Classifying Edits to Variability in Source Code
Appendix

July 11, 2022

This appendix accompanies our submission *Classifying Edits to Variability in Source Code* submitted to the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) in March 2022.

Our appendix consists of four parts. In Section 1, we present an extended formalization of our concepts in the `Haskell` programming language to show that variation trees and variation tree diffs can be parameterized in their node types to support further language constructs. In Section 2, we prove that variation tree diffs are complete regarding edits to variation trees and that our elementary edit patterns are complete and unambiguous. In Section 3, we show that edit patterns from related work (Al-Hajjaji et al. [2016], Stănciulescu et al. [2016]) are either composite edit patterns built from our elementary edit patterns or similar to our elementary edit patterns. In Section 4, we include the complete results of our validation for all 44 datasets.

Contents

1 Extended Formalization

In this section, we show how we can parameterize variation trees and variation tree diffs by their set of supported node types, which is fixed to {`artifact`, `mapping`, `else`} in the paper. As an example, we then provide an extension of our definitions of variation trees and variation tree diffs to also support `elif` directives.

We reformulate our definitions from the paper in the `Haskell` programming language. This has the following benefits:

Error Prevention. By compiling the source code we can ensure type correctness and thus a correct encoding of our definitions.

Extensibility. `Haskell` provides suitable mechanisms to formulate possible extension points of our definitions. In particular, we can define how variation trees and variation tree diffs can be parameterized in their node types using type classes.

Explicit Requirements. `Haskell` forces us to make requirements on our inputs explicit (with type class constraints). Thus, we can and have to explicitly list all requirements we impose on the used logic and set of node types. This verifies that we indeed require only some operators (with their usual semantics).

Referential Transparency. As `Haskell` is a pure functional programming language with referential transparency, we can perform proofs using equational reasoning (i.e., substituting definitions).

Transition to Proof Assistants. `Haskell` is a language halfway between a practical language and a proof assistant, such as CoQ, Isabelle or Agda. Thus, our code will be easier to adapt to these tools should we desire to do further and more rigorous formal proofs in the future.

1.1 Logic

While we use propositional logic to map implementation artifacts to features in the examples of our paper, our concepts support any kind of logic as long as it supports conjunction $\wedge$ and negation $\neg$ (which we in fact only need for `else` nodes as we will see later) and has a neutral value *true*. We thus make the requirements to the used logic explicit such that we can later state which parts and functions require certain properties of the logic. We formulate each requirement as a type class (loosely similar to interfaces in object-oriented programming) that states that certain functions are defined for a type `f` (abbreviation of `formula`):

```haskell
class Negatable f where
    lnot :: f -> f

class HasNeutral f where
    ltrue :: f

class Composable f where
    land :: [f] -> f

class Comparable f where
    limplies :: f -> f -> Bool
```

The first type class says that a type `f` is `Negatable` if there exists a function `lnot` that takes a value of type `f` and returns a value of type `f`. A concrete implementation of `Negatable` for a concrete type `f` then has to provide an definition for `lnot` and ensure that it entails the respective semantics (i.e., a negation of a formula). Analogous, the other type classes say that a type `f` (1) has a neutral value if there exists a value `ltrue` of type `f`, (2) is composable (i.e., supports conjunction $\wedge$) in terms of an operator `land` that takes a list of values and returns their conjunction[1], and (3) is comparable if two values can be compared in terms of implication by a function `limplies` (see Section 4 in the original paper). To ensure that names are unique, we prepend each function name with `l`, which stands for `logic`. (We continue this naming scheme when necessary.)

Propositional formulas as we use in our paper and our tooling indeed satisfy all these requirements:

```haskell
data PropositionalFormula a =
      PTrue
    | PFalse
    | PVariable a
    | PNot (PropositionalFormula a)
    | PAnd [PropositionalFormula a]
    | POr [PropositionalFormula a]
    deriving (Eq)

instance Negatable (PropositionalFormula a) where
```

[1] `[x]` is syntax (sugar) for a list of values of type `x`.

```
    lnot PTrue = PFalse
    lnot PFalse = PTrue
    lnot p = PNot p

instance HasNeutral (PropositionalFormula a) where
    ltrue = PTrue

instance Composable (PropositionalFormula a) where
    land [] = PTrue
    land l = PAnd l

instance Comparable (PropositionalFormula a) where
    limplies a b = isTautology (POr [lnot a, b])
```

We define propositional formulas as a sum type that reads as follows: A `PropositionalFormula` is either (1) the value *true*, (2) the value *false*, (3) a variable with a value `a`, (4) a negation $\neg$ of a formula, (5) a conjunction $\wedge$ of a list of formulas, or (6) a disjunction $\vee$ of a list of formulas. We parameterize `PropositionalFormula`s by a type `a` that determines which values are stored in variables.[2] For example, the type `a` could be `String` if variables should be named, or `Int` if variables should be indexed. `PropositionalFormula`s support all of the four requirements we introduced which we show by providing an instance of each requirement type class. We omit the definition of the auxiliary function `isTautology` here that invokes a SAT solver on a given formula to determine whether the given formula is a tautology.

1.2 Variation Trees

We now translate our definition of variation trees from the paper to its `Haskell` equivalent which allows us to (1) make the requirements to the used logic explicit by referencing the type classes introduced in the last section, and (2) formulate the set of node types as a parameter for defining a variation tree. Let us recall our original definition:

Definition 2.2 (Variation Tree). A *variation tree* (V, E, r, τ, λ) is a tree with nodes V, edges $E \subseteq V \times V$, and root node $r \in V$. Each edge $(x, y) \in E$ connects a child node x with its parent node y, denoted by $p(x) = y$. The node type $\tau(v) \in \{\texttt{artifact}, \texttt{mapping}, \texttt{else}\}$ identifies a node $v \in V$ either as representing an implementation artifact, a feature mapping, or an else branch. The label $\lambda(v)$ is a propositional formula if $\tau(v) = \texttt{mapping}$, a

[2]In object-oriented programming, such a type parameter is usually known as a *generic* type (e.g., an equivalent Java definition would be `class PropositionalFormula<A>`).

reference to an implementation artifact if $\tau(v) = \texttt{artifact}$, or empty if $\tau(v) = \texttt{else}$. The root r has type $\tau(r) = \texttt{mapping}$ and label $\lambda(r) = \mathit{true}$. An `else` node can only be placed directly below a non-root `mapping` node and a `mapping` node has at most one child of type `else`.

To reference nodes V, we introduce a `U`nique `U`niversal `ID`entifier as an alias for `Int`:

```haskell
type UUID = Int
```

We can then define nodes, edges, and finally variation trees.

```haskell
data VTNode l f = VTNode UUID (l f)

data VTEdge l f =
    VTEdge {
        childNode  :: VTNode l f,
        parentNode :: VTNode l f
    }

data VariationTree l f = VariationTree [VTNode l f] [VTEdge l f]
```

All data types are parameterized by a label set type `l` and formula type `f`. The formula type `f` describes the used logic as introduced earlier, one possible type being `PropositionalFormula a`. The type `l` describes the set of node types, which is determined by τ in our original definition. In our paper, the set of node types is fixed to $\{\texttt{artifact}, \texttt{mapping}, \texttt{else}\} = im(\tau)$. Yet, variation trees are more general: They are also valid without `else` statements but can also be extended by further statements (e.g., `elif`). We thus model the set of available node types as the type parameter `l` here and explain requirements for it later in detail.

A `VTNode` consists of its identifier and a label of type `(l f)`, which means that the label of the node is itself parameterized in the formula type `f`. We store the type $\tau(v)$ and label $\lambda(v)$ within a node v in terms of the label `(l f)` for two reasons: First, by storing properties in nodes instead of accessing them through dedicated functions τ and λ we do not have to manually ensure that the respective functions are defined for all nodes in a variation tree (and only for those nodes). Second, the type of the label $\lambda(v)$ of a node v depends on the node's type $\tau(v)$. This implementations matches our original definition because we could define the functions τ and λ of variation trees to just return the respective values that are stored within a node, but we omit this functions for brevity here.

Edges consist of a `childNode` and `parentNode`. We define edges here as a record type instead of a simple algebraic data type `data VTEdge l f = VTEdge (VTNode l f) (VTNode l f)` to avoid confusion about which node is the child and which node is the parent.

We define variation trees as the type `VariationTree l f` that has a list of nodes `[VTNode l f]` and edges `[VTEdge l f]`.

To define feature mappings and presence conditions, we have to be able to access the parent $p(v)$ of a node v:

```haskell
import Data.List

parent :: VariationTree l f -> VTNode l f -> Maybe (VTNode l f)
parent (VariationTree _ edges) v =
    fmap parentNode (find (\edge -> childNode edge == v) edges)
```

To get the parent of a node `v` in a given `VariationTree` with `edges`, we first find the edge whose child node is `v` (via `find (\edge -> childNode edge == v) edges`) and then return the `parentNode` stored in that edge.[3]

To complete our formalization of variation trees in `Haskell`, we now define requirements for node types τ and labels λ. As mentioned earlier, the type of the label $\lambda(v)$ of a node v depends on the node's type $\tau(v)$. As we did for the used logic in Section 1.1, we define our requirements to node types and labels using a type class:

```haskell
type ArtifactReference = String

class VTLabel l where
    makeArtifactLabel :: ArtifactReference -> l f
```

[3]For those not familiar with `Haskell`: The function `find :: (a -> Bool) -> [a] -> Maybe a` takes a predicate `a -> Bool` and returns the first element in a given list `[a]` for which the predicate evaluates to *true*. In case no such element exists, `find` returns `Nothing`. In particular, the return type `Maybe a` either represents a found value (`Just a`) or represents failure in terms of the value `Nothing`. In Java, C#, C++, etc. `Nothing` would correspond to `null`. While in Java, any reference type can have value `null`, no type can do so in `Haskell`. `Maybe` is a type that explicitly makes a type nullable. To extract the `parentNode` of the found edge, we thus use `fmap parentNode` that either returns the `parentNode` of a found edge, or does nothing in case no element was found. You may read `fmap parentNode m` as `if (m is (Just a)) then (Just (parentNode a)) else Nothing`.

```
    makeMappingLabel :: (Composable f) => f -> l f

    featuremapping :: VariationTree l f -> VTNode l f -> f
    presencecondition :: VariationTree l f -> VTNode l f -> f
```

Nodes of type `artifact` and `mapping` are the basic types which we always require in variation trees. We thus require a label set type `l` to offer functions `makeArtifactLabel` and `makeMappingLabel` to create labels for `artifact` and `mapping` nodes from a reference to an artifact or a logical formula `f` respectively. We may create a label for an `artifact` node from an `ArtifactReference`, which we plainly set to string here (e.g., a file name, function name, or any other way to reference an artifact) but could be changed later. The `makeMappingLabel` function creates a label for a `mapping` node from a formula `f`. Therefore, `f` must to be `Composable` (i.e., support conjunctions $\wedge$) to be able to define feature mappings and presence conditions.[4] We make this requirement explicit using the type class constraint `(Composable f)`. Lastly, the feature mappings and presence condition of a node in a variation tree with the given label set type `l` have to be available via the functions `featuremapping` and `presencecondition`.

An example for a possible set of label types `l` was presented in our paper with the node type set $\{$`artifact`, `mapping`, `else`$\}$. The implementation of our `VTLabel` type class is given by the functions F and PC in Equations 1 and 2 respectively. We give another example for the minimal node type set $\{$`mapping`, `artifact`$\}$ here. Therefore, we use generalized algebraic datatypes (GADTs) as they allow us to add type class constraints to each constructor:

```
{-# LANGUAGE GADTs #-}

data MinimalLabels f where
    Artifact :: ArtifactReference -> MinimalLabels f
    Mapping :: (Composable f) => f -> MinimalLabels f
```

The type `MinimalLabels f` only allows for labels `Artifact` and `Mapping`. For `Mapping` nodes, we require the used logic `f` to be `Composable` as required

[4] For those not familiar with `Haskell`: We can make requirements on the argument types of functions explicit using the `=>` operator. For example, a function `foo :: (Composable f) => f -> [f]` is a function `f -> [f]` that is only defined for types `f` that are instances of the `Composable` type class. In Java, such a requirement would be expressed using `extends` in declaration of generic arguments. For example, an equivalent declaration of `foo` in Java would be `<f extends Composable<f>> List<f> foo(f x) { ... }`.

by `VTLabel` type class definition. The instance for `VTLabel` is the same as for the node type set {`artifact`, `mapping`, `else`} presented in our paper but without `else` and translated to `Haskell` here:

```haskell
import Data.Maybe (fromJust)

instance VTLabel MinimalLabels where
    makeArtifactLabel = Artifact
    makeMappingLabel = Mapping

    featuremapping tree node@(VTNode _ label) = case label of
        Artifact _ -> fromJust $ featureMappingOfParent tree node
        Mapping f -> f
    presencecondition tree node@(VTNode _ label) = case label of
        Artifact _ -> parentPC
        Mapping f -> land [f, parentPC]
        where
            parentPC = fromJust $ presenceConditionOfParent tree node
```

To obtain the feature mapping and presence condition of the parent node,[5] we make use of the following helper functions:

```haskell
ofParent :: (VTNode t f -> f) -> VariationTree t f -> VTNode t f -> Maybe f
ofParent property tree node = property <$> parent tree node

featureMappingOfParent :: VTLabel t =>
    VariationTree t f -> VTNode t f -> Maybe f
featureMappingOfParent tree = ofParent (featuremapping tree) tree

presenceConditionOfParent :: VTLabel t =>
    VariationTree t f -> VTNode t f -> Maybe f
presenceConditionOfParent tree = ofParent (presencecondition tree) tree
```

The function `ofParent` returns a formula `f` from the parent of a given node in a tree, where the formula is extracted using the given `property` function. Both `featureMappingOfParent` and `presenceConditionOfParent` make use

[5]For those not familiar with `Haskell`: The operator `name@pattern` enables pattern matching on a value `pattern` while referring to the whole value as `name`. In particular, `node@(VTNode _ label)` matches all nodes, such that we can access the node's `label` (as if we wrote just `(VTNode _ label)` in the first place without using `@`), but allows us at the same time to refer to the whole node by the name `node`.

of `ofParent` to retrieve the `featuremapping` or `presencecondition` respectively.

Finally, we define the root of variation trees. As we fixed the root r to have type $\tau(r) = \texttt{mapping}$ and label $\lambda(r) = \mathit{true}$, we introduce a constant for it, such that it is the same for all `VariationTree`s:

```haskell
root :: (HasNeutral f, Composable f, VTLabel l) => VTNode l f
root = VTNode 0 (makeMappingLabel ltrue)
```

To create the root, we require our logic `f` to have a neutral element `ltrue` such that we can fix its formula to $\lambda(r) = \mathit{true}$. Because the root is a node of type `mapping`, we have to create a respective label for it using the `makeMappingLabel` function that requires the used logic `f` to be `Composable`. Moreover, the function `makeMappingLabel` is only defined for labels, so we have to require that the given label type `l` is indeed a valid set of labels `VTLabel`. We fix the `UUID` of the root to 0.

1.3 Variation Tree Diffs

We now formulate variation tree diffs as an extension of variation trees in `Haskell`. Let us again first recall their original definition:

Definition 3.1 (Variation Tree Diff). A variation tree diff is a rooted directed connected acyclic graph $D = (V, E, r, \tau, \lambda, \Delta)$ with nodes V, edges $E \subseteq V \times V$, root node $r \in V$, node types τ, node labels λ, and a function $\Delta : V \cup E \to \{+, -, \bullet\}$ that defines if a node or edge was added $+$, removed $-$, or unchanged $\bullet$, such that $\mathrm{project}(D, t)$ is a variation tree for all times $t \in \{\mathrm{b}, \mathrm{a}\}$.

To reason about variation tree diffs, and in particular the variation trees before and after the edit, we introduced the time $t \in \{\mathrm{b}, \mathrm{a}\}$ in our paper. Moreover, we also defined a function exists that checks whether an element with a diff type from $\{+, -, \bullet\}$ exists at a certain time $t \in \{\mathrm{b}, \mathrm{a}\}$. We thus translate these definitions to `Haskell`:

```haskell
data Time = BEFORE | AFTER
    deriving (Eq, Show)
data DiffType = ADD | REM | NON
    deriving (Eq, Show)

existsAtTime :: Time -> DiffType -> Bool
existsAtTime BEFORE ADD = False
existsAtTime AFTER REM = False
existsAtTime _ _ = True
```

Analogous to our definition, we can introduce variation tree diffs as the `data` type `VariationDiff l f` that is defined the same as a `VariationTree` but additionally has the function `Delta l f` that assigns a `DiffType` to each node and edge:

```haskell
type Delta l f = Either (VTNode l f) (VTEdge l f) -> DiffType
data VariationDiff l f = VariationDiff [VTNode l f] [VTEdge l f] (Delta l f)
```

In order to be able to pass both `VTNode`s and `VTEdge`s as arguments to a function of type `Delta l f`, we set its domain to `Either (VTNode l f) (VTEdge l f)` which means that any value passed to the function must either be a `VTNode l f` or a `VTEdge l f` (realised in the paper by a set union $\cup$).

As described in our paper, every variation tree diff is designed to describe exactly two variation trees: The variation tree that existed before the edit and the variation tree after the edit. In our paper and in this appendix, we refer to these variation trees as the *projections* of a variation tree diff. We may obtain the projection a given variation tree diff at a certain time $t \in \{\mathrm{b}, \mathrm{a}\}$ with the following function:

```haskell
project :: Time -> VariationDiff l f -> VariationTree l f
project t (VariationDiff nodes edges delta) = VariationTree
    (filter (existsAtTime t . delta . Left)  nodes)
    (filter (existsAtTime t . delta . Right) edges)
```

The function `project` takes a `VariationDiff` and returns a `VariationTree` with exactly those nodes and edges from the diff that exist at time `t`. Therefore, we use the function `filter` that takes a predicate and a list and returns a list that contains all elements for which the given predicate evaluates to *true*. Here, we check that a given node or edge `existsAtTime t` which we do by obtaining its diff type via `delta`. Since `delta` does not take a node or edge as input directly, but an `Either`, we have to wrap the given node or edge first using the respective constructors `Left` and `Right`.[6]

[6]For those not familiar with `Haskell`: The data type `data Either a b = Left a | Right b` describes a generic sum type that may inhabit exactly one of two values (similar as for `PropositionalFormula` we saw earlier). A value of `Either a b` is either a `Left a` storing a value of type `a` or `Right b` storing a value of type `b`. There are multiple ways

1.4 Extension: Elif Directives

We now show that we can extend variation trees to also support `#elif` directives. While in principle, an `#elif` can be expressed as a `mapping` node below an `else` node, inspecting `#elif` directives explicitly may be desirable for increased granularity. In fact, we also include the node type `elif` in our tool `DiffDetective` for our validation. We thus introduce a new node type set called `WithElif` which includes the new node type `elif` next to `artifact`, `mapping`, and `else` nodes:

```haskell
data WithElif f where
    Artifact :: ArtifactReference -> WithElif f
    Mapping :: Composable f => f -> WithElif f
    Else :: (Composable f, Negatable f) => WithElif f
    Elif :: (Composable f, Negatable f) => f -> WithElif f
```

The labels `Artifact` and `Mapping` are defined the very same as for our `MinimalLabels` introduced earlier: We may construct an `Artifact` label from an `ArtifactReference`, and we may construct a `Mapping` label from a `Composable` formula `f`. As in our paper, `Else` labels do not hold any value but we require the used logic `f` to be `Composable` and `Negatable` to be able to define feature mappings and presence conditions of `Else` nodes. The same requirements arise for `Elif` labels but in contrast to `Else` labels, an `Elif` also stores a formula just as `Mapping` does.

We can now define feature mappings and presence conditions for this new label set by showing that `WithElif` is an instance of `VTLabel`:

```haskell
instance VTLabel WithElif where
    makeArtifactLabel = Artifact
    makeMappingLabel = Mapping

    featuremapping tree node@(VTNode _ label) = case label of
        Artifact _ -> fromJust $ featureMappingOfParent tree node
        Mapping f -> f
        Else ->            notTheOtherBranches tree node
        Elif f -> land [f, notTheOtherBranches tree node]
```

to construct such a sum type in object-oriented languages. One way (in Java) is to create an interface `interface Either<A, B> {}` with two possible implementations `class Left<A, B> implements Either<A, B> { A a; }` and `class Right<A, B> implements Either<A, B> { B b; }`.

```
    presencecondition tree node@(VTNode _ label) = case label of
        Artifact _ -> parentPC
        Mapping f -> land [f, parentPC]
        Else ->   land [
            featuremapping tree node,
            presencecondition tree (getParent (correspondingIf tree node))
            ]
        Elif _ -> land [
            featuremapping tree node,
            presencecondition tree (getParent (correspondingIf tree node))
            ]
        where
            parentPC = fromJust $ presenceConditionOfParent tree node
            getParent = fromJust . parent tree

notTheOtherBranches :: (Composable f, Negatable f) =>
    VariationTree WithElif f -> VTNode WithElif f -> f
notTheOtherBranches tree node = land $ lnot <$> branchesAbove tree node

branchesAbove :: VariationTree WithElif f -> VTNode WithElif f -> [f]
branchesAbove tree node = branches tree (fromJust (parent tree node))

branches :: VariationTree WithElif f -> VTNode WithElif f -> [f]
branches _ (VTNode _ (Mapping f)) = [f]
branches tree node@(VTNode _ (Elif f)) = f : branchesAbove tree node
branches tree node = branchesAbove tree node

correspondingIf :: VariationTree WithElif f ->
                   VTNode WithElif f ->
                   VTNode WithElif f
correspondingIf _ fi@(VTNode _ (Mapping _)) = fi
correspondingIf tree node =
    correspondingIf tree . fromJust $ parent tree node
```

The feature mapping and presence condition of `Artifact` and `Mapping` nodes are defined the very same as for our `MinimalLabels` and as in the paper. The feature mapping and presence condition of `Else` nodes are more complicated than in our definitions in the paper that are valid for the node type set $\{\texttt{artifact}, \texttt{mapping}, \texttt{else}\}$. The key difference is, that the extension by `elif` nodes now enables chains of `elif` and `else` branches, as in the following example:

```
#if A
```

```
  foo();
#elif B
  bar();
#elif C
  baz();
#else
  lol();
#endif
```

Thus, when determining feature mappings and presence copnditions for `Else` and `Elif` nodes, we have to consider all other branches above the current node in a potential chain. To do so, we use several helper functions:

`notTheOtherBranches` retrieves the formulas of all branches above a given node with `branchesAbove tree node`, then negates all formulas using `lnot <$>`[7] and finally conjuncts all negated formulas via `land`. Thus, `notTheOtherBranches` returns the condition that has to be satisfied in order to reach a given node in a chain. For example, for `#elif C` in the above example, `notTheOtherBranches` would return $\neg A \wedge \neg B$.

`branchesAbove` returns the formulas of all branches in a chain that are above a given node by invoking `branches` on the parent of the given node. For example, (in pseudo code) `branchesAbove (#elif C) = branches (parent of #elif C) = [A, B]`.

`branches` returns all branches in a chain starting from a given node. The chain ends at the first `Mapping` node when traversing the chain upwards, thus `branches` just returns the formula of the mapping in this case. If `branches` finds an `Elif` instead, it returns a list consisting of its formula `f` together with all formulas above the `elif` in the chain. `Artifact` nodes are skipped (third case).

`correspondingIf` returns the `mapping` node at the top of a chain by traversing the tree upwards from a given node until it finds the `mapping` and returns that `mapping`.

With these helper functions, we then define `featuremapping` and `presence-condition` for `Else` and `Elif` nodes.

The feature mapping of an `Else` node is the conjunction of the negation of the conditions of all the other branches because the code in an else branch is included if and only if every branch above the else evaluates to *false* (i.e.,

[7] `f <$> x` is syntactic sugar for `fmap f x`.

its negation evaluates to *true*). The same applies for `Elif` nodes except that an `Elif` comes with its own condition `f`. The feature mapping of an `Elif` is thus also given by `notTheOtherBranches tree node` but in conjunction with its own formula `f`.

The presence condition is defined the same for `Else` and `Elif` nodes except that their individual feature mappings are different. The presence condition of an `Else` or `Elif` node is a conjunction of (1) its own feature mapping and (2) the presence condition of any outer annotations. The reason is that the own feature mapping (1) handles all nodes in the current if-elif-else chain but this chain might be nested again in other outer annotations (2). These outer annotation are above the `correspondingIf` of the current chain, and thus we obtain the `presencecondition` of the parent of the `correspondingIf`.

While `else` and `elif` statements belong to the basic elements of most programming languages, their formal evaluation is intricate as shown above. In fact, `else` and `elif` help developers by shifting some complexity from program specification (i.e., development) to program evaluation (i.e., compilation or interpretation). Thus, the definition of `featuremapping` and `presencecondition` is much more complex for the node type set $\{\texttt{artifact}, \texttt{mapping}, \texttt{else}, \texttt{elif}\}$ (i.e., `WithElse`) than for $\{\texttt{artifact}, \texttt{mapping}, \texttt{else}\}$ (defined in our paper). This is the reason why we decided to discuss `elif` statements in the appendix rather than the actual paper.

2 Proofs

In this section, we provide the full proofs for the completeness of variation tree diffs and for the completeness and unambiguity of our catalog of elementary edit patterns.

2.1 Completeness of Variation Tree Diffs

In this section, we prove the completeness of variation tree diffs as a model for edits to variation trees. Therefore, we use our `Haskell` definitions introduced in the previous section.

Theorem 1. *Variation tree diffs are complete regarding variation trees, meaning that the difference between any two variation trees can be described in terms of a variation tree diff.*

To prove Theorem 1, we have to show that we can construct a variation tree diff `d` for any two variation trees `t` and `u`, such that

```
project BEFORE d == t
```

and

```
project AFTER  d == u.
```

By definition of variation tree diffs, these two laws have to be satisfied. These laws can be seen as axiomatic requirements to any diffing technique: Any diffing technique should describe the difference between two states of a data structure such that we can retrieve both states of the data structure. This ensures that the produced diff `d` holds enough information to actually represent all differences between both states.[8]

[8]Sometimes, diffs are condensed meaning that they only describe a local change to a data structure without storing the entire old state `t`. For example, *unix diffs* of an edited text file (e.g., a git diff) usually show just the changed lines surrounded by additional unchanged lines that serve as context to locate the change in the old state of the text file (cf. Listing 2 in our paper). Similarly, also variation tree diffs may be condensed and in fact we condense variation tree diffs in our tool `DiffDetective` for our validation by removing all non-edited subtrees. When diffs are condensed, the `project` function also has to take the old state `t` of the diffed data structure as input as one can neither construct the old state `t` nor the new state `u` from just a condensed diff. In this case the first law `project BEFORE d t == t` is trivially satisfied for any kind of diffed data structure because we could define `project` to just return `t` when the given time is `BEFORE`. Projecting the diff `d` to the new state becomes harder because the diff `d` has to be applied to / embedded into the old state `t` to yield the new state `u`. Here, we do not respect condensed diffs

Proof of Completeness. To prove completeness of variation tree diffs, we have to show that given any two variation trees `t` and `u`, there exists at least one variation tree diff `d` that satisfies the above requirements. To find one such variation tree diff, we provide a diffing function that takes two variation trees and describes their differences in terms of a variation tree diff. As argued in our paper, there are many possible ways to construct diffs, so we define the simplest possible diffing function we could think of and refer to it as `naiveDiff`.

We assume that the `UUID`s of the nodes in both input trees to `naiveDiff` are unique (i.e., there are no two nodes with the same `UUID` across both trees). Otherwise we would have to create a matching of the input trees first and create new `UUID`s out of the matching, which would unnecessarily complicate the proof. We thus assume all given `UUID`s to be unique already which does not limit the validity of our proof because the given trees are finite and thus there exists a numeration of the nodes such that all nodes have unique `UUID`s. Without loss of generality, let the `UUID` of the root be 0 (cf. Section 1.2).

Our `naiveDiff` creates a variation tree diff that marks all nodes and edges of the old tree as removed and all nodes and edges of the new tree as added, except for the root that remains unchanged:

```haskell
{-# LANGUAGE LambdaCase #-}

naiveDiff :: (HasNeutral f, Composable f, VTLabel l) =>
    VariationTree l f -> VariationTree l f -> VariationDiff l f
naiveDiff
    (VariationTree nodesBefore edgesBefore)
    (VariationTree nodesAfter edgesAfter)
    =
    VariationDiff
    (root : nodesWithoutRoot (nodesBefore <> nodesAfter))
    (edgesBefore <> edgesAfter)
    delta
        where
            nodesWithoutRoot nodes = [n | n <- nodes, n /= root]
            delta = \case
```

explicitly as they can be seen as an extension to full diffs that store the entire old state. This does not limit the validity of our proof for completeness as (1) we show that there always exists a valid full diff for two variation trees, and (2) condensed diffs can be and usually are constructed from condensing a full diff. Thus, by showing that variation tree diffs are complete as full diffs, also their condensed diffs are complete.

```
                Left node ->
                    if node == root then
                        NON
                    else if node `elem` nodesBefore then
                        REM
                    else if node `elem` nodesAfter then
                        ADD
                    else
                        error "Given node is not part of variation diff!"
                Right edge ->
                    if edge `elem` edgesBefore then
                        REM
                    else if edge `elem` edgesAfter then
                        ADD
                    else
                        error "Given edge is not part of variation diff!"
```

For two given variation trees

```
VariationTree nodesBefore edgesBefore
```

and

```
VariationTree nodesAfter  edgesAfter
```

`naiveDiff` creates a `VariationDiff` with all nodes from both input trees `nodesBefore <> nodesAfter`[9] but with only a single root below which both trees are inserted. Thus, `naiveDiff` removes the roots from both input node sets via `nodesWithoutRoot` but reinserts the root at the beginning of the `VariationDiff`'s node set. The resulting `VariationDiff` contains exactly the edges from both input trees. Finally, the produced `VariationDiff` is equipped with the function `delta` which is defined to flag (1) the root as unchanged `NON`, (2) all old nodes and edges as removed `REM` and (3) all new nodes and edges as inserted `ADD`. The function `delta` is undefined for nodes or edges that were not part of the original variation trees, thus issuing an error for those elements.

To prove the completeness of variation tree diffs, we show that a variation tree diff created with `naiveDiff` is a valid variation tree diff by showing that its projections are indeed the initial two variation trees. Let `t` and `u` be any two variation trees of the same type (i.e., using the same logic `f` and the same label type `l`):

[9] `<>` concatenates two lists (or more generally: composes two monoidal values).

```
t :: VariationTree l f
t = VariationTree nodesBefore edgesBefore

u :: VariationTree l f
u = VariationTree nodesAfter edgesAfter
```

We show that the following two equalities hold:

```
project BEFORE (naiveDiff t u) == t
project AFTER  (naiveDiff t u) == u
```

We start by proving the first equality using equational reasoning (i.e., we substitute the definitions of our functions). We describe our proof steps in comments (preceded by `--`).[10]

```
project BEFORE (naiveDiff t u)
-- Substitute t and u
== project BEFORE (naiveDiff
    (VariationTree nodesBefore edgesBefore)
    (VariationTree nodesAfter edgesAfter))
-- Substitute naiveDiff
== project BEFORE (VariationDiff
    (root : nodesWithoutRoot (nodesBefore <> nodesAfter))
    (edgesBefore <> edgesAfter)
    delta) -- defined exactly as in the definition for naiveDiff
-- Substitute project
== VariationTree
    (filter
        (existsAtTime BEFORE . delta . Left)
        (root : nodesWithoutRoot (nodesBefore <> nodesAfter))
    )
    (filter
        (existsAtTime BEFORE . delta . Right)
        (edgesBefore <> edgesAfter)
    )
```

By definition of `delta` we know that

$$\forall \texttt{ e `elem` edgesBefore: delta (Right e) == REM}$$

[10]Note that the proof is not a valid `Haskell` program but uses our `Haskell` definitions.

and that

$$\forall$$ `e 'elem' edgesAfter:  delta (Right e) == ADD`.

By definition of `existsAtTime` we know that `existsAtTime BEFORE x` is *true* iff `x /= ADD`. Thus, exactly the edges in `edgesBefore` exist at time `BEFORE`. We get:

```
== VariationTree
    (filter
        (existsAtTime BEFORE . delta . Left)
        (root : nodesWithoutRoot (nodesBefore <> nodesAfter))
    )
    edgesBefore
-- Substitute nodesWithoutRoot
== VariationTree
    (filter
        (existsAtTime BEFORE . delta . Left)
        (root : [n | n <- (nodesBefore <> nodesAfter), n /= root])
    )
    edgesBefore
```

By definition of `delta` we know that

$$\forall$$ `n 'elem' nodesBefore:  delta (Left n) == REM`

and

$$\forall$$ `n 'elem' nodesAfter:  delta (Left n) == ADD`

and

`delta (Left root) = NON`.

By definition of `existsAtTime` we know that `existsAtTime BEFORE x` is *true* iff `x /= ADD`. Thus, all nodes in `nodesBefore` and the `root` exist at time `BEFORE` but not the nodes in `nodesAfter`. We get:

```
== VariationTree
    (root : [n | n <- nodesBefore, n /= root])
    edgesBefore
-- Assuming that
--     root == head nodesBefore,
-- or assuming that
--     nodesBefore is a set and not a list,
-- we get:
== VariationTree
    nodesBefore
    edgesBefore
== t
```

The other proof for `project AFTER (naiveDiff t u) == u` is analogous. We have to replace all occurrences of `BEFORE` in the equations and reasoning by `AFTER` to retrieve the dual sets `nodesAfter` and `edgesAfter`, and finally the second variation tree `u`. □

2.2 Completeness and Unambiguity of Elementary Edit Patterns

We prove that our catalog of elementary edit patterns is complete (i.e., every `artifact` node matches at least one pattern) and unambiguous (i.e., every `artifact` node matches at most one pattern). This means that every `artifact` node matches exactly one pattern.

Theorem 2. *Every node in a variation tree diff with node type `artifact` is classified by exactly one elementary pattern.*

Proof. The elementary edit patterns distribute into three groups, one for each type of edit of a given `artifact` node c (i.e., *Add**, *Rem**, or unchanged patterns). The type of change made to c is given by $\Delta(c)$ which is exactly one value of $\{+, -, \bullet\}$. This means, exactly one clause of added(c), removed(c), and unchanged(c) evaluates to *true* while the others evaluate to *false*. Thus, c can only match a pattern of at most one of the three groups.

The patterns within each group are mutually exclusive in their remaining clauses: Each pattern contains the negation of at least one clause of each other pattern of the same group. Thus, every `artifact` node matches at most one pattern, meaning that elementary patterns are unambiguous.

Moreover, every possible evaluation of the shared terms of the patterns within a group is covered by at least one pattern. Thus, every `artifact` node matches at least one pattern, meaning that elementary patterns are complete. □

3 Composite Edit Patterns

In this section, we show that edit patterns defined in related work Al-Hajjaji et al. [2016], Stănciulescu et al. [2016] are either (1) a subtype of or equivalent to one of our elementary edit patterns, or (2) indeed a composition of our elementary edit patterns and thus a composite edit pattern. For each pattern from related work, we show its definition or example from the original paper together with the corresponding variation tree diff (which is not part of the original paper but constructed by us). In the variation tree diff, we label `artifact` nodes directly with their corresponding elementary edit pattern (as each `artifact` node is classified by exactly one elementary pattern). In this sense, we provide a visual proof that the corresponding pattern (or at least an example of it) from related work is equivalent to one of our elementary patterns or that it is a composite pattern.

3.1 Al-Hajjaji et al. [2016]

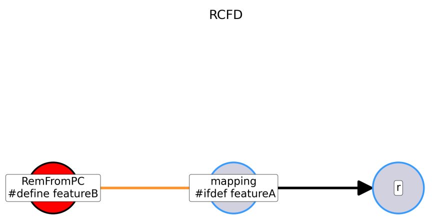
Al-Hajjaji et al. provide a set of mutation operators to preprocessor-based variability. We consider all edits to source code and preprocessor directives here but not those to the variability model. Al-Hajjaji et al. define all patterns in terms of a natural language description and a generic example. Each example is given as a state before and a state after the edit but not as a unix diff as we do in our paper. For comparability, we translate each example to a unix diff here. A further discussion and comparison to our work is part of the related work section of our paper.

3.1.1 Feature Dependency Operators

Al-Hajjaji et al. distinguish edits to source from edits to macro definition (that may describe dependencies between features). The feature dependency operators describe changes to the feature mapping of `#define` statements to conditionally activate or deactivate other features. Both operators correspond to elementary patterns of our catalog. While, we do not distinguish between `#define` directives and pure source code in `artifact` nodes for our elementary edit patterns, such a differentiation is still possible when inspecting the label of `artifact` nodes.

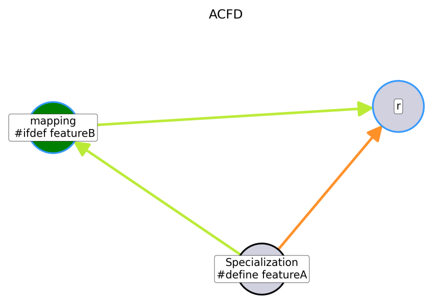
RCFD – Remove Conditional Feature Definition

```
 #ifdef featureA
-#define featureB
 #endif
```



ACFD – Add Condition to Feature Definition

```
+#ifdef featureB
 #define featureA
+#endif
```

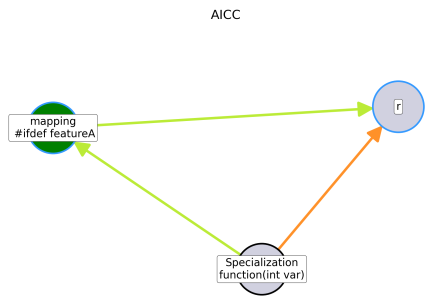


3.1.2 Variability-Mapping Operators

The variability-mapping operators by Al-Hajjaji et al. [2016] describe edits the preprocessor directives that describe feature mappings. All operators correspond to elementary edit patterns.

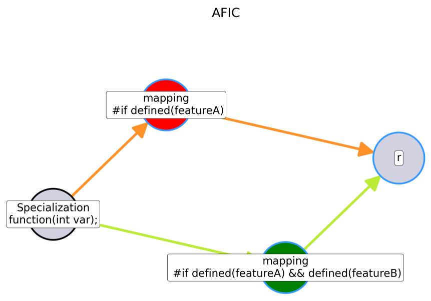
AICC – Adding ifdef Condition around Code

```
+#ifdef featureA
 function(int var)
+#endif
```



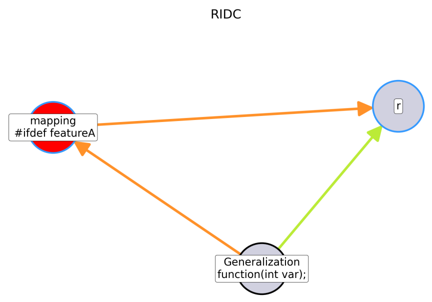
AFIC – Adding Feature to ifdef Condition

```
-#if defined(featureA)
+#if defined(featureA) && defined(featureB)
 function(int var);
 #endif
```



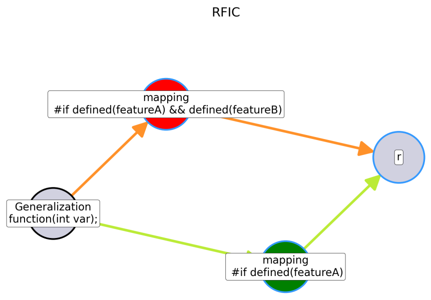
RIDC – Removing ifdef Condition

```
-#ifdef featureA
 function(int var);
-#endif
```



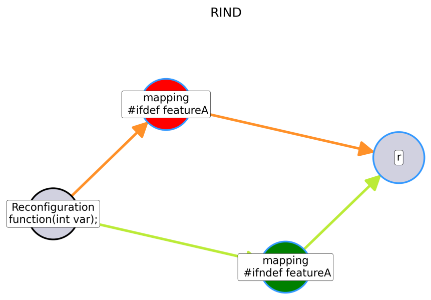
RFIC – Removing Feature of ifdef Condition

```
-#if defined(featureA) && defined(featureB)
+#if defined(featureA)
 function(int var);
 #endif
```



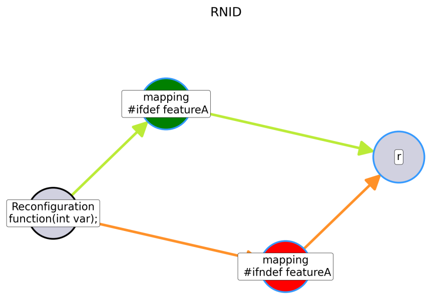
RIND – Replacing ifdef Directive with ifndef Directive

```
-#ifdef featureA
+#ifndef featureA
 function(int var);
 #endif
```



RNID – Replacing ifndef Directive with ifdef Directive

```
+#ifdef featureA
-#ifndef featureA
 function(int var);
 #endif
```



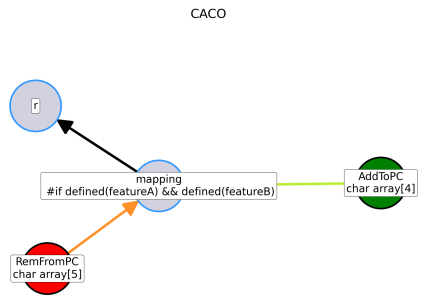
3.1.3 Domain Artifact Operators

The domain artifact operators describe changes to source code.

CACO – Conditionally Applying Conventional Operator

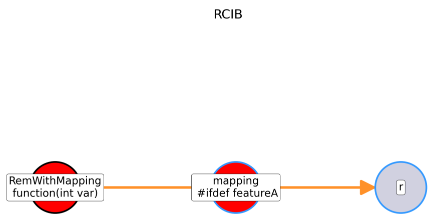
CACO applies conventional source code mutation operators in a variability-aware way. It modifies source code that has a certain presence condition. In a diff, such a modification occurs as the removal and insertion of source code and thus CACO is a composite edit built from a *RemFromPC* and *AddToPC* pattern application.

```
 #if defined(featureA) && defined(featureB)
-char array[5]
+char array[4]
 #endif
```



RCIB – Removing Complete ifdef Block

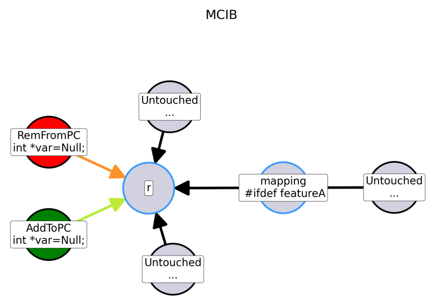
```
-#ifdef featureA
-function(int var)
-#endif
```



MCIB – Moving Code around ifdef Blocks

MCIB moves code around an `#ifdef` block. As discussed in the discussion section for our edit patterns in the paper, describing moves in terms of unix diffs is ambiguous: It is subject to the differ's (or developer's) choice to consider the code or the preprocessor directives as moved, as both can be the case. Here, we show the move of source code as envisioned by Al-Hajjaji et al..

```
  ...
-int *var=Null;
 #ifdef featureA
  ...
 #endif
+int *var=Null;
  ...
```



3.1.4 Conclusion

As described in our paper, the operators by Al-Hajjaji et al. are similar to our patterns. Yet, the operators are incomplete, as for example *AddWith-Mapping* and thus a non-empty subset of edits is missing. On the other hand, the operators distinguish more cases for single elementary patterns, for example if a `#define` directive or source code was specialized. Our catalog of elementary could be extended by distinguishing such sub-cases for different elementary patterns in the future (in particular, by adding further clauses to the patterns definitions), while remaining complete.

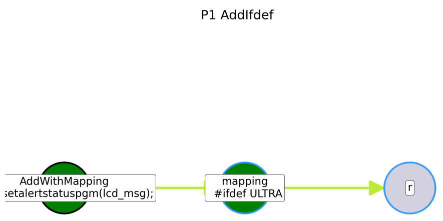
3.2 Stănciulescu et al. [2016]

Stănciulescu et al. provide a set of edit patterns for edits to variability in source code, yet without being complete and facing overlap and ambiguity. A discussion and comparison to our work is part of the related work section of our paper.

3.2.1 Code-Adding Patterns

P1 AddIfdef

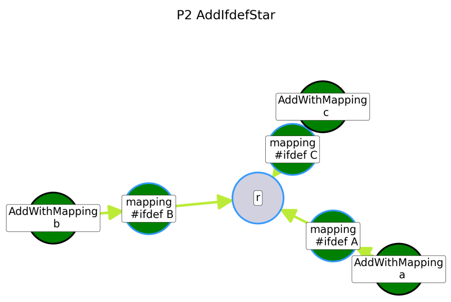
```
+ #ifdef ULTRA_LCD
+  lcd_setalertstatuspgm(lcd_msg);
+ #endif
```



P2 AddIfdef*

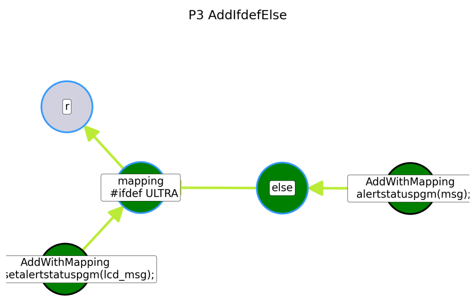
AddIfdef* is the repeated application of the AddIfdef pattern (two or more times). Thus, AddIfdef* is a composite pattern, built from two or more *AddWithMapping* patterns. We show an example with three consecutive applications of the AddIfdef pattern:

```
+ #ifdef A
+  a
+ #endif

+ #ifdef B
+  b
+ #endif

+ #ifdef C
+  c
+ #endif
```



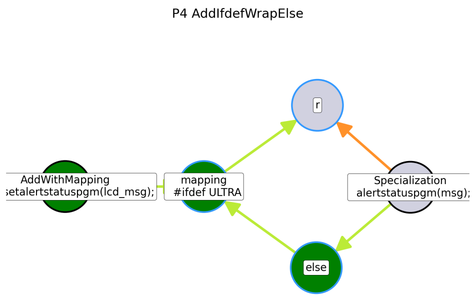
P3 AddIfdefElse

```
+ #ifdef ULTRA_LCD
+   lcd_setalertstatuspgm(lcd_msg);
+ #else
+   alertstatuspgm(msg);
+ #endif
```



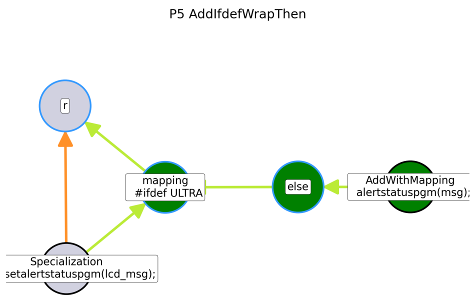
P4 AddIfdefWrapElse

```
+ #ifdef ULTRA_LCD
+   lcd_setalertstatuspgm(lcd_msg);
+ #else
    alertstatuspgm(msg);
+ #endif
```



P5 AddIfdefWrapThen

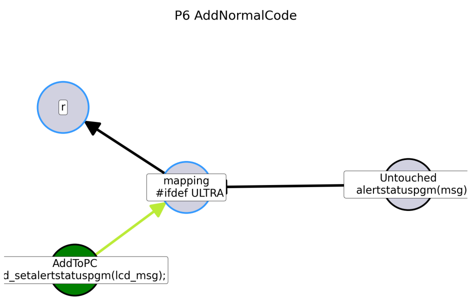
```
+ #ifdef ULTRA_LCD
    lcd_setalertstatuspgm(lcd_msg);
+ #else
+   alertstatuspgm(msg);
+ #endif
```



P6 AddNormalCode

This pattern is explained without an example and described in natural language. AddNormalCode describes the insertion of source code within another presence condition, which can also be *true*. We constructed the following example from its natural language description (and the example that was given by Stănciulescu et al. for the dual RemNormalCode pattern). This pattern corresponds to our *AddToPC* pattern.

```
   #ifdef ULTRA_LCD
+    lcd_setalertstatuspgm(lcd_msg)
     alertstatuspgm(msg);
   #endif
```



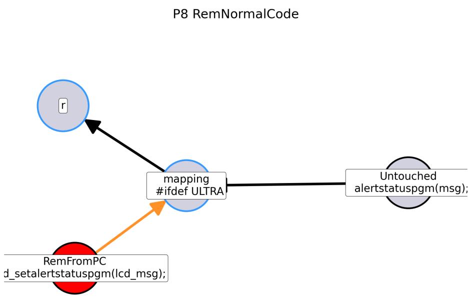
P7 AddAnnotation

This pattern matches fixes to syntactically incorrect annotations by insertion of `#ifdef` or `#endif` directives, and whitespace changes. This pattern is neither supported by `DiffDetective` nor the variation control system by Stănciulescu et al..

3.2.2 Code-Removing Patterns

P8 RemNormalCode

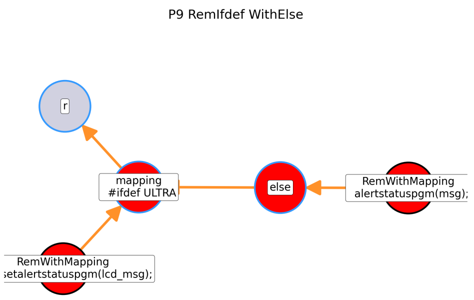
```
   #ifdef ULTRA_LCD
-    lcd_setalertstatuspgm(lcd_msg)
    alertstatuspgm(msg);
   #endif
```



P9 RemIfdef

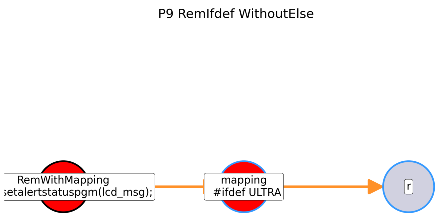
This pattern has two cases and thus actually describes two patterns. RemIfdef matches the removal of source code with its surrounding #ifdef and #else annotations

```
- #ifdef ULTRA_LCD
-  lcd_setalertstatuspgm(lcd_msg);
- #else
-  alertstatuspgm(msg);
- #endif
```



or without an #else annotation:

```
- #ifdef ULTRA_LCD
-  lcd_setalertstatuspgm(lcd_msg);
- #endif
```



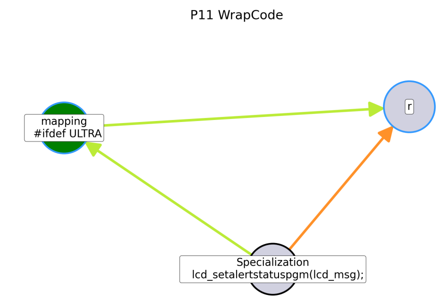
P10 RemAnnotation

This pattern matches fixes to syntactically incorrect annotations by removal of #ifdef or #endif directives. This pattern is neither supported by DiffDetective nor the variation control system by Stănciulescu et al..

3.2.3 Other Patterns

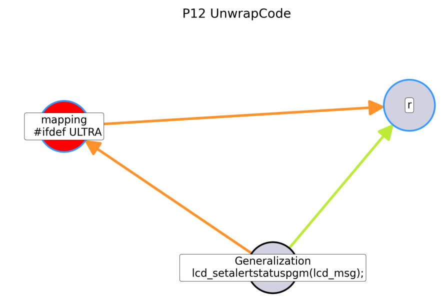
P11 WrapCode

```
+ #ifdef ULTRA_LCD
    lcd_setalertstatuspgm(lcd_msg)
+ #endif
```



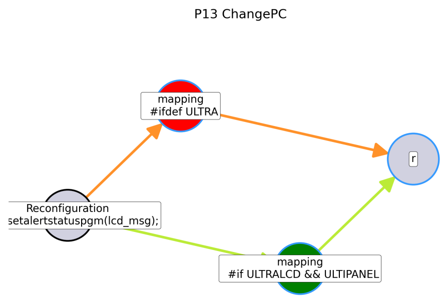
P12 UnwrapCode

```
- #ifdef ULTRA_LCD
    lcd_setalertstatuspgm(lcd_msg)
- #endif
```



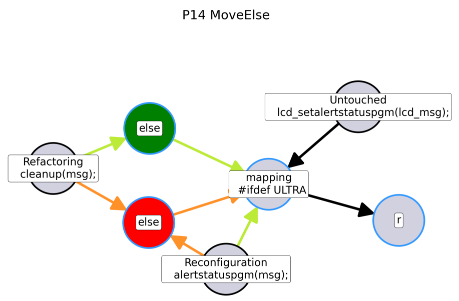
P13 ChangePC

```
- #ifdef ULTRA_LCD
+ #if ULTRALCD && ULTIPANEL
    lcd_setalertstatuspgm(lcd_msg)
  #endif
```



P14 MoveElse

```
  #ifdef ULTRA_LCD
    lcd_setalertstatuspgm(lcd_msg)
- #else
    alertstatuspgm(msg);
+ #else
    cleanup(msg);
  #endif
```



3.2.4 Conclusion

As described in our paper, the patterns by Stănciulescu et al. inspired our work. In particular, we addressed the following three problems of the patterns by Stănciulescu et al. in our work:

Ambiguity. The patterns lack a formal description and are explained on the examples presented above. Thus, one has to come up with its own method for matching these patterns when one wants to re-implement the detection of the patterns by Stănciulescu et al.. Thereby it is not clear how some patterns were exactly defined (e.g., if further code is allowed between some line edits or not such as in WrapCode or UnwrapCode).

Incompleteness. The patterns by Stănciulescu et al. are incomplete. The insertion or deletion of just an `#else` branch is not covered: These operations are explicitly excluded from the AddIfdef and RemIfdef patterns Stănciulescu et al. [2016] and no other patterns matches the insertion or deletion of an `#else` branch. Such edits are covered in our catalog by *AddWithMapping* and *RemWithMapping* (thus AddIfdef can be seen as a subtype of *AddWithMapping*). Elif directives are not explicitly mentioned by Stănciulescu et al.. Moreover, some composite patterns miss their inverse operation (AddIfdef*, AddIfdefWrapElse, AddIfdefWrapThen), and *Untouched* is missing. Furthermore, Stănciulescu et al. [2016] report that not all edits the history of Busybox could be classified with their patterns.

Overlap. Edits can be classified by more than one pattern. For example, it is undefined if an occurrence of AddIfdefWrapElse should be considered as an application of AddifdefWrapElse or an application of AddIfdef and WrapCode. With the distinction between elementary and composite patterns, we explicitly account for this overlap in our paper.

4 Complete Validation Results

For processing Marlin, we used the same settings as Stănciulescu et al. [2016] to be comparable. This means that we

1. considered only files within the `Marlin` subdirectory,
2. ignored `arduino` files,
3. only considered file modifications (as for the other datasets)
4. inspected exactly all files of type `c`, `cpp`, `h`, and `pde`.

We also accounted for the custom `ENABLED` and `DISABLED` macros in Marlin explicitly where `ENABLED` acts similar as `defined` and `DISABLED` tests whether a macro is undefined or set to 0. We did not implement custom treatments for other datasets.

In the following we present the full validation results for each dataset both in absolute (Table 1) and in relative values (Table 2).

Name	Domain	#total commits	#processed commits	#diffs	#artifact nodes	AddToPC	AddWithMapping	RemFromPC	RemWithMapping	Specialization	Generalization	Reconfiguration	Refactoring	runtime	avg. runtime per processed commit	median runtime per processed commit
clamav	antivirus program	10,659	8,234	25,761	294,500	147,114	6,840	128,698	5,384	2,193	1,674	1,532	1,065	308.0s	37.2ms	18ms
freebsd	operating system	272,207	179,770	729,879	9,778,773	4,812,808	264,075	4,210,889	226,081	71,658	73,635	60,473	59,154	26,601.2s	147.0ms	29ms
gcc	compiler framework	191,405	122,784	416,765	3,062,884	1,523,600	48,740	1,393,883	38,500	12,518	11,232	9,737	24,674	106,361.6s	863.1ms	157ms
openldap	LDAP directory service	23,938	17,634	56,583	379,651	180,136	13,900	155,610	11,652	5,176	5,715	2,427	5,035	894.9s	50.5ms	24ms
postgresql	database system	52,934	31,447	158,250	1,736,201	880,598	14,862	815,758	12,459	3,511	3,425	1,566	4,022	9,238.9s	293.0ms	59ms
mpsolve	mathematical software	1,773	1,326	5,395	51,544	26,663	205	24,421	102	72	6	51	24	26.3s	19.8ms	9ms
mplayer-svn	media player	37,992	22,806	46,291	327,830	157,781	10,432	139,333	7,943	3,108	3,240	4,191	1,802	1,059.1s	45.8ms	20ms
linux	operating system	1,072,601	870,447	1,875,894	12,540,430	6,429,832	116,828	5,714,788	115,507	30,917	43,025	24,697	64,836	104,189.5s	119.1ms	41ms
sylpheed	e-mail client	2,682	1,820	4,237	30,718	16,943	752	11,959	392	380	217	17	58	70.5s	38.6ms	28ms
sendmail	mail transfer agent	86	0	0	0	0	0	0	0	0	0	0	0	0.0s	–ms	–ms
vim	text editor	15,354	14,453	39,547	361,775	146,072	13,355	114,386	12,525	3,222	8,129	2,258	61,828	2,706.8s	187.2ms	128ms
gnumeric	spreadsheet application	24,144	15,668	58,999	586,763	303,091	5,681	269,566	4,534	1,520	1,316	822	233	2,424.8s	154.2ms	64ms
xorg-server	X server	17,788	14,401	53,691	529,290	245,915	26,308	222,516	14,887	4,480	7,298	2,175	5,711	760.1s	52.7ms	18ms
tcl	program interpreter	24,414	10,189	25,767	410,801	182,891	23,385	171,748	21,315	3,290	3,673	1,048	3,451	1,494.0s	146.1ms	73ms
godot	game engine	40,944	18,867	107,847	1,272,304	612,816	42,715	540,406	38,565	6,812	4,687	18,188	8,115	4,890.5s	257.9ms	46ms
openvpn	security application	3,128	2,357	8,673	83,973	37,941	3,745	34,570	2,314	681	1,433	1,579	1,710	139.0s	58.9ms	35ms
privoxy	proxy server	7,558	2,634	4,278	37,710	18,764	1,439	15,254	1,083	362	283	446	79	89.5s	33.3ms	22ms
libssh	network	5,352	4,439	8,465	56,481	29,332	1,441	23,625	1,012	604	201	138	128	72.3s	16.2ms	10ms
marlin	3d printing	19,260	14,610	84,351	580,133	215,751	69,192	198,286	49,396	3,262	9,267	17,695	17,284	1,344.6s	91.5ms	23ms
opensolaris	operating system	11,422	9,691	53,928	999,302	501,904	16,915	450,919	11,603	8,178	5,349	1,114	3,320	2,135.8s	220.2ms	54ms
sqlite	databases	8,664	6,358	15,643	175,270	86,420	4,060	75,469	3,207	1,585	1,154	722	2,653	398.9s	62.6ms	44ms
busybox	embedded systems	17,447	14,486	41,159	398,788	179,935	14,631	170,612	14,517	3,976	2,666	4,961	7,490	706.2s	48.4ms	18ms
lynx	web browser	125	0	0	0	0	0	0	0	0	0	0	0	0.0s	–ms	–ms
libxml2	XML library	5,146	3,767	8,979	150,356	69,585	9,305	58,328	3,464	5,215	1,155	256	3,048	470.1s	124.5ms	74ms
emacs	text editor	154,155	37,946	71,329	600,393	271,579	18,648	248,209	15,253	5,946	5,611	3,698	31,449	8,194.5s	211.8ms	87ms
apache-httpd	web server	32,953	15,986	35,092	289,032	142,766	5,094	128,435	6,046	1,746	2,006	889	2,050	1,227.2s	74.8ms	41ms
minix	operating system	7,153	4,624	35,416	591,999	271,228	28,686	251,046	21,457	4,277	2,963	8,426	3,916	373.5s	79.3ms	9ms
lighttpd	web server	4,433	3,480	9,521	79,836	36,707	2,960	33,087	1,975	783	435	210	3,679	94.9s	27.2ms	13ms
gnuplot	plotting tool	11,766	6,550	14,761	150,671	71,826	5,993	64,027	4,136	1,338	1,783	675	893	369.8s	56.1ms	34ms
xfig	vector graphics editor	9	4	19	85	44	2	37	0	1	0	0	1	0.2s	43.5ms	66ms
subversion	revision control system	60,037	36,204	88,580	742,916	384,812	5,016	344,148	3,910	1,570	1,755	980	725	7,586.8s	208.5ms	63ms
xterm	terminal emulator	112	108	1,280	29,260	14,535	1,406	12,433	494	180	99	56	57	26.9s	249.0ms	255ms
xine-lib	media library	114	0	0	0	0	0	0	0	0	0	0	0	0.0s	–ms	–ms
gimp	graphics editor	47,836	31,700	168,678	1,801,508	913,529	12,075	853,440	12,367	3,608	2,924	2,342	1,223	3,434.5s	107.8ms	36ms
berkeley-db-libdb	database system	7	1	485	3,326	1,760	58	1,431	32	28	0	0	17	2.4s	2,358.0ms	2,358ms
cpython	program interpreter	112,196	28,446	59,127	951,893	466,454	23,247	430,065	11,548	8,509	4,736	1,821	5,513	3,941.6s	133.4ms	47ms
cherokee-webserver	web server	5,853	2,170	7,879	46,746	24,236	595	20,853	588	119	181	69	105	42.0s	18.9ms	10ms
php	program interpreter	127,632	65,415	175,855	3,016,271	1,476,025	58,963	1,384,638	41,057	16,359	12,432	11,530	15,267	51,334.6s	783.6ms	40ms
pidgin	instant messenger	40,097	27,354	72,271	730,596	364,173	11,649	330,159	11,404	4,418	3,304	4,098	1,391	2,971.0s	108.0ms	45ms
glibc	programming library	38,349	23,481	191,212	855,285	392,240	33,172	364,266	24,864	11,107	6,553	6,887	16,196	3,737.4s	158.2ms	13ms
dia	diagramming software	6,666	3,694	16,943	146,820	75,346	1,810	65,538	1,952	845	1,067	142	120	114.1s	30.7ms	13ms
parrot	virtual machine	49,989	13,807	40,701	944,273	464,332	8,473	446,755	7,897	2,125	1,741	1,250	11,700	1,396.8s	99.2ms	27ms
irssi	IRC client	6,346	4,052	9,736	53,285	28,592	543	23,156	492	180	139	150	33	45.3s	11.1ms	6ms
ghostscript	postscript interpreter	22,186	14,962	71,754	816,015	396,737	21,820	363,153	15,657	6,552	2,991	3,829	5,276	1,310.7s	86.3ms	30ms
total	–	2,594,912	1,708,172	4,901,021	45,695,687	22,602,813	949,016	20,305,900	777,571	242,411	239,500	203,145	375,331	352,586.7s	205.4ms	41ms

Table 1: Absolute results.

Name	Domain	#total commits	#processed commits	#diffs	#artifact nodes	AddToPC	AddWithMapping	RemFromPC	RemWithMapping	Specialization	Generalization	Reconfiguration	Refactoring	runtime	avg. runtime per processed commit	median runtime per processed commit
clamav	antivirus program	10,659	8,234	25,761	294,500	50.0%	2.3%	43.7%	1.8%	0.7%	0.6%	0.5%	0.4%	308.0s	37.2ms	18ms
freebsd	operating system	272,207	179,770	729,879	9,778,773	49.2%	2.7%	43.1%	2.3%	0.7%	0.8%	0.6%	0.6%	26,601.2s	147.0ms	29ms
gcc	compiler framework	191,405	122,784	416,765	3,062,884	49.7%	1.6%	45.5%	1.3%	0.4%	0.4%	0.3%	0.8%	106,361.6s	863.1ms	157ms
openldap	LDAP directory service	23,938	17,634	56,583	379,651	47.4%	3.7%	41.0%	3.1%	1.4%	1.5%	0.6%	1.3%	894.9s	50.5ms	24ms
postgresql	database system	52,934	31,447	158,250	1,736,201	50.7%	0.9%	47.0%	0.7%	0.2%	0.2%	0.1%	0.2%	9,238.9s	293.0ms	59ms
mpsolve	mathematical software	1,773	1,326	5,395	51,544	51.7%	0.4%	47.4%	0.2%	0.1%	0.0%	0.1%	0.0%	26.3s	19.8ms	9ms
mplayer-svn	media player	37,992	22,806	46,291	327,830	48.1%	3.2%	42.5%	2.4%	0.9%	1.0%	1.3%	0.5%	1,059.1s	45.8ms	20ms
linux	operating system	1,072,601	870,447	1,875,894	12,540,430	51.3%	0.9%	45.6%	0.9%	0.2%	0.3%	0.2%	0.5%	104,189.5s	119.1ms	41ms
sylpheed	e-mail client	2,682	1,820	4,237	30,718	55.2%	2.4%	38.9%	1.3%	1.2%	0.7%	0.1%	0.2%	70.5s	38.6ms	28ms
sendmail	mail transfer agent	86	0	0	0	–%	–%	–%	–%	–%	–%	–%	–%	0.0s	–ms	–ms
vim	text editor	15,354	14,453	39,547	361,775	40.4%	3.7%	31.6%	3.5%	0.9%	2.2%	0.6%	17.1%	2,706.8s	187.2ms	128ms
gnumeric	spreadsheet application	24,144	15,668	58,999	586,763	51.7%	1.0%	45.9%	0.8%	0.3%	0.2%	0.1%	0.0%	2,424.8s	154.2ms	64ms
xorg-server	X server	17,788	14,401	53,691	529,290	46.5%	5.0%	42.0%	2.8%	0.8%	1.4%	0.4%	1.1%	760.1s	52.7ms	18ms
tcl	program interpreter	24,414	10,189	25,767	410,801	44.5%	5.7%	41.8%	5.2%	0.8%	0.9%	0.3%	0.8%	1,494.0s	146.1ms	73ms
godot	game engine	40,944	18,867	107,847	1,272,304	48.2%	3.4%	42.5%	3.0%	0.5%	0.4%	1.4%	0.6%	4,890.5s	257.9ms	46ms
openvpn	security application	3,128	2,357	8,673	83,973	45.2%	4.5%	41.2%	2.8%	0.8%	1.7%	1.9%	2.0%	139.0s	58.9ms	35ms
privoxy	proxy server	7,558	2,634	4,278	37,710	49.8%	3.8%	40.5%	2.9%	1.0%	0.8%	1.2%	0.2%	89.5s	33.3ms	22ms
libssh	network	5,352	4,439	8,465	56,481	51.9%	2.6%	41.8%	1.8%	1.1%	0.4%	0.2%	0.2%	72.3s	16.2ms	10ms
marlin	3d printing	19,260	14,610	84,351	580,133	37.2%	11.9%	34.2%	8.5%	0.6%	1.6%	3.1%	3.0%	1,344.6s	91.5ms	23ms
opensolaris	operating system	11,422	9,691	53,928	999,302	50.2%	1.7%	45.1%	1.2%	0.8%	0.5%	0.1%	0.3%	2,135.8s	220.2ms	54ms
sqlite	databases	8,664	6,358	15,643	175,270	49.3%	2.3%	43.1%	1.8%	0.9%	0.7%	0.4%	1.5%	398.9s	62.6ms	44ms
busybox	embedded systems	17,447	14,486	41,159	398,788	45.1%	3.7%	42.8%	3.6%	1.0%	0.7%	1.2%	1.9%	706.2s	48.4ms	18ms
lynx	web browser	125	0	0	0	–%	–%	–%	–%	–%	–%	–%	–%	0.0s	–ms	–ms
libxml2	XML library	5,146	3,767	8,979	150,356	46.3%	6.2%	38.8%	2.3%	3.5%	0.8%	0.2%	2.0%	470.1s	124.5ms	74ms
emacs	text editor	154,155	37,946	71,329	600,393	45.2%	3.1%	41.3%	2.5%	1.0%	0.9%	0.6%	5.2%	8,194.5s	211.8ms	87ms
apache-httpd	web server	32,953	15,986	35,092	289,032	49.4%	1.8%	44.4%	2.1%	0.6%	0.7%	0.3%	0.7%	1,227.2s	74.8ms	41ms
minix	operating system	7,153	4,624	35,416	591,999	45.8%	4.8%	42.4%	3.6%	0.7%	0.5%	1.4%	0.7%	373.5s	79.3ms	9ms
lighttpd	web server	4,433	3,480	9,521	79,836	46.0%	3.7%	41.4%	2.5%	1.0%	0.5%	0.3%	4.6%	94.9s	27.2ms	13ms
gnuplot	plotting tool	11,766	6,550	14,761	150,671	47.7%	4.0%	42.5%	2.7%	0.9%	1.2%	0.4%	0.6%	369.8s	56.1ms	34ms
xfig	vector graphics editor	9	4	19	85	51.8%	2.4%	43.5%	0.0%	1.2%	0.0%	0.0%	1.2%	0.2s	43.5ms	66ms
subversion	revision control system	60,037	36,204	88,580	742,916	51.8%	0.7%	46.3%	0.5%	0.2%	0.2%	0.1%	0.1%	7,586.8s	208.5ms	63ms
xterm	terminal emulator	112	108	1,280	29,260	49.7%	4.8%	42.5%	1.7%	0.6%	0.3%	0.2%	0.2%	26.9s	249.0ms	255ms
xine-lib	media library	114	0	0	0	–%	–%	–%	–%	–%	–%	–%	–%	0.0s	–ms	–ms
gimp	graphics editor	47,836	31,700	168,678	1,801,508	50.7%	0.7%	47.4%	0.7%	0.2%	0.2%	0.1%	0.1%	3,434.5s	107.8ms	36ms
berkeley-db-libdb	database system	7	1	485	3,326	52.9%	1.7%	43.0%	1.0%	0.8%	0.0%	0.0%	0.5%	2.4s	2,358.0ms	2,358ms
cpython	program interpreter	112,196	28,446	59,127	951,893	49.0%	2.4%	45.2%	1.2%	0.9%	0.5%	0.2%	0.6%	3,941.6s	133.4ms	47ms
cherokee-webserver	web server	5,853	2,170	7,879	46,746	51.8%	1.3%	44.6%	1.3%	0.3%	0.4%	0.1%	0.2%	42.0s	18.9ms	10ms
php	program interpreter	127,632	65,415	175,855	3,016,271	48.9%	2.0%	45.9%	1.4%	0.5%	0.4%	0.4%	0.5%	51,334.6s	783.6ms	40ms
pidgin	instant messenger	40,097	27,354	72,271	730,596	49.8%	1.6%	45.2%	1.6%	0.6%	0.5%	0.6%	0.2%	2,971.0s	108.0ms	45ms
glibc	programming library	38,349	23,481	191,212	855,285	45.9%	3.9%	42.6%	2.9%	1.3%	0.8%	0.8%	1.9%	3,737.4s	158.2ms	13ms
dia	diagramming software	6,666	3,694	16,943	146,820	51.3%	1.2%	44.6%	1.3%	0.6%	0.7%	0.1%	0.1%	114.1s	30.7ms	13ms
parrot	virtual machine	49,989	13,807	40,701	944,273	49.2%	0.9%	47.3%	0.8%	0.2%	0.2%	0.1%	1.2%	1,396.8s	99.2ms	27ms
irssi	IRC client	6,346	4,052	9,736	53,285	53.7%	1.0%	43.5%	0.9%	0.3%	0.3%	0.3%	0.1%	45.3s	11.1ms	6ms
ghostscript	postscript interpreter	22,186	14,962	71,754	816,015	48.6%	2.7%	44.5%	1.9%	0.8%	0.4%	0.5%	0.6%	1,310.7s	86.3ms	30ms
total	–	2,594,912	1,708,172	4,901,021	45,695,687	49.5%	2.1%	44.4%	1.7%	0.5%	0.5%	0.4%	0.8%	352,586.7s	205.4ms	41ms

Table 2: Relative results.

References

Mustafa Al-Hajjaji, Fabian Benduhn, Thomas Thüm, Thomas Leich, and Gunter Saake. Mutation Operators for Preprocessor-Based Variability. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 81–88. ACM, 2016.

Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*, pages 323–333. IEEE, 2016.