Classifying Edits to Variability in Source Code
Appendix

This appendix accompanies our paper *Classifying Edits to Variability in Source Code* published at the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

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 diffs can be parameterized in their node types to support further language constructs. In Section 2, we prove that variation diffs are complete regarding edits to variation trees and that our edit classes 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 edits built from our edit classes or similar to our edit classes. 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 diffs by their set of supported node types, which in the paper is fixed to $\{$`artifact`, `mapping`, `else`$\}$. As an example, we then provide an extension of our definitions of variation trees and variation diffs to also support `elif` directives.

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

Type Correctness. 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 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 $\neg$ and has a neutral value *true* (in fact, we only need negation for `else` nodes as we will see later). 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 `f`ormula):

```
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 a 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 `l`ogic. (We continue this naming scheme when necessary.)

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

```
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`.

```haskell
    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 the type class of each requirement. 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, l) 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

[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>`).

branch. The label $l(v)$ is a propositional formula if $\tau(v) =$ `mapping`, a reference to an implementation artifact if $\tau(v) =$ `artifact`, or empty if $\tau(v) =$ `else`. The root r has type $\tau(r) =$ `mapping` and label $l(r) = \mathit{true}$. 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`:

```
type UUID = Int
```

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

```
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 $l(v)$ 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 l 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 $l(v)$ of a node v depends on the node's type $\tau(v)$. This implementation matches our original definition because we could define the functions τ and l of variation trees to just return the respective values that are stored within a node, but we omit these 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 l. As mentioned earlier, the type of the label $l(v)$ 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
    makeMappingLabel :: (Composable f) => f -> 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`.

```
    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 in the paper, 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

[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`.

use 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) = $ `mapping` and label $l(r) = \mathit{true}$, we introduce a constant for it, such that it is the same for all `VariationTrees`:

```
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 $l(r) = \mathit{true}$. 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 Diffs

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

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

To reason about variation 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`:

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

existsAtTime :: Time -> DiffType -> Bool
existsAtTime BEFORE ADD = False
```

```haskell
existsAtTime AFTER REM = False
existsAtTime _ _ = True
```

Analogous to our definition, we can introduce variation 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 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 diff. We may obtain the projection a given variation 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 mul-

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:

```
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`:

```
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]
```

---

tiple ways 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; }`.

```haskell
    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 {`artifact`, `mapping`, `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 `presencecondition` 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 diffs and for the completeness and unambiguity of our catalog of edit classes. The proofs for completeness and unambiguity are based on a proof scheme each, which can be reused to prove the respective property for other, custom catalogs of edit classes.

2.1 Completeness of Variation Diffs

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

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

To prove Theorem 1, we have to show that we can construct a variation 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 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 diffs may be condensed and in fact we condense variation 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

Proof of Completeness. To prove completeness of variation diffs, we have to show that given any two variation trees `t` and `u`, there exists at least one variation diff `d` that satisfies the above requirements. To find one such variation diff, we provide a diffing function that takes two variation trees and describes their differences in terms of a variation 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 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
```

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 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 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 diffs, we show that a variation diff created with naiveDiff is a valid variation 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 Proofs for Edit Classes

In this section, we prove that our catalog of edit classes is complete (i.e., every `artifact` node is in at least one class) and unambiguous (i.e., every `artifact` node is in at most one class). This means that every `artifact` node is in exactly one class.

For each proof, we first introduce a proof scheme that captures the proof's idea and structure. The purpose of the scheme is to provide instructions on how to prove the property of interest for any (valid) set of edit classes, not just the one we propose in our paper. Thus, each scheme is parameterized in a set of edit classes.

We then employ the introduced scheme to prove that our catalog of edit classes satisfies the respective property. The proof thereby also serves as an example on how to use the proof scheme, which is useful when building other edit class catalogs.

2.2.1 Completeness of Edit Classes

Theorem 2. *Every node in a variation diff with node type `artifact` is classified by at least one edit class.*

Proof Scheme. Let P be the set of edit class definitions which we want to prove to be complete, where each class' definition $p \in P$ is a predicate over an artifact node in a variation diff. To prove the completeness of P, we have to show that for all artifact nodes c, at least one predicate evaluates to *true*. We thus disjunct all predicates $p \in P$ because a disjunction evaluates to *true* if at least one of its clauses evaluates to *true*. We can thus prove the completeness of P by proving that the following formula is a tautology:

$$\forall c. \bigvee_{p \in P} p(c).$$

Verifying that this formula is a tautology can be done in multiple ways. Using equational reasoning, we could show that this formula simplifies to *true*. Another way is using a SAT solver because a formula φ is a tautology iff its negation is unsatisfiable (i.e., $\neg SAT(\neg\varphi)$). When using a SAT solver, references to the variable c may have to be replaced by a boolean constant.

Proof. Following our proof scheme, we prove the completeness of our edit class catalog by showing that

$$\forall c. \bigvee_{p \in P} p(c)$$

is a tautology by equational reasoning. Let c be any artifact node from a variation diff. Let P be the set of predicates defining the edit class catalog proposed in our paper:

$$\begin{aligned} P = \{ & \mathit{AddWithMapping}, \mathit{AddToPC}, \\ & \mathit{RemWithMapping}, \mathit{RemFromPC}, \\ & \mathit{Specialization}, \mathit{Generalization}, \\ & \mathit{Reconfiguration}, \mathit{Refactoring}, \mathit{Untouched}\}. \end{aligned}$$

We get:

$$\begin{aligned} & \bigvee_{p \in P} p(c) \\ \equiv \quad & \mathit{AddWithMapping}(c) \\ & \vee \mathit{AddToPC}(c) \\ & \vee \mathit{RemWithMapping}(c) \\ & \vee \mathit{RemFromPC}(c) \\ & \vee \mathit{Specialization}(c) \\ & \vee \mathit{Generalization}(c) \\ & \vee \mathit{Reconfiguration}(c) \\ & \vee \mathit{Refactoring}(c) \\ & \vee \mathit{Untouched}(c) \end{aligned}$$

$$
\begin{aligned}
\equiv\ & (\text{added}(c) \wedge \,\text{added}(M_{\text{a}}(c))) \\
& \vee(\text{added}(c) \wedge \neg\,\text{added}(M_{\text{a}}(c))) \\
& \vee(\text{removed}(c) \wedge \,\text{removed}(M_{\text{b}}(c))) \\
& \vee(\text{removed}(c) \wedge \neg\,\text{removed}(M_{\text{b}}(c))) \\
& \vee(\text{unchanged}(c) \wedge \neg(\text{PC}_{\text{b}}(c) \models \text{PC}_{\text{a}}(c)) \wedge (\text{PC}_{\text{a}}(c) \models \text{PC}_{\text{b}}(c))) \\
& \vee(\text{unchanged}(c) \wedge (\text{PC}_{\text{b}}(c) \models \text{PC}_{\text{a}}(c)) \wedge \neg(\text{PC}_{\text{a}}(c) \models \text{PC}_{\text{b}}(c))) \\
& \vee(\text{unchanged}(c) \wedge \neg(\text{PC}_{\text{b}}(c) \models \text{PC}_{\text{a}}(c)) \wedge \neg(\text{PC}_{\text{a}}(c) \models \text{PC}_{\text{b}}(c))) \\
& \vee(\text{unchanged}(c) \wedge (\text{PC}_{\text{b}}(c) \models \text{PC}_{\text{a}}(c)) \wedge (\text{PC}_{\text{a}}(c) \models \text{PC}_{\text{b}}(c)) \wedge (\mathit{path}_{\text{b}}(c) \neq \mathit{path}_{\text{a}}(c))) \\
& \vee(\text{unchanged}(c) \wedge (\text{PC}_{\text{b}}(c) \models \text{PC}_{\text{a}}(c)) \wedge (\text{PC}_{\text{a}}(c) \models \text{PC}_{\text{b}}(c)) \wedge (\mathit{path}_{\text{b}}(c) = \mathit{path}_{\text{a}}(c))) \\
\equiv\ & (\text{added}(c) \wedge (\text{added}(M_{\text{a}}(c)) \vee \neg\,\text{added}(M_{\text{a}}(c)))) \\
& \vee(\text{removed}(c) \wedge (\text{removed}(M_{\text{b}}(c)) \vee \neg\,\text{removed}(M_{\text{b}}(c)))) \\
& \vee\Big(\text{unchanged}(c) \wedge \\
& \quad ((\neg(\text{PC}_{\text{b}}(c) \models \text{PC}_{\text{a}}(c)) \wedge (\text{PC}_{\text{a}}(c) \models \text{PC}_{\text{b}}(c))) \\
& \quad \vee((\text{PC}_{\text{b}}(c) \models \text{PC}_{\text{a}}(c)) \wedge \neg(\text{PC}_{\text{a}}(c) \models \text{PC}_{\text{b}}(c))) \\
& \quad \vee(\neg(\text{PC}_{\text{b}}(c) \models \text{PC}_{\text{a}}(c)) \wedge \neg(\text{PC}_{\text{a}}(c) \models \text{PC}_{\text{b}}(c))) \\
& \quad \vee((\text{PC}_{\text{b}}(c) \models \text{PC}_{\text{a}}(c)) \wedge (\text{PC}_{\text{a}}(c) \models \text{PC}_{\text{b}}(c)) \wedge (\mathit{path}_{\text{b}}(c) \neq \mathit{path}_{\text{a}}(c))) \\
& \quad \vee((\text{PC}_{\text{b}}(c) \models \text{PC}_{\text{a}}(c)) \wedge (\text{PC}_{\text{a}}(c) \models \text{PC}_{\text{b}}(c)) \wedge (\mathit{path}_{\text{b}}(c) = \mathit{path}_{\text{a}}(c))))\Big) \\
\equiv\ & (\text{added}(c) \wedge \mathit{true}) \\
& \vee(\text{removed}(c) \wedge \mathit{true}) \\
& \vee\Big(\text{unchanged}(c) \wedge \\
& \quad ((\neg(\text{PC}_{\text{b}}(c) \models \text{PC}_{\text{a}}(c)) \wedge (\text{PC}_{\text{a}}(c) \models \text{PC}_{\text{b}}(c))) \\
& \quad \vee((\text{PC}_{\text{b}}(c) \models \text{PC}_{\text{a}}(c)) \wedge \neg(\text{PC}_{\text{a}}(c) \models \text{PC}_{\text{b}}(c))) \\
& \quad \vee(\neg(\text{PC}_{\text{b}}(c) \models \text{PC}_{\text{a}}(c)) \wedge \neg(\text{PC}_{\text{a}}(c) \models \text{PC}_{\text{b}}(c))) \\
& \quad \vee((\text{PC}_{\text{b}}(c) \models \text{PC}_{\text{a}}(c)) \wedge (\text{PC}_{\text{a}}(c) \models \text{PC}_{\text{b}}(c)) \wedge \mathit{true}))\Big) \\
\equiv\ & \text{added}(c) \vee \text{removed}(c) \\
& \vee(\text{unchanged}(c) \wedge \mathit{true}) \\
\equiv\ & \text{added}(c) \vee \text{removed}(c) \vee \text{unchanged}(c) \\
\equiv\ & \mathit{true}
\end{aligned}
$$

because exactly one clause of added(c), removed(c), and unchanged(c) evaluates to *true* while the others evaluate to *false* (because the the type of change made to c is given by $\Delta(c)$ which is exactly one value of $\{+, -, \bullet\}$). □

2.2.2 Unambiguity of Edit Classes

Theorem 3. *Every node in a variation diff with node type* `artifact` *is classified by at most one class.*

Proof Scheme Let P be the set of edit class definitions which we want to prove to be unambiguous, where each class' definition $p \in P$ is a predicate over an artifact node in a variation diff. To prove the unambiguity of P, we have to show that for all artifact nodes c, at most one predicate evaluates to *true*. This means, that all predicates are alternative to each other; when a predicate p evaluates to *true*, all other predicates q must yield *false*:

$$\forall c. \forall p, q \in P, p \neq q. p(c) \Rightarrow \neg q(c).$$

Another, equivalent interpretation of this formula is, that for any disjunct pair of predicates p, q, not both predicates can evaluate to *true* simultaneously.

$$\forall c. \forall p, q \in P, p \neq q. \neg(p(c) \land q(c)).$$

By showing that either formula is a tautology, we prove that P is unambiguous.

Proof. Following our proof scheme, we prove the unambiguity of our edit class catalog by showing that

$$\forall c. \forall p, q \in P, p \neq q. \neg(p(c) \land q(c)).$$

is a tautology. Let c be any artifact node from a variation diff. Let P be the edit class catalog proposed in our paper:

$$\begin{aligned} P = \{ & \mathit{AddWithMapping}, \mathit{AddToPC}, \\ & \mathit{RemWithMapping}, \mathit{RemFromPC}, \\ & \mathit{Specialization}, \mathit{Generalization}, \\ & \mathit{Reconfiguration}, \mathit{Refactoring}, \mathit{Untouched}\} \end{aligned}$$

Each of our classes $p \in P$, by definition, is a conjunction of (sub-)predicates S_p (i.e., $p = \bigwedge_{s \in S_p} s(c)$). For example, *AddWithMapping* is a conjunction of the two (sub-)predicates $\mathrm{added}(c)$ and $\mathrm{added}(M_{\mathrm{a}}(c))$. This means, each class can only evaluate to *true* if all its (sub-)predicates S_p evaluate to *true*.

Given any two classes $p, q \in P$, we see that there is always at least one (sub-)predicate $\psi \in S_p$ with $\psi \models \neg\kappa$ and $\kappa \in S_q$. (Note, that the three predicates added(c), removed(c), and unchanged(c) are alternative to each other by definition.) Thus, each class contains at least one (sub-)predicate that will become *false* when another class evaluates to *true*. Thus, no two disjunct classes can evaluate to *true* simultaneously. $\square$

3 Composite Edits

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

3.1 Al-Hajjaji et al. [2016]

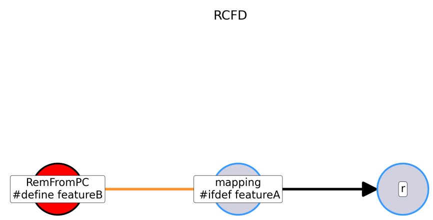
Al-Hajjaji et al. provide a set of mutation operators to preprocessor-based variability. We consider all edits to source code and preprocessor directives here but not those to the variability model. Al-Hajjaji et al. define all operators in terms of a natural language description and a generic example. 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 code from edits to macro definitions (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 classes of our catalog. While, we do not distinguish between `#define` directives and pure source code in `artifact` nodes for our edit classes, such a differentiation is still possible when inspecting the label of `artifact` nodes.

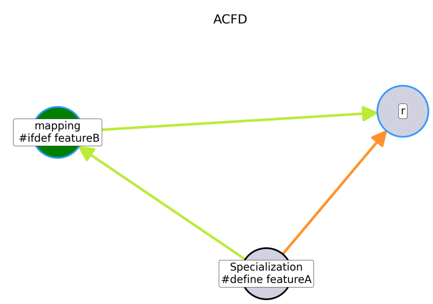
RCFD – Remove Conditional Feature Definition

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



ACFD – Add Condition to Feature Definition

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

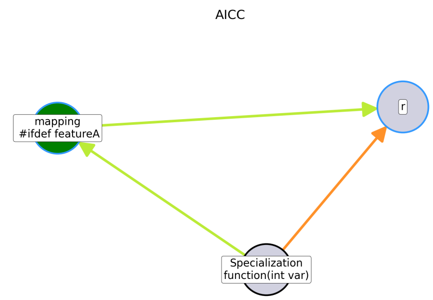


3.1.2 Variability-Mapping Operators

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

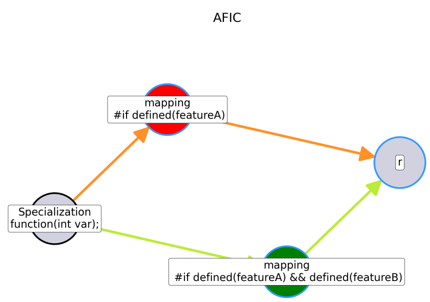
AICC – Adding ifdef Condition around Code

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



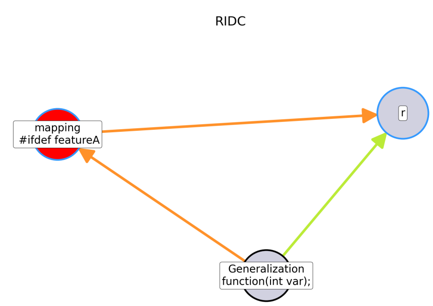
AFIC – Adding Feature to ifdef Condition

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



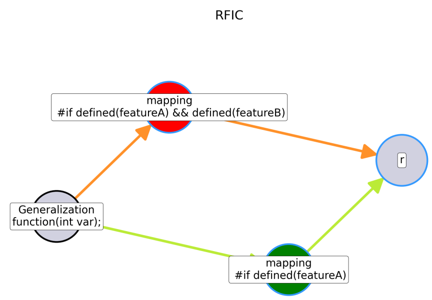
RIDC – Removing ifdef Condition

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



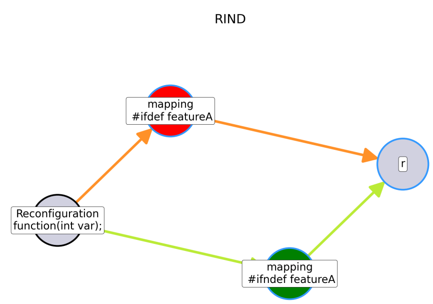
RFIC – Removing Feature of ifdef Condition

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



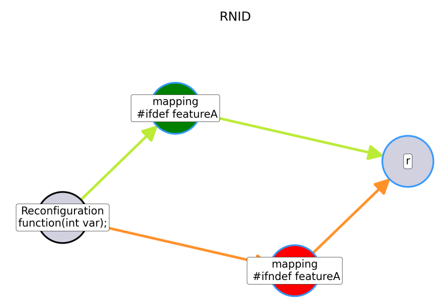
RIND – Replacing ifdef Directive with ifndef Directive

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



RNID – Replacing ifndef Directive with ifdef Directive

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



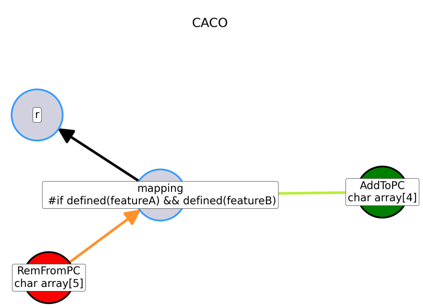
3.1.3 Domain Artifact Operators

The domain artifact operators describe changes to source code.

CACO – Conditionally Applying Conventional Operator

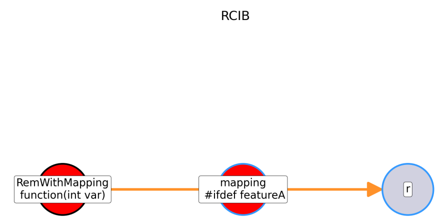
CACO applies conventional source code mutation operators in a variability-aware way. It modifies source code that has a certain presence condition. In a diff, such a modification occurs as the removal and insertion of source code and thus CACO is a composite edit built from a *RemFromPC* and *AddToPC*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

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



RCIB – Removing Complete ifdef Block

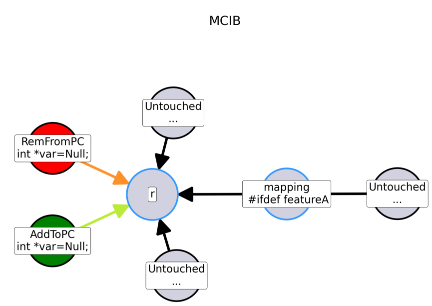
```
-#ifdef featureA
-function(int var)
-#endif
```



MCIB – Moving Code around ifdef Blocks

MCIB moves code around an `#ifdef` block. As discussed in the discussion section for our edit classes 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 classes. Yet, the operators are incomplete, as for example *AddWithMapping* and thus a non-empty subset of edits is missing. On the other hand, the operators distinguish more cases for single classes, for example if a `#define` directive or source code was specialized. Our catalog of classes could be extended by distinguishing such sub-cases for different classes in the future (in particular, by adding further clauses to the definitions of classes), while remaining complete.

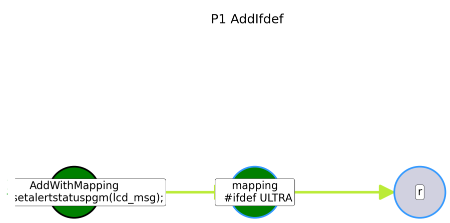
3.2 Stănciulescu et al. [2016]

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

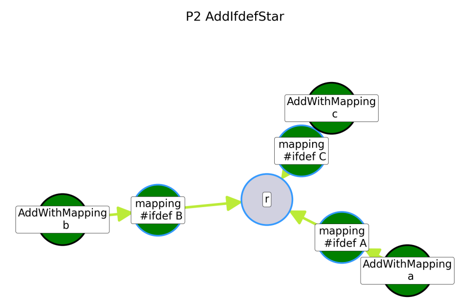
```
+ #ifdef ULTRA_LCD
+  lcd_setalertstatuspgm(lcd_msg)
+ #endif
```



P2 AddIfdef*

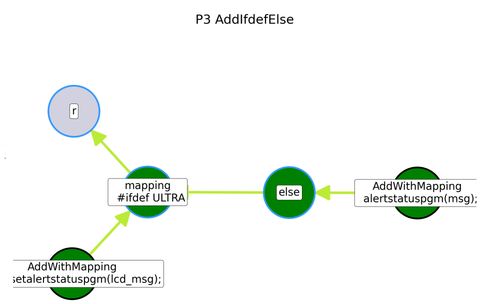
AddIfdef* is the repeated application of the AddIfdef pattern (two or more times). Thus, AddIfdef* is a composite edit pattern, built from two or more *AddWithMapping* instances. We show an example with three consecutive applications of the AddIfdef pattern:

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

+ #ifdef B
+  b
+ #endif

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



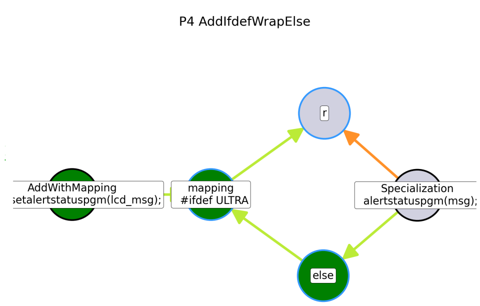
P3 AddIfdefElse

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



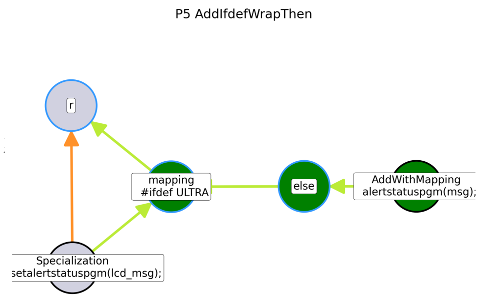
P4 AddIfdefWrapElse

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



P5 AddIfdefWrapThen

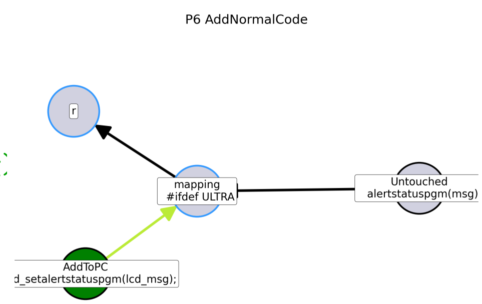
```
+ #ifdef ULTRA_LCD
    lcd_setalertstatuspgm(lcd_msg)
+ #else
+   alertstatuspgm(msg);
+ #endif
```



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* class.

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



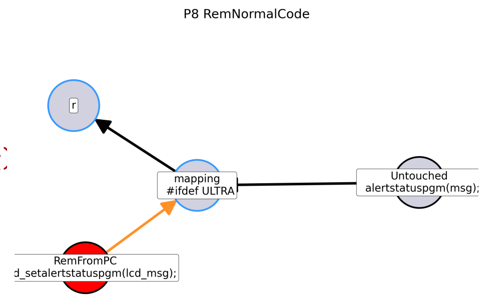
P7 AddAnnotation

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

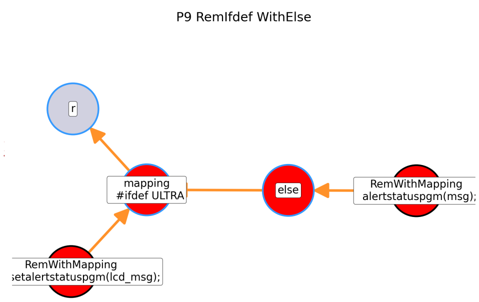
```
  #ifdef ULTRA_LCD
-   lcd_setalertstatuspgm(lcd_msg
   alertstatuspgm(msg);
  #endif
```



P9 RemIfdef

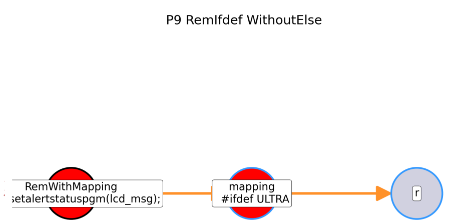
This pattern has two cases and thus actually describes two patterns. RemIfdef matches the removal of source code with its surrounding `#ifdef` and `#else` annotations

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



or without an `#else` annotation:

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

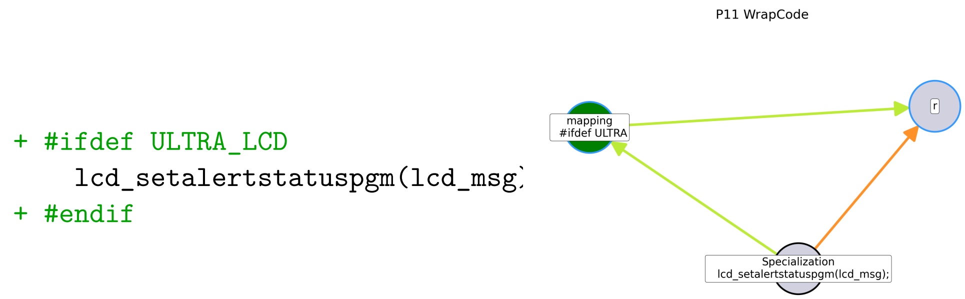


P10 RemAnnotation

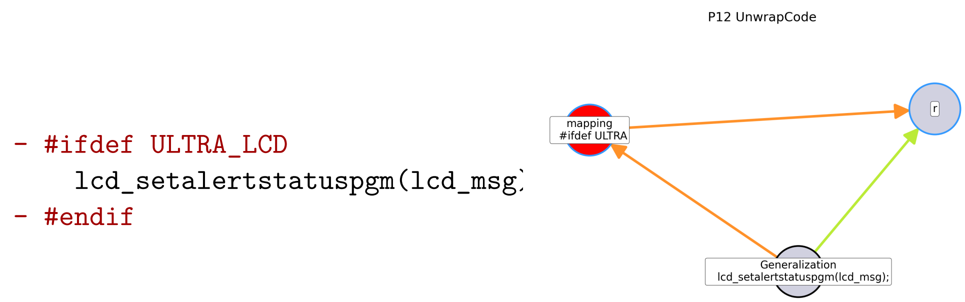
This pattern matches fixes to syntactically incorrect annotations by removal of `#ifdef` or `#endif` directives. This pattern is neither supported by our catalog of edit classes nor the variation control system by Stănciulescu et al..

3.2.3 Other Patterns

P11 WrapCode

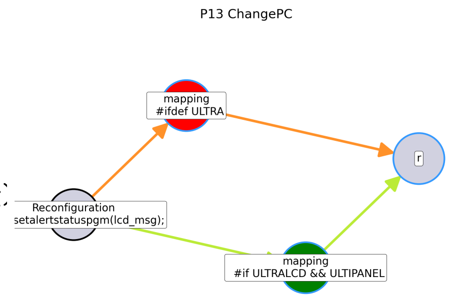


P12 UnwrapCode



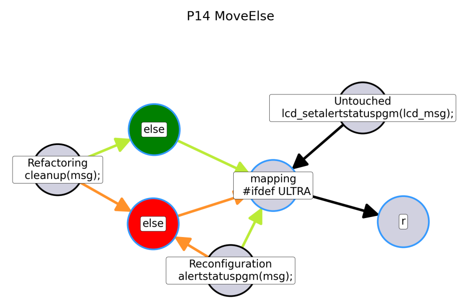
P13 ChangePC

```
- #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 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 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 classes and composite edit 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	294,314	147,118	6,836	128,702	5,380	2,193	1,674	1,529	882	87.4s	10.5ms	5ms
freebsd	operating system	272,207	179,753	729,831	9,747,920	4,815,697	261,187	4,213,463	223,507	71,578	73,579	60,241	28,668	5,418.2s	29.6ms	5ms
gcc	compiler framework	191,405	122,777	416,750	3,040,492	1,524,145	48,195	1,394,357	38,026	12,384	11,164	9,217	3,004	6,023.5s	47.4ms	21ms
openldap	LDAP directory service	23,938	17,633	56,581	376,091	180,201	13,835	155,675	11,587	5,169	5,715	2,424	1,485	299.4s	16.8ms	6ms
postgresql	database system	52,934	31,447	158,241	1,734,906	880,642	14,818	815,802	12,415	3,505	3,418	1,560	2,746	859.8s	26.7ms	10ms
mpsolve	mathematical software	1,773	1,326	5,395	51,544	26,663	205	24,421	102	72	6	51	24	7.6s	5.6ms	3ms
mplayer-svn	media player	37,992	22,803	46,288	327,632	157,789	10,424	139,341	7,935	3,108	3,240	4,185	1,610	405.9s	17.3ms	5ms
linux	operating system	1,072,601	870,429	1,875,864	12,483,635	6,431,565	115,095	5,716,418	113,877	30,783	42,941	24,307	8,649	10,829.9s	12.2ms	6ms
sylpheed	e-mail client	2,682	1,820	4,237	30,718	16,943	752	11,959	392	380	217	17	58	18.7s	10.1ms	8ms
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	39,535	301,848	146,654	12,773	114,921	11,990	3,203	8,094	2,249	1,964	544.0s	37.6ms	24ms
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	353.0s	22.2ms	11ms
xorg-server	X server	17,788	14,401	53,690	526,911	245,937	26,286	222,543	14,860	4,480	7,288	2,082	3,435	174.9s	12.1ms	5ms
tcl	program interpreter	24,414	10,188	25,762	408,717	182,935	23,342	171,796	21,268	3,288	3,670	1,048	1,370	233.0s	22.6ms	12ms
godot	game engine	40,944	18,867	107,845	1,267,555	613,239	42,292	540,778	38,193	6,791	4,676	18,153	3,433	2,687.2s	141.8ms	7ms
openvpn	security application	3,128	2,357	8,673	82,594	37,973	3,713	34,592	2,292	681	1,433	1,572	338	38.0s	16.0ms	9ms
privoxy	proxy server	7,558	2,634	4,278	37,710	18,764	1,439	15,254	1,083	362	283	446	79	30.1s	10.8ms	7ms
libssh	network	5,352	4,439	8,465	56,440	29,369	1,404	23,661	976	604	201	138	87	21.3s	4.8ms	3ms
marlin	3d printing	19,260	14,607	84,332	567,164	215,781	69,162	198,315	49,367	3,166	9,240	16,416	5,717	549.0s	37.4ms	10ms
opensolaris	operating system	11,422	9,691	53,928	997,484	501,954	16,865	450,953	11,569	8,177	5,348	1,106	1,512	363.0s	37.3ms	15ms
sqlite	databases	8,664	6,358	15,643	172,947	86,524	3,956	75,568	3,108	1,584	1,154	722	331	130.8s	20.4ms	13ms
busybox	embedded systems	17,447	14,485	41,146	393,324	180,133	14,433	170,797	14,332	3,973	2,665	4,858	2,133	244.8s	16.8ms	6ms
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	8,978	149,490	69,601	9,289	58,341	3,451	5,215	1,155	256	2,182	124.8s	32.9ms	20ms
emacs	text editor	154,155	37,943	71,321	571,109	271,660	18,567	248,295	15,167	5,914	5,602	3,127	2,777	1,349.1s	33.0ms	14ms
apache-httpd	web server	32,953	15,985	35,090	287,547	142,786	5,074	128,455	6,026	1,739	2,005	885	577	384.3s	22.4ms	7ms
minix	operating system	7,153	4,624	35,413	590,559	271,330	28,584	251,131	21,372	4,289	2,975	7,659	3,219	82.1s	16.6ms	4ms
lighttpd	web server	4,433	3,480	9,521	76,382	36,730	2,937	33,108	1,954	730	432	204	287	38.1s	10.8ms	6ms
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	86.6s	12.9ms	9ms
xfig	vector graphics editor	9	4	19	84	44	2	37	0	1	0	0	0	0.0s	8.5ms	12ms
subversion	revision control system	60,037	36,204	88,578	742,737	384,817	5,011	344,153	3,905	1,569	1,755	978	549	705.1s	19.1ms	8ms
xterm	terminal emulator	112	108	1,280	29,260	14,535	1,406	12,433	494	180	99	56	57	12.6s	116.2ms	111ms
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	168,676	1,800,906	913,538	12,066	853,449	12,358	3,608	2,924	2,342	621	660.9s	20.3ms	8ms
berkeley-db-libdb	database system	7	1	485	3,314	1,760	58	1,431	32	28	0	0	5	0.6s	595.0ms	595ms
cpython	program interpreter	112,196	28,444	59,123	947,950	466,515	23,186	430,128	11,485	8,506	4,736	1,820	1,574	661.9s	20.4ms	8ms
cherokee-webserver	web server	5,853	2,170	7,879	46,746	24,236	595	20,853	588	119	181	69	105	16.7s	7.1ms	4ms
php	program interpreter	127,632	65,413	175,846	3,010,633	1,476,592	58,396	1,385,124	40,571	16,354	12,428	11,452	9,716	2,051.8s	30.8ms	7ms
pidgin	instant messenger	40,097	27,354	72,271	730,359	364,174	11,648	330,161	11,402	4,418	3,304	4,097	1,155	662.5s	24.0ms	8ms
glibc	programming library	38,349	23,480	191,204	840,886	393,256	32,156	365,428	23,702	11,087	6,526	6,328	2,403	756.1s	31.6ms	4ms
dia	diagramming software	6,666	3,694	16,943	146,820	75,346	1,810	65,538	1,952	845	1,067	142	120	30.8s	8.1ms	4ms
parrot	virtual machine	49,989	13,806	40,699	934,193	464,463	8,342	446,884	7,768	2,125	1,741	1,233	1,637	236.1s	15.7ms	6ms
irssi	IRC client	6,346	4,052	9,736	53,275	28,592	543	23,156	492	180	139	150	23	14.9s	3.6ms	2ms
ghostscript	postscript interpreter	22,186	14,962	71,753	814,078	397,290	21,267	363,652	15,158	6,460	2,985	3,819	3,447	363.9s	23.3ms	8ms
total	–	2,594,912	1,708,111	4,900,820	45,413,708	22,612,208	939,623	20,314,666	768,806	241,706	239,159	198,435	99,105	37,558.1s	21.5ms	7ms

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	294,314	50.0%	2.3%	43.7%	1.8%	0.7%	0.6%	0.5%	0.3%	87.4s	10.5ms	5ms
freebsd	operating system	272,207	179,753	729,831	9,747,920	49.4%	2.7%	43.2%	2.3%	0.7%	0.8%	0.6%	0.3%	5,418.2s	29.6ms	5ms
gcc	compiler framework	191,405	122,777	416,750	3,040,492	50.1%	1.6%	45.9%	1.3%	0.4%	0.4%	0.3%	0.1%	6,023.5s	47.4ms	21ms
openldap	LDAP directory service	23,938	17,633	56,581	376,091	47.9%	3.7%	41.4%	3.1%	1.4%	1.5%	0.6%	0.4%	299.4s	16.8ms	6ms
postgresql	database system	52,934	31,447	158,241	1,734,906	50.8%	0.9%	47.0%	0.7%	0.2%	0.2%	0.1%	0.2%	859.8s	26.7ms	10ms
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%	7.6s	5.6ms	3ms
mplayer-svn	media player	37,992	22,803	46,288	327,632	48.2%	3.2%	42.5%	2.4%	0.9%	1.0%	1.3%	0.5%	405.9s	17.3ms	5ms
linux	operating system	1,072,601	870,429	1,875,864	12,483,635	51.5%	0.9%	45.8%	0.9%	0.2%	0.3%	0.2%	0.1%	10,829.9s	12.2ms	6ms
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%	18.7s	10.1ms	8ms
sendmail	mail transfer agent	86	0	0	0	–%	–%	–%	–%	–%	–%	–%	–%	0.0s	–ms	–ms
vim	text editor	15,354	14,453	39,535	301,848	48.6%	4.2%	38.1%	4.0%	1.1%	2.7%	0.7%	0.7%	544.0s	37.6ms	24ms
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%	353.0s	22.2ms	11ms
xorg-server	X server	17,788	14,401	53,690	526,911	46.7%	5.0%	42.2%	2.8%	0.9%	1.4%	0.4%	0.7%	174.9s	12.1ms	5ms
tcl	program interpreter	24,414	10,188	25,762	408,717	44.8%	5.7%	42.0%	5.2%	0.8%	0.9%	0.3%	0.3%	233.0s	22.6ms	12ms
godot	game engine	40,944	18,867	107,845	1,267,555	48.4%	3.3%	42.7%	3.0%	0.5%	0.4%	1.4%	0.3%	2,687.2s	141.8ms	7ms
openvpn	security application	3,128	2,357	8,673	82,594	46.0%	4.5%	41.9%	2.8%	0.8%	1.7%	1.9%	0.4%	38.0s	16.0ms	9ms
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%	30.1s	10.8ms	7ms
libssh	network	5,352	4,439	8,465	56,440	52.0%	2.5%	41.9%	1.7%	1.1%	0.4%	0.2%	0.2%	21.3s	4.8ms	3ms
marlin	3d printing	19,260	14,607	84,332	567,164	38.0%	12.2%	35.0%	8.7%	0.6%	1.6%	2.9%	1.0%	549.0s	37.4ms	10ms
opensolaris	operating system	11,422	9,691	53,928	997,484	50.3%	1.7%	45.2%	1.2%	0.8%	0.5%	0.1%	0.2%	363.0s	37.3ms	15ms
sqlite	databases	8,664	6,358	15,643	172,947	50.0%	2.3%	43.7%	1.8%	0.9%	0.7%	0.4%	0.2%	130.8s	20.4ms	13ms
busybox	embedded systems	17,447	14,485	41,146	393,324	45.8%	3.7%	43.4%	3.6%	1.0%	0.7%	1.2%	0.5%	244.8s	16.8ms	6ms
lynx	web browser	125	0	0	0	–%	–%	–%	–%	–%	–%	–%	–%	0.0s	–ms	–ms
libxml2	XML library	5,146	3,767	8,978	149,490	46.6%	6.2%	39.0%	2.3%	3.5%	0.8%	0.2%	1.5%	124.8s	32.9ms	20ms
emacs	text editor	154,155	37,943	71,321	571,109	47.6%	3.3%	43.5%	2.7%	1.0%	1.0%	0.5%	0.5%	1,349.1s	33.0ms	14ms
apache-httpd	web server	32,953	15,985	35,090	287,547	49.7%	1.8%	44.7%	2.1%	0.6%	0.7%	0.3%	0.2%	384.3s	22.4ms	7ms
minix	operating system	7,153	4,624	35,413	590,559	45.9%	4.8%	42.5%	3.6%	0.7%	0.5%	1.3%	0.5%	82.1s	16.6ms	4ms
lighttpd	web server	4,433	3,480	9,521	76,382	48.1%	3.8%	43.3%	2.6%	1.0%	0.6%	0.3%	0.4%	38.1s	10.8ms	6ms
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%	86.6s	12.9ms	9ms
xfig	vector graphics editor	9	4	19	84	52.4%	2.4%	44.0%	0.0%	1.2%	0.0%	0.0%	0.0%	0.0s	8.5ms	12ms
subversion	revision control system	60,037	36,204	88,578	742,737	51.8%	0.7%	46.3%	0.5%	0.2%	0.2%	0.1%	0.1%	705.1s	19.1ms	8ms
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%	12.6s	116.2ms	111ms
xine-lib	media library	114	0	0	0	–%	–%	–%	–%	–%	–%	–%	–%	0.0s	–ms	–ms
gimp	graphics editor	47,836	31,700	168,676	1,800,906	50.7%	0.7%	47.4%	0.7%	0.2%	0.2%	0.1%	0.0%	660.9s	20.3ms	8ms
berkeley-db-libdb	database system	7	1	485	3,314	53.1%	1.8%	43.2%	1.0%	0.8%	0.0%	0.0%	0.2%	0.6s	595.0ms	595ms
cpython	program interpreter	112,196	28,444	59,123	947,950	49.2%	2.4%	45.4%	1.2%	0.9%	0.5%	0.2%	0.2%	661.9s	20.4ms	8ms
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%	16.7s	7.1ms	4ms
php	program interpreter	127,632	65,413	175,846	3,010,633	49.0%	1.9%	46.0%	1.3%	0.5%	0.4%	0.4%	0.3%	2,051.8s	30.8ms	7ms
pidgin	instant messenger	40,097	27,354	72,271	730,359	49.9%	1.6%	45.2%	1.6%	0.6%	0.5%	0.6%	0.2%	662.5s	24.0ms	8ms
glibc	programming library	38,349	23,480	191,204	840,886	46.8%	3.8%	43.5%	2.8%	1.3%	0.8%	0.8%	0.3%	756.1s	31.6ms	4ms
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%	30.8s	8.1ms	4ms
parrot	virtual machine	49,989	13,806	40,699	934,193	49.7%	0.9%	47.8%	0.8%	0.2%	0.2%	0.1%	0.2%	236.1s	15.7ms	6ms
irssi	IRC client	6,346	4,052	9,736	53,275	53.7%	1.0%	43.5%	0.9%	0.3%	0.3%	0.3%	0.0%	14.9s	3.6ms	2ms
ghostscript	postscript interpreter	22,186	14,962	71,753	814,078	48.8%	2.6%	44.7%	1.9%	0.8%	0.4%	0.5%	0.4%	363.9s	23.3ms	8ms
total	–	2,594,912	1,708,111	4,900,820	45,413,708	49.8%	2.1%	44.7%	1.7%	0.5%	0.5%	0.4%	0.2%	37,558.1s	21.5ms	7ms

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. doi: 10.1145/2866614.2866626.

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. doi: 10.1109/ICSME.2016.88.