands for *diff*, as we need the abstract argument combined with some sort of diffing structure. This can be seen as a list of changes made to the view. What is important is that we need to have a function $empty : A \to D(A)$ that can transform an abstract structure into a diffing structure. This function should, as the name suggests, deliver an empty diff structure. We will use this function in new lens laws.

Finally, we have the *create* function, this function has changed from the original in the same way the *get* function changed, we have only added a configuration argument E to it.

Since we have changed the signature of the lens functions, we also have to adapt the lens laws accordingly:

$$\begin{array}{ll}
put\,(empty\,(get\ c\ e))\ \ c\ e\ e = c & \textsc{GetPut} \\
get\,(put(d(a)\ c\ e_g\ e))\ \ (e \otimes e_g) = a & \textsc{PutGet} \\
get\,(create\ a\ e)\ \ e = a & \textsc{CreateGet}
\end{array}$$

Here, we have that e_g is the configuration used to get to the view, which is put back using the *put* function. How the first two laws interact with the system as a whole can also be seen in Figure 4.1.

4.2 Virtual Platform

We want to work towards proving the defined lens laws for our instance of the lens. Before we start working on that, let us first look at how we should fill in the abstract types in order to fit our *get* and *put* functions:

$$\begin{aligned}
&C = \mathcal{A} \\
&E = Exp\,(\mathcal{F}) \\
&A = \lceil \mathcal{A} \rceil \\
&D(A) = \lceil \mathcal{A} \rceil \times \mathcal{P}\,(\mathcal{I})
\end{aligned}$$

Indeed, if we fill in these types for the different functions we get:

$$\begin{aligned}
&get &&: \mathcal{A} \times Exp\,(\mathcal{F}) \rightarrow \lceil \mathcal{A} \rceil \\
&put &&: \lceil \mathcal{A} \rceil \times \mathcal{P}\,(\mathcal{I}) \times \mathcal{A} \times Exp\,(\mathcal{F}) \times Exp\,(\mathcal{F}) \rightarrow \mathcal{A} \\
&create &&: \lceil \mathcal{A} \rceil \times Exp\,(\mathcal{F}) \rightarrow \mathcal{A}
\end{aligned}$$

We want to minimise the restrictions put on the ambition in the *put* operator. But if we want to stay true to the lens laws defined just above, we need at least some limitation, to illustrate the issue, see Figure 4.2. This example is structured much like in Figure 4.1, where we start in the top-left corner and work our way to the top-right corner. We start with an asset tree with three children, with A, $\neg B$ and $\neg A$ as presence conditions respectively. To get our view, we take the entire tree and *get* it using A as our choice expression. This leads to the removal of the third child as $\neg A \wedge A \notin SAT$. Next, we edit our view such that a_1 is changed and a_4 is added. Finally, we want to *put* our new asset tree back into the original asset tree. To do this, we choose as ambition expression just B. Note that the changed asset set contains a_1 and a_4, for now, we assume that the identities of the assets a_1 and a_4 have the same name. The result of our operation is as expected, we now have two versions of a_1, one is the original (a_1) and the new one is the edited version (a_1'). We also see that a_4 now has B as its presence condition and that it is placed after the previously hidden a_3. Now, we run into the following problem: the PutGet law says that we need to have some expression e such that $get(r_3, e) = r_2'$. Currently, this is not possible, as we want both a_1', a_2 and a_4 in our result. In other words, we do not want a_1 and a_3. The way to do this is to choose $e = A \wedge B$. This does not work for us though, as it would lead to r_2' without a_2, as $\neg B$ would not hold anymore.

The example shows that we need some limitation in terms of what expression you can choose for the ambition. To satisfy the lens laws we restrict the ambition in the *put* operator in the following way: for every asset a in $\mathcal{A}_{new}$, with am being the ambition and ch being the choice expression:

$$ch \wedge am \wedge \mathsf{PC}(a) \in SAT$$

This says that the choice expression, together with the ambition and any asset presence condition must be satisfiable. We will now show that this limitation indeed makes our definition of *put* adhere to the lens laws.

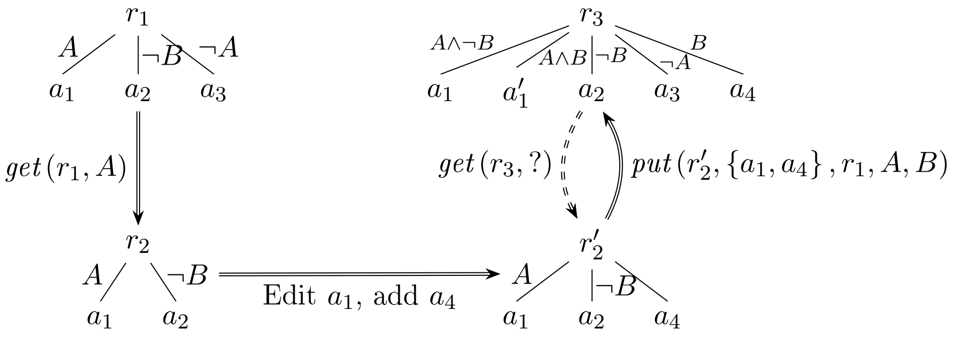


Figure 4.2: Example of relationships between lens operators on the Virtual Platform showing why we need a restriction on the ambition expresion.

4.3 Proving the lens laws

We want to show that the proposed lens laws hold for the formalisation of the operators created for the virtual platform. We will start with the GETPUT law and then prove the PUTGET law.

Theorem 4.1. *The formalisations of* get *and* put *in Sections 3.4 and 3.5 adhere to the* GETPUT *law.*

Proof. As we know, the GETPUT law says that when we use the *put* operator directly after the *get* operator, we should result in the same asset tree. Our law uses an empty function that needs to be defined on the $D(A)$ type. So in our case we need a function $empty : \lceil \mathcal{A} \rceil \to \lceil \mathcal{A} \rceil \times \mathcal{P}(\mathcal{I})$. This is a simple function that takes an optional asset and returns that asset in combination with an empty set. So $empty\ x = x, \emptyset$. By definition, the *put* operator only makes changes to assets in the set of changed assets. Our *empty* function always creates an empty set of changes and thus no changes will be applied. This ultimately means that the result of the *put* operator, when applied with an empty change set, will always be the concrete argument given to it. This is even stronger than the lens law we need to prove: in our method, the "ambition" argument to the function may even differ from the "choice" argument:

$$put\,(get\ c\ e_1)\ \emptyset\ c\ e_1\ e_2 = c$$

This proves that the GETPUT law is always satisfied. □

Theorem 4.2. *The formalisations of* get *and* put *in Sections 3.4 and 3.5 adhere to the* PUTGET *law.*

Proof. The PUTGET law states that we should be able to get back to a possibly edited view from the result of the *put* operator applied on that edited view. We of course know the typing of the operators on the Virtual Platform and thus can fill in our definitions of $d(a)$ and $\otimes$. The $d(a)$ we can replace with two arguments, one for the asset (a) and one for the set of changed assets ($\mathcal{P}(\mathcal{I})$). The binary operator combining the configurations is replaced with the conjunction operator ($\wedge$):

$$\mathit{get}\,(\mathit{put}(a\ i\ c\ e_g\ e))\ \ (e \wedge e_g) = a$$

We start off by stating that all assets a' in the tree of a have a presence condition such that:

$$e_g \wedge \mathsf{PC}(a') \in \mathit{SAT} \tag{4.1}$$

We know this since that is what the *get* operator used to retrieve the assets. And by the restriction we introduced earlier, we also know that all assets a' in the tree of a have that just *before* applying the *put* operator:

$$e_g \wedge e \wedge \mathsf{PC}(a') \in \mathit{SAT} \tag{4.2}$$

If this does not hold for any of the assets in the tree, we cannot apply the operator. Note that this second equation is a stronger version of the first one. Also note that this second equation contains $e_g \wedge e$, which is exactly what the *get* operator in the law uses to obtain the result. This means that all the assets that are visible in a will also be obtained after applying the *get* operator. But since the *put* operator can change the presence conditions of the assets, we have to look at the possible changes it can make before we can conclude that the law holds.

1. If an asset was edited, we want that after the *get* operator, the edited version is visible, not the original version of that asset. We are sure that this will happen since:

   - The "old" asset used to satisfy (4.1), but will be changed such that the presence condition of it is combined with the negation of the ambition ($\neg e$), that means that the new presence condition can never satisfy the *choice* expression of the final *get* operator.
   - The "new" asset already satisfied (4.2) before applying the *put* operator, and it will only be changed such that the ambition is combined with the ambition (e), this means that (4.2) will keep holding.

2. If an asset was added, it satisfied (4.2) before the *put* operator, and it will only get the ambition (e) added to its presence condition, so similar to what we have seen before, (4.2) will keep holding. This means that the added asset will indeed be in the result of the *get* operator.

3. If an asset was deleted, the “old” asset used to satisfy (4.1), but similarly to what we have seen before, this asset cannot be in the result of the *get* operator, since it gets the negation of the ambition ($\neg e$) appended to its presence condition.

This means that any changes in the asset tree cannot lead to the assets not being in the result of the *get* operator. We now still have to show that any assets that did not show up after the initial *get* operator, will also not be in the result of the *get* operator as seen in the lens law. We know that this is the case as these “hidden” assets did not satisfy (4.1) and the choice expression used in the *get* operator in the lens law uses a stronger version, namely (4.2). That means the initially hidden assets will stay hidden as their presence conditions have not been affected (the *put* operator only changes the presence conditions of visible assets.)

At this point, we know that all the assets in the view will stay in the view after applying the final *get* operator. We can also conclude that the (vertical) order does not change (the parent-child relationships), since the *put* operator does not change those relationships. One final thing we should show is that alignment conflicts cannot lead to problems with this law.

As we know, alignment issues can happen when we create new assets as siblings of assets that have their siblings hidden by the *get* operator. It does not matter where the programmer decides to place this new asset, however. This is because as we have already seen, the hidden assets will stay hidden after the final *get* operator application. This means that the location of the new asset does not change relatively from the other shown assets.

With this final knowledge, we can say for sure that the asset tree in the view does not change if we apply the *get* operator after a *put* operator if we choose the ambition as elaborated on, proving that the PUTGET law holds. □

Example 4.1 (Continuation of Example 3.4)**.** In the previous example, where we have shown how the *put* operator works, we already hinted on that the figure visualising the example (Figure 3.4) resembles the figure visualising how a lens works (now Figure 4.1). Let us now exemplify the arrows defining the lens laws by applying the laws. We start with the easiest, the GETPUT law. This one is easy, because our *put* operator works by looping through all the changes, of which we have none. The *empty* function that is applied on the result simply gives back an empty set of changes. Because of this, we result in the original view f_1.

The PUTGET law is somewhat more involved. We want to go from f_1' to f_2' using an application of the *get* operator. The *choice* in this case, is a combination of the original choice and the ambition. We have also already figured out that this combination is a conjunction operator ($\wedge$). Since our original choice was $\neg C$ and the ambition was G, our new choice is $\neg C \wedge G$ and the full application of the *get* operator looks like $get\,(f_1', \neg C \wedge G)$. We know that the asset labelled by f_1' will pass through the *get* operator as it does not have a presence condition (and we know that the feature model is valid with our new choice). We can look at every child of this local root asset and see which ones are valid for the *get* operator:

- The asset labelled by c_1 does not pass the operator, as its presence condition is $\neg G$. This presence condition is not satisfiable with our choice.
- The asset labelled by c_1' does pass the checks, as its presence condition (G) is satisfiable with our choice (and the feature model).
- The asset labelled by c_4 has the same presence condition as c_1' and will thus also be in the result of the *get* operator. We already know that the child of this asset (m_1) will also be in the result set as it does not have any further presence conditions or feature models.
- The asset labelled by c_2 will not be in the result set as its presence condition contradicts the *choice*.
- The asset labelled by c_3 will also not be in the result set. Its presence condition contradicts the choice in combination with the feature model of the system.

As is clear, we end up with f_1', c_1' and c_4. These are exactly the assets that are also present in the edited view f_2'.

Chapter 5

Implementation

In this chapter, we will present the implementation of the operators in the Virtual Platform. The implementation of the Virtual Platform is not a one-to-one match of the formalisation created in this work, our formalisation abstracts away some details that are present in the implementation. We aim to give an overview of our implementation. After the overview, we go into more detail of selected, particularly interesting parts of the implementation.

The implementation itself is written in Scala[1] and can be found in the source code of the Virtual Platform[2], which is open source. The Virtual Platform contains many different operators, the *get* and *put* functions are thus added as two new operators.

5.1 *Get* Operator

In Listing 1, we can see the source code for the *get* operator. The first check in the operator is the availability of a feature model somewhere in the ancestors of the asset that the operator is applied on. We need this to save the folded feature models and presence conditions. This is also a difference between the implementation and the formalisation: the formalisation sees the feature model as an expression, while the implementation sees the feature model as a tree of features, together with a list of expressions that can be used as cross tree constraints. We want to access the cross-tree constraints and thus need some feature model to save them into. This check can be found in lines 2-5.

[1]`https://www.scala-lang.org/`

[2]`https://bitbucket.org/easelab/vp/src/master/`

Listing 1 Implementation of the *get* operator

```
override def perform(): Boolean = {
  if (getImmediateAncestorFeatureModel(Some(asset)).isEmpty) {
    log.error("Cannot apply operator on an asset without a
        feature model")
    return false
  }

  val assetClone = asset.doClone()

  var parentFM: Option[FeatureModel] = None
  if (assetClone.parent.isDefined) {
    parentFM = assetClone.parent.get.foldFeatureModels()
  }
  if (assetClone.featureModel.isDefined && parentFM.isDefined) {
    assetClone.featureModel.get.crossTreeConstraints :+=
        parentFM.get.asExpression
  } else if (assetClone.featureModel.isEmpty) {
    assetClone.featureModel = parentFM
  }

  if (assetClone.parent.isDefined) {
    assetClone.featureModel.get.crossTreeConstraints :+=
        assetClone.parent.get.foldPresenceConditions()
  }

  assetClone.featureModel.get.crossTreeConstraints :+= choice

  val result = applyChoice(assetClone)

  if (result.isEmpty) {
    log.warn("The given choice resulted in an empty asset tree")
    false
  } else {
    val root = asset.getRoot()
    if (root.isDefined) {
      root.get.unassignedAssets :+= result.get
    }
    true
  }
}
```

To continue, we first clone the target asset, such that we can use it to change it without changing the original asset. This cloning is done using the `.doClone()` method in line 7.

Next, we fold the parent feature models using the `.foldFeatureModels()` method, this method results in an optional new feature model. We want to save this folded feature model in either the cross tree constraints of the feature model of the current asset, or as the feature model of the current asset in general. This is depending on if the current asset already has a feature model. This is done in lines 9-17. The same is then done for the presence conditions in the ancestors of the current asset. This time we do not have to check if the current asset has a feature model, as we already know this from the previous step. The folded presence conditions are saved in lines 19-23.

Lastly, the actual pruning of the assets is done using the `applyChoice()` method. We will cover this in more detail next, assume for now that this method results in an optional asset (line 25). Then in lines 27-36 the result is parsed. If there is no result (the target asset was not included in the result), the user is warned about this. Otherwise, we save the new asset in the *unassignedAssets* of the root asset. This is another difference between the implementation and the formalisation, we want to keep the full tree connected in the implementation. To do this, we have to include the cloned asset in the original asset tree. This is done by attaching the cloned assets to the *unassignedAssets*.

Listing 2 Implementation of the *applyChoice* method

```scala
private def applyChoice(localAsset: Asset): Option[Asset] = {
  var expression = localAsset.presenceCondition & choice
  if (localAsset.featureModel.isDefined) {
    expression &= localAsset.featureModel.get.asExpression
  }
  if (SATSolver.solve(expression) == UNSAT()) {
    None
  } else {
    val cloneAsset = localAsset.doClone()
    cloneAsset.children = localAsset.children.flatMap(
        applyChoice)
    Some(cloneAsset)
  }
}
```

5.1.1 The *applyChoice* Method

The workings of the *get* operator have been explained in the previous section, but this explanation skips over the most important function of the operator.

This function actually does the filtering of the assets to decide which ones should end up in the result. It is called *applyChoice* and its implementation can be found in Listing 2.

The function takes one argument, which is the asset that we are currently looking at (*localAsset*). We first want to create the expression which we want to check for satisfiability, we do this by creating a conjunction with the *choice*, the local presence condition, and the local feature model. The feature model is optional, so we have to do an extra check before using it. This expression is built in lines 2-5.

Next, we actually apply a SAT solver to check the expression for satisfiability, for this, we use the SAT solver available in the Virtual Platform. If this is not satisfiable (the result is *UNSAT*), we return *None* to indicate that the asset should not appear in the result asset set. If the expression was satisfiable, we want to return the asset. Before returning it however, we want to apply the filtering function on the children of the asset.

5.2 *Put* Operator

Listing 3 Implementation of the *put* operator

```
override def perform(): Boolean = {
  if (!doSanityChecks()) return false
  filterChangedAssets()

  for (changedAsset <- changedAssets) {
    if (targetAssetMap.contains(changedAsset)) {
      targetAssetMap(changedAsset).presenceCondition &= !ambition
    }
    if (sourceAssetMap.contains(changedAsset)) {
      sourceAssetMap(changedAsset).presenceCondition &= ambition
      insertNewAsset(sourceAssetMap(changedAsset))
    }
  }

  val targetRoot = targetAsset.getRoot().get
  targetRoot.unassignedAssets = targetRoot.unassignedAssets filterNot {_.name == asset.name}

  true
}
```

The main function of the implementation of the *Put* operator closely matches the definition we gave in pseudocode in Algorithm 1. The operator starts by doing a number of sanity checks on the assets on which the operator was applied. These checks make sure that the changed assets are valid, the target asset is actually the source of the new asset, if the new asset is correctly created using the *get* operator, and finally if the new asset tree satisfies the limitation we created for the *put* operator. This is done in line 2.

In the next line, we filter the changed assets, such that changed assets that are children of another changed asset are removed from the list (line 3). The following for loop enumerates the changed assets and applies the necessary changes to the original asset tree. At this point it should be noted that even before calling the *put* operator, two maps are created to easily access the old and new assets from their respective asset trees. These are called the *targetAssetMap* for the original asset tree and *sourceAssetMap* for the new asset tree. These maps can be used to check if the changed assets are edited, created or deleted. The necessary changed are made to the presence conditions (adding the ambition or the negation of the ambition) and the new asset is inserted in the asset tree using the `insertNewAsset` method in case this is needed. We will go into more detail of this function later. This process can be found in lines 5-13. Lastly, we remove the cloned asset tree from the *unassignedAssets* field of the target asset tree, as the *get-put* cycle is now complete (lines 15-16).

5.2.1 The *insertNewAsset* Method

Before, we have seen the general structure of how the *put* operator is implemented. Let us now look at the most important function of the implementation, namely the one that actually inserts the new assets into the source asset tree. This function is called *insertNewAsset*, it takes as an argument the asset to add (called *newAsset*). Its implementation can be found in Listing 4.

The method is mainly split up into three cases: the first case is an edit operation, in which we already know where to place the new asset (lines 2-5). The second case is the case where it is an addition of a new asset, where the ordering is important (lines 6-20). The last case is a new asset where the ordering does not matter (lines 21-24).

Edit Operation

The first case is an edit operation of an asset. We have a new asset to add, and we want to place it right beside the asset that was edited. An asset can be added using the `insertInList` method, for which we only need the new

Listing 4 Implementation of the *insertNewAsset* method

```scala
private def insertNewAsset(newAsset: Asset): Unit = {
  if (targetAssetMap.contains(newAsset.name)) {
    val oldAsset = targetAssetMap(newAsset.name)
    val oldAssetIndex = oldAsset.parent.get.children.indexOf(
        oldAsset)
    oldAsset.parent.get.children = Put.insertInList(oldAsset.
        parent.get.children, newAsset, oldAssetIndex + 1)
  } else if (Put.childrenAreOrdered(newAsset.parent.get.
      assetType)) {
    val targetParent = targetAssetMap(newAsset.parent.get.name)

    val lowerBound = ...
    val upperBound = ...
    var newIdx = -1
    if (lowerBound == upperBound) {
      newIdx = lowerBound
    } else {
      do {
        print(f"Please select an index for asset ${newAsset.name
            }")
        newIdx = scala.io.StdIn.readInt()
      } while (newIdx > upperBound || newIdx < lowerBound)
    }
    targetParent.children = Put.insertInList(targetParent.
        children, newAsset, newIdx)
  } else {
    val targetParent = targetAssetMap(newAsset.parent.get.name)
    targetParent.children = Put.insertInList(targetParent.
        children, newAsset, idx = 0)
  }
}
```

index of the asset. To get the index, we first get the original of the edited asset in line 3. With this asset, we can decide the index using the `indexOf` method in line 4. The insertion in the list of children is then done in line 5.

Ordered Addition Operation

As mentioned in the formalisation, when an asset is added and we have multiple choices for the new location, we have to ask the programmer where they want to place the new asset. We first decide the lower- and upper bounds for the location in lines 7-11. If these bounds are equal, we have only one possible location and do not have to ask the programmer for help (lines 12-13). Otherwise, we ask the programmer for help (lines 14-19). Note that the message the programmer gets is simplified in the listing shown here. Lastly, we insert the new asset in the chosen location in line 20.

Unordered Addition Operation

Lastly, we have the unordered addition operation. This is used for assets that do not have to be ordered (such as files). Here, we simply add the new asset to the first position of the list of children. This is the simplest variant as no index calculation has to be done. We can simply get the target parent and add the new child to it (lines 21-24).

5.3 Implementation Remarks

In this chapter, we have seen the major parts of the implementation of the operators. One last remark that should be made is the shortcoming of the current implementation. The Virtual Platform of course does not work without serialisation and deserialisation of the code that is represented using the asset trees. Without these steps, editing the code is near impossible. We have not implemented the (de)serialisation of the cloned asset trees, because the Virtual Platform currently cannot (de)serialise presence conditions that are not disjunctions of features. We need more complex presence conditions to support our operators. Before support is created for (de)serialisation of presence conditions in the shape of any expressions, our created operators are not fully finished. The core logic is still implemented and it can still be used as a proof of concept, simply without actually having the source files generated.

Example 5.1 (Continuation of Example 3.4)**.** In this example, we will recre-

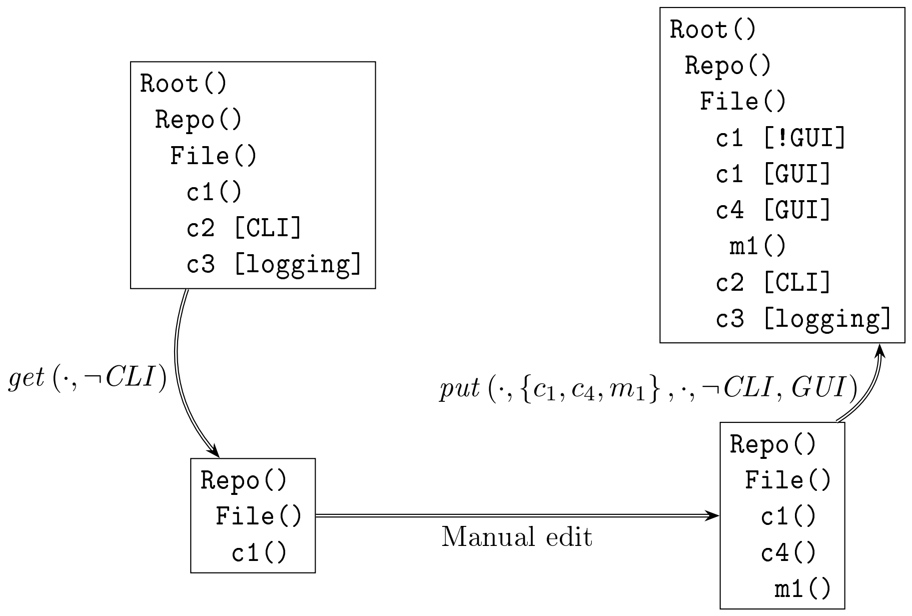


Figure 5.1: Example 3.4 recreated using the Virtual Platform.

ate the example of the *put* operator in the implementation in the Virtual Platform. The result of the previous example of the *put* operator can be found in Figure 3.4. The visualisation that the Virtual Platform gives is seen in Figure 5.1. Here, a tree structure is shown of how the assets relate to each other. An identation means that the asset is a child asset of the asset above. The name of the asset is shown together with an optional presence condition between square braces. It can be seen that this represents the same structure as seen in the previous example using the *put* operator from the formalisation. The only thing that the Virtual Platform asked of us is that we have to give a location for c_4, as it can be put it multiple places: this is the alignment conflict. The way the Virtual Platform asks this is as follows:

```
An alignment conflict has occurred!
1-> c1 <-2-> c2 <-3
Please select an index for asset c4 (within the range [1, 3]):
```

This represents the children of the *File* asset, we see c_1 and c_2 together with numbers pointing to them. We can now choose either 1, 2 or 3 to select a location: 1 to place the new asset *before* c_1, 2 to place it *after* c_1, and 3 to place it *after* c_2. We choose 2 here, to adhere to the example.

Chapter 6

Evaluation

This chapter will cover the evaluation of the operators we designed and formalised in Chapter 3. We will perform this evaluation by looking at common editing operations in software systems and seeing how well our system applies to these operations.

Note that we only referred to Chapter 3 as to what this chapter will cover. This is because we only cover the formalisation of the operators here, not the implementation (from Chapter 5). The implementation is not evaluated at large, given the scale of the thesis. The evaluation that *is* in place for the implementation are the unit tests created for it.

6.1 Context and Methodology

The editing operations we will look into are a collection of fourteen editing operations, structured into three groups (code-adding, code-removing and other), that have been identified in [29]. These editing operations have been extracted from commit data from the Marlin 3D printer firmware[1]. This was a suitable repository as it is highly configurable and large (more than 140 features and 40,000 lines of code). The code-adding patterns are for adding variability (P1–P3), partly wrapping existing code with variability (P4 and P5), adding code without variability (P6), and repairing annotations (P7). Code-removing patterns exist for removing non annotated code (P8), removing code with variability (P9), and for repairing annotations (P10). The other patterns include wrapping and unwrapping code with and from their variability (P11 and P12), changing presence conditions of code (P13),

[1]`https://github.com/MarlinFirmware/`

$$\begin{tikzcd}
r_1 \arrow[d, -] & & & r_3 \arrow[dl, -] \arrow[dr, -, "A"] & \\
a_1 \arrow[d, Rightarrow, "get\,(r_1, true)"'] & & a_1 & & a_2 \\
r_2 \arrow[d, -] & & & r'_2 \arrow[u, Rightarrow, "put\,(r'_2, \{a_2\}, r_1, true, A)"'] \arrow[dl, -] \arrow[dr, -] & \\
a_1 \arrow[rr, Rightarrow, "\text{Manual edit}"'] & & a_1 & & a_2
\end{tikzcd}$$

Figure 6.1: Workflow for adding a new asset, relevant for patterns *P1* through *P3*.

and moving code between different variability blocks (P14). Our evaluation is structured along the groups.

In this evaluation, we follow the same process as Stănciulescu et al.: we take a look at every edit operation and see how our method supports them. Supporting in this context means that we can fulfill the edit operation with our definitions of the *get* and *put* operators. We will see that this comes down to picking suitable choices and ambitions. We keep the same naming scheme for the edit operations from the previous work, even though they are centred around `#ifdef` statements. This is acceptable, because our presence conditions can be seen as `#ifdef` statements and vice versa. This naming scheme also eases comparability, which we will do after applying the edit operations. For our system to be usable, we want to be able to support all of the operations. We will also take a closer look at the restriction we placed on the *put* operator, as we do not want it to restrict us from using any common edit pattern.

6.2 Code-Adding Patterns

P1 AddIfdef, P2 AddIfdef*, P3 AddIfdefElse

The first three patterns are all grouped since they have a common goal: adding new code with a presence condition. They were previously split up because a distinction was made between adding one and multiple assets, and adding just an `#ifdef` and adding an `#ifdef` combined with a `#else` clause. For us, adding new code always means adding a new asset. To do this, we can *get* the tree with a trivial choice (`True`) and *put* the tree back with as ambition the desired presence condition. We have to take care however, there is a limitation on the *put* operator. We cannot have any asset in the editing

$$\begin{tikzcd}
r_1 \arrow[d, dash] & & & r_3 \arrow[dl, dash, "\neg A"'] \arrow[dr, dash, "A"] & \\
a_1 \arrow[d, Rightarrow, "get\left(r_1, true\right)"'] & & a_1 & & a_1' \\
r_2 \arrow[d, dash] & & & r_2' \arrow[d, dash] \arrow[ul, Rightarrow, "put\left(r_2', \left\{a_1\right\}, r_1, true, A\right)"'] & \\
a_1 \arrow[rrr, Rightarrow, "\text{Manual edit}"'] & & & a_1 &
\end{tikzcd}$$

Figure 6.2: Workflow for wrapping assets with presence conditions, relevant for patterns *P4* and *P5*.

view that has a presence condition which is not satisfiable with the ambition. In particular, this means that we cannot have any asset a in our view such that $\mathsf{PC}\left(a\right) \wedge \mathit{ambition} \notin \mathit{SAT}$. This process can be seen in Figure 6.1.

P4 AddIfdefWrapElse, P5 AddIfdefWrapThen

These edit operations are for the cases where an existing asset without a presence condition needs to get a presence condition, and another asset is created with the negation of that presence condition. As with the previous system, we can support this workflow by applying *get* with the trivial expression True, to then edit that asset and apply the *put* operator with the ambition set to the desired presence condition of the edited line. This process can also be seen in Figure 6.2.

P6 AddNormalCode

In this edit operation, no changes are made to variability. In this case, we should use the *get* operator with as choice the presence condition of the code we want to edit (this might be the trivial choice.) After editing we should again use the same ambition as choice to ensure we do not change any presence conditions.

P7 AddAnnotation

This edit operation is not relevant to us, it is about repairing broken preprocessor annotations. We do not work with `#ifdef` statements, however.

$$\begin{tikzcd}
& r_1 \arrow[dl, dash, "A"'] \arrow[dr, dash, "B"] \arrow[dd, Rightarrow, "get\,(r_1, true)"'] & & & r_3 \arrow[d, dash, "A"'] \\
a_1 & & a_2 & & a_1 \\
& r_2 \arrow[dl, dash, "A"'] \arrow[dr, dash, "B"] & & & r'_2 \arrow[d, dash, "A"'] \arrow[u, Rightarrow, "put\,(r'_2, \{a_2\}, r_1, true, true)"'] \\
a_1 & & a_2 \arrow[rr, Rightarrow, "\text{Manual edit}"'] & & a_1
\end{tikzcd}$$

Figure 6.3: Workflow for removing assets, relevant for patterns *P8* and *P9*.

6.3 Code-Removing Patterns

P8 RemNormalCode, P9 RemIfdef

These patterns cover removing normal code, and code *with* preprocessor annotations. For us, there is no difference, as the presence conditions are embedded in the assets. To cover this pattern, we can apply the *get* operator with any choice that is satisfiable with the presence condition of the target asset. Then we can delete the asset to finally apply the *put* operator with any ambition such that the old presence condition in conjunction with the negation of the ambition is not satisfiable. Using as ambition True always works. A visualisation of this pattern can be seen in Figure 6.3.

P10 RemAnnotation

Similarly to *P7*, this is not a relevant operation for us. This edit operation is about removing annotations that were unintentionally left by removing some other code. These ill-formed annotations cannot occur in our system as the presence conditions are embedded in the assets.

6.4 Other Patterns

P11 WrapCode

This pattern wraps non-variational code (with a trivial presence condition) with some presence condition. The way we can do this is by obtaining the asset with some choice (for example, the trivial choice), removing the asset, to then *put* the changes back using an ambition equal to the negation of the

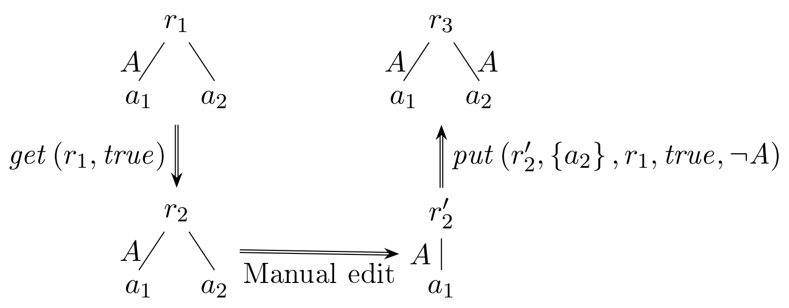


Figure 6.4: Workflow for giving an asset a non-trivial presence condition, relevant for pattern *P11*. In this case, we want to give a_2 a presence condition of A.

desired presence condition. Deleting an asset makes the presence condition of the original asset a conjunction with the negation of the ambition. This pattern can be seen in Figure 6.4.

P12 UnwrapCode

In contrast to the previous pattern, this pattern removes the presence condition from an asset. That is, we want to change the presence condition such that it becomes the trivial one. We can do this by trying to add a new asset with the negation of the asset its presence condition. But the better way to support this is to remove the presence condition from the asset manually, since adding an asset with the negation of the presence condition requires copying and pasting the same code twice.

P13 ChangePC

Changing the presence condition is an operation that is limited in our system. It can be like *P11*, where we "added" onto the old presence condition, then we can delete the asset and use the negation of what we want to add to the presence condition as the ambition. But we cannot remove any parts of a presence condition by applying the *put* operator.

P14 MoveElse

Moving the `else` part of a preprocessor annotation is not a sensible action to us. Since we are not working with literal `#ifdef` annotations. For us, moving the `else` part is nothing more than changing multiple presence conditions. Which we have covered in the previous edit operations.

6.5 Comparison

We will now compare the applicabilities of the edit operations above to the previous method. The previous method uses the exact same choices and ambitions as we did to "implement" these operations. This means that even with the restriction of the *put* operator, we can still support all of the edit patterns. The first conclusion we can thus draw is that our method has at least the level of applicability of the previous method. The fact that we can apply the same choices and ambitions to reach the same goals should not be surprising. As we noted on before, we do not principally create a new method. The current work and the work that this is based on have operators with equal goals and (nearly) equal parameters. We *want* them to do the same, but we aimed to loosen restrictions of the previous work.

A specific goal of this work is to loosen the restriction of the *ambition* expression, in comparison to the previous method. To do this, we needed the *put* operator to not be reliant on the *choice* expression. We know that this is not possible, hence the need for the restriction on the *put* operator. But we still see an improvement in terms of applicability, which was already visible in Figure 3.4. The idea is that our definition of *put* only uses the *choice* expression in internal checks. When applying changes, it only uses the *ambition* expression. This is in contrast to the previous system, where new `#ifdef` statements were created with both the *choice* and the *ambition*. In particular, this means that we can create a new feature while hiding certain other features from the view, as seen in the figure. If we were to apply the same strategy in the previous method, the new code would be added with as presence condition $GUI \land \neg CLI$. This means that the new feature can only be enabled given that the *CLI* feature is not. This same idea can be used in other editing patterns, which leads to the fact that the choice used to obtain the views in the patterns *P1*-*P5*, *P8*, *P9* and *P11*-*P13* are completely free to the user, as long as the needed assets are obtained by them (their presence condition is satisfiable with the choice). Of course, the edit operations still have to admit to the restriction in the *put* operator. We will discuss the effects of this limitation in greater detail in the next subsection.

The biggest limitations we came across are most noticeable in patterns *P12* through *P14*, which were equally limited in the previous system. We should note that when working with the Virtual Platform, a separate operator to change the presence condition would be able to tackle these patterns. A final important difference with the previous system is that the new system is not (always) fully automatic: the user sometimes needs to choose the location of a new asset when an Alignment Conflict has occurred. This can happen when certain assets are not present in the view.

6.6 Put Limitation

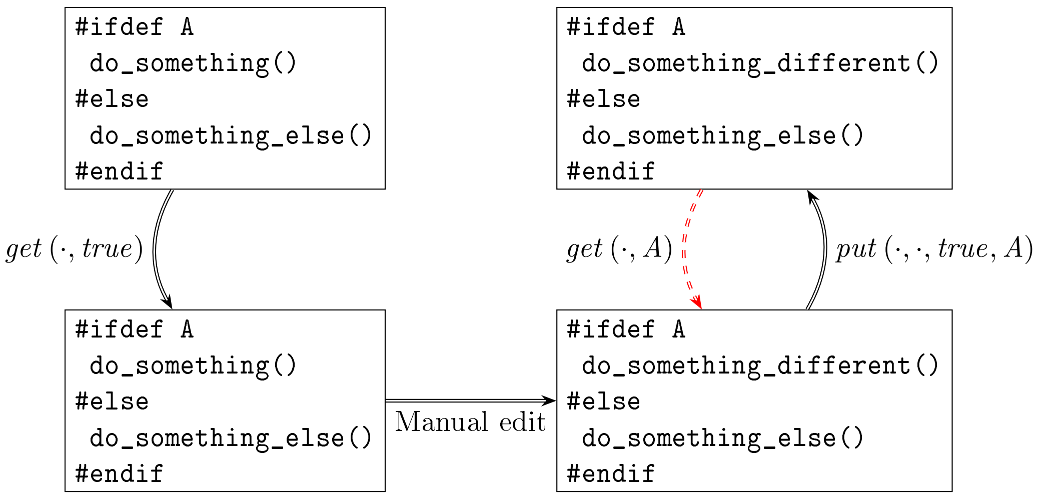
The limitation we introduced for the *put* operator in Section 4.2 ensures that the PutGet law is satisfied. In all operations we do with the *put* operator, we have to make sure that this limitation is met. This of course limits the operations we can do. We now discuss the severity of this limitation.

Again, the limitation is as follows, for every asset a in the view, the following must hold:

$$\mathit{choice} \wedge \mathit{ambition} \wedge \mathsf{PC}(a) \in \mathit{SAT}$$

Some parts of this limitation already logically follow from the *get* operator. We already know that for every asset a in the view, $\mathit{choice} \wedge \mathsf{PC}\,(a) \in \mathit{SAT}$. This is by definition of the *get* operator – if this would not hold, the asset would not be in the view. At this point we can split the limitation up into two parts. Firstly, we must make sure that $\mathit{choice} \wedge \mathit{ambition} \in \mathit{SAT}$, secondly, we must make sure that for every asset a in the view, $\mathit{ambition} \wedge \mathsf{PC}\,(a) \in \mathit{SAT}$. The first part is a single expression that we must satisfy. It limits us in such a way that we cannot have an ambition that is the negation of the choice. We can still solve those situations, however. By deleting or editing existing assets, the existing assets get the negation of the ambition added to their presence conditions. So if we were to choose the ambition equal to the choice, then the negation of the choice is – in a way – used as the ambition. The second limitation then is that every asset has to be satisfiable with the chosen ambition. This can lead to some difficulties, we might get the view with the trivial choice (True) and edit some assets. If we want to *put* these changes back with some ambition, all assets, even those not edited, must satisfy that their presence condition is satisfiable with this ambition. In other words: the presence conditions of the assets in the view must *all* be satisfiable with the ambition expression. There are two ways to avoid this: we can either apply the *get* operator to an asset such that the assets breaking this limitation are out of the view, or we can choose a *choice* in the *get* operator such that it already includes the ambition.

Since this last restriction seems rather strict, we wonder how the previous method in [29] managed to overcome this issue. In this process, we figured



```
#ifdef A
 do_something()
#else
 do_something_else()
#endif
```

```
#ifdef A
 do_something_different()
#else
 do_something_else()
#endif
```

```
#ifdef A
 do_something()
#else
 do_something_else()
#endif
```

```
#ifdef A
 do_something_different()
#else
 do_something_else()
#endif
```

Figure 6.5: Example showing how the system in [29] does not adhere to the PutGet law. The edited view (bottom-right) does not follow from the red edge coming from the edited source (top-right). Note that this same limitation applies to our system, hence the restriction we applied on the *put* operator.

that their method does not satisfy the PutGet law. With our ambition restriction applied to their method, this problem is solved. An example showing how the previous method does not adhere to the PutGet law can be found in Figure 6.5. Here, we have two preprocessor annotations, which we check out using the trivial choice *true*. We then edit the line covered by the first annotation to finally recombine the changes with as ambition A. After minimisation, we get to the code shown in the top-right. Now the PutGet law says that we should be able to go back to this view using the *get* operator with the initial expression and the ambition as the choice expression. In this case, this is just A. But then we get to the problem. When we apply the *get* operator with as choice A, we do not get this view, but rather just the line that we have edited.

A final note on the limitation in terms of the implementation in the Virtual Platform is that it is not a necessary limitation in terms of usability. The restriction is purely there for the PutGet law. If the user wants to ignore this law, they could still apply the *put* operator without satisfying the law. It just might not be possible to extract that exact view from the source again. In the implementation in the Virtual Platform, this is done using an error that can be configured to be a warning instead.

Chapter 7

Related Work

7.1 View-Based Editing

As noted above, the most closely related work is that of Stănciulescu et al., where a projection-based variation control system is designed [29]. For this, they use so-called *Choice Calculus* [32] to define the system, a formal language to model variation in software. This work relieved the restrictions on the *put* operator from previous work of Walkingshaw and Ostermann [33] by using a looser definition of the *Edit-Isolation Principle.* The principle says that the *put* operator cannot affect code outside of the view given by the *get* operator. The work of Walkingshaw and Ostermann did not make use of an ambition operator yet, which avoids the complications that we deal with in our work. For this setup, they proved the lense laws, as the present work does for the broader scope of view-based editing with ambitions.

There is a lot of research into easing programming in variant-rich software. In the work of Kästner et al., an IDE is created to visualise software variabilities in the source code. To this end, they use a tool called *CIDE* which was created in earlier work [18, 17]. They aim to help developers to understand and explore individual features. Another visualisation tool similar to *CIDE* is created by Heidenreich et al.. MappingViews is a tool that visualises different views of variational software to the programmer [13]. Nestor et al. present more work on visualisation which scales better to medium and large software product lines [24]. These tools are limited to visualisation, they do not deal with the view-update problem.

Kersten and Murphy created an Eclipse plugin called Mylar that uses task contexts to improve programmer productivity [19]. Their plugin stores structural relationships of programs in so-called task contexts. Programmers can then use these contexts to quickly swap between contexts. This tool is more of a tracing tool and does not work on the view-update problem.

This research direction has roots in earlier research on software maintainability. Weiser has shown in 1984 how *Slicing* can help understand programs [34]. With this approach, it is possible to select several variables and see the flow of the data in this program. In the same year, Linton created a way to store programs in a database such that different views, cross-sections or slices could easily be extracted or updated [21]. These tools are also for visualisations.

Chu-Caroll et al. created a system called VSC, which stands for visual separation of concerns, this system can provide certain views of the source code in terms of features [6]. VSC, like the Virtual Platform, stores programs using finer-grained artefacts. They do this on a storage-based level, such that it becomes language independent. Hofer et al. try to tackle the problem of multiple views at the filesystem level using the Leviathan filesystem [14]. The idea is that other tools all need special development environments, by having variant views at the filesystem level, other tools can be used without issues. While VSC does not work with write-back (the view-update problem), the Leviathan filesystem can do this. It is however limited to changing only non-variational code. Aside from that, it sometimes uses heuristics to decide where changed code should go in the source.

More in terms of variability management, we of course have the Virtual Platform, which aims to bridge the gap between *Clone & Own* and a full *Integrated Platform* [22]. The Virtual Platform is relevant to us, as we created our implementation onto it. Similar to the Virtual Platform, research has also been done on tools to transition from *Clone & Own* to an integrated platform, for example by Schwägerl et al. in SuperMod [26]. ECCO is a framework and tool for easing the work on variational software created by Fischer et al.. It can find common parts and create new software variants using those parts [10].

7.2 Lenses

Lenses were introduced by Foster et al. as a solution to the view-update problem that originates from databases [11]. Here, lenses were formalised and lens laws were created.

Many different types and classes of lenses have been created in the meanwhile. All of the lenses noted here differ from the first definition of lenses but are not able to carry configuration data. Hence the need for us to create *Vari-*

ational Lenses. There is a distinction between Symmetric and Asymmetric lenses [15]. Asymmetric lenses usually have a source and a view, where the view contains a subset of the information of the source. Symmetric lenses, on the other hand, have two sets that both contain some information that the other set does not have.

Boomerang, a language created by Bohannon et al., can create lenses that work between string data [4]. The user creates a lens by writing it in the specification of Boomerang and can then use it to transform strings. The language is based on regular expressions. Boomerang also contains logic to deal with alignment problems within the data. To this end, Matching Lenses were created [3] by Barbosa et al.. Matching lenses expand on the earlier definition of Dictionary Lenses which were also included in Boomerang in [4]. These lenses make use of extra data structures to save the ordering of the data, this data structure can then be used in the definition of the *put* operator. Miltner et al. have created a way to synthesise Bijective Lenses from examples of in- and output. The resulting lens is defined in the Boomerang language [23]. Bijective lenses differ from the standard lenses in that they do not hide data going from the source to the view, rather they only change the structure of it. Another extension to Boomerang is Quotient Lenses, created by Foster et al.. Quotient lenses can contain extra functions to normalise input or output data [12]. This way, spacing or newlines can be normalised before going through the *get* and *put* functions.

Monadic Lenses deal with the fact that most lenses are pure, while most programming languages have side effects. In [1], different monadic lenses are discussed.

A more applied version of a lens is for example Lenses for Web Data [25]. Here, lenses are implemented to create forms for web pages and to obtain the data from them. This is implemented in Haskell.

An example of symmetric lenses are Edit Lenses, created by Hofmann et al.. Edit lenses work by transforming between representations using descriptions of changes instead of the entire views [16].

Our new *Variational Lenses* can be seen as asymmetric lenses, as we only hide data in one direction (from the source to the view). We also take some inspiration from edit lenses, however, as we use a "diff" structure in the *put* operator. This structure can be seen as the descriptions of changes made, but in contrast to edit lenses, these structures by themselves are not sufficient to make the operator work.

Chapter 8

Conclusion

In this work, we created and formalised a lens for view-based editing. From this formalisation specifically for the Virtual Platform, we derived a more general type of lens, which we call *Variational Lenses*. This lens differs from other lenses since it can carry configuration with it and enables it to be a framework for view-based editing. In the evaluation, we have shown that the system can comply with all common edit operations that were previously established. We have also shown that this new definition allows for more general use than the most closely related previous work. In particular, we can create and edit independent features in a limited view, which was not possible in the previous work. The downside of the system is that we had to include a limitation to the *put* operator of the lens, since without it, one of the lens laws would not hold. We also established that this lens law does not hold in the previous system without this new restriction. Finally, we implemented the lens in the Virtual Platform as two new operators.

The definition of the Variational Lenses was extracted from the formalisation of the *get* and *put* operators. With this new formalisation, we had to adapt the existing lens laws for our new lens and we have proven that our implementation of the lens adheres to these adapted lens laws.

8.1 Future Work

There are several possible directions for future work. First, it would be worthwhile to study the usability of our proposed operations using a user study, in which developers could solve some program evolution tasks in a certain view and use our framework to keep the view synchronized with the overall platform. Second, while we achieved our results in the context of a specific variability management system (Virtual Platform), it would be interesting to further increase the scope of our contribution by investigating alternative program representations.

Another direction for future work could be done in terms of an editor that better supports view-based editing. It is currently challenging to save the mapping of features to lines of code, as code files are merely text files. To save mappings, we need either some form of comment (that would be visible to the developer), or an external file with the mapping. The latter of the two methods comes with the benefit that it is not visible to the user, but maintaining this line mapping is difficult when the file is edited. An editor that can save the line mappings internally, thus keeping the feature-to-line information, may improve the usability of view-based editing. We expect that it might even remove the need for ambitions in certain editing cases.

Bibliography

[1] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Reflections on monadic lenses. *A List of Successes that can Change the World*, pages 1–31, 2016.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. Software product lines. In *Feature-oriented software product lines*, pages 3–15. Springer, 2013.

[3] Davi MJ Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C Pierce. Matching lenses: alignment and view update. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 193–204, 2010.

[4] Aaron Bohannon, J Nathan Foster, Benjamin C Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 407–419, 2008.

[5] Jan Bosch. *Design and use of software architectures: adopting and evolving a product-line approach.* Pearson Education, 2000.

[6] Mark C Chu-Carroll, James Wright, and Annie TT Ying. Visual separation of concerns through multidimensional program storage. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 188–197, 2003.

[7] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 25–34. IEEE, 2013.

[8] Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(1):1–27, 2011.

[9] J-M Favre. Preprocessors from an abstract point of view. In *Proceedings of WCRE'96: 4rd Working Conference on Reverse Engineering*, pages 287–296. IEEE, 1996.

[10] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *2014 IEEE International conference on software maintenance and evolution*, pages 391–400. IEEE, 2014.

[11] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3):17–es, 2007.

[12] J Nathan Foster, Alexandre Pilkiewicz, and Benjamin C Pierce. Quotient lenses. *ACM Sigplan Notices*, 43(9):383–396, 2008.

[13] Florian Heidenreich, Ilie Savga, and Christian Wende. On controlled visualisations in software product line engineering. In *SPLC (2)*, pages 335–341, 2008.

[14] Wanja Hofer, Christoph Elsner, Frank Blendinger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Toolchain-independent variant management with the leviathan filesystem. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, pages 18–24, 2010.

[15] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Symmetric lenses. *ACM SIGPLAN Notices*, 46(1):371–384, 2011.

[16] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Edit lenses. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 495–508, 2012.

[17] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 311–320. IEEE, 2008.

[18] Christian Kästner, Salvador Trujillo, and Sven Apel. Visualizing software product line variabilities in source code. In *SPLC (2)*, pages 303–312, 2008.

[19] Mik Kersten and Gail C Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, 2006.

[20] Duc Le, Eric Walkingshaw, and Martin Erwig. # ifdef confirmed harmful: Promoting understandable software variation. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 143–150. IEEE, 2011.

[21] Mark A Linton. Implementing relational views of programs. *ACM SIGSOFT Software Engineering Notes*, 9(3):132–140, 1984.

[22] Wardah Mahmood, Daniel Strüber, Thorsten Berger, Ralf Lämmel, and Mukelabai Mukelabai. Seamless variability management with the virtual platform. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1658–1670. IEEE, 2021.

[23] Anders Miltner, Kathleen Fisher, Benjamin C Pierce, David Walker, and Steve Zdancewic. Synthesizing bijective lenses. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

[24] Daren Nestor, Luke O'Malley, Aaron Quigley, Ernst Sikora, and Steffen Thiel. Visualisation of variability in software product line engineering. 2007.

[25] Raghu Rajkumar, Nate Foster, Sam Lindley, and James Cheney. Lenses for web data. *Electronic Communications of the EASST*, 57, 2014.

[26] Felix Schwägerl, Thomas Buchmann, and Bernhard Westfechtel. Supermod—a model-driven tool that combines version control and software product line engineering. In *2015 10th International Joint Conference on Software Technologies (ICSOFT)*, volume 2, pages 1–14. IEEE, 2015.

[27] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The variability model of the linux kernel. *VaMoS*, 10(10):45–51, 2010.

[28] Henry Spencer and Geoff Collyer. #ifdef considered harmful, or portability experience with c news. In *USENIX Summer 1992 Technical Conference (USENIX Summer 1992 Technical Conference)*, San Antonio, TX, June 1992. USENIX Association.

[29] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. Concepts, operations, and feasibility of a projection-based variation control system. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 323–333. IEEE, 2016.

[30] Frank J Van der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action: the best industrial practice in product line engineering.* Springer Science & Business Media, 2007.

[31] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*, pages 45–54. IEEE, 2001.

[32] Eric Walkingshaw and Martin Erwig. A calculus for modeling and implementing variation. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 132–140, New York, NY, USA, 2012. Association for Computing Machinery.

[33] Eric Walkingshaw and Klaus Ostermann. Projectional editing of variational software. *ACM SIGPLAN Notices*, 50(3):29–38, 2014.

[34] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.