Change Pattern Detection for Optimising Incremental Static Analysis

Cindy Wauters*, Jens Van der Plas*, Quentin Stiévenart†, Coen De Roover*
*Vrije Universiteit Brussel, Brussels, Belgium
{cindy.suzy.wauters, jens.van.der.plas, coen.de.roover}@vub.be
†Université du Québec à Montréal, Montréal, Canada
stievenart.quentin@uqam.ca

***Abstract*—Static analyses can be used by developers to compute properties of a program, enabling e.g., bug detection and program verification. However, reanalysing a program from scratch upon every change is time-consuming, especially in settings where code changes often, such as within IDEs. To avoid such full reanalyses, incremental analyses instead reuse parts of the previous analysis result, and reanalyse the changed code as necessary.**

While incrementality improves the analysis time, we introduce a complementary approach that further reduces the analysis time. A traditional incremental analysis updates previous analysis results without domain-specific knowledge. However, the effect of particular source code changes on analysis results can be predicted. Performing a traditional incremental analysis of the changed code might therefore be unnecessary. Instead, we propose to detect code change patterns of which the effect on analysis results can be predicted and to update these results accordingly, saving potentially expensive computations.

In this paper, we explore the idea of adapting the analysis results for behaviour-preserving change patterns. In particular, we consider consistent renamings, inverted conditionals, and moved function definitions within Scheme programs. We implemented our approach and evaluated it on 30 programs. We show decreases in incremental analysis time between 3% and 99% on 25 programs that contain at least one behaviour-preserving change pattern.

***Index Terms*—Static Program Analysis, Incremental Analysis, Modular Analysis, Refactoring**

I. Introduction

Static analysis is a useful tool for developers as it computes program properties without the need for running the program. It can be used for bug detection and program verification. Unfortunately, performing a static analysis can be very time-consuming. A code base might also change often, in which case performing an analysis of the entire program upon every change might not be preferred. Incremental analyses, therefore, reuse parts of the results of the previous static analysis of that program to speed up the process.

We build on top of previous work by Van der Plas et al. [1], [2], which introduces an approach to incrementalise effect-driven modular analyses [3]. Their idea is to reanalyse changed code, and to make use of dependencies to reanalyse dependent code if an analysis result did change. While this approach reduces the analysis time for most of their benchmarks, the incremental analysis can still take longer than expected, especially when there are many dependencies on the changed component(s). Additionally, this approach does not employ domain-specific knowledge. However, changes with a predictable effect on the analysis results exist, and these can be processed *prior to* the incremental analysis.

As an example, consider a static analysis for the Scheme[1] program where Listing 1 represents a program before a change and Listing 2 represents the same program after a change. In this example, a renaming is performed: the parameter `x` of the function `f` is renamed to `y`. During the incremental analysis, every reference to `x` will eventually be updated to `y`. This happens by performing a reanalysis of the changed code. Table I shows the analysis results before and after this reanalysis. We will discuss this example in more detail in Section IV.

Because we know how a renaming affects the analysis results, we can instead perform the result updates immediately, avoiding the more expensive incremental analysis by means of abstract interpretation. In this work, we introduce a novel approach to make an incremental analysis more efficient, by detecting change patterns that have predictable effects on existing analysis results in advance, and by updating the analysis results accordingly. Our approach is lightweight and ensures that only changes not following known patterns will need to be reanalysed. Concretely, we make the following contributions:

- We propose three change patterns with a predictable effect on the analysis result. We focus on behaviour-preserving change patterns, namely:
  - *Consistent renamings*: renaming a variable or function and updating all references to it,
  - *Inverted conditionals*: negating the condition of an if expression and switching its then and else branches,
  - *Moved function definitions*: moving a function definition up or down (one or more) scope level(s), to make the definition more local or global;
- After the detection of change patterns, the results of the initial modular analysis are updated accordingly;
- We evaluate our approach and find a decrease in running time between 3% and 99% on 25 out of 30 programs that contain at least one of these refactorings (with some programs also containing other changes).

[1]Without loss of generality, we present our approach for this famous Lisp dialect. It is representative of other dynamically-typed, higher-order programming languages such as Python or JavaScript, for which no static call graph can be computed and to which our approach can be ported.

Listing 1
OLD VERSION OF THE CODE

```
(define f (lambda (x)
            (define g (lambda() x))
            (g)))
(f 5)
```

Listing 2
NEW VERSION OF THE CODE

```
(define f (lambda (y)
            (define g (lambda() y))
            (g)))
(f 5)
```

TABLE I
STORE OF AN ANALYSIS AFTER THE INITIAL ANALYSIS (LEFT) AND AFTER UPDATING ACCORDING TO THE CONSISTENT RENAMING (RIGHT).

Address	Abstract value
Return ((λ () x)@1:22, {x@0:19})	Int
Return ((λ (x) ... (g))@0:10, {})	Int
Variable x@0:19	Int
Variable f@0:8	((λ (x) ... (g))@1:22, {})
Variable g@1:20	((λ () x)@1:22, {x@0:19})
...	...

Address	Abstract value
Return ((λ () y)@1:22, {y@0:19})	Int
Return ((λ (y) ... (g))@0:10, {})	Int
Variable y@0:19	Int
Variable f@0:8	((λ (y) ... (g))@1:22, {})
Variable g@1:20	((λ () y)@1:22, {y@0:19})
...	...

Our contributions aim to make static analysis within IDEs more efficient so it becomes within reach of every developer. In doing so, bugs will be easier to detect and understand, allowing for easier bug fixes.

In this paper, we first provide an overview of the required background. Afterwards, we describe our approach, going into both the detection of behaviour-preserving change patterns and into the updating the analysis results, which we illustrate in Section IV using the previous example containing a renaming. Then, we evaluate our approach and discuss the results, future work, and related work.

II. BACKGROUND

This paper consists of two main topics: incremental modular static analysis, and detecting behaviour-preserving change patterns (refactorings).

A. *Incremental Modular Static Analysis*

A modular analysis [4] divides the program under analysis into different parts (e.g., function definitions), called modules, each of which can have multiple runtime instantiations (e.g., function calls). The reification of such an instantiation in the analysis is called a component, and a modular analysis analyses its components in isolation. Modular analyses are scalable [5] and lend themselves well to programs written in highly dynamic languages with support for higher-order functions. We focus on a function-modular effect-driven analysis as proposed by Nicolay et al. [3], as this is built upon by Van der Plas et al. [1]. The analysis of Nicolay et al. divides the program into function definitions, but the concepts we introduce generalise to other modular analyses as well.

Although analysed in isolation, components may interact with each other, meaning that the analysis result of one component can depend on the analysis result of another. For example, when a function is called, it receives argument values from its caller. A component therefore might have to be reanalysed when another component updated an analysis result. An effect-driven modular analysis is therefore composed of two interleaved fixed-point computations: an *intra-component analysis* and an *inter-component analysis*. The former analyses one component at a time, while the latter takes care of which components have to be analysed next.

Van der Plas et al. [1] propose an incremental analysis that uses this modularity by scheduling the components that correspond to changed modules for analysis. Due to the dependencies between the components, other components can still be scheduled for reanalysis later, when it is inferred that they are also affected by the changes. Their approach is applied to both a function-modular and a thread-modular analysis, and their results show a 6% to 99% decrease in analysis time on 14 out of 16 benchmark programs. In this paper, we only focus on function-modular analysis. In a function-modular analysis, components correspond to function calls: each function call is analysed in isolation during the intra-component analyses. Functions can be dependent on each other (a function can call another function), which creates dependencies between the different components. Therefore, when the analysis results of one function call is updated, every component that is dependent on that result will be reanalysed accordingly.

In this paper, we use a type analyses performed using abstract interpretation [6]. However, our approach is also applicable to other static analyses that do not use abstract interpretation, as well as to static analyses through abstract interpretation with other abstract domains.

B. *Behaviour-preserving Change Patterns*

Code bases are constantly updated and changed throughout their lifespan. Sometimes, the code base is updated only structurally, which does not change the actual program behaviour. This is known as refactoring [7]. Many developers perform refactorings to improve the readability and usability of their code bases. Besides improving the quality of the software, it can also lead to better development productivity [8]. In fact, up to 16% of all changes in code bases may be refactorings [9]. Additionally, Murphy-Hill [10] found that up to 41% programming sessions contained refactorings, with multiple refactorings being performed at once.

III. APPROACH

In this section, we discuss the detection of specific change patterns and the corresponding updating of the analysis results.

A. Detecting Behaviour-preserving Change Patterns

Before we can update the initial analysis result, we must first detect the changes that follow a specific pattern, i.e., the changes with known effects on the analysis results. Unfortunately, refactorings might be performed incorrectly [11], [12], thereby changing the behaviour of the program and leaving the effect on the analysis results unpredictable. Such errors can occur even when automated refactoring tools are available [13], due to bugs in the automated tools themselves [14]. To update the analysis results correctly, we only want to detect refactorings that are applied correctly.

In order to focus only on the refactorings, rather than on change detection as a whole, we assume that there is already a mechanism in place to detect changes in source code; we focus on detecting whether those changes indeed follow a specific change pattern. Detecting changes can be done by using tools or annotations in the code base. Our approach follows an extended version of the change annotations used by Van der Plas et al. [1]. Programs are pre-annotated with the changes, containing both the old and the new version of the code in one file. Annotations may indicate updates, insertions or deletions.

To avoid false positives in the detection of patterns, currently, our approach does not always detect nested refactorings (such as a moved function definition that has also been renamed). The incremental static analysis by Van der Plas et al. [1] is *sound*, i.e., there are no *false negatives* in its results. To preserve soundness, it is important to avoid *false positives* during the pattern-detection phase: a change that does not follow a particular change pattern should never be flagged as one that does. A change pattern that is not detected (false negative) will simply be subject to reanalysis without updating. This means that the incremental analysis will not be faster than before, but it preserves its soundness. False positives in the pattern-detection phase can, however, lead to incorrect updates, which can make the analysis unsound.

This paper focusses on three change patterns in particular. We decided on consistent renamings and moved function definitions due to their popularity in real-life systems. While inverted conditionals occur less frequently in real-world code, this pattern poses interesting challenges for our approach. Together, these patterns provide new insights that can be used when implementing other change patterns in the future.

1) Consistent Renaming: One of the most frequently performed refactorings is the renaming of an identifier [10], with many IDEs providing automated tools to perform this refactoring. A renaming is *consistent* if it does not cause variable capturing, and if *every* reference to the renamed identifier is updated. A renaming that is consistent is behaviour-preserving, whereas a renaming that is not consistent may have an effect on the behaviour of the program. For example, forgetting to update a reference to a renamed function might lead to an error, as there now is a call to a function that no longer exists.

In order to detect consistent renamings, we use a locally nameless representation [15] of the piece of code before and after the change. To this end, every bound variable in an updated expression is replaced by its De Bruijn index, whereas free variables are kept as-is. If the two versions of the code are different in their locally nameless representation, they cannot be a consistent renaming of each other. If the two locally nameless representations are the same, we need to check that the free variables in the expression are unchanged in the expression's environment. That is, free variables still need to reference the same variable and function definitions. This can only be violated if there were multiple changes in the file other than the (potential) consistent renaming. While this can lead to some false negatives, especially when multiple functions or variables have been renamed across the program (as free variables within an expression that contains a consistent renaming may then also be updated, yielding unequal locally nameless representations), we avoid false positives.

2) Inverted Conditional: A conditional (e.g., an if expression) is inverted in a behaviour-preserving way when its condition is negated (for example, by adding `not` or by changing a relational operator) and its branches are swapped. Detecting this change can be done by textually comparing the old else branch and new then branch (as well as the other way around), and the old and new conditions.

3) Moved Function Definition: A function definition can be moved elsewhere in the program, thereby possibly changing its scope. This change is behaviour-preserving if (1) the function has not moved outside the scope of any of its callers, which will lead to errors, (2) the free variables used in the function body are all still in scope and reference the same definitions as before, and (3) no variable capturing has occurred. This variable capturing is possible if, e.g., there already existed a variable or function with the same name as the moved function definition within its new scope. If the function is moved outside of the scope of its callers, for example, violating criterion (1), the program will yield an error which might not have existed before (meaning there is a change in behaviour).

We can detect this type of change by comparing all function definitions that have been removed from the code against all of those that have been added to the code. If two function definitions are textually identical, this may indicate a moved function definition. Then, we need to check whether all their free variables still refer to the same function and variable definitions as before the move. If this is the case, we check if all callers of the removed function now call the inserted function. However, note that in the case of a recursive function, the recursive call will now call a different function (i.e., the inserted one).

B. Updating the Analysis Results

Once we detect that a change matches a particular behaviour-preserving change pattern, the analysis results can be updated accordingly. We now look into the constituents of these analysis results that require updating once a pattern is detected. These constituents stem from the analysis state space depicted in Table II, based on the state space defined by Nicolay et al. [3]. One of the most important parts is the *store* of the analysis, σ, which maps addresses in the heap (e.g., return addresses or variable addresses) to their abstract

TABLE II
SIMPLIFIED STATE SPACE OF THE INTER-COMPONENT ANALYSIS.

$s \in \sigma$	=	$(Addr + K) \rightarrow Val$
$dep \in Deps$	=	$AddrDep \rightarrow Cmp$
$mapping \in TrackMap$	=	$Exp \rightarrow \mathrm{Set}(Cmp)$
$v \in Visited$	::=	$cmp : v \mid \epsilon$
$AddrDep$	::=	$\mathbf{addrDep}(a)$
$cmp \in Cmp$	::=	$\mathbf{cmp}(\mathbf{clos}(\lambda,\ \rho),\ \kappa)$
$\rho \in Env$	=	$Var \rightarrow Addr$
$a \in Addr$	::=	$\mathbf{retAdd}(cmp) \mid \mathbf{varAdd}(\mathrm{var},\ \kappa) \mid \ldots$
$val \in Val$	::=	$\mathbf{clos}(\lambda,\ \rho) \mid type$
$var \in Var$	::=	$\mathbf{identifier}(string,\ pos)$
$exp \in Exp$	::=	$\mathrm{var} \mid \mathbf{expr}(e,\ pos) \mid \lambda$
$\lambda \in Lambda$	::=	$\mathbf{expr}(\mathbf{lam}(args,\ exp),\ pos)$
$pos \in Positions$	::=	*line:column*
$\kappa \in K$		a finite set of contexts
$type \in Type$		a finite set of types
$args \in Args$		a finite set of variables

value (i.e., the computed type of a value for our type analysis). Note that expressions from the analysed program can occur at various levels in the analysis results. For instance, return addresses in the store contain the component (i.e., function call) the analysis returned from. Components in turn consist of a lambda expression from the source code, its definition environment, and the calling context for which it was analysed. Environments in turn contain variables from the source code. Thus, the state space of the analysis consists of highly nested parts that also require updating.

1) General updating: First, we look into the updating that is the same across all change patterns. Every element in the state space eventually contains either variables and/or expressions. For example, an address dependency (*AddrDep*) could contain a return address containing a component. This component, in turn, is a closure with a lambda (an expression), an environment ρ (a map of variables to addresses, which in turn can contain expressions etc.) and a context κ (although we do not go into details of contexts here, context-sensitive analyses can be supported).

We use two sets to keep track of what updating is required. First, *ReExp*: (*Exp*, *Exp*) keeps track of *replaced expressions* in the program. The set consists of tuples where the first element is an expression before a change, and the second element is the corresponding expression after the change. Second, *ReVar*: (*Var*, *Var*) keeps track of *replaced variables*; it contains variable definitions before and after the change. Our approach constructs these sets upon the detection of behaviour-preserving change patterns, i.e., during the detection phase.

Using *ReExp*, an expression at any level in the analysis results can be updated using the following case-based function:

$$\mathrm{updateExp}(e) = \begin{cases} ne & \text{if } (e,\ ne) \in ReExp \\ e & \text{if } (e,\ \_) \notin ReExp \\ & \quad \wedge \neg\mathrm{hasSubExpressions}(e) \\ usub(e) & \text{otherwise} \end{cases}$$

where $usub(e)$ maps *updateExp* to each of e's subexpressions.

The *updateExp* function takes an expression and has three possible outcomes. If the given expression e exists as the first element of a tuple in *ReExp*, the expression has been changed and the second element of that tuple (i.e., the new expression) will be returned. If e does not have any subexpression, and e itself is *not* present as the first element in the *ReExp* set, the expression has not changed and can therefore be returned itself. Finally, if the expression is not present as the first element of a tuple in the *ReExp* set, but it does have subexpressions, each of the subexpressions should be checked for required updating. This happens in case a change is nested somewhere deep inside an expression.

Afterwards, variables in environments can be updated similarly, making use of the *ReVar* set as follows:

$$\mathrm{updateEnv}(\rho) = \begin{cases} \rho[nv \mapsto a] \backslash \{v\} & \text{if } (v,\ nv) \in ReVar \\ & \quad \wedge\ v \in \rho \\ \rho & \text{otherwise} \end{cases}$$

where $a = \mathbf{varAddr}(nv, \kappa')$.

If an environment contains a variable which exists as the first element in a tuple of *ReVar*, that variable should be replaced in the environment by the second element of the tuple in the *ReVar* set. Because environments map variables to *variable addresses*, this new variable should be mapped to a variable address containing the new variable, as well as an updated version of the context of the old variable address, κ (κ'), in the case of a context-sensitive analysis.

Depending on the change pattern, additional updates might be required. These are be described below.

2) Consistent Renaming: In the case of a consistent renaming, there exists a one-to-one mapping for every expression and variable in the analysis results from before the change to after. Therefore, no special updating is required in this case, and the changes can be performed as described above.

3) Inverted Conditional: Inverted conditionals require some additional updating next to the updating process described above. While many expressions do have a one-to-one mapping (for example, a mapping from the old then branch to the new else branch), new expressions may be introduced. For example, a `not` can be introduced around the condition. In this case, the `not` expression has never been analysed in this context before, meaning there are no analysis results for it yet (and therefore it cannot be updated). Similarly, a relational operator can be changed, e.g., > may be replaced by <=.

To remedy this, we only allow changes where a `not` expression is removed. For relational operators, we use the fact that the analysis framework we are extending defines some relational operators in terms of others. For example, > can be defined using `not` and <= as follows: `(define (> x y) (not (<= x y)))` Therefore, the "negated" conditional (> to <=) might already have an analysis result. It is important that this result already exists (e.g., in the case of a type analysis, this would mean that the abstract value returned by <= is a boolean). Otherwise, it will be missing from the analysis results, thereby making them unsound. If this result does *not* yet exist, it is important to perform the analysis of the component regardless of it being a behaviour-preserving change pattern. Therefore, we also restrict some of the negated relational operators. Thus, to guarantee soundness, we under-detect this pattern, reanalysing it in some cases despite being a behaviour-preserving change pattern. In other cases, we update the analysis results as described.

In the case a `not` expression is removed, we have to remove the component (and its analysis results related to it) as well, unless `not` is still used elsewhere in the program.

4) Moved function definition: Moved function definitions impact the analysis results in more than one way. While there exists a one-to-one mapping from every (sub-)expression of the moved function definition before and after the move, the surrounding expressions should also be updated:

- Due to the change in the function definition's scope, the function definition will now belong to a different (set of) component(s), as it is moved elsewhere;
- If the function is moved, it is no longer a subexpression of where it was previously defined, but a subexpression of where it is defined now. In the state space, *TrackMap* keeps track of which components contain a given expression;
- This change pattern can affect many environments within the analysis result. A function moved *down*, outside of the scope of other functions, therefore leaves those environments. A function moved *up*, causes it to be in the scope of more functions than before. In both cases, closures in the analysis results need to be updated to reflect their new environments.

In the framework we are extending, the environments do *not* contain all variables in scope, but only those that are used by the lambda expression the environments belong to. We will use this knowledge when updating the environments, as this allows us to reason in terms of the free variables.

Thus, in addition to applying the updating rules *updateExp* and *updateEnv* described before, we also apply the *updateMv* rule. *updateMv* is applied to *all* environments that exist within the analysis results, as all may be affected. *fv* is a function that takes a lambda expression and returns its free variables (both the names and definition sites are returned to ensure the correct variable is referenced). *nv* is the name and definition site of the moved function definition *after* the move, whereas *v* is the name and position of the moved function definition *before* the move. As every environment in the analysis result is located within a closure, there is always a lambda expression that corresponds to a given environment that is being updated by *updateMv*. *e* is the lambda expression that belongs to the environment that is currently being updated, and *ne* is the corresponding lambda expression in the updated program.

$$\text{updateMv}(\rho) = \begin{cases} \rho[nv \mapsto a]\backslash\{v\} & \text{if } nv \in \text{fv}(ne) \land v \in \text{fv}(e) \\ \rho[nv \mapsto a] & \text{if } nv \in \text{fv}(ne) \land v \notin \text{fv}(e) \\ \rho\backslash\{v\} & \text{if } nv \notin \text{fv}(ne) \land v \in \text{fv}(e) \\ \rho & \text{otherwise} \end{cases}$$

where $a = \textbf{varAddr}(nv, \kappa')$ as before.

In the first case, the moved function is called by another function both before and after the move. Here, it is important that the moved function is *not* defined within the body of the calling function. Therefore, when looking at the body of the calling function in isolation, the name of the moved function is a free variable ($nv \in \text{fv}(ne) \land v \in \text{fv}(e)$). The environment of the calling function must be updated: *nv* is inserted into the environment, mapping to its new variable address *a*, and the old mapping of *v* is removed. Note that in this case, the variable referencing the moved function remains a free variable within the body of the calling function ($nv \in \text{fv}(ne)$), i.e., the function did not move into the body of the calling function.

If there is a function in which the moved function definition was nested initially (so *v* was bound in *e* due to its definition site within *e*: $v \notin \text{fv}(e)$), but due to the move the function is no longer nested within the former, then the definition no longer exists within the body of the enclosing lambda and references to it within the body of this lambda become free variables ($nv \in \text{fv}(ne)$). Therefore, *nv* must be added to the environment, so that calling function can still use it.

Third, if there is a function of which the old version did reference the moved function using a free variable ($v \in \text{fv}(e)$), but it does *not* after the move ($nv \notin \text{fv}(ne)$), it means that the function definition was moved into the body of this function. In this case, the moved function definition can be removed from the environment of the now enclosing function, as the moved function is now defined within its body.

Finally, if no of the above cases apply ($nv \notin \text{fv}(ne) \land v \notin \text{fv}(e)$), that means the lambda associated with the environment that is being updated does not make use of the moved function definition, and nothing should be updated.

Note that, if we were to keep track of all variables in scope, rather than only those that are referenced, *updateMv* would be slightly different: in this case it would be necessary to add or remove the variable to which to the moved function is bound more often as it would appear in more environments.

Finally, to update *TrackMap* correctly, the *main* component is scheduled for reanalysis. This will trigger no additional reanalyses as it does not update the store (due to it being updated according to the rules described above), and will therefore still be faster than performing a full incremental reanalysis. We discuss this further in Section VI.

IV. Example

We illustrate the updating phase described in Section III-B on the example program from Section I[2], in which `x` is consistently renamed to `y`. For a consistent renaming, the most basic form of the updating is required: there is a one-to-one mapping of the expressions before and after the renaming.

The sets *ReExp* and *ReVar* were computed during the detection phase, based on the detection of the renaming. In this case, the only (sub)expressions that have been changed are the two references to `x`. For this program, *ReExp* is as follows: {(`x`@0:19, `y`@0:19), (`x`@1:32, `y`@1:32)} while *ReVar* is {(`x`@0:19, `y`@0:19)}. Recall that @*line:column* denotes the

[2]To improve readability, we omit part of the state space for simplicity. For instance, we use 'Return (`(λ (x) x)`@0:10, {})' rather than '**retAddr**(**cmp**(**clos**(`(λ (x) x)`, 0:10, {}), ϵ))'.

line number and column of the expression. This position is important, as two expressions that are textually the same might exist in the program, but that does not mean both should be updated (they could, e.g., be in a different scope which makes that they may not yield the same result).

ReExp and *ReVar* will then be used to update every part of the analysis result. First we go over the addresses in the store and loop over each one. Imagine we want to update the first element in the store, namely the return address of the function `g`, which contains the body expression of `g`, `(λ () x)`@1:22, and its definition environment {`x`@0:19}. As `(λ () x)`@1:22 is absent from *ReExp*, each of its subexpressions is considered. This causes the subexpression `x`@1:32 to be updated to `y`@1:32, yielding `(λ () y)`@1:22 for the new body of `g`.

To update the environment of `g`, i.e., {`x`@0:19}, each element is considered. As *ReVar* contains the tuple (`x`@0:19, `y`@0:19), the variable is replaced by `y`@0:19, yielding the updated environment {`y`@0:19}. This process is repeated for every element in the analysis result, so that all necessary expressions and environments are updated.

This differs from the incremental analysis by Van der Plas et al. [1], where the component corresponding to the call of `g` on line 3 and eventually also the component corresponding to the call of `f` on line 5 would be scheduled for reanalysis using abstract interpretation.

Note that the store is not the only place where updating might be necessary: it is also required to loop over the *Deps* map (which keeps track of dependencies), *TrackMap* (which keeps track of which expression belongs to which component), and *Visited* (the set of analysed components). Recall that this example only talks about a consistent renaming, which has no additional updating required other than the one-to-one mapping. If a moved function definition or an inverted conditional is detected, it will also require additional updating, as explained in Section III-B1.

V. Evaluation

We compare our approach to the incremental analysis proposed by Van der Plas et al. [1], henceforth referred to as the *baseline incremental analysis*, or simply *the baseline*. We answer the following research questions:

- **RQ1** *Upon a source code change, what is the impact of our approach on the running time of an incremental analysis?*
- **RQ2** *What are the differences in impact between the three different behaviour-preserving change patterns?*
- **RQ3** *What is the overhead of detecting changes and updating the analysis results?*

Additionally, to confirm the soundness of our results, we conducted soundness tests [16] on a total of 90 programs to ensure that our implementation is sound, i.e., by ensuring that there are no false negatives in the analysis results (false positives are allowed). This means that the analysis correctly over-approximates the program behaviour.

A. Experimental Setup

We extended the incremental context-insensitive function-modular type analysis of the baseline [1] with pattern detection rules for verifying whether the source code under analysis has been changed according to one of three supported change patterns (cf. Section III-A). We also extended the baseline with the machinery needed to update the existing analysis results according to a detected change pattern (cf. Section III-B). Changes in the program that do not follow a change pattern are reanalysed as before using the baseline.

To evaluate our approach, we manually created three mutations of 10 different Scheme programs, each mutation containing a different behaviour-preserving change pattern. The 10 original Scheme programs are summarised in Table III. Thus, in total, we obtain 30 programs containing different change patterns. We also include *additional* changes that are not behaviour-preserving in 12 of the 30 resulting programs. These changes can be found in the three mutations of the `freeze`, `leval`, `machine-sim`, and `multiple-dw` programs. Some of the non-behaviour-preserving changes originate from the benchmark suite of Van der Plas et al. [1]. As mentioned before, these are not handled by our approach but will be reanalysed using the baseline incremental analysis.

To exemplify the modified programs containing the manually added refactorings, we show an excerpt of the three mutations of the `nbody` program in Listing 3, Listing 4, and Listing 5. Listing 3 shows the `rand` function with an inverted conditional. In this case, we changed the relational operator of the if expression and swapped its branches. Listing 4 shows the same `rand` procedure, but this time consistent renamings have been applied to the let-bindings (`h` becomes `high` and `l` becomes `low`) and all references are also updated. Finally, Listing 5 shows the `nbody` program but with a moved function definition. As `random` is the only function that calls `rand`, `rand` does not have to be a global function and can be moved down: `rand` is deleted on the top level and inserted locally in the body of `random`.

For the experiments where we measured running times, we ran both the baseline and the proposed approach for 10 warm-up runs, followed by measured 25 runs to calculate the average running time on each benchmark program. All benchmarks were run on a machine with 8GB RAM and an Intel Core i5-7200U processor with 2 physical and 4 logical cores. We measured both the full running time and the running times of each individual phase (detecting the changes, updating the analysis results, and reanalysing where necessary). Additionally, we also look into the number of intra-component analyses performed during the reanalysis phase (if any) during a separate run, as a different way to quantify the amount of work performed by the resulting incremental analysis phase.

VI. Results

A. RQ 1: Impact on Total Running Time

In RQ1, we look at the difference in total running time between our new approach and the baseline. Table IV shows

TABLE III
USED BENCHMARK PROGRAMS, THEIR LINES OF CODE AND DESCRIPTION. MUTATIONS OF THE BOTTOM 4 PROGRAMS CONTAIN OTHER CHANGES ALONGSIDE A REFACTORING.

name	LOC	description
`browse`	161	Creates and browses through a data base
`matrix`	617	Computes maximal matrices
`mceval`	239	Meta-circular evaluator for Scheme
`nbody`	1 205	Performs calculations related to n-body problem
`nboyer`	625	Logic program evaluator
`peval`	497	Partial evaluator for Scheme
`freeze`	325	Adds "freeze" to meta-circular evaluator
`leval`	379	Lazy evaluator for Scheme
`machine-sim`	964	Compiles to machine code and simulates
`multiple-dw`	404	non-deterministic evaluator

that on our 30 programs, the running time of the incremental analysis decreased for 25 programs but increased for 5 others.

For 4 of the 5 programs with an increase in time, the increase is less than 35 milliseconds, with the exception of `machine-sim` with a renaming. This could mean that for small programs, the overhead of detecting and updating might be too high compared to simply performing an incremental analysis. At the same time, we see decreases in running time as well, on both big and small programs. While some benchmarks only have a decrease of a couple of milliseconds, the program `multiple-dw` sees a decrease of multiple seconds for all three of the introduced change patterns. The program `peval` with a moved function definition has the highest decrease, going from almost 12 seconds to under a second. This is due to the fact that the baseline incremental approach scheduled several expensive component reanalyses for this benchmark.

We can also look at the number of intra-component analyses performed by both the baseline algorithm and the new method. These results can be found in Table V. In terms of intra-component analyses, we see a decrease for all of the 30 benchmarks, meaning that the fixed-point of the reanalysis phase is obtained in fewer analysis steps. If the only change present in the program is either a renaming or an inverted conditional, we can bring the number of intra-component analyses to zero. In some cases however, the baseline analysis also performs few reanalyses. The number of components that will be reanalysed depends on how many components are dependent on the one that has changed. For example, if the parameter of a small function with no side effects that is called only once is renamed, it will have little impact on the analysis results of all the other components, and few will be triggered for reanalysis. If, on the other hand, something that has a larger impact on the program is renamed, more components will be triggered for analysis. For a moved function definition, we often still have one or two reanalyses to ensure the soundness of the results. As discussed in section III-B1, this is to ensure the correctness of *TrackMap*. However, this is still a significant decrease in the number of intra-component analyses for each of the programs, even ones containing other changes.

Our results thus show that an increase in running time is not always due to more intra-component reanalyses. The increase in running time can also be caused by, e.g., the overhead

Listing 3. `nbody` benchmark with an inverted conditional

```
...
(define (rand)
  (let* ((hi (quotient (car *seed*) 127773))
         (lo (modulo (car *seed*) 127773))
         (test (- (* 16807 lo) (* 2836 hi))))
    (<update>
       (if (> test 0)
         (set-car! *seed* test)
         (set-car! *seed* (+ test 2147483647)))
       (if (<= test 0)
         (set-car! *seed* (+ test 2147483647))
         (set-car! *seed* test)))
    (car *seed*)))

(define random (lambda (n)  (modulo (abs (rand)) n)))
...
```

Listing 4. `nbody` benchmark with a consistent renaming

```
...
(define (rand)
  (let* (((<update> h high)
             (quotient (car *seed*) 127773))
         ((<update> l low)
             (modulo (car *seed*) 127773))
         (test (- (* 16807 (<update> l low))
                  (* 2836 (<update> h high)))))
    (if (> test 0)
      (set-car! *seed* test)
      (set-car! *seed* (+ test 2147483647)))
      (car *seed*)))

(define random (lambda (n) (modulo (abs (rand)) n)))
...
```

Listing 5. `nbody` benchmark with a moved function definition

```
...
(<delete>
  (define (rand)
    (let* ((hi (quotient (car *seed*) 127773))
           (lo (modulo (car *seed*) 127773))
           (test (- (* 16807 lo) (* 2836 hi))))
    (if (> test 0)
      (set-car! *seed* test)
      (set-car! *seed* (+ test 2147483647)))
      (car *seed*))))

(define random
  (let (<insert> (rand (lambda ()
    (let* ((hi (quotient (car *seed*) 127773))
           (lo (modulo (car *seed*) 127773))
           (test (- (* 16807 lo) (* 2836 hi))))
      (if (> test 0)
        (set-car! *seed* test)
        (set-car! *seed* (+ test 2147483647)))
        (car *seed*))))))
    (lambda (n)
      (modulo (abs (rand)) n)))
...
```

created by the detection of the change patterns or by the updating of the analysis results; we discuss this in more detail in RQ3. Some components can also take longer to analyse than others, which means that even if there are fewer intra-component analyses performed, they may take a longer time.

B. RQ 2: Comparison of the Change Patterns

We also look at each of the change patterns separately to investigate whether our approach is more effective for specific change patterns. Table IV shows that for 2 of the benchmarks

TABLE IV
AVERAGE RUNNING TIMES FOR EACH OF THE BENCHMARK PROGRAMS EXPRESSED IN MILLISECONDS. RUNNING TIME CALCULATED ON 25 RUNS, AFTER 10 WARM-UP ROUNDS. Δ SHOWS THE DIFFERENCE IN TOTAL TIME.

	Inverted conditional			**Consistent renaming**			**Moved function definition**		
	baseline	new	Δ	baseline	new	Δ	baseline	new	Δ
browse	186	18	-90,32%	175	9	-94,86%	203	44	-78,33%
matrix	33	27	-18,18%	33	29	-12,12%	47	48	2,13%
mceval	22	19	-13,64%	22	21	-4,55%	26	29	11,54%
nbody	64	60	-6,25%	65	57	-12,31%	65	96	47,69%
nboyer	923	149	-83,86%	945	161	-82,96%	813	166	-79,58%
peval	150	70	-53,33%	170	39	-77,06%	11751	99	-99,16%
freeze	6547	6561	0,21%	4743	4467	-5,82%	5110	4681	-8,40%
leval	2368	2279	-3,76%	3530	3404	-3,57%	2310	2213	-4,20%
machine-sim	12140	12927	6,48%	9974	9367	-6,09%	15512	12833	-17,27%
multiple-dw	15119	11329	-25,07%	18285	12898	-29,46%	17338	11021	-36,43%

TABLE V
NUMBER OF INTRA-COMPONENT ANALYSES PERFORMED BY THE BASELINE AND THE NEW APPROACH, I.E., THE NUMBER OF COMPONENTS THAT ARE EVENTUALLY (RE)ANALYSED IN BOTH APPROACHES. Δ REFERS TO THE DIFFERENCE BETWEEN THE BASELINE AND THE NEW APPROACH.

	Inverted conditional			**Consistent renaming**			**Moved function definition**		
	baseline	new	Δ	baseline	new	Δ	baseline	new	Δ
browse	118	0	-118	115	0	-115	143	1	-142
matrix	4	0	-4	4	0	-4	20	1	-19
mceval	6	0	-6	6	0	-6	19	1	-18
nbody	3	0	-3	4	0	-4	7	1	-6
nboyer	37	0	-37	39	0	-39	44	2	-42
peval	4	0	-4	5	0	-5	181	1	-180
freeze	1564	1553	-11	1564	1549	-15	1620	1673	-53
leval	797	795	-2	782	779	-3	805	791	-14
machine-sim	1035	1027	-8	932	924	-8	949	933	-16
multiple-dw	1111	1003	-108	1121	1119	-2	1364	988	-366

with inverted conditionals and 3 of the benchmarks with a moved function definition, the new approach is slower than the baseline incremental analysis. The new approach proposed in this paper is faster than the baseline for all the benchmark programs with a consistent renaming. At the same time, we see the biggest improvement in running time for `peval` with a moved function definition. This could be due to the fact that inverted conditionals and consistent renamings of variables can be local to a specific component, meaning very few components will be scheduled for reanalysis in the baseline incremental analysis. In these cases, detecting change patterns and updating existing analysis results might create too much overhead for smaller programs. Moved function definitions can, in contrast, affect multiple components due to changes in environments, leading the baseline to schedule several component reanalyses.

C. RQ3: Overhead Created by Detecting Changes

In order to investigate the overhead created by the updating of the analysis results, we look at each individual phase of our approach, namely detecting changes and refactorings, updating the analysis results, and reanalysing any other changes that may exist. These results can be found in Table VI. As inverted conditionals and renamed identifiers lead to no reanalyses being performed if they are the only changes present, programs that contain these refactorings spend 0ms in the reanalysis phase. For all of our benchmark programs, both detecting the refactorings and updating the analysis results take only a few milliseconds.

For inverted conditionals, we see that `freeze` and `machine-simulator` do not spend a lot of time in the detecting and updating phase. However in *RQ1*, we saw that both of these benchmarks are slower than the baseline algorithm, despite having fewer intra-component analyses (as seen in *RQ2*). One possible reason for this is that, despite the fewer intra-component analyses being performed, the ones that are reanalysed are more difficult. The order in which components are added to the worklist can also influence the running time of the analysis [17], and while both the new approach and the baseline use the same LIFO worklist algorithm, the baseline approach also has to reanalyse the inverted conditional, meaning it will add an extra component to the worklist, which can lead to a different order.

For programs such as `freeze` that do have other changes present, we see that the time spent on the detection of patterns and on the updating of results takes only a few milliseconds, whereas the reanalysis of all the other changes present is more expensive. Moved function definitions always lead to some reanalysis. However, in these cases, the reanalysis that is performed also only takes a few milliseconds, whereas reanalysing other changes takes a longer time. Therefore, in these programs, we see that the reanalysis of the changes takes longer than the detecting and updating of the analysis results.

Additionally, we see little difference in the detection and updating phases across refactorings. This is because all possible refactorings are tested for when a change is found during the detection phase. The updating phase also does not differ a lot between the different behaviour-preserving change patterns. This is due to the fact that all analysis results need to be traversed during the updating phase regardless of which pattern is detected, meaning that each pattern will cause looping over the store σ, the *Deps* map, the *Visited* set and *TrackMap*, to inspect and potentially update each item.

We also do not see a connection between lines of code and time to update the analysis results for our benchmark programs. `nbody` is the largest program in terms of lines of code, however, `nboyer`, with almost half the lines of code, spends the most time in the updating phase across refactorings. The time needed to update the analysis results does therefore not correspond to lines of code, but to the number of entries in the store, and to the number of dependencies in the program. When there are more entries in the store or more dependencies, more analysis results require checking and updating.

VII. Limitations and Future Work

We leverage domain-specific knowledge about the impact of change patterns on analysis results to speed up an incremental modular analysis that did not employ such knowledge. The expertise required to add support for additional change patterns might hinder the applicability of our approach at large. However, designers of static analyses are likely to already posses this knowledge. Our approach can be ported to other dynamically-typed languages such as Python or JavaScript, but this may require insights about the analysed language.

We limited ourselves to context-insensitive type analyses in this work, although the approach supports context-sensitive analyses too. For the consistent renaming and inverted conditional change patterns, we have already implemented the required context updates for argument sensitivity and call-site sensitivity. A preliminary evaluation revealed that this increases the time required for the updating phase. On the other hand, the baseline incremental analysis is also more expensive for context-sensitive analyses. Therefore, a more detailed evaluation is required.

We only evaluated our work on programs that contain at least one of the three behaviour-preserving change patterns. Programs that do not contain any of these changes will incur the overhead of change pattern detection, which cannot be regained by saving on component reanalyses. However, for most of our benchmarks, the time spent in the detection phase is much shorter than the time spent reanalysing the components, keeping the overhead minimal.

Furthermore, we use annotations to find changes in the program, rather than compare commits.

Finally, our evaluation is limited to the three behaviour-preserving change patterns presented in this work. However, it can be extended as many other behaviour-changing patterns or refactorings exist [7], as well as other changes that might have a predictable effect on the analysis results. An example of such a change is adding a print statement at the very end of a function definition. When performing a type analysis, we know that this addition has the predictable effect of changing the return type of that function to `void`. While there might be some additional updates required for these other patterns (as with the moved function definitions and inverted conditionals), the approach stays the same: every expression or variable that has been changed according to a pattern should be updated in the analysis results accordingly.

VIII. Related Work

Our work builds on top of the work on incremental static analysis of Van der Plas et al. [1]. Our work does not change the incremental analysis itself, but adds a step at the beginning of an incremental update, which detects specific change patterns and updates the analysis results accordingly. We look into the related work of both refactoring detection as well as incremental static analysis.

A. Refactoring Detection

Over the years, many studies [18]–[37] have been conducted on refactorings [7] and their detection. However, some approaches are prone to more false positives [38], e.g., due to reliance on similarity thresholds, which we avoid as mentioned in Section III-A.

Over the years, many techniques relying on detecting similarity between two program versions have been proposed. One of the earliest refactoring detection strategies was created by Demeyer et al. [18], who use a set of change metrics. Weissgerber and Diehl [25] use a set of rules in combination with clone detection. For example, for a method to be renamed, it must exist in the same class and have the same return type. Ref-finder [20], [39] also uses a set of rules in combination with a similarity threshold that is based on the longest common subsequence. However, this technique is also not ideal for renamed function definitions of recursive functions, for example, as references within the body of the function will also have to be updated, hampering similarity detection. The RefactoringCrawler [19] tool first renders a lightweight AST of the program and then uses Shingles encoding [40]. It uses a user-provided similarity threshold. Refdiff [26], [27] is a tool that works on multiple programming languages, and uses relations between entities, together with a similarity threshold. To find similarities, they use *Term Frequency–Inverse Document Frequency* (TF-IDF). RefDetect [22] also works on multiple languages and uses a string alignment algorithm, and also allows for detecting, for example, a function definition that has moved and that has another change in its body as well. All these techniques that rely on similarity are unfortunately vulnerable to false positives. If the similarity threshold is too low, many false negatives will occur. At the same time, if the threshold is too high, there will be more false positives. Therefore, such similarity thresholds are not sufficient in our

TABLE VI
TIME IN MILLISECONDS SPENT BY THE ANALYSIS IN EACH OF THE PHASES.

	Inverted conditional			**Consistent renaming**			**Moved function definition**		
	Detection	Updating	Reanalysis	Detection	Updating	Reanalysis	Detection	Updating	Reanalysis
browse	3	13	0	1	7	0	5	27	8
matrix	3	22	0	3	24	0	4	32	9
mceval	3	14	0	4	15	0	3	19	3
nbody	13	45	0	12	42	0	16	69	7
nboyer	18	129	0	12	147	0	19	132	11
peval	8	60	0	4	33	0	4	35	58
freeze	5	19	6535	3	13	4448	3	17	4659
leval	4	14	2259	6	20	3375	4	16	2190
machine-sim	12	46	12866	11	40	9313	13	50	12766
multiple-dw	5	20	11302	4	17	12873	5	25	10988

setting as we avoid false positives to ensure soundness of the analysis results.

RefactoringMiner [23], [41] uses a bottom-up approach using the AST of the program. While their approach leaves less room for errors than the ones using similarity threshold, in our approach we made use of the fact that the code has change annotations. Hence, we know where changes are and we can focus on pattern matching.

Stroggylos and Spinellis [31] as well as Ratzinger et al. [32] analyse commit messages to detect refactorings in code bases. However, this technique is not ideal in our setting, as we do not know which refactorings have been performed in this case, users do not always say they have refactored, and refactorings might have been performed incorrectly.

Additionally, there are also techniques that detect refactorings in the IDE as they are occurring [33], [34], [42], [43]. However, the static incremental analysis we build upon is not yet integrated into an IDE.

B. Incremental Static Analysis

Our work does not propose a new technique to perform incremental static analysis. Rather, we build on top of an existing incremental static analysis and make it more efficient when refactorings are present. There have been multiple studies on performing incremental static program analysis. Not all are applicable to dynamic, higher-order languages however, and many have different approaches to how the incremental analysis is performed. Nichols et al. [44] propose an incremental analysis for JavaScript. Their approach requires a mapping from old program points to new program points, which is similar to what we use when performing the updating. However, every program point has to be reanalysed at least once, whereas we try to avoid reanalysis as much as possible. In addition, the incremental analysis of Van der Plas et al. [1], only reanalyses the affected program parts.

IncA [45]–[48] uses Datalog-based graph patterns. Domain-specific knowledge about these graph patterns might be used to implement our approach in their setting.

Andromeda [49] uses a support graph to perform specifically incremental demand-driven taint analysis. Another technique using support graphs is proposed by Saha and Ramakrishnan [50]. Programs need to be specified as Horn clauses, a requirement that is not needed by the approach we built upon. Also, our approach can also be applied to other analyses than taint analysis. Garcia-Contreras et al. [51], [52] also require Horn clauses for a context-sensitive incremental modular analysis. However, the approach uses programmer-defined lexical modules, and thus does not allow for thread-modular analyses, for example.

Unlike the technique of Van der Plas et al. [1], many incremental static analysis techniques require a static call graph [53]–[59]. Liu et al. [60] do not require a statically known call graph, and preserve precision in their incremental analysis. However, their approach is limited to flow-insensitive analyses.

IX. CONCLUSION

In this work, we propose a method to decrease the running time of an incremental static analysis by updating previous analysis results according to change patterns detected in the code change. As change patterns have a predictable effect on the analysis results, the results can be updated directly, thereby avoiding a more expensive reanalysis of the changes. We find that for 25 of our 30 benchmarks, there is indeed a decrease in running time between 3% and 99% (*RQ1*). For consistent renamings, all of the benchmarks were faster, while for inverted conditionals 2 of the 10 benchmarks were slower. For moved function definitions, 3 of the 10 benchmarks were slower, though we also see our best result in a program with a moved function definition (*RQ2*). From *RQ3*, we can also conclude that for our benchmark programs, there is little overhead created by our approach to detect refactorings and to update the analysis results. Especially in bigger programs with many dependencies for which reanalysing components can be expensive, and for change patterns that may otherwise schedule many components for reanalysis, our approach can speed up existing incremental analyses.

ACKNOWLEDGEMENTS

This work was partially supported by *Research Foundation – Flanders (FWO)* (grant number 11F4822N), by *CS ICON project APAX*, and by *“Cybersecurity Initiative Flanders”*.

REFERENCES

[1] J. Van der Plas, Q. Stiévenart, N. Van Es, and C. De Roover, "Incremental flow analysis through computational dependency reification," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2020, pp. 25–36.

[2] J. Van der Plas, Q. Stiévenart, and C. De Roover, "Result invalidation for incremental modular analyses," in *Proceedings of the 24th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2023, Boston, MA, USA, January 15-17, 2023*, 2023.

[3] J. Nicolay, Q. Stiévenart, W. De Meuter, and C. De Roover, "Effect-driven flow analysis," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2019, pp. 247–274.

[4] P. Cousot and R. Cousot, "Modular static program analysis," in *Compiler Construction - 11th International Conference, CC 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Proceedings*. Springer Verlag, 2002, pp. 159–179.

[5] Q. Stiévenart, J. Nicolay, W. De Meuter, and C. De Roover, "A general method for rendering static analyses for diverse concurrency models modular," *Journal of Systems and Software*, vol. 147, pp. 17–45, 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2018.10.001

[6] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.

[7] M. Fowler, K. Beck, J. Brant, and W. Opdyke, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[8] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A case study on the impact of refactoring on quality and productivity in an agile team," in *IFIP Central and East European Conference on Software Engineering Techniques*. Springer, 2007, pp. 252–266.

[9] Z. Xing and E. Stroulia, "Refactoring practice: How it is and how it should be supported - an eclipse case study," in *2006 22nd IEEE International Conference on Software Maintenance*, 2006, pp. 458–468.

[10] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[11] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2012, pp. 104–113.

[12] M. Di Penta, G. Bavota, and F. Zampetti, "On the relationship between refactoring actions and bugs: a differentiated replication," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 556–567.

[13] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, 2016, pp. 858–870.

[14] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 185–194.

[15] A. Charguéraud, "The locally nameless representation," *Journal of automated reasoning*, vol. 49, no. 3, pp. 363–408, 2012.

[16] E. S. Andreasen, A. Møller, and B. B. Nielsen, "Systematic approaches for increasing soundness and precision of static analyzers," in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, 2017, pp. 31–36.

[17] N. Van Es, Q. Stiévenart, J. Van der Plas, and C. De Roover, "A parallel worklist algorithm for modular analyses," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2020, pp. 1–12.

[18] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '00. Association for Computing Machinery, 2000, pp. 166–177. [Online]. Available: https://doi.org/10.1145/353171.353183

[19] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *Proceedings of the 20th European Conference on Object-Oriented Programming*, ser. ECOOP'06. Springer-Verlag, 2006, pp. 404–428.

[20] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.

[21] R. Stevens, C. De Roover, C. Noguera, and V. Jonckers, "A history querying tool and its application to detect multi-version refactorings," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR13)*, 2013.

[22] I. H. Moghadam, M. Ó. Cinnéide, F. Zarepour, and M. A. Jahanmir, "Refdetect: A multi-language refactoring detection tool based on string alignment," *IEEE Access*, vol. 9, pp. 86 698–86 727, 2021.

[23] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 03, pp. 930–950, mar 2022.

[24] C. De Roover and K. Inoue, "The ekeko/x program transformation tool," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 53–58.

[25] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 2006, pp. 231–240.

[26] D. Silva and M. T. Valente, "Refdiff: detecting refactorings in version histories," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 269–279.

[27] D. Silva, J. Silva, G. J. D. S. Santos, R. Terra, and M. T. O. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, 2020.

[28] G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, and T. Massoni, "Analyzing refactorings on software repositories," in *2011 25th Brazilian Symposium on Software Engineering*, 2011, pp. 164–173.

[29] Z. Xing and E. Stroulia, "Refactoring detection based on umldiff change-facts queries," in *2006 13th Working Conference on Reverse Engineering*, 2006, pp. 263–274.

[30] C. De Roover and R. Stevens, "Building development tools interactively using the ekeko meta-programming library," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 429–433.

[31] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?" in *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*, 2007, pp. 10–10.

[32] J. Ratzinger, T. Sigmund, and H. C. Gall, "On the relation of refactorings and software defect prediction," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. Association for Computing Machinery, 2008, pp. 35–38. [Online]. Available: https://doi.org/10.1145/1370750.1370759

[33] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 211–221.

[34] S. R. Foster, W. G. Griswold, and S. Lerner, "Witchdoctor: Ide support for real-time auto-completion of refactorings," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 222–232.

[35] R. Stevens and C. De Roover, "Extracting executable transformations from distilled code changes," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 171–181.

[36] R. Stevens, T. Molderez, and C. De Roover, "Querying distilled code changes to extract executable transformations," *Empirical Software Engineering*, vol. 24, pp. 491–535, 2019.

[37] C. Noguera, A. Kellens, C. De Roover, and V. Jonckers, "Refactoring in the presence of annotations," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 337–346.

[38] P. Hegedűs, I. Kádár, R. Ferenc, and T. Gyimóthy, "Empirical evaluation of software maintainability based on a manually validated refactoring dataset," *Information and Software Technology*, vol. 95, pp. 313–327, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584916303561

[39] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: A refactoring reconstruction tool based on logic query templates," 01 2010, pp. 371–372.

[40] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE, 1997, pp. 21–29.

[41] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. Association for Computing Machinery, 2018, pp. 483–494. [Online]. Available: https://doi.org/10.1145/3180155.3180206

[42] X. Ge and E. Murphy-Hill, "Manual refactoring changes with automated refactoring validation," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. Association for Computing Machinery, 2014, pp. 1095–1105. [Online]. Available: https://doi.org/10.1145/2568225.2568280

[43] X. Ge, S. Sarkar, J. Witschey, and E. Murphy-Hill, "Refactoring-aware code review," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017, pp. 71–79.

[44] L. Nichols, M. Emre, and B. Hardekopf, "Fixpoint reuse for incremental javascript analysis," in *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2019. Association for Computing Machinery, 2019, pp. 2–7. [Online]. Available: https://doi.org/10.1145/3315568.3329964

[45] T. Szabó, S. Erdweg, and M. Voelter, "Inca: A dsl for the definition of incremental program analyses," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. Association for Computing Machinery, 2016, pp. 320–331. [Online]. Available: https://doi.org/10.1145/2970276.2970298

[46] T. Szabó, G. Bergmann, S. Erdweg, and M. Voelter, "Incrementalizing lattice-based program analyses in datalog," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, oct 2018. [Online]. Available: https://doi.org/10.1145/3276509

[47] T. Szabó, S. Erdweg, and G. Bergmann, "Incremental whole-program analysis in datalog with lattices," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1–15. [Online]. Available: https://doi.org/10.1145/3453483.3454026

[48] T. Szabó, "Incrementalizing static analyses in datalog," Ph.D. dissertation, Mainz, 2021.

[49] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and Scalable Security Analysis of Web Applications," in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering, FASE 2013, Rome, Italy, March 16-24, 2013*, V. Cortellessa and D. Varró, Eds. Berlin, Heidelberg, Germany: Springer, 2013, pp. 210–225.

[50] D. Saha and C. R. Ramakrishnan, "Incremental and Demand-driven Points-To Analysis Using Logic Programming," in *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal*, P. Barahona and A. P. Felty, Eds. ACM, 2005, pp. 117–128.

[51] I. Garcia-Contreras, J. F. M. Caballero, and M. V. Hermenegildo, "An Approach to Incremental and Modular Context-Sensitive Analysis," 2018. [Online]. Available: http://oa.upm.es/53067/

[52] I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo, "Incremental and Modular Context-sensitive Analysis," *Theory and Practice of Logic Programming*, vol. 21, no. 2, p. 196–243, 2021.

[53] S. Arzt and E. Bodden, "Reviser: Efficiently Updating IDE-/IFDS-Based Data-Flow Analyses in Response to Incremental Program Changes," in *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, Hyderabad, India, May 31-June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. New York, NY, USA: ACM Press, 2014, pp. 288–298.

[54] M. G. Burke, "An interval-based approach to exhaustive and incremental interprocedural data-flow analysis," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 341–395, 1990. [Online]. Available: https://doi.org/10.1145/78969.78963

[55] M. D. Carroll and B. G. Ryder, "Incremental data flow analysis via dominator and attribute updates," in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, J. Ferrante and P. Mager, Eds. ACM Press, 1988, pp. 274–284. [Online]. Available: https://doi.org/10.1145/73560.73584

[56] L. L. Pollock and M. L. Soffa, "An incremental version of iterative data flow analysis," *IEEE Trans. Software Eng.*, vol. 15, no. 12, pp. 1537–1549, 1989. [Online]. Available: https://doi.org/10.1109/32.58766

[57] J. Yur, B. G. Ryder, and W. Landi, "An Incremental Flow- and Context-Sensitive Pointer Aliasing Analysis," in *Proceedings of the 1999 International Conference on Software Engineering, ICSE 1999, Los Angeles, CA, USA, May 16-22, 1999*, B. W. Boehm, D. Garlan, and J. Kramer, Eds. ACM, 1999, pp. 442–451.

[58] F. K. Zadeck, "Incremental Data Flow Analysis in a Structured Program Editor," in *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, Montreal, Canada, June 17-22, 1984*, M. S. V. Deusen and S. L. Graham, Eds. ACM, 1984, pp. 132–143.

[59] S. Zhan and J. Huang, "ECHO: Instantaneous In Situ Race Detection in the IDE," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 775–786. [Online]. Available: https://doi.org/10.1145/2950290.2950332

[60] B. Liu, J. Huang, and L. Rauchwerger, "Rethinking incremental and parallel pointer analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 41, no. 1, pp. 1–31, 2019.