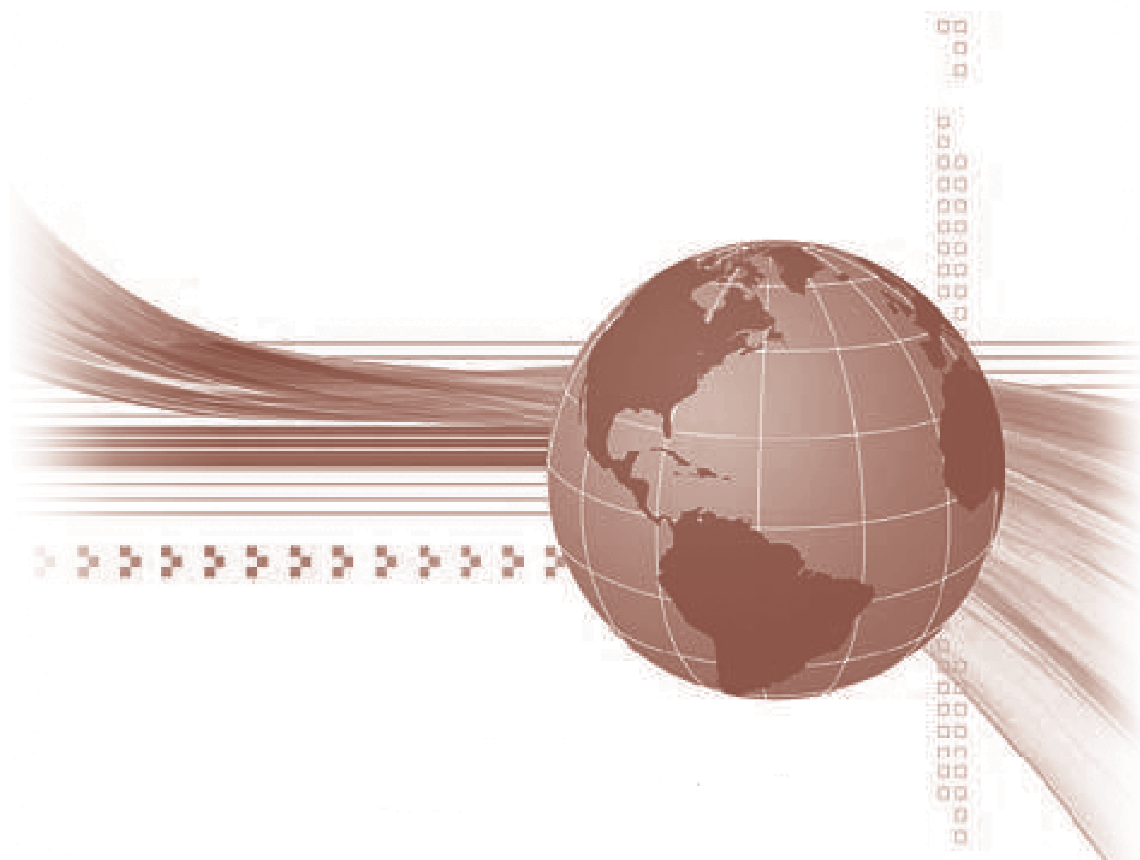
STUDIA UNIVERSITATIS
BABEŞ-BOLYAI



INFORMATICA

2/2018

STUDIA

UNIVERSITATIS BABEŞ-BOLYAI
INFORMATICA

No. 2/2018

July - December

EDITORIAL BOARD

EDITOR-IN-CHIEF:

Prof. Horia F. Pop, Babeş-Bolyai University, Cluj-Napoca, Romania

EXECUTIVE EDITOR:

Prof. Gabriela Czibula, Babeș-Bolyai University, Cluj-Napoca, Romania

EDITORIAL BOARD:

Prof. Osei Adjei, University of Luton, Great Britain
Prof. Anca Andreica, Babeş-Bolyai University, Cluj-Napoca, Romania
Prof. Florian M. Boian, Babeş-Bolyai University, Cluj-Napoca, Romania
Prof. Sergiu Cataranciuc, State University of Moldova, Chișinău, Moldova
Prof. Wei Ngan Chin, School of Computing, National University of Singapore
Prof. Laura Dioșan, Babeş-Bolyai University, Cluj-Napoca, Romania
Prof. Farshad Fotouhi, Wayne State University, Detroit, United States
Prof. Zoltán Horváth, Eötvös Loránd University, Budapest, Hungary
Assoc. Prof. Simona Motogna, Babeş-Bolyai University, Cluj-Napoca, Romania
Prof. Roberto Paiano, University of Lecce, Italy
Prof. Bazil Pârv, Babeş-Bolyai University, Cluj-Napoca, Romania
Prof. Abdel-Badeeh M. Salem, Ain Shams University, Cairo, Egypt
Assoc. Prof. Vasile Marian Scuturici, INSA de Lyon, France

YEAR Volume 63 (LXIII) 2018
MONTH DECEMBER
ISSUE 2

S T U D I A

UNIVERSITATIS BABEȘ-BOLYAI

INFORMATICA

2

EDITORIAL OFFICE: M. Kogălniceanu 1 • 400084 Cluj-Napoca • Tel: 0264.405300

SUMAR – CONTENTS – SOMMAIRE

PAPERS FROM THE 12TH CONFERENCE MACS 2018

A. Poór, T. Kozsik, M. Tóth, I. Bozó, *Compiler Front End Fusion: Undo Desugaring in Language Processing Tools* 5

M. Komáromi, I. Bozó, M. Tóth, *An Efficient Graph Visualisation Framework for RefactorErl* 21

A. Reale, P. Kiss, C. Ferrari, B. Kovács, L. Szilágyi, M. Tóth, *Application Functions Placement Optimization in a Mobile Distributed Cloud Environment* 37

Z. Parragi, Z. Porkoláb, *Instrumentation of C++ Programs Using Automatic Source Code Transformations* 53

G. Morse, D. Lukács, M. Tóth, *Incremental Decompilation of Loop-Free Binary Code: Erlang* 66

R. Kovács, G. Horváth, *An Initial Prototype of Tiered Constraint Solving in the Clang Static Analyzer* 88

REGULAR PAPERS

L.M. Crivei, *Incremental Relational Association Rule Mining of Educational Data Sets* 102

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 2, 2018
DOI: 10.24193/subbi.2018.2.01

COMPILER FRONT END FUSION: UNDO DESUGARING IN LANGUAGE PROCESSING TOOLS

ARTÚR POÓR, TAMÁS KOZSIK, MELINDA TÓTH, AND ISTVÁN BOZÓ

Abstract. Compiler front ends often perform *desugaring* on the source code while constructing the abstract syntax tree (AST). A programming language processing tool (such as a refactoring tool) working with the desugared AST perceives the code at this abstract level, and loses information on the rich syntax used in the actual source code. This paper discusses the concept of *front end fusion*, a technique which may help language processing tools to retain the syntactic sugar information on the source code in the presence of desugaring compiler front ends. We propose a hybrid front end created from two separate front ends: one provided by the compiler, which offers type information, and another one, which provides the details of the concrete syntax used in the source code. Specifically, we show how to construct a hybrid front end in a language processing tool for the Scala programming language.

1. Introduction

Programming language processing tools provide invaluable help during software development and maintenance. They can statically analyse source code for debugging, code upgrade or grokking purposes, and they can perform source code transformations and refactoring as well. These tools typically need to be able to parse and pretty print source code, and may also require semantic information, e.g. the type of expressions and the result of name resolution.

There are two major approaches to implement language processing tools. Firstly, the tool may have a custom lexer, parser, type checker, static semantic analyser, and pretty printer built in, and tailored for, the tool (*standalone*

Received by the editors: April 17, 2018.
1991 *Mathematics Subject Classification.* 68N15, 68N20.
1998 *CR Categories and Descriptors.* D.3.4 [**Programming languages**]: Processors – *Compilers*.
Key words and phrases. parser, abstract syntax tree, compiler front end, syntactic sugar, desugaring, refactoring.
This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

approach). Secondly, the *external* approach relies on some existing, widely-used compiler infrastructure for the given programming language. However, as we shall see, both approaches have disadvantages.

Older compilers provide no convenient ways to access the output of the compiler front end. For example, earlier versions of GCC (before v4.5) offer intermediate files for observing available information between different compilation phases – which is a rather inconvenient input to a language processing tool. This is where the standalone approach may be needed: the tool needs to re-implement (a part of) the compiler front end for the language. Obviously, this is quite expensive from the tool developer's point of view. Moreover, this makes the tool more vulnerable against the evolution of the programming language. Modern compilers such as Clang [10, p. 32], GHC [3] or Scalac [14] provide APIs to access (annotated) abstract syntax trees (AST) at different stages of the compilation process. Annotated ASTs may convey not only syntactic information, but semantic information (e.g. types) as well. This turns out to be a useful input for a language processing tool – a clear benefit of the external approach.

Rich languages offer a great amount of syntactic sugar, so that programmers can write terse, expressive, and easy-to-read code. The syntactic sugar, however, is typically eliminated from the AST. The compiler replaces certain fancy programming language constructs with semantically equivalent simpler constructs (often referred to as *core language* constructs). This *desugaring* process results in loss of information, which can be a disadvantage of the external approach: the language processing tool will be unable to reproduce the original, syntactically rich source code. Although syntactic sugar does not affect the meaning of a program (with respect to core language constructs), it does have a significant impact on readability and maintainability – i.e. code quality. Therefore recovery of syntactic sugar in a language processing tool is an essential issue. For instance, we would like to observe the original, rich syntax, when the tool communicates analysis results back to the programmer, or pretty prints the code.

As the main contribution, this paper proposes the concept of *front end fusion*: a technique to preserve syntactic sugar for a programming language processing tool, if the external compiler infrastructure used by the tool applies desugaring during the construction of annotated abstract syntax trees. We propose a *hybrid front end*, a language processing tool front end, which is the result of front end fusion: it is hybrid because it combines external and standalone front ends. The presented approach performs a fusion of a custom standalone parser and an external compiler infrastructure when creating the hybrid front end. The main advantage of the presented methodology is to

rely on an external compiler front end, use the static semantic information calculated by the compiler, and replace its parsed information with a "non-desugared" syntax tree.

In the presentation below we show how to assemble a hybrid front end for Scala. The concrete problem to solve is to obtain an AST representing the rich, sugared syntax of a Scala source code, and annotate its nodes with type information provided by the desugaring Scala compiler.

The rest of the paper is structured as follows: in Section 2 we present desugaring in Scala, and provide a few examples. Section 4 describes some difficulties in front end fusion, and Section 5 provides the fusing algorithm. Section 6 presents a discussion about the presented methodology. Finally, in Sections 7 and 8 we present related work and conclude the paper.

2. Desugaring

Scala is a particularly good language to study desugaring, since it heavily relies on syntactic sugars. For example, in this language one-argument methods can be invoked without dot and a pair of parentheses as well. This makes both `args contains "--help"` and `args.contains("--help")` valid. Furthermore, the anonymous (or lambda-) function that increases an integer by one may be written as `_ + 1`, which will be expanded into `x => x + 1`. The **`for`**-loop is also a syntactic sugar, and not part of the core language. The loop that prints powers of two to the standard output is the following:

```
for (e <- List(0, 1, 2, 3, 4)) println(Math.pow(2, e))
```

This may as well be written using the `foreach` method:

```
List(0, 1, 2, 4).foreach{ e => println(Math.pow(2, e)) }
```

Lastly, the expression which overwrites an element of an array is as follows:

```
val xs : Array[String] = Array("zero", "one", "")
xs(2) = "two"
```

The second line may also be written as

```
xs.update(2, "two")
```

In all of these examples the compiler rewrites the former to the latter during parsing. An important consequence of these and the many other syntactic sugars is that Scala is especially well-suited for embedding languages (e.g. creating embedded domain-specific languages, EDSLs). However, syntactic sugars are rather ubiquitous, and can be found in other languages as well. In Java, anonymous functions are syntactic sugars for instances of classes with suitable "functional interface". Anonymous functions have the benefit that they are easier to construct and pass around, especially when working with

streams. Another nice example of syntactic sugar is the do-notation of Haskell, which makes it possible to write imperative style code in a purely functional language.

Syntactic sugar can be defined in terms of *rewriting rules*. A rewriting rule specifies the equivalent language constructs of the core language, thus it gives semantics to a syntactic sugar. The application of the rewrite rules may take place at different phases of the compilation process. For example, semicolon inference (e.g. in Scala and Eiffel) may be performed by the lexer. Operator syntax in Scala is rewritten to method calls during parsing. Finally, lambda-functions are rewritten to occurrences of `PartialFunction` or `Function` objects after typing, in a separate phase. In the end, however, the compiler front-end can output a desugared annotated abstract syntax tree containing the constructs of the core language.

Desugaring is not an injective function, different source code may result in the same desugared AST. On the one hand, the desugared AST is convenient to work with in the compiler, which is only interested in whether the code is semantically correct, and in the meaning of the code. On the other hand, the desugared AST may be too abstract to work with in a static analyser, in a refactoring tool, or in a pretty-printer, where the faithful reproduction of the original source code is expected.

Another source of information loss about the syntax used in the source code is demonstrated by the following example. Consider a simple Scala class, which implements a counter. It has a hidden mutable variable `count`, an `increment` procedure to increase `count` by one, and a `get` function to retrieve the current value.

```
class Counter {
 private var count : Int  = 0
 def increment()    : Unit = count = count + 1
 def get()          : Int  = count
}
```

The compilation technique used in the compiler turns the hidden mutable variable into even more hidden ("object-private"), generates a getter (`count`) and a setter (`count_=`) method, and rewrites every access to the `count` field to an invocation of the getter, and every update to an invocation of the setter. Shall we consider this as removal of syntactic sugar? Or is this `Counter` example a counter-example to desugaring? In any case, when pretty-printing the AST constructed by the compiler, the class looks quite different compared to the original source code.

```
class Counter {
  private[this] var count : Int = 0
```

```
  private def count : Int = count
  private def count_=(newVal : Int) : Unit = count = newVal
  def increment() : Unit = count_=(count.+(1))  // + is a
 method in the Int class
  def get() : Int = count
}
```

On a side note, this code may seem broken because of a name conflict between the field and its getter method. If we investigate the AST directly, we discover that the name of the field is not "`count`", but "`count `". The extra space character in the name of the field is not handled properly by the standard pretty-printer, and this causes the confusion. The right way to pretty-print the AST would be to use a so-called "literal identifier", as follows.

```
  private[this] var count : Int = 0
  private def count : Int = count
```

All in all, this example also makes it clear that the abstract representation of the code in the compiler-generated AST may lose too much syntactical information about the source code.

3. Hybrid front end

In the presence of a desugaring compiler and an independent parser producing accurate, syntactically sugared ASTs, a hybrid front end can be assembled. The hybrid front end produces an AST which is built from the AST of the custom parser, which avoids desugaring, and preserves all the syntactical information available in the source code. Then, this AST is combined with the desugared AST constructed by the desugaring compiler front end, which contains collected and inferred static semantic information. In this approach only a parser (and a lexer) may need to be developed, and the "hard part", the semantic analyses including name resolution and typing can be carried out by an existing tool, the compiler. This combination of the *standalone* and *external* approaches should be a good trade-off for many cases.

In this paper we investigate how to build such a hybrid front end for the Scala language. Scala is selected as case study for its richness in syntactic sugars, its desugaring compiler. Fortunately, there is no need to develop an accurate parser for Scala: Scalameta, an open-source meta-programming library [13], suits our needs. The proposed hybrid front end relies on Scalameta to parse source codes, and on the Scala compiler to resolve names and infer types. In other words, the parser of the Scala compiler is "replaced" by the parser of Scalameta, as illustrated on Figure 1.

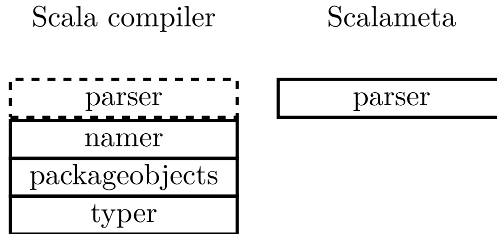


FIGURE 1. Phases of the hybrid front end for Scala.

4. Difficulties in front end fusion

Our main goal is to propose a hybrid front end for Scala producing syntactically rich AST annotated with proper type information. The front-end constructs the AST using Scalameta, and attaches type information computed by the Scala compiler. This annotated AST is an excellent input for various language processing tools.

In order to find the type of expressions represented by the nodes of the Scalameta AST in the typed, desugared AST produced by the compiler, a matching between the two ASTs must be established. The two ASTs have a very similar structure, apart, of course, from the nodes representing a syntactic sugar in the Scalameta AST and their desugared counterparts in the compiler AST. However, the two tools use different names for the same syntactic categories. Literals are represented with nodes of type `Literal` in the Scala compiler, while Scalameta uses type-specific specializations of the `Lit` type. Therefore, in the case of the selected two tools, a matching between the two ASTs based on node types is cumbersome to define. The position information attached to AST nodes proved to be a better basis for the matching. The details of this typing technique will be discussed in Section 5.1. Before that, we investigate two issues which can hamper our fusion approach.

4.1. **Position consistency.** Position based matching works when both terminal and non-terminal nodes have information about their positions in the source file. Position ranges of non-terminal nodes are synthesized from the positions of their children.

We can say that a node from one of the ASTs and a node from the other AST are in *same-position relationship*, if the position ranges defined by their tokens are equal. The same-position relationship between the two tools is position consistent if it is a one-to-one relationship. In this case, the fact that two nodes from the two ASTs are in same-position relationship guarantees that they are the roots of subtrees representing the same code fragment.

Unfortunately, Scalameta and the Scala compiler are not position consistent. Some of the desugaring transformations can result in position inconsistencies. An example will be presented in Section 5 (Figure 3).

4.2. **Preservation of types in desugaring.** When we copy type information from the typed AST to the sugared AST, we identify matching AST nodes using the same-position relationship. If a node c in the compiler AST is in this relationship with a node s in the sugared AST, we copy the type information from c to s. This approach is correct, if c and s represent Scala expressions of the same type.

In the case of desugaring, however, nodes in the same position in the two ASTs may refer to Scala expressions of different types. Consider, for example, the `increment` method of `Counter` in Section 2, where the assignment to the `count` variable is desugared to the invocation of the setter method `count_=`. Here, the `count` variable on the left-hand side of the assignment operator is in same position relationship with the setter method. Note that the type of `count` is `Int`, and the type of `count_=` is the function type `(Int) : Unit`. Hence it is an error to copy the type information from the compiler AST to the sugared one with respect to these two AST nodes.

Section 2 offers examples of type-preserving desugarings as well. When desugaring `args contains "--help"` to `args.contains("--help")`, the types for all pairs of nodes in same position relationship are identical. The same holds for desugaring the anonymous function `_ + 1` to `x => x + 1`.

Note that nodes inserted by the compiler during desugaring do not cause a problem if they are inserted to “unused” positions.

The construction of the hybrid front end would be easy if position consistency and type preservation held. In that case nodes in same-position relationship would represent the same expression, thus they would have the same type. Unfortunately, these properties do not hold for the chosen front ends: the Scalameta library and the Scala compiler. This may lead to annotating with ambiguous and even incorrect types, as we shall see in Section 5.1.

5. Fusion of two compiler front ends

Now we need to investigate how to use Scalameta and the Scala compiler together. The hybrid front end traverses the sugared and the desugared ASTs from top to bottom. The output is an annotated AST, which includes all terminal and non-terminal nodes of the sugared AST, as well as the semantic information of the desugared AST, as presented in the rest of this section. We conclude with challenges posed by Scala compiler desugarings.

5.1. **Typing a sugared AST.** Our type copying algorithm annotates the sugared AST with semantic information from the desugared AST. The algorithm is given in pseudo-code below. The procedure TYPE takes two nodes, a sugared one and a desugared one, as parameters. The nodes need not be in same-position relationship. The procedure searches for same-position counterparts in the desugared subtree. The procedure ANNOTATE appends the type of a match to a list of possible types for $node_s$. Then the typing algorithm for children of $node_s$ is done, this time the match is set as root of the desugared AST. Note that this choice restricts the search for same-position counterpart to the subtree of the match.

If there is no match found for $node_s$ then there still may be matches for children of $node_s$, so the typing continues.

```
TYPE(node_s, node_d)
  let node_s be the root of sugared and node_d the root of the
    desugared abstract syntax subtrees
  matches = SAME-POSITION(node_s, node_d)
  for match in matches
    ANNOTATE(node_s, TYPE(match))
    for child in CHILDREN(node_s)
      TYPE(child, match)
  if EMPTY(matches)
    for child in CHILDREN(node_s)
      TYPE(child, node_d)
```

The function SAME-POSITION returns a set of desugared nodes that are in same-position relationship with $node_s$. The function performs a recursive depth-first traversal of the desugared subtree. The operator “includes” checks whether a position range of a node is between the start and end of position range of another node. The function SAME-POSITION uses “includes” to skip unrelated parts. In case of position consistency, the function SAME-POSITION always returns a singleton set.

```
SAME-POSITION(node_s, node_d)
  let matches be an empty set of desugared nodes
  if POSITION(node_s) == POSITION(node_d)
    ADD(node_d, matches)
  for child in CHILDREN(node_d)
    if POSITION(child) includes POSITION(node_s)
      UNION(SAME-POSITION(node_s, child), matches)
  return matches
```

We demonstrate the typing algorithm using the desugaring examples from Section 2. We show how to type the anonymous function `_ + 1` and the array element overwrite `xs(2) = "two"`.

For the anonymous function, we are required to use the following class definition because the compiler accepts only complete compilation units.

```
class C {
  val inc : Int => Int = _ + 1
}
```

The sugared and desugared ASTs of the expression `_ + 1` is illustrated on Figure 2. The nodes `ApplyInfix` and `Function` are the roots of the subtrees in the two ASTs. They are in same-position relationship. For each of the children of `ApplyInfix`, the algorithm searches for same-position counterparts in the subtree of `Function`. It annotates `Placeholder` correctly with the `Int` type. However, `ApplyInfix` has ambiguous type because it has two same-position counterparts (a result of the violation of position consistency): the algorithm annotates with `Int` and `Int => Int`. Also, the algorithm does not annotate `Name("+")` since the node is not in same-position relationship with any nodes.

For the array element overwrite, we use the following program:

```
object O {
  def main(args : Array[String]) {
    val xs : Array[String] = Array("zero", "one", "")
    xs(2) = "two"
  }
}
```

The sugared and desugared ASTs of the assignment `xs(2) = "two"` is shown on Figure 3. Every node in the sugared can be annotated since each node is in one or more same-position relationships. The types of `Update`, `Int(2)` and `String("two")` are `Unit`, `Int`, `String`, respectively. However, `Name("xs")` receives two distinct types: the correct type `Array[String` and the type of the method `update`, which is `(Int, String) : Unit`. Again, this is a result of violation of position consistency.

6. Evaluation

The problem to be solved is implementation of a suitable front end for a variety of external language processing tools. Most common features that these tools offer are *static analysis* and *program code transformation*. Many tools statically analyse the code at hand, and even perform transformation

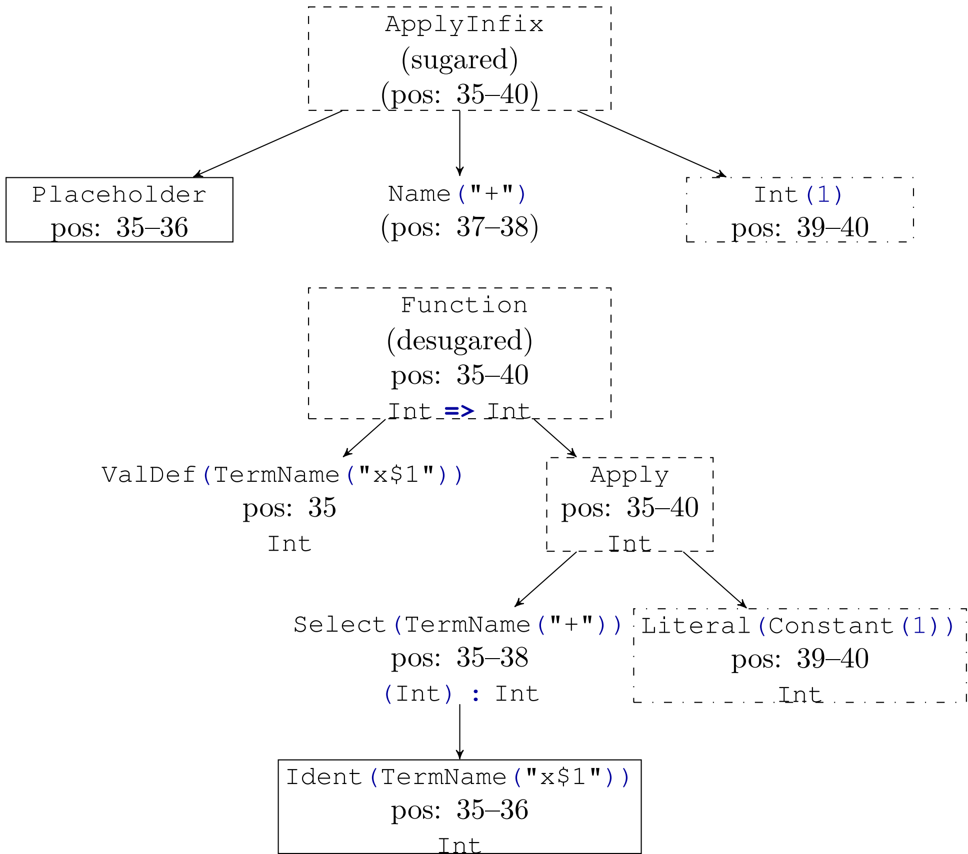


FIGURE 2. Sugared and desugared ASTs of `_ + 1`. Positions are given in offsets.

based on information from static analysis, combining the two features. We elaborate on the effect of hybrid front ends on these features in what follows.

6.1. **Benefits in program code transformation.** External tools which perform source code transformations would benefit from a front end that generates a more accurate source code representation. A typical workflow consists of parsing, locating the code to be transformed in the AST, transformation and pretty printing the AST.

A hybrid front end may improve locating the code in the AST in specific cases. Depending on the compiler front end infrastructure and the order and organisation of the compilation phases, the AST may become subject to optimizations and compile time meta-programming. By the time the external tool

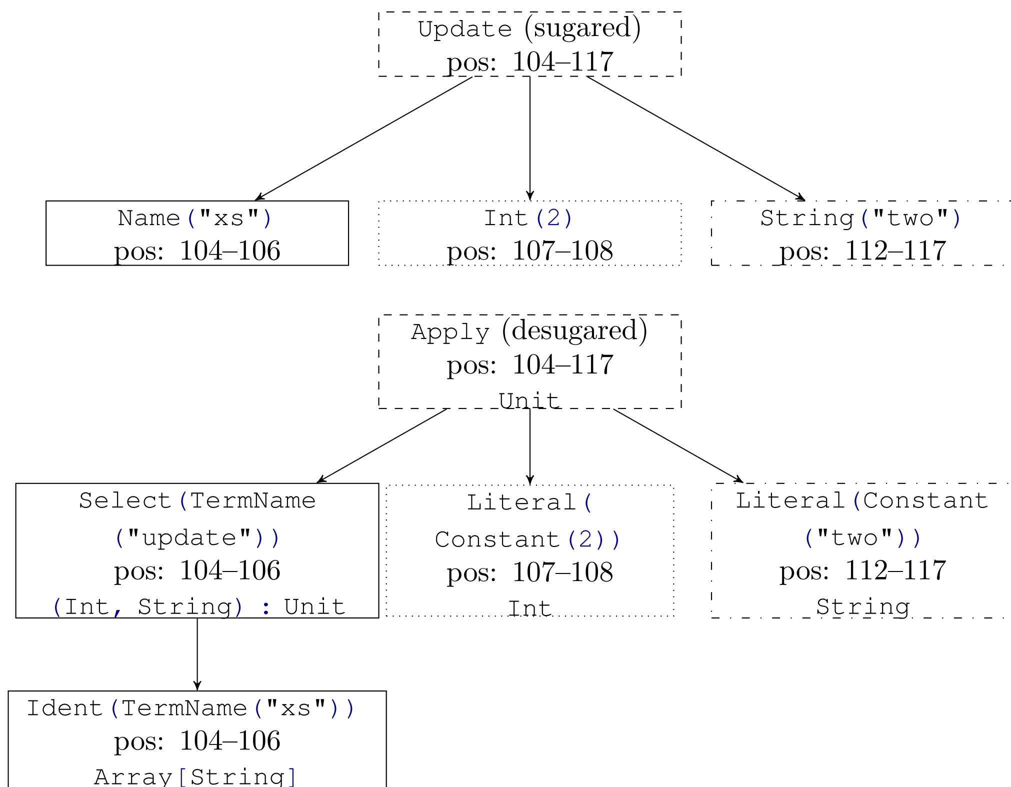


FIGURE 3. Sugared and desugared ASTs of `xs(2) = "two"`. Positions are given in offsets.

receives the AST, constant expressions may be folded, and meta-programming constructs are expanded into generated code. It may happen that the programmer specifies a (part of a) meta-programming construct as the subject of a code transformation, and the compiler front end replaces it with its expansion in the AST, thus the search in the AST fails.

Benefit in pretty printing is clear. A hybrid front end retains lexical and syntactical information on the code. The retained information, which includes syntactic sugars, comments and whitespaces, helps the pretty printer to generate code that pleases the programmer. Without this information, as a side effect, **`for`**-loops may become `foreach` functions, invaluable documentation comments may be lost, and tabs may be replaced with spaces or vice versa, throwing away careful indentation.

6.2. **Effect on static analysis.** The difference between ASTs in representation of the same statement, such as the assignment `counter = counter + 1`,

may cause ambiguity in interprocedural semantic analyses. This statement can be regarded as an assignment, where the value from the right hand side flows to the left hand side, or as an invocation of the setter method `counter_=` in the class `Counter`.

We can resolve this ambiguity in the following way. When the user does not define a setter for `counter`, the compiler will generate a trivial setter. In that case, the analysis can treat the statement as an assignment. Otherwise, the assignment regarded as an invocation of the setter method.

7. Related work

Several programming languages offer syntactic sugars to make software development more convenient and efficient. At first, we present a resugaring technique in Scala, then we investigate other languages as well.

7.1. Resugaring in Scala.

An alternative approach to build a language processing tool is to further enhance the annotated AST provided by an external tool by undoing the desugaring and adding the sugared syntax tree to the representation. In [12] we introduced a *resugaring* algorithm for Scala by linking two ASTs, a sugared and a desugared, together. The output is a joint AST which includes terminal and non-terminal nodes of both ASTs and the links between them. Figure 4 illustrates the relevant fragment of the joint AST of the `Counter` class, with a sugared AST constructed with Scalameta, and a desugared AST provided by the *scalac* compiler.

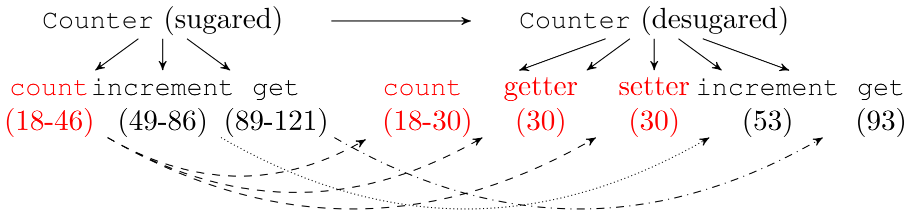


Figure 4. Resugared AST of the `Counter` class. Positions are given in offsets.

The links between the nodes of the two ASTs are established by an algorithm that traverses the two ASTs simultaneously in level-order. The algorithm is presented on Figure 5.

7.2. Scalameta.

Our choice of parser library, Scalameta [13], comes with capability to annotate ASTs with types. The Scalameta compiler plugin collects information from compiler into semantic database. During parsing, Scalameta

```
RESUGAR(tree_s, tree_d)
  let tree_s be the root of sugared and tree_d the root of the
    desugared AST
  edges = RESUGAR-CHILDREN(tree_s, tree_d)
  if POSITION(tree_s) overlaps POSITION(tree_d)
    ADD(edges, EDGE(tree_s, tree_d))
  return edges

RESUGAR-CHILDREN(tree_s, tree_d)
  let edges be an empty set of links between the nodes
  let mapping be an empty mapping from positions to nodes
  for i = 1 to NUM-CHILDREN(tree_d)
    ADD(mapping,
        POSITION(CHILDREN(tree_d, i)),
        CHILDREN(tree_d, i))
  for i = 1 to NUM-CHILDREN(tree_s)
    if CHILDREN(tree_s, i) has a matching node in mapping
      let match be the desugared node with overlapping
        position in mapping
      UNION(edges,
            RESUGAR-CHILDREN(CHILDREN(tree_s, i), match))
    return edges
```

FIGURE 5. Resugaring algorithm

consults the semantic database and exposes types in its Semantic API. Similarly to our fused front end, Scalameta also uses position information.

So far, Semantic API is limited to symbol types and name resolution. Annotating complex expressions is a work in progress.

7.3. **Syntactic sugars in other languages.** The records in Erlang are taken as syntactic sugar, thus are translated to tuple expressions by the compiler. A record of n fields is substituted with a tuple of $n+1$ elements, where the very first element is the name of the record and the following elements are the values of the fields (listed in the defined field order).

There are two major refactoring tools for Erlang, RefactorErl [5] and Wrangler. These tools use different approaches in source code processing. RefactorErl follows a standalone approach: it uses its own analyser framework to make every bit of information available. Even the layout, comment, preprocessor constructs and record information are stored in the Semantic Program

Graph (SPG), thus the source code restoration with its context is straightforward. Opposed to RefactorErl, Wrangler [9] is closer to the external approach, since it uses the standard *syntax tools* [1] library that comes with Erlang OTP. Atop of the standard parser, however, the tool annotates ASTs with additional information with macro, record and layout information.

The Glasgow Haskell Compiler [11] uses several well-separated phases in compilation process. Type checking is performed right before desugaring phase, thus extracting representation after type checking phase includes information on syntactic sugar constructs. Haskell-tools [4] refactoring framework uses this representation for further analysis and transformation – this is an external approach.

7.4. **Literate Programming.** Donald Knuth's literate programming [8] allows us to generate documentation and code from a single `WEB` file. The *weaving* process generates TEX document, which can be rendered in human-readable format. The *tangling* process generates code (say, C), which can be compiled and run.

It may be possible to reconstruct the original `WEB` file from TEX document and code. This problem is similar to ours: assuming that front ends for TEX and C can be fused, a hybrid front end could produce a `WEB` file from TEX documentation and C code of the same program. The C code carries information to restore section structure of the `WEB` file. The section names and documentation in each section are part of the TEX file. Annotation comments (e.g. `/*8:*/` and `/*:8*/`) and TEX macros (e.g. `\X8:`) provide a way to establish connection between C code and TEX file. In contrast, we used position information in our hybrid front end for Scala.

7.5. **Preprocessor constructs.** Preprocessor constructs, such as macros, can also be considered as a special form of desugaring. In the original source code a macro application is presented, but usually well before the static semantic analysis a preprocessor substitutes the macro application with the corresponding macro body, and the compiler builds the annotated AST from the expanded macro body. This raises a similar problem as the desugaring in a language processing tool. For example, the source code needs to be pretty printed after a refactoring transformation with the original macro applications kept.

For Erlang, the tool RefactorErl provides a custom parser to store both the original code and the preprocessed one [7, 6]. This makes the pretty printing after refactoring straightforward, and the static analysis more accurate on the expanded AST.

The C programming language also provides a powerful macro system. The tool CRefactory [2] introduces a standalone approach to solve the same issue by preserving the preprocessor directives during parsing.

8. Conclusion

In this paper, we elaborated on how to implement a programming language processing tool in order to minimize the effort. We showed that building upon modern compiler infrastructure helps, but it comes at the price of losing information, due to desugaring. We presented an approach, *front end fusion*, to circumvent this. We proposed an algorithm to construct an annotated abstract syntax tree by fusing the ASTs of the Scala compiler and the Scalameta library.

The presented algorithm is based on the simultaneous traversing of the ASTs to be fused while considering the position consistency of the desugared nodes. We also discussed the need of type-preserving desugaring in terms of the fusion, and presented the solution for the Scala-specific deviations.

We have implemented and evaluated our methodology by creating a language processing tool for Scala with the aim of providing a refactoring framework for parallelisation. Probably, the presented approach may be used for other programming languages as well.

9. Acknowledgement

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013).

We would like to thank the anonymous reviewers for calling our attention to literate programming.

References

[1] Ericsson AB. Erlang Syntax Tools User's Guide. http://erlang.org/doc/apps/syntax_tools/users_guide.html, 2018.

[2] Alejandra Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2005. AAI3199001.

[3] Adam Gundry. A Typechecker Plugin for Units of Measure. *SIGPLAN Not.*, 50:11–22, August 2015.

[4] Haskell-tools Refact. A GHC based toolset for Haskell programming. http://haskelltools.org, 2018.

[5] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Melinda Tóth, István Bozó, and Roland Király. Modeling semantic knowledge in erlang for refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT, Sp. Issue, Studia Universitatis Babeş-Bolyai, Series Informatica*, volume 54, pages 7–16, 2009.

[6] Róbert Kitlei, I. Bozó, Tamás Kozsik, Máté Tejfel, and Melinda Tóth. Analysis of preprocessor constructs in erlang. In *Proceedings of the 9th ACM SIGPLAN Erlang Workshop*, pages 45–55, Baltimore, USA, September 2010.

[7] Róbert Kitlei, László Lövei, Tamás Nagy, Zoltán Horváth, and Tamás Kozsik. Layout preserving parser for refactoring in Erlang. *Acta Electrotechnica et Informatica*, 9(3):54–63, 2009.

[8] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[9] Huiqing Li and Simon Thompson. Tool support for refactoring functional programs. In *Partial Evaluation and Program Manipulation*, San Francisco, California, USA, January 2008. Assoc of Computing Machinery.

[10] Bruno Cardoso Lopes and Rafael Auler. *Getting Started with LLVM Core Libraries*. Packt Publishing, 1st edition, 2014.

[11] Simon Marlow and Simon Peyton-Jones. The glasgow haskell compiler, 2012. in The Architecture of Open Source Applications (Volume II: Structure, Scale, and a Few More Fearless Hacks), http://aosabook.org/en/ghc.html.

[12] Artúr Poór and Tamás Kozsik. Resugaring: Undo desugaring in language processing tools. Thessaloniki, Greece, 2017. To appear in the Proceedings of the Symphosium of Computer Languages and Tools.

[13] Scalameta. Metaprogramming library for Scala. http://scalameta.org, 2018.

[14] Dean Wampler and Alex Payne. *Programming Scala – Scalability = Functional Programming + Objects*. O'Reilly Media, 2nd edition, December 2014.

Eötvös Loránd University, Budapest, Hungary
Email address: `{poor_a, kto, toth_m, bozo_i}@inf.elte.hu`

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 2, 2018
DOI: 10.24193/subbi.2018.2.02

AN EFFICIENT GRAPH VISUALISATION FRAMEWORK FOR REFACTORERL

MÁTYÁS KOMÁROMI, ISTVÁN BOZÓ, AND MELINDA TÓTH

Abstract. Graph visualisation is a well-known and researched field of graphical informatics. Several good algorithms were developed and reviewed by our days. However, most of the graph drawing tools mainly focus on static drawing generation. In this paper we present an approach that is efficient enough to visualise the user-requested parts (views) of a relatively large Semantic Program Graphs of Erlang projects in soft real-time. With the presented approach the visualised graphs can be traversed interactively, by changing between different levels of detailed information, which may support code comprehension in the RefactorErl framework.

1. Introduction

Graph visualisation is a popular research topic, and several algorithms and tools exist that are ready to use. However, the increasing size of the nodes and links among them to visualise on the graph makes the layout calculation more complicated and slow.

Graph visualisation is often used in tools supporting static and dynamic source code comprehension. It is very convenient to denote the relations/dependencies among program entities using a graph view.

RefactorErl [11, 22] is a static source code analysis and transformation tool for Erlang [10]. Besides the more than 20 refactorings, the tool provides several functionalities to support program comprehension: semantic queries,

Received by the editors: 31 March 2018.

2010 *Mathematics Subject Classification.* 68N01, 65D18.

1998 *CR Categories and Descriptors.* I.3.1 [**COMPUTER GRAPHICS**]: Hardware Architecture – *Parallel processing*; I.3.5 [**COMPUTER GRAPHICS**]: Computational Geometry and Object Modeling – *Physically based modelling*.

Key words and phrases. graph visualisation, layout generation, physically based modelling, code comprehension.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

static-dynamic call analyses, data-flow analysis, dependence analyses, etc. The results of several analyses can be visualised as graphs.

Also, the tool itself uses a graph based intermediate representation for the source code: the Semantic Program Graph [16]. The SPG contains lexical, syntactic, and semantic information about the source code. These three layers generate huge amount of data (nodes and edges) from the source code. Even for a module with few hundreds of lines of code (LOC) the standard visualisation tools, such as Graphviz [5], are hardly able to generate a proper view.

Although, it depends on the complexity of the source code, but in general, there are 50-times more nodes and edges in the graph than lines of code in the source code.

When analysing millions of LOC in industrial scale software, or when a single Erlang application is analysed, having more than twenty thousands of LOC, the graph visualisation is almost impossible. Thus we decided not to visualise the entire graph, but only the relevant parts for the user. We needed a graph that can be traversed fully interactively, switching between the levels of information.

The main contribution of this paper is a solution to the above presented problem. We are providing a graph visualisation method and a new component *gview* for RefactorErl that is capable of handling real Erlang projects. We demonstrate different views available through the new component and evaluate the performance on different open-source projects.

2. Related work

Graph visualisation has been subject to research since long time ago, many good visualisation tools are available for use today.

2.1. **Graphviz.**

Graphviz [5] is an open source graph visualisation software. It supports many input formats, specifications, and algorithms for presenting graphs. However, Graphviz is unable to render graphs with high node count, in an interactive manner, as experienced during the development of the user interface of RefactorErl. Rendering the main view of Mnesia into an svg file with Graphviz, consisting of around 2200 nodes, took around 3700 seconds (more than an hour). After the layout generation, opening the generated svg file in a browser took 4 minutes. The layout for the very same view can be generated by *gview* in 2 minutes, cached, and then displayed interactively.

2.2. **Wolfram Mathematica.**

Wolfram Mathematica [9]: The Wolfram Language provides functions for the aesthetic drawing of graphs. Algorithms

implemented include spring embedding, spring-electrical embedding, high-dimensional embedding, radial drawing, random embedding, circular embedding, and spiral embedding. In addition, algorithms for layered/hierarchical drawing of directed graphs as well as for the drawing of trees are available.

2.3. **MSAGL.** MSAGL [6]: MSAGL is a .NET library and tool for graph layout and viewing. MSAGL was developed in Microsoft by Lev Na-chmanson, Sergey Pupyrev, Tim Dwyer, Ted Hart, and Roman Prutkin.

2.4. **Erlgraph.** Erlgraph [2] is an application which enables a d3js force directed graph to connect to an Erlang VM. The plotted data consists of the active processes of the running application and the messages they send or receive. D3.js is a JavaScript library for manipulating documents based on data. D3 aids creating data driven animations using HTML, SVG, and CSS. D3 is said to emphasis on modern web standards, which gives full capability of modern browsers without tying to a proprietary framework. Erlgraph provides an insight into the underlying mechanism of an Erlang application in runtime. Erlgraph is an extremely useful tool for visualising how processes of the project interact with each other. What different in *gview* and Erlgraph is that while Erlgraph realises a dynamic (runtime) analysis of Erlang code, *gview* targets static analysis of the Erlang project at hand, which means no code needs to be executed.

3. Background

Gview is built upon Flib [4]. Flib (at the time of writing this paper) is a single-author OpenGL development library for C++. It supports creating and handling Windows, OGL contexts and OpenGL objects. It also has a GUI system, graphical and linear mathematical tools. On windows platforms, Flib uses the standard Windows API for window creation and management, on Linux platforms, it uses the XLib windowing system. Consequently, OpenGL context creation is done using WGL and GLX respectively. The Flib API documentation generated by doxygen can be found on the online repository [3].

4. Visualisation framework: *gview*

4.1. **Overview.** Dot [18] is a general purpose graph describing language capable of representing directed and non-directed graphs alike, with extra information options on both edges and nodes. RefactorErl supports exporting all the data from semantic graphs of loaded Erlang files to a single dot file. Therefore, by parsing this exported dot file *gview* is able to create a layout for the views of the graphs and display these views interactively. The architecture of the software can be seen on Figure 1.

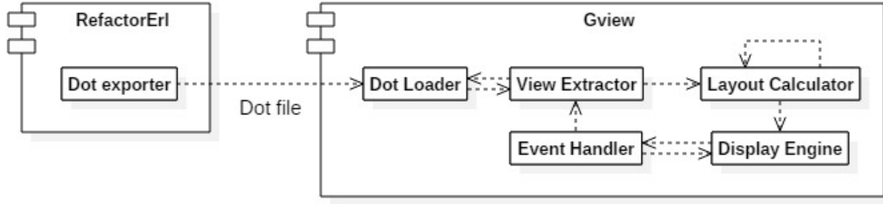


FIGURE 1. The architecture of *gview*.

4.2. **Dot files.** The main reason of using dot files is the existing support for it in RefactorErl and the ease of implementing a custom parser in C++. On the other hand, using dot files as intermediate data representation is quite rigid. Having a file changed one must regenerate the dot file in RefactorErl, and on every startup, *gview* has to parse the whole dot file. The data-flow of the program can be seen on Figure 2.

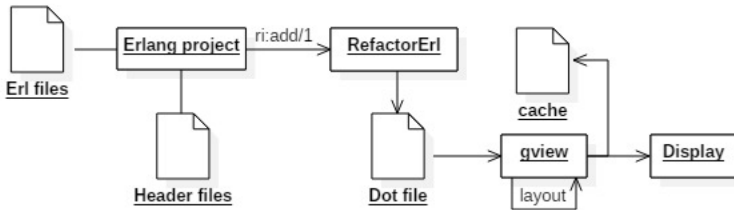


FIGURE 2. Data-flow in the visualisation process.

4.3. **Rendering.** The rendering is done using OpenGL [21] with the supporting classes of Flib. We preferred OpenGL because it is hardware close and very fast and, unlike DirectX, portable across operating systems. The drawing data is generated on the fly after each iteration of the layout algorithm, thick lines are tessellated into triangles, circles into regular polygons by the C++ implementation. Extra information on vertices for anti-aliasing is also added in this process. The drawing data is then uploaded to the graphics card and drawn as a single batch, avoiding the cost of setting up many drawing calls. The anti-aliasing is done by our shader program on the GPU.

4.4. **Layout.** Our layout calculating algorithm is a modified version of two-dimensional N-Body simulation, known as Force-Directed Layout (FDL) [14]. In a classic N-Body simulation we would have N objects, each pair exerting attracting gravitational force, proportional to the mass and inverse square of the distance, to each other. However our FDL algorithm considers these objects (nodes of the graph) to have electric charges, instead of gravitational effects, and thus repel one-another. Furthermore edges between nodes of the graph are represented as springs of logarithmic strength, meaning the force they exert is logarithmically proportional to the distance they stretch across. This way the edges attract nodes they connect. After the initial setup Acting net forces are calculated in every iteration for each node, then we update the position of nodes according to these net forces and the elapsed simulation time. Forces are taken to act instantaneously, which means they are applied directly to the position of nodes not on their speed.

There are many toggleable elements of this simulation; the strength of the springs, the amount charge a node has, the stepping time between iterations of the simulation, and initial positions of the nodes. Choosing these parameters were done on an empirical basis; we experimented with them until the results looked good. Therefore, by modifying the charge of nodes or the strength of the springs we can change the final spacing among edges or nodes. This way we can emphasis parts of the displayed graph.

4.5. **Related libraries.** Beside the Flib there are many other excellent frameworks for OpenGL development.

SDL (Simple Directmedia Layer) [12] is one of the oldest of these frameworks, it has outstanding wide system and hardware support. However, its interface was designed for C not C++, SDL does not use object-oriented paradigms. SFML (Simple and Fast Multimedia Layer) [8] is another excellent choice for OpenGL development. It is completely object oriented (by the C++ binding) with cross-platform support, but it lacks the GUI module. Qt [7] is a professional and robust framework with good GUI and OpenGL support.

Flib (developed by the author of this paper) brings the required OpenGL window and context management classes and wrapper classes for the mostly used GL object sand it has a GUI module which we use for simple text output. It also has a robust event handling system and very convenient graphic classes such as vectors and matrices.

4.6. **GUI Framework.** Flib provides GUI classes on top of OpenGL. *Gview* uses Flib to automatically open a window, an OGL context associated with this window, and a GUI context which is responsible for storing GUI related data

such as fonts and shaders. The GUI context also has a main GUI layout where the application can attach GUI elements. The GUI elements are structured in a tree pattern: each GUI element may have a parent layout, each layout may have any number of children elements, layouts are GUI elements too.

The events, draw calls, and update calls are forwarded down the hierarchical structure each time. To detect node selection and change the current view, we use the event listener functionality of Flib to translate mouse events such as movements, button down, button up, and more complex events as click or double click. The graph transformation is also done with the built-in classes of Flib, these are responsible for calculation of scaling, offset, and rotation values that can be used for generation of the displayed mesh.

4.7. **Mesh Generation.** Although a mesh is generated on the CPU after each iteration of the layout calculation by *gview*, using the CPU for this task is perfectly sufficient since the displayed part of the graph is expected to have at most 3000 nodes and 10000 edges (not even Mnesia has this much in the main view) which results in at most tens of thousands of triangles. For this, the task is easily handled by an average CPU these days.

Having the layout (the node positions) of the calculated graph, we tessellate it into triangles and lines. The data to be transferred is generated in a batch, thus it can be sent in one operation what makes the upload fast. After the mesh calculation, the drawing can be performed with a single call. Uploading to the GPU is done by buffer object streaming, dropping the buffer object before each upload and creating a new one. This enables the GL to complete drawing commands referring the previous buffer while uploading the new one.

Uploading the mesh is done using the designated interface of Flib for simplicity. Tessellation of lines uses the built-in tessellation functionality from Flib which automatically includes distance-field data needed by the anti-aliasing technique. The data is drawn using the drawing API of Flib as well. Results of the tessellation can be seen on Figure 3.

4.8. **Dynamic Level of Detail.** A very useful technique for graphical applications is Dynamic Level Of Detail [23] (DLOD), it involves altering the detailedness of an object (how many triangles it has or what textures it uses) based on how small that object appears on the screen. This technique is effective because our eyes can not make out the difference on small objects, we use DLOD in *gview* to reduce workload on CPU when the mesh is generated. As for an example, circles that represent module or file nodes get drawn as regular n-sided polygons. The value of n is based on the zooming, the more it is zoomed in, the larger the value of n is. Application of DLOD is shown on Figure 4.

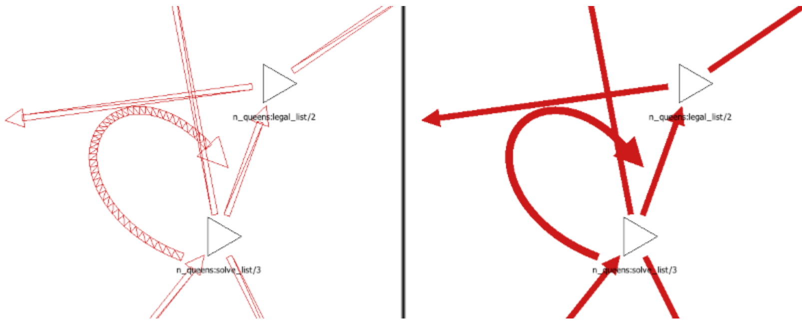


FIGURE 3. Tessellation of the edges (the thickness of the edges was increased for demonstration).

To calculate the number of sides our polygon needs, when approximating a circle, let us say the circle has r radius given in pixels on the screen. The size of the radius in pixels can be calculated from the resolution at which the rendering is done and the zooming level. Considering the formula for the circle perimeter $2*r*\pi$ we can approximate the number of pixels on the perimeter of the circle easily, as $p = 2*r*$M_PI in C++. When drawing a circle of radius r, we ought to approximate a curve of length p. After taking measurements, we found that using $p/4$ line segments produces satisfactory images. It is also worth noting, that letting the number of sides drop below 3 is trivially pointless.

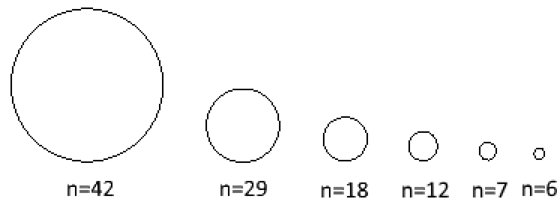


FIGURE 4. Number of sides (denoted as n) of regular polygons representing circles with different zooming level.

4.9. **Anti-aliasing.** Anti-aliasing [13] is done using distance fields, through a technique called distance-to-edge anti-aliasing (DEAA) [17]. The main idea is that, if we know the distance from the edge of the primitive when processing a fragment, then we can set the transparency to drop when getting near the edge of the primitive. Thus, having the transparency stored in the alpha channel of the fragment, the OpenGL blending mechanism will ensure it will be displayed

transparently. The effect of using anti-aliasing can be seen in Figure 5. The distance function is calculated as the minimum of the distances to the edges of the mesh. For this purpose, we store the distances in separate channels (red channel holds the distance from the left edge, green channel from the right edge etc.). Calculating these distances on a per-vertex basis and using the OpenGL built-in linear interpolation functions, we can approximate the distance to the edges of the mesh as the minimum of the interpolated distances.

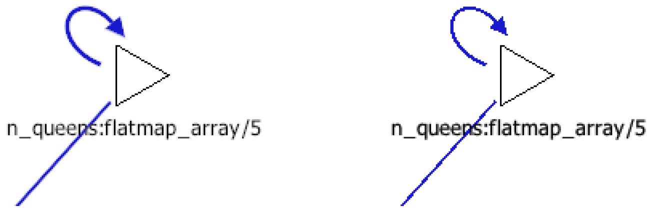


FIGURE 5. Call and recursive edge with and without antialiasing.

5. Generating the layout

The layout of a displayed graph is defined as a list containing points (two-dimensional) for each node of the graph. The algorithm described below takes an initial layout and processes it in iterations. We use the Force-directed layout algorithm, that is described in more detail in Section 4.4. For experimenting, we have defined and implemented three different versions of the algorithm (Sections 5.1, 5.2, and 5.3). The efficiency of different implementations can be seen on Figure 6, where we plotted the number of iterations it took to generate the final layout for tested views, against the number of connections in the views, by connections we mean the number of potential forces acting in the simulation. The number of these connections is in $\mathcal{O}(n^2 + e)$, where n is the number of nodes and e is the number of edges.

5.1. **CPU implementation.** The first implementation uses only the CPU without any optimization; it simply iterates through all node pairs and sums up the forces then applies the forces to the node positions (instantaneous forces). The calculation of forces the nodes exert on other nodes takes $\mathcal{O}(n^2)$ time where n is the number of nodes. Summing the spring forces takes time proportional to the number of edges e: $\mathcal{O}(e)$. Applying the forces on n nodes take $\mathcal{O}(n)$ time.

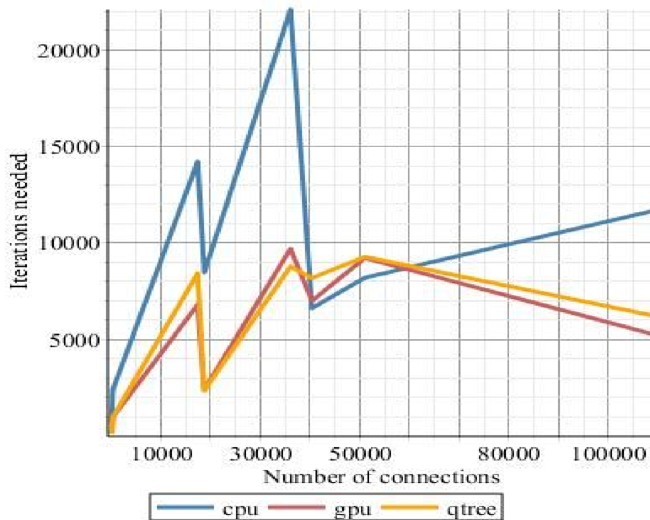


FIGURE 6. Iterations needed to reach final state plotted against number of connections (n^2+e) in view.

5.2. **QuadTrees.** The second implementation uses quad trees for space partitioning to reduce computation time. These quadtrees have regions as nodes, generated by recursively dividing the points in a region into two sets of the equal size, using horizontal splitting lines and vertical splitting lines alternately. This method is also called as Barnes-Hut algorithm [19]. The recursion stops when a certain minimum of nodes in a region is reached. Therefore, when calculating the net force on a node we can traverse the generated tree and approximate the net force exerted by all the nodes in a region of the tree in constant time when it is far enough from the currently processed node. This technique results in $\mathcal{O}(n * log(n))$ time complexity when the distribution of the nodes is even. The problem is that the constant factor of the complexity is quite big as the tree has to be regenerated in each iteration (and not CPU cache friendly). On large views this method outperforms the trivial CPU implementation.

5.3. **GPU implementation.** The third implementation is the parallel equivalent on GPU of the first one using OpenGL Compute Shaders. We chosed OpenGL over Cuda or OpenCL because it has good support for AMD and Intel cards too, integrates nicely with the rest of the drawing code, and requires no extra libraries. The graph is sent to the GPU in adjacency matrix representation in a texture image. A kernel is then dispatched for each node

of the graph which computes the net force acting on that node. The GPU implementation reuses the calculated data and thus data is only streamed to the CPU once every frame (typically 20 iterations). As a consequence, when the number of nodes is smaller than the number of shader cores, the GPU is not used efficiently. This issue could be remedied using a divide and conquer approach: each kernel only calculates the net force on a node exerted by a portion of other nodes. Then another shader is dispatched to sum up the subresults.

5.4. **Alternatives.** Stress majoring [15] is another good algorithm that we looked into using. However, on lack of time it was not implemented. Thus, it is not discussed in this paper.

5.5. **Caching.** Generating the final layout of a graph view is very computational and time costly. Therefore, caching the generated layouts can save important resources and speed up *gview*. Caches need to be stored permanently, thus they are saved as external binary files using the file streams of the C++ standard library.

One way to implement caching is having a cache file for each view of the graph. This potentially results in thousand of files per graph, although they can be loaded separately resulting in less memory usage.

The other way is keeping one cache file per project. This way, the cache management is easier and clearer. The resulting cache file sizes depend on the opened views. The cache files barely reach 50kB even on the largest test project: Mnesia [20].

6. RefactorErl graph views

Plotting the whole graph exported to the dot file would be pointless and computationally extremely expensive. Consequently, we define views of the graph and only plot one of these views at a time reducing workload and letting the user concentrate on one aspect of the project.

6.1. **Main view.** The main view consists of all the modules and functions defined or referenced in the application. Each function is linked to the module they are defined in. One can change the view by clicking on a module node, then the view for that module is shown.

6.2. **Module view.** The module view is similar to the main view, but it displays only one module and the functions defined by the actual module. It has a ROOT node through which one can return to the main view. One can change the view by clicking on a function node, than the view for that function is brought up. The massive main view of Mnesia is shown on Figure 7.

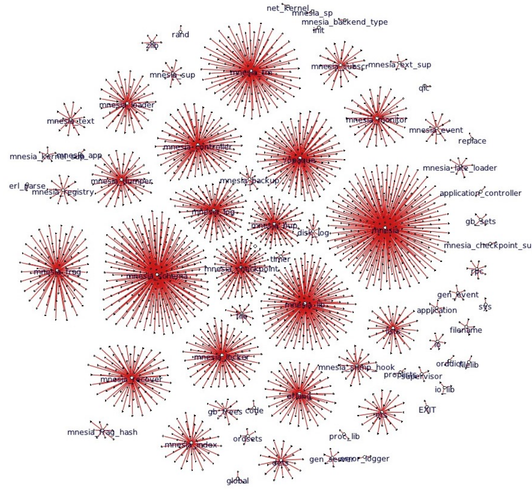


Figure 7. The main view of Mnesia containing more than 2500 nodes and edges.

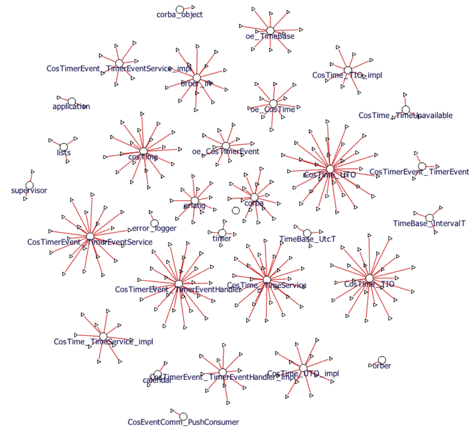


Figure 8. The main view of the CosTime application.

6.3. **Function view.** The function view plots a function and all functions that directly or indirectly call/get called by it (Figure 9). The plot depth of the call graph can be adjusted, this depth is an argument of the view. Currently the depth is not limited, but in the future we plan to limit it to a reasonable size.

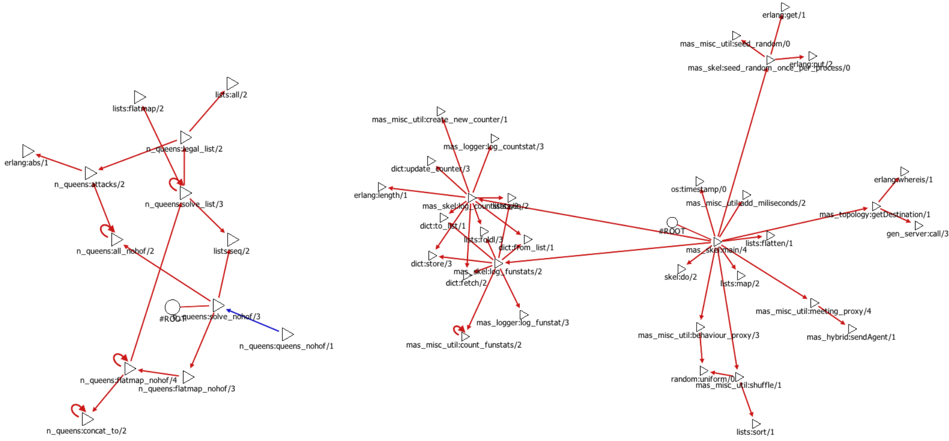


Figure 9. Deep function views of GreenErl and MAS projects.

6.4. **Visual elements.** The displaying engine in *gview* supports various graphical elements to provide information to the user. The shape of the nodes can be customised and set for example to triangle, square, or circle. The width of edges can be adjusted, the colour of the edges, the displayed text below nodes, and the shape of the arrows of edges can be customised as well. Through these options the view generator is able to highlight the differences between the semantic entities (function, module, etc.) of the displayed graph. Some of the available elements are shown on Figure 10.

7. Evaluation

To measure the performance of *gview* and profile it we tested it on several open-source projects.

The largest project was the Mnesia [20], a robust, distributed database management system, written in Erlang. With more than 2500 functions (and around 25 thousands LOC) this is by far the largest project *gview* was tested on. Plotting all the texts of the main view interactively could be considered a challenge alone. Part of the main view is shown on Figure 11.

CosTime [1] is an Erlang implementation of the OMG CORBA Time and TimerEvent Services. It has many modules with few functions, thus it was a good candidate to test *gview* on. Main view of CosTime can be seen on Figure 8.

The measurements were made on loading of dot files and generating the views. We can conclude that the loading of dot files was the major slowdown on startup. The loading may take more than 90% of the start time. Other

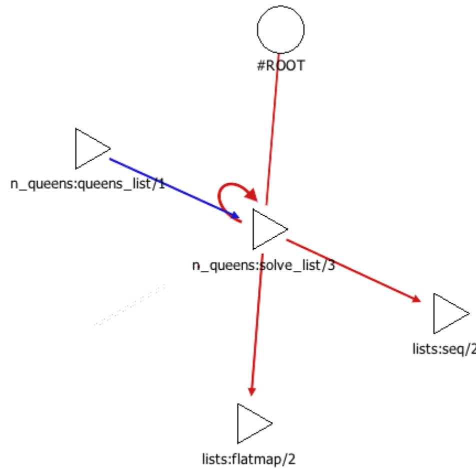


FIGURE 10. Some of the customisable visual elements: triangles, circles, edges, recursive edges, etc.

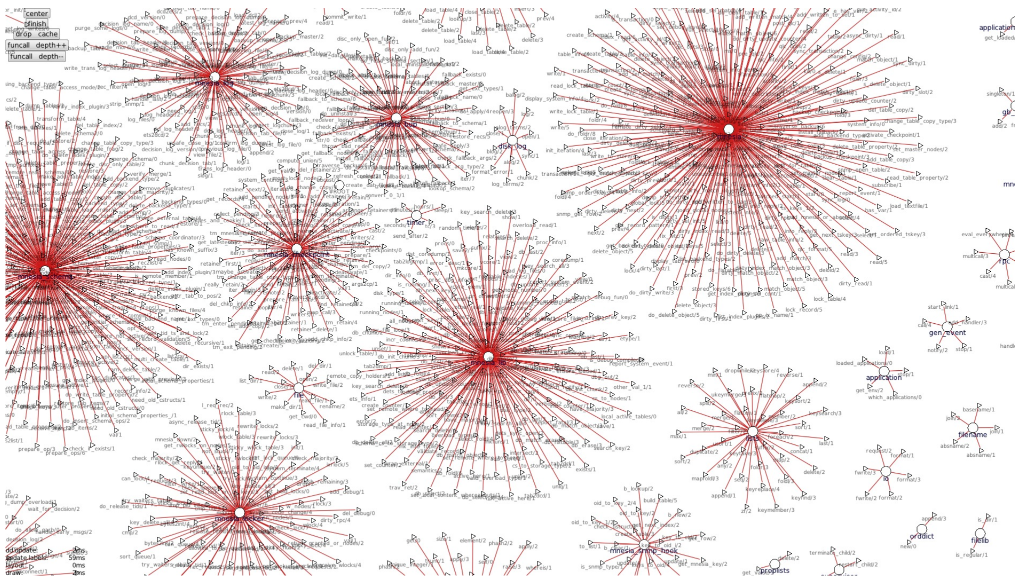


FIGURE 11. The massive amount of functions in Mnesia.

events performed on startup, such as window or OGL context creation take negligible time compared to loading and interpreting dot files.

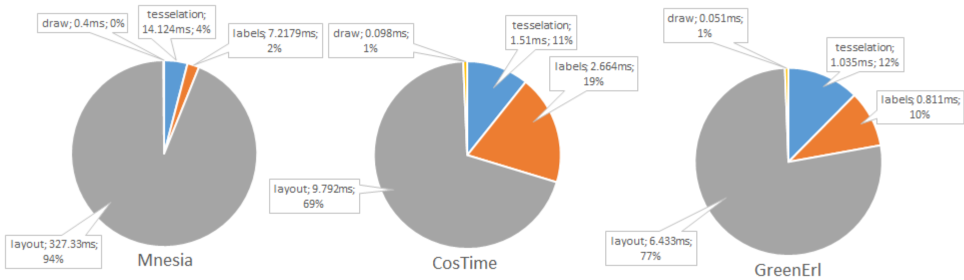


FIGURE 12. The average time taken for layout generation, tessellation, label setting, and drawing the view.

When displaying a view, the most time is spent on generating the layout (70% and up). Label placement and mesh generation could run in real-time (without layout calculation). Figure 12 demonstrates distribution of time spent on different stages. Analysing the charts, it is obvious that described caching mechanism largely improves the efficiency and ensures real-time response on large views.

As an efficiency test, we have compared the execution times of *gview* and the old Graphviz based dependence graph drawing component of RefactorErl. We have exported a graph of the Mnesia application to a dot file. The graph generation with *gview* needed cc. 90 seconds, which could be then interactively used. To generate to same graph in SVG with Grapviz took more than an hour, and because of the amount of nodes displaying and browsing the content was hard to manage.

Measurements were done on a laptop, running Windows 10, with Intel(R) Core(TM) i5-5250U CPU @ 1.60GHz and 8GB of memory, with Intel(R) HD Graphics 6000 integrated GPU, using a TOSHIBA MQ02ABD100H 1TB HDD 5400Hz, which could be considered a low-end setup today. Using a 7th generation Intel processor and a much faster SSD could potentially improve the results of some of these benchmarks.

8. Conclusion and future work

RefactorErl framework has several graphical and command-line interfaces, that support refactorings, static code analysis, and code comprehension as well. The tool uses a Semantic Program Graph as an intermediate representation of the source code. The SPG includes static semantic information beside the syntactic and lexical information. The conversion of the SPG to an SVG file with Graphviz was possible only on relatively small graphs. There was

high demand for an efficient and interactive graph visualisation tool that led us to create the *gview* component presented in this paper.

RefactorErl has support for exporting the SPG to a dot file, thus we used this functionality and the dot format for representing the graph. These files are then processed by our custom dot parser implemented in C++. The views of the exported graph are generated and visualised using OpenGL. We have used Flib for GUI and OGL object management. To calculate the layout of the graph we used Force-directed layout generation. To improve the performance of the interactive viewer, the resulted layout is saved to cache files to avoid the continuous need of recalculating. To enhance the visual quality of the rendered scene we used anti-aliasing. Dynamic Level Of Detail is applied to reduce geometry, which is then tessellated using Flib and drawn in a single batch. We made the appearance fully customisable, thus the users are able to use different shapes and arrows for different semantic entities and relations among them accordingly. The user is able to switch between views using the cursor; pointing on a node and clicking brings up a more detailed view associated with that node.

Although static data access through dot files was a good starting point, it turned out that processing/parsing the dot file is the bottleneck in graph generation. In the future we plan to replace it with dynamic graph information acquired directly from RefactorErl.

Also, using the GPU for parallelisation of mesh tessellation is also an appropriate subject for future research.

References

[1] The CosTime application. http://erlang.org/doc/apps/cosTime/cosTime.pdf. [access date: Jun. 2, 2018].

[2] Erlgraph on GitHub. https://github.com/aol/erlgraph/. [access date: Jun. 4, 2018].

[3] Flib documentation. http://makom789.web.elte.hu/docs/index.html. [access date: Jun. 2, 2018].

[4] Flib project github page. https://github.com/Frontier789/Flib/. [access date: Jun. 2, 2018].

[5] Graphviz homepage. https://www.graphviz.org/. [access date: Jun. 2, 2018].

[6] Microsoft automatic graph layout homepage. https://www.microsoft.com/en-us/research/project/microsoft-automatic-graph-layout/. [access date: Jun. 2, 2018].

[7] Qt — cross-platform software development for embedded and desktop. https://www.qt.io/. [access date: Jun. 2, 2018].

[8] Simple and Fast Multimedia Library. https://www.sfml-dev.org/. [access date: Jun. 2, 2018].

[9] Wolfram Mathematica homepage. https://www.wolfram.com/mathematica/. [access date: Jun. 2, 2018].

[10] Joe Armstrong. *Programming Erlang*. The Pragmatic Bookshelf, 2nd edition, October 2013.

[11] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máté Tejfel, and Melinda Tóth. RefactorErl, Source Code Analysis and Refactoring in Erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools*, Tallin, Estonia, 2011.

[12] Fabian Fagerholm. Simple Directmedia Layer (SDL). 2006.

[13] A. R. Forrest. Antialiasing in practice. In Rae A. Earnshaw, editor, *Fundamental Algorithms for Computer Graphics*, pages 113–134, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[14] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.

[15] Emden R. Gansner, Yehuda Koren, and Stephen North. Graph drawing by stress majorization. In János Pach, editor, *Graph Drawing*, pages 239–250, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[16] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, and Roland Király. Modeling Semantic Knowledge in Erlang for Refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009*, volume 54(2009) Sp. Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, July 2009.

[17] Jorge Jimenez, Diego Gutierrez, Jason Yang, Alexander Reshetov, Pete Demoreuille, Tobias Berghoff, Cedric Perthuis, Henry Yu, Morgan McGuire, Timothy Lottes, Hugh Malan, Emil Persson, Dmitry Andreev, and Tiago Sousa. Filtering approaches for real-time anti-aliasing. In *ACM SIGGRAPH 2011 Courses on*, page 6, 2011.

[18] Eleftherios E. Koutsofios and Stephen C. North. Drawing graphs with dot. *Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ*, 1991.

[19] Tancred Lindholm. N-body algorithms. http://www.cs.hut.fi/ ctl/NBody.pdf. [access date: Jun. 2, 2018].

[20] Håkan Mattsson, Hans Nilsson, and Claes Wikström. Mnesia - a distributed robust dbms for telecommunications applications. *practical aspects of declarative languages*, pages 152–163, 1999.

[21] Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. OpenGL® Shading Language. 2004.

[22] Melinda Tóth and István Bozó. Static Analysis of Complex Software Systems Implemented in Erlang. In *Central European Functional Programming School*, volume 7241 of *Lecture Notes in Computer Science*, pages 440–498. Springer, 2012.

[23] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In *Proceedings of the 7th conference on Visualization '96*, volume 25, pages 327–334, 1996.

ELTE, Eötvös Loránd University, Pázmány Péter sétány 1/C, Budapest 1117, Hungary

Email address: {makom789, bozoistvan, tothmelinda}@caesar.elte.hu

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 2, 2018
DOI: 10.24193/subbi.2018.2.03

APPLICATION FUNCTIONS PLACEMENT OPTIMIZATION IN A MOBILE DISTRIBUTED CLOUD ENVIRONMENT

ANNA REALE, PÉTER KISS, CHARLES FERRARI, BENEDEK KOVÁCS,
LÁSZLÓ SZILÁGYI, AND MELINDA TÓTH

Abstract. Distributed Computing in 5G Mobile Networks is a potential requirement for certain applications that depends on low latency and information sharing through or with data information sources. Such applications may be observed as a distributed application. We present a tool and method to optimize the deployment of distributed applications, dividing it into Modules, in a 5G Mobile Network environment. To do so we apply an approximation algorithm for the Path Computation and Function Placement Problem described in [1]. We show that under certain circumstances it is beneficial to deploy parts of such applications in a Cloud Computing environment with Distributed Cloud resources at the Mobile Network Edge. We verify our findings with an example, an Augmented Reality application.

1. Introduction

5G mobile networks promise high bandwidth and low latency on the radio interface for both downlink and uplink data [2] which capability will enable new type of applications and services. Such mobile applications include Augmented Reality (AR), Virtual Reality, Gaming and many other bandwidth heavy and latency sensitive applications, potentially applied for critical use cases such as Intelligent Transportation Systems or Surveillance.

Deploying an application on a 5G network with distributed cloud capabilities involves the choice of were to allocate what parts of the applications. It depend both on the application itself and on the involved network. In this

Received by the editors: March 31, 2018.

2010 *Mathematics Subject Classification.* 68U20,68U35,90C35.

1998 *CR Categories and Descriptors.* C.4[**PERFORMANCE OF SYSTEMS**]: Modeling techniques– ; C.2.4[**COMPUTER-COMMUNICATION NETWORKS**]:Distributed Systems– *Distributed applications* .

Key words and phrases. 5G, Distributed Cloud,Distributed Computing, Application Partitioning, Edge Computing, Augmented Reality.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

work we propose a method and tool to facilitate planning of refactoring of an existing applications into modules. To do so we calculate the best placement of the modules on the network compute servers, given user profile and context conditions informations such as: typical user request, policy per type of user (SLA), available bandwidth, network node types, available computation power and cost of computation on the device, in the distributed edge and central cloud.

We have chosen a resource demanding AR application for our test. We assume that such applications can benefit both from involving low latency external computation power and significant sized, but affordable, storage capabilities. To validate our assumptions we apply the mentioned tool and measure the application properties under certain circumstances and network constraints.

Main contributions of this work is the proposal of a method to automatize application partitioning and placement in a 5G/Edge environment. We introduce a possible tool-set implementing our method, its experimental setup and evaluation.

Most works on this topic focuses only on the problem of task partitioning and placement, while they seldom addresses issues of multiple users and load balancing. For this purpose in our work we integrate an approach from network service placements and apply a variation of the approximation algorithm for the Path Computation and Function Placement Problem described in [1].

2. Background

In the following section we give an informal description of the problem we address and contextualize it by referring to related works.

2.1. **Problem Statement.** To partition an application and to deploy its modules in a 5G network with edge computing resources, we need to calculate what is the (sub)optimal grouping of the components and their placement that maximizes the usage of network capacities in a given instant. Giving a flexible method to automate this process enables applications to adapt to environment changes through dynamical reallocation of resources.

This task can be reduced in three main steps:

(1) Model the application through hybrid analysis (using both static analysis and heuristics from dynamic profiling of the given application);
(2) Calculate a partition to divide the application in modules minimizing their interactions and communication cost while maximizing the responsiveness and perceived performances;
(3) Decide best placements of the modules in the given network;

2.2. **Related Works.** State-of-the-art Application Partitioning Algorithms (APAs), applied to distributed processing, still face many issues and challenges. An extensive summary concerning APAs in Mobile Cloud Computing is proposed in [3]. Based on the model used as an input to the partition, the authors identify three main categories of existing solution: graph-based, Linear Programming (LP) or hybrid solutions between the two.

2.2.1. *Application Partitioning.* Solutions based on graph representations of the applications may use data flow graph to represent data dependencies between operations [4, 5, 6],

while class dependency graphs can be used to describe the structure of an application [7, 4].

The authors in[8] partition object-oriented programs by generating an Object Relation Graph (ORG) to estimate the runtime objects and their interactions, and then applying graph partitioning to this ORG. In [9] a two-layer graph structure is used, in which a second graph, the Target Graph (TG) accounts for the various target infrastructures and distribution objectives.

Graph-based APAs require efficient manual annotation techniques, it is up to the programmer to balance the metrics and specify metrics function. In addition, a greatresource overhead is generated in case of applications with a large number of components. Finally the performance of graph-based solutions depends on the application characteristics: the analysis is easily performed if the applications is already modularized somehow. On the other hand, LP based solutions always produce optimal results for a particular objective function [10, 11, 12]. LP APAs need dynamic scheduling techniques, extra profiling and resource monitoring, thus they also cause high overheads.

Hybrid solutions extract the important features of graph-based APAs and LP-based APAs in order to improve the performance and mitigate overheads but, in most cases, at the expenses of generating only a sub-optimal partition [13, 14, 6, 15, 16].

In this work we plan to create a tool to help to define a set of candidates for partitions according to different network conditions. Such partition database could be used in the future to allow dynamic reallocation of the application, based only on light dynamic profiling of the context it runs in.

2.2.2. *Application Modeling and graph partitioning.* The NP-hard graph partitioning problem is a fundamental issue in many other domains of computer science, such as parallel processing [17] and load balancing [18]. In grid computing the graph partitioning problem has been used to define parallel tasks

to be deployed on heterogeneous infrastructures. As stated by [16] many proposed algorithms, such as MiniMax, VHEM, QM, PaGrid, and MinEX, use a multilevel paradigm, while others use simulated annealing [19].

In the literature, decomposition techniques based on graphs [20] involve three macro steps: (1) Identify level of granularity for the partitions elements or tasks; (2) Analyze the application with task dependency and interaction graphs, (3) Map possible valid partitions.

Properties of tasks that affect the quality of mapping are: feasibility of task generation, size of tasks and size of data handled by the task or passed between two of them.

In fact, one needs to take in consideration the interaction between the partitioned task: they often share data and may have a precise sequential order [21, 22].

In scheduling the interaction graph is used to represent the application dividing it into tasks. Nodes in the graph are the tasks while their weights denote the amount of work to be performed by the task. Edges represent the interactions between tasks. Generally edges are undirected, when directed they are used to show the direction of the flow of data (if the flow is unidirectional). Weights on edges contain the cost of communication. Shared data may imply synchronization protocols (mutual exclusion, etc) to ensure consistency.

In distributed systems theory, the interaction graph is also referred as the Control Flow Graph (CFG). A CFG is a representation, using graph notation, of all paths that might be traversed through a program during its execution. The graph provides the structure of the program as a whole, among others, making explicit all of the paths that are induced by a conditional branch. A function dependency graph, for example, is a sub-graph of this graphs, having has partition granularity the function. Dependency between functions implies interaction (calls or data passing) between them.

A Call Graph (CG) is a dependency graph that represents calling relationships between functions in a computer program. Each node denotes a procedure and each $edge(f, g)$ indicates that procedure f calls g. Thus, a cycle in the graph indicatesrecursive calls.

Call graphs are results of a basic program analysis, that can be used for model programs, or as a basis for further analyses. Call graphs can be dynamic or static. A dynamic call graph is a record of an execution of the program, for example as output by a profiler. Thus, a dynamic call graph can be exact, but only describes one execution of the program.

In object-oriented languages the potential target method(s) of many calls cannot be precisely determined solely by an examination of the source code

[23]. Thus, to build the call graph, it is necessary to have inter-procedural data and control-flow analysis of the program.

Granularity. Program partitioning has been used in application offloading for resource-constrained devices. Previous works propose computation offloading at different levels of granularity: Module level [5], Method level [13], Object level [8, 9], Thread level [14, 6], Component level [10, 16]. Various metrics can help to decide a good level of granularity for the partition of a graph. For instance, the critical path, being the longest directed path between any start and finish nodes, indicates what is the shortest time needed to execute. The time can be calculated from it's length, computed by sum of the weights of the traversed nodes. The average degree of concurrency, that is the total amount of work divided by critical path length is also a common metric. Related to the size of the partitions we consider important the size of the data associated with tasks, because it helps to minimize volume of data-exchange and maximize data locality. Also the size of context is an indicator of how affordable or expensive the communication between tasks can be.

2.2.3. *Placement Models.* Appropriate resource allocation is a very old issue in different disciplines. In this section, we present two resource allocation problems in computer networks: placement of Virtual Machines (VM) in cloud computing and placement of Virtual Networks Function (VNF).

VM Placement. With the term VM placement we refer to the process of selecting the most appropriate physical machines for VMs. According to [24], objectives of VM placement are maximizing resource utilization, reliability and availability. There are several approaches to VM placement in the literature [25, 26, 27], some variants even consider dynamic placement and multi-clouds placement. For instance, [25] uses traffic-aware VM placement to improve the network scalability in data center, defining it as an hard optimization problem solved by a two-tier approximation algorithm to overcome very large sizes.

Service Chain Placements in NFV. Service Function Chaining (SFC) [28] aims to overcome the limitation of static deployment models applying algorithms that can optimally map SFC to substrate network. This category of algorithms is referred as "Virtual Network Functions Placement (VNFP)" algorithms [29]. As explained in [30], in this category of placement problems, we are given a physical network, VNF specifications, and a set of service requests. The algorithm performs the three following steps:

(1) Calculate an optimal number of needed VNF types, all the VNFs that should be instantiated compose a set.
(2) Place VNFs to physical nodes such that the demand of VNFs do not exceed the capacity of physical nodes;

(3) Assign service requests to VNFs such that the demand of service requests do not exceed the capacity of VNFs.

However, the three steps are not independent, and their order depends from the implementation of the algorithm and the problem statement. For example, [31, 32] give an Integer Linear Programming(ILP) formulation; many others, preferring fast heuristics to allow real time decisions [33], propose a dynamic programming; [34] gives a Mixed ILP formulation and a heuristic algorithm that solve the problem incrementally, which can solve the problem for incoming flows without impacting existing flows. Among the meta-heuristic solutions, [35] proposes a method based on genetic algorithms while [36] considers a greedy algorithm and a tabu search-based algorithm.

Although in our example we will not work with VNF specific algorithms, we claim that our methods may be applied to them as well. This is especially true for network functions such as User Plane Function and special observability, monitoring, tracing, logging and analytics VNFs.

3. A model for application partitioning and deployment

In the following sections we provide formal description of the models and methods used to construct our simulation toolset, followed by a description of the real application we used as first input for it.

3.1. Models.

To map an application based on functions granularity we construct a function dependency graph. In scheduling and load balancing, this method is used when the application can be described from the static definition of the dependency graph and the function sizes are known.

Determining an optimal mapping of the function dependency graph becomes solvable if there are good heuristics available to estimate the data flow and a structured call graph. In our case we use a static analyzer tool to generate the function call graph from the source code. Then we run the application and collect for each function, using a non-intrusive dynamic profiler, the percentage of runtime spent in it. In addition, for each link between two functions, we collect the number of times the callers calls the callee. We normalize those results and store them in the call graph as node and edge weights. The normalized edge weights will define the dependency between the two connected functions, thus to estimate how to separate the application to reduce such interactions it will be enough to use a minimum edge-cut strategy. The node weight is a useful information to estimate the complexity of the computations handled by the function, this value can be used to balance the partitions or to deploy different optimization strategies. For example if we want for the User Equipment (UE) accessing the application to save energy, we could want to

concentrate the computational load on the Edge or on the Cloud the UE can connect to.

It is important to stress how we are deploying a context-insensitive construction of the application call graph. In fact, each node will represent exactly a single contour: an analysis-time representation of a function.

3.2. **Application Partitions.** Having our weighted graph, we partition it using Multilevel [37] version of the Kernighan - Lin [38] algorithm (MLKL). We choose the multilevel strategy to be able to handle potentially large function call graphs. After running the algorithm, the application will be divided into a number of Modules where the ceiling for the number of partitions can be selected by the user, and the weight of each Module and the interaction frequencies between them are derived from the original graph. The directed graph resulting as an output of the partition steps represents the due interactions between the modules. This new level of abstraction means that we lose information like when and for how long two specific functions in two modules will interact at running time. Such information also depends on the user interaction with the application itself and can vary from instance to instance.

We decided to adopt a pessimistic approach in the module deployment phase, taking as the weight of the assumption making that we want to instantaneously run all the modules.

3.2.1. *MLKL and METIS.* MLKL is a Multilevel Version of the KL algorithm. It means that the algorithm is applied in three repeated phases: Coarsen, Partition and Uncoarsen.

First, the algorithm coarsen down the graph by merging connected vertices until a small graph is obtained. Then this graph is partitioned and uncoarsened again, while optimizing the partition in each uncoarsening step using KL as refinement function.

The KL algorithm is iterative. It starts with an initial partition and in each iteration it finds two subsets which guarantee a smaller edge-cut. If such subsets exist, then it moves them to the other part and this becomes the partition for the next iteration. The algorithm continues by repeating the entire process. In the implementation proposed by [39] the KL algorithm computes for each vertex v a quantity called gain which is the decrease (or increase) in the edge-cut if v is moved to the other part. The algorithm terminates when the edge-cut does not decrease after x number of vertex moves and those last moves are undone to get the maximum edge cut.

3.3. **Network Model.** Now we test our partition behavior in our network to see what configuration gets the maximum out of the same network conditions.

Thus we deploy all tasks in all nodes and see what are used the most to satisfy network demand considering network capacities.

The network is a fixed set of computational resources and communication links. The network is represented by a graph $N = (V, E)$, where V is the set of nodes and E is the set of edges.

We classify nodes in three categories: UE, Edge Cloud Servers and Central Cloud Servers. Note that the classes are disjoint and our proposed method works with other types of disjoint classification of nodes as well.

Nodes and edges have capacities. The capacity of an edge $e \in E$ is denoted by $c(e)$, and the capacity of a node $v \in V$ is denoted by $c(v)$. All capacities are positive integers. $c(e)$ represents the available bandwidth between the two network nodes; $c(v)$ depends on the amount of available computational resources and the cost of accessing them. We suppose several UEs

that request services from the application. Each of these services may be different on the *Service type* and the *Location* of the involved nodes. Examples of such services can be a video upstream or augmented downlink video. Each Module is a part of the application that, combined, can solve a certain service request.

3.4. **Service Request.** A service request for user j is specified by a tuple $s_j = (G_j, d_j, b_j, U_j)$, where the components are as follows:

$G_j = (M_j, Y_j)$ is a directed (acyclic) graph called the place-and-route graph (pr-graph). There is a single source and a single sink, that corresponds to the node requesting the service. We denote the source and sink nodes in G_j by $ns_j \in M_j$ and $nt_j \in M_j$, respectively. The other vertices correspond to services or processing stages of a request. The edges of the pr-graph are directed and indicate precedence relations between pr-vertices.

The demand of a request s_j is d_j and its benefit is b_j. Demand is computed from the cost of running a complete module. The benefit is the benefit of serving that precise request of service. It should be calculated from the SLA, but it depends on the network owner as well. By scaling, we may assume that $min_j\{b_j\} = 1$.

We map the User Equipment service request s_j as the realization of a path trough the directed partition graph representing the application. In this case the demand of a Module can be calculated over the cost of each function composing the Module that composes the specific service request. The routing cost from one Module to the other become than the overhead or transmission cost brought by the selected Module interaction scheme. For example, the size of the data to be transferred from one Virtual Machine to the other to keep the state consistent trough all their network instances [40]. The impact of the service request on the network thus can vary only based on the location of

the Modules. To specify the possible realization of a pr-graph in the physical network we use a function $U_j : M_j \cup Y_j \to 2^V \cup 2^E$ where $U_j(m)$ is a set of "allowed" nodes in N that can perform module m, and $U_j(y)$ is a set of "allowed" edges of N that can implement the precedences and routing requirement that corresponds to y. We now define for each service request s_j the product network $pn(N, s_j)$. The node set of $pn(N, s_j)$, denoted by V_j, is defined as $V_j \triangleq \cup_{y \in Y_j}(U_j(y) \times y)$. We refer to the subset $U_j(y) \times y$ as the y-layer in the product graph. The edge set of $pn(N, sj)$, denoted E_j, consists of two types of edges $E_j = E_{j,1} \cup E_{j,2}$ defined as follows:

(1) Routing edges connect vertices in the same layer, they represent the physical links in the network.
$E_{j,1} = \{((u, y), (v, y)) \mid y \in Y_j, (u, v) \in U_j(y)\}$
(2) Processing edges connect two copies of the same network vertex in different layers, representing the move from one Module to the consecutive one in the service chain specified in Y.
$E_{j,2} = \{((v, y), (v, y')) \mid y \neq y' \in Y_j$ edges with common endpoint m, and $v \in U_j(m)\}$

PCPF problem. The substrate network $N = (V, E)$ and a set of service requests $\{s_i\}_{i \in I}$ described as stated before, are the necessary input for the solution we used for Path Computation and Function Placement Problem (PCFP). The goal is to compute valid realizations $\tilde{P} = \{\tilde{p}_i\}_{i \in I'}$ for a subset of the requests $I' \subseteq I$ so that $\tilde{P}$ satisfies the capacity constraint of N and maximize the total benefit $\sum_{i \in I'} b_i$. For our work, we apply the fractional relaxation of PCFP-problem described in [1]. This is a variation of Raghavan's randomized rounding algorithm for general packing problems [41].

3.4.1. *Experiment Setup.* We created a generic setup for Multi Access Edge Computing partitioning and distribution. It is composed by four resource constrained devices connected with an edge server through redundant networks, where different network setups can be tried. The application has initially all the processing activities done in the server, which collects information from the four connecting devices and performs the processing.

The connections used for the experiment explained in this article were carried with wireless 5 GHz and Ethernet connections, where the client devices were equipped with 100 megabits network shields.

The client devices were equipped with cameras using Sony IMX219 sensors, streaming real-time video to the server. The camera was configured to create frames of 640x480 pixels, 25 frames per second and 4:3 aspect ratio. The connection between the clients and the server was an UDP connection.

3.4.2. *Measurement Tools.* The measurements used to configure the tool are grouped in three independent areas namely: (1) Network performance; (2) Computational performance, and (3) Software processing cost. Each of them composed as described below:

- *Network Performance*: available resources, Jitter between nodes (Latency variation), Locality UE-Edge-Cloud;
- *Computational Performance*: Machine capabilities, Network connection speed, Processor capabilities, Memory availability;
- *Software processing cost*: Dependency between two functions (number of calls), Resource usage from App (Average memory Usage Mb per function), Cost of the software execution (processor cycles that are required to execute each function of the software).

The software measurements were taken using instrumented profiling tools, Valgrind [42] and our self-produced tools.

3.5. **Modeling the example application.** During our experiment, we chose to start at a function level granularity for our applications, to be able to partition it into Modules. A typical AR application has the following chain of services: *capture, preprocessing, detection, recognition, tracking, rendering.* Each of this service calls a sequence of *Modules*. Note that these *Modules* may be different for different applications. Another example can be a partitioning of a Linear Unicast service which may have the following modules: Streaming, Origination, Manipulation, Encapsulation, Encryption, Encoding, according to [43].

In our first example (Figure 1) we show the result of running the partitioning only on the call graph of the *capture* service (involving camera calibration), where different colors refers to different modules and the number of requested partitions was 5. As second example (Figure 2), we show the Function Call graph generated only by the camera calibration part of the application 2 on the edges the calls between functions and on the nodes the CPU clocks.

The result of the whole AR application partition is shown in Figure 3a. The $Start$ node represents the interface with the User Equipment, the $Main$ node is the partition in which the known entry point of the program execution is located.

The arrows are the interaction between $Modules$. For example, we know that $Main$ can receive data and be called by $M1$, but every call from the $Main$ goes either to $M2$ or to $M4$. In the construction of service requests we kept the following interaction constraints: if the service needed by UE is contained in $M1$ the shortest possible request path became $\{(Start, Main)(Main, M2)$ $(M2, M1)(M1, Main)\}$.

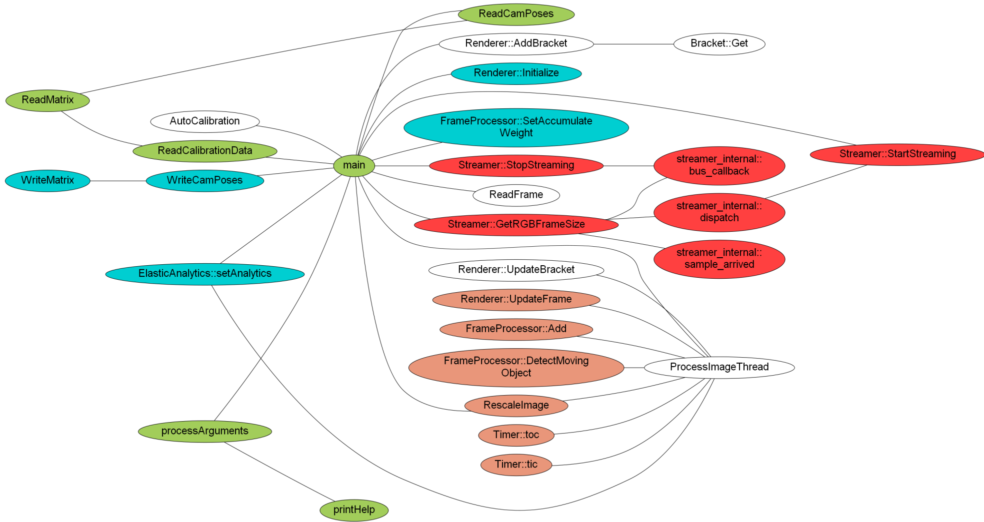


FIGURE 1. Partitioned Call Graph for an Image Capture Service

The Simulation Network will represent the possible interactions between nodes. In our simulation, we decided to allow both direct UE to Edge and UE to Cloud communications (Figure 3b). We consider an average transmission overhead in the range of few ms (1 or 2) between UE and Edge nodes, of 25 ms betweens Edge and Local Clouds and of the sum of the two (26 or 27) between UE and Cloud. The resulting pr-graph

is shown in Figure 3c. We normalize the capacities of the network nodes based on available memory. We experimented on a SLA scenario where we want to reduce the computation time at a minimum overhead.

For the same computation demand we define the benefit of a chosen deployment path based on the computation cost (we estimated the Edge to be four times more expensive than the cloud) and the average transmission overhead. Both weights were calculated as the coefficient of variation of the relative measures registered on the Experiment Setup.

In all the generated simulations a deployment was proposed for which 12 contemporary simulated user requests where served, respecting the capacity constraints of different networks, obtaining maximal benefit flows like the one shown in Figure 3d. On average, the benefit was higher than running everything on the device: a complete run on the single device lasted on average 9444 ms, while the average run on our simulations saved from 667 ms up to 3904 ms with maximum average communication overhead per request being 1152 ms (Table 1).

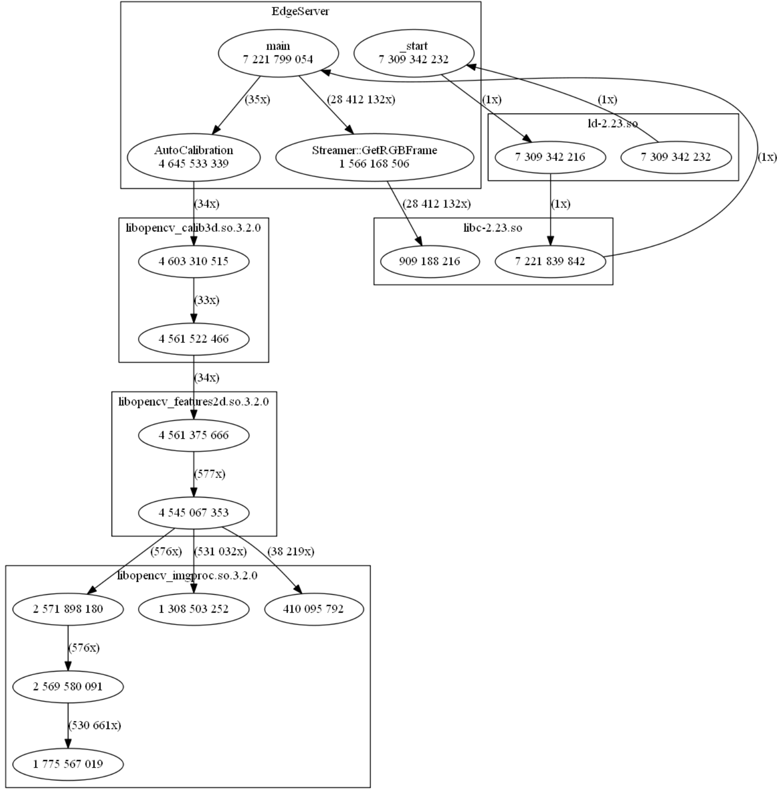


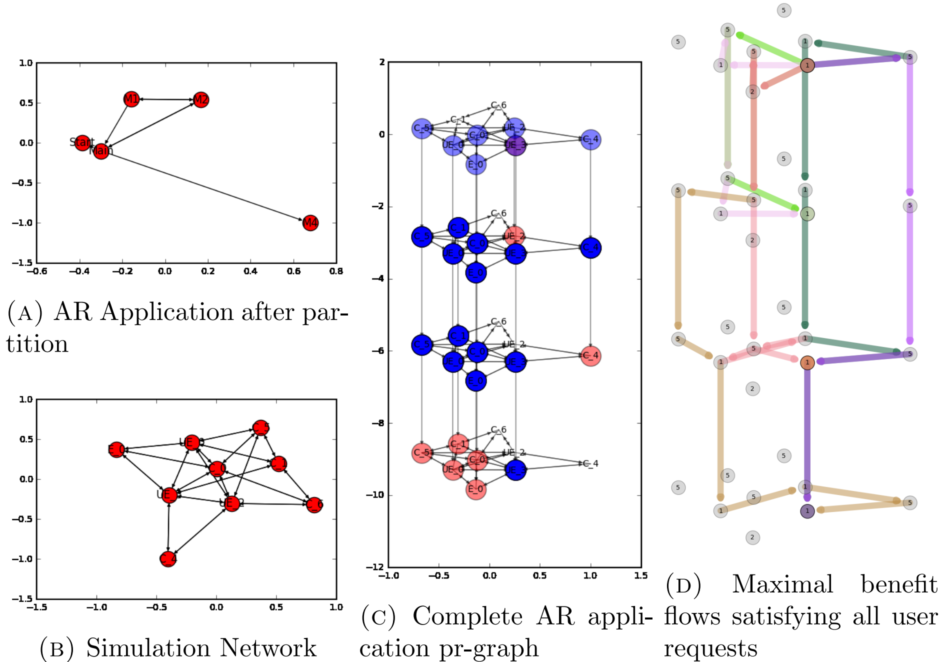
FIGURE 2. Camera calibration call graph

	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8
Edges	4	2	5	5	5	3	4	4
UEs	3	6	3	3	3	5	4	4
Local Cloud	1	1	1	1	1	1	1	1
Average overhead per request (ms)	37	58	144	29	1152	583	84	148
Average benefit per request (ms)	3941	1348	3743	3348	1820	2841	3340	2395
Average final benefit (ms)	3904	1289	3598	3319	667	2258	3255	2246

TABLE 1. Experiment result: benefits of partitioning and deployment of the same application on different networks topologies

4. Conclusions and Future Work

In this work we described the methods and the algorithms we used to develop a first prototype of our tool to partition and deploy an application in a



(A) AR Application after partition

(B) Simulation Network

(C) Complete AR application pr-graph

(D) Maximal benefit flows satisfying all user requests

FIGURE 3. Models of a real application

5G distributed network. We believe this is the minimum analysis to be performed to, in the future, be able to implement a dynamic reallocation of the applications based on variation of the context conditions. The problem was divided in three steps. First selecting the application granularity and construct a graph model. Than reduce it into Modules by solving the NP-hard graph partitioning problem it represents; finally implement and apply a fractional relaxation of the Path Computation and Function Placement Problem as described by [1].

Simulation were run with various simultaneous request of service. For our specific set up and our AR application, there is a possibility to implement a distributed scenario with a reasonably low overhead.

The next step would be to implement the new application partition suggested by the framework and locate them in the physical network to verify how close our simulations are to reality. We hope by running the new deployment to be able to perfect the parameters we used to describe the network capacities and the benefits of the distributed execution. Interesting measures to validate the outcome on different AR applications could be quality and

efficiency related measures: for example Video Quality as Average Bit rate expressed in Kbps.

5. Acknowledgements

This research has been co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013). We would like to thanks our university supervisors and the 5G Ericsson Garage team: Gábor Fábián, Zoltán Gera.

References

[1] G. Even, M. Rost, and S. Schmid, "An approximation algorithm for path computation and function placement in sdns," in *International Colloquium on Structural Information and Communication Complexity*, pp. 374–390, Springer, 2016.

[2] E. AB, "5g systems, enabling the transformation of industry and society," white paper, ERICSSON, 2017.

[3] J. Liu, E. Ahmed, M. Shiraz, A. Gani, R. Buyya, and A. Qureshi, "Application partitioning algorithms in mobile cloud computing: Taxonomy, review and future directions," *Journal of Network and Computer Applications*, vol. 48, pp. 99–117, 2015.

[4] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, "Calling the cloud: enabling mobile phones as interfaces to cloud applications," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, p. 5, Springer-Verlag New York, Inc., 2009.

[5] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A framework for partitioning and execution of data stream applications in mobile cloud computing," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 23–32, 2013.

[6] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden, "Wishbone: Profile-based partitioning for sensornet applications.," in *NSDI*, vol. 9, pp. 395–408, 2009.

[7] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic, "Adaptive offloading inference for delivering applications in pervasive computing environments," in *Pervasive Computing and Communications, 2003.(PerCom 2003). Proceedings of the First IEEE International Conference on*, pp. 107–114, IEEE, 2003.

[8] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. J. Giuli, and X. Gu, "Towards a distributed platform for resource-constrained devices," in *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pp. 43–51, IEEE, 2002.

[9] L. Wang and M. Franz, "Automatic partitioning of object-oriented programs for resource-constrained mobile devices with multiple distribution objectives," in *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on*, pp. 369–376, IEEE, 2008.

[10] L. Yang, J. Cao, and H. Cheng, "Resource constrained multi-user computation partitioning for interactive mobile cloud applications," *Technical reort. Department of Computing, Hong Kong Polytechnical University*, 2012.

[11] D. Kovachev, "Framework for computation offloading in mobile cloud computing," *IJIMAI*, vol. 1, no. 7, pp. 6–15, 2012.

[12] M.-R. Ra, B. Priyantha, A. Kansal, and J. Liu, "Improving energy efficiency of personal sensing applications with heterogeneous multi-processors," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pp. 1–10, ACM, 2012.

[13] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pp. 49–62, ACM, 2010.

[14] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*, pp. 301–314, ACM, 2011.

[15] M. Goraczko, J. Liu, D. Lymberopoulos, S. Matic, B. Priyantha, and F. Zhao, "Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems," in *Proceedings of the 45th annual design automation conference*, pp. 191–196, ACM, 2008.

[16] T. Verbelen, T. Stevens, F. De Turck, and B. Dhoedt, "Graph partitioning algorithms for optimizing software deployment in mobile cloud computing," *Future Generation Computer Systems*, vol. 29, no. 2, pp. 451–459, 2013.

[17] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel computing*, vol. 26, no. 12, pp. 1519–1534, 2000.

[18] H. Meyerhenke, B. Monien, and S. Schamberger, "Graph partitioning and disturbed diffusion," *Parallel Computing*, vol. 35, no. 10-11, pp. 544–569, 2009.

[19] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[20] F. Berman, "High-performance schedulers," *The grid: blueprint for a new computing infrastructure*, vol. 67, pp. 279–309, 1999.

[21] D. L. Long and L. A. Clarke, "Task interaction graphs for concurrency analysis," in *Proceedings of the 11th international conference on Software engineering*, pp. 44–52, ACM, 1989.

[22] M. Naghibzadeh, "Modeling workflow of tasks and task interaction graphs to schedule on the cloud," *CLOUD COMPUTING 2016*, p. 81, 2016.

[23] D. Grove and C. Chambers, "Ibm research report an assessment of call graph construction algorithms," 06 2000.

[24] M. Masdari, S. S. Nabavi, and V. Ahmadi, "An overview of virtual machine placement schemes in cloud computing," *Journal of Network and Computer Applications*, vol. 66, pp. 106–127, 2016.

[25] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *INFOCOM, 2010 Proceedings IEEE*, pp. 1–9, IEEE, 2010.

[26] R. A. da Silva and N. L. da Fonseca, "Algorithm for the placement of groups of virtual machines in data centers," in *Communications (ICC), 2015 IEEE International Conference on*, pp. 6080–6085, IEEE, 2015.

[27] J. Chase, R. Kaewpuang, W. Yonggang, and D. Niyato, "Joint virtual machine and bandwidth allocation in software defined network (sdn) and cloud computing environments," in *Communications (ICC), 2014 IEEE International Conference on*, pp. 2969–2974, IEEE, 2014.

[28] J. Halpern and C. Pignataro, "Service function chaining (sfc) architecture," tech. rep., 2015.

[29] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," in *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*, pp. 171–177, IEEE, 2015.

[30] Y. Xie, Z. Liu, S. Wang, and Y. Wang, "Service function chaining resource allocation: A survey," *arXiv preprint arXiv:1608.00095*, 2016.

[31] T. Taleb, M. Bagaa, and A. Ksentini, "User mobility-aware virtual network function placement for virtual 5g network infrastructure," in *Communications (ICC), 2015 IEEE International Conference on*, pp. 3879–3884, IEEE, 2015.
[32] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pp. 1346–1354, IEEE, 2015.
[33] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "On orchestrating virtual network functions," in *Network and Service Management (CNSM), 2015 11th International Conference on*, pp. 50–56, IEEE, 2015.
[34] A. Mohammadkhan, S. Ghapani, G. Liu, W. Zhang, K. Ramakrishnan, and T. Wood, "Virtual function placement and traffic steering in flexible and dynamic software defined networks," in *Local and Metropolitan Area Networks (LANMAN), 2015 IEEE International Workshop on*, pp. 1–6, IEEE, 2015.
[35] M. Bouet, J. Leguay, and V. Conan, "Cost-based placement of virtualized deep packet inspection functions in sdn," in *Military Communications Conference, MILCOM 2013-2013 IEEE*, pp. 992–997, IEEE, 2013.
[36] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and S. Davy, "Design and evaluation of algorithms for mapping and scheduling of virtual network functions," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pp. 1–9, IEEE, 2015.
[37] B. Hendrickson and R. W. Leland, "A multi-level algorithm for partitioning graphs.," *SC*, vol. 95, no. 28, pp. 1–14, 1995.
[38] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations research*, vol. 21, no. 2, pp. 498–516, 1973.
[39] G. Karypis and V. Kumar, "Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0," tech. rep., 1995.
[40] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan, "Just-in-time provisioning for cyber foraging," in *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pp. 153–166, ACM, 2013.
[41] P. Raghavan, *Randomized rounding and discrete ham-sandwich theorems: provably good algorithms for routing and packing problems*. University of California. Computer Science Division, 1986.
[42] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, pp. 89–100, ACM, 2007.
[43] W. Scott, "Content delivery networks (cdn): Caching principles, architecture, and resource optimization." `https://www.slideshare.net/hacktivism/cisco-live-content-delivery-networks-cdn`, 2017. Online; accessed 29-March-2018.

Faculty of Informatics, ELTE university, Budapest, Pázmány Péter stny. 1/C., 1117 Hungary

Ericsson Hungary Research and Development Center, Budapest, Magyar tudósok körútja 11, 1117 Hungary

Email address: {anna.reale, axx6v4, svu938, toth_m}@inf.elte.hu
Email address: {benedek.kovacs,laszlo.szilagyi}@ericsson.com

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 2, 2018
DOI: 10.24193/subbi.2018.2.04

INSTRUMENTATION OF C++ PROGRAMS USING AUTOMATIC SOURCE CODE TRANSFORMATIONS

ZSOLT PARRAGI AND ZOLTÁN PORKOLÁB

Abstract. The main tool for programmers is always the compiler, but there are also many other tools to help the development process. Some of these tools work on the source code of the program, analyzing, measuring or transforming it. Implementing a source based tool is a complex task, especially for complex languages such as C++. In recent years the C++ language received an easy-to-use library for developing such software, in the form of clang tooling. However, this library only focuses on processing a single translational unit of the program, independently to the other parts of the build process. Tools which ignore this big picture could result in failures when used on larger projects, or incorrect runtime behavior. In this paper, we describe some of these challenges encountered in real-world C++ projects and propose possible solutions for future tools to fix or mitigate the issues.

1. Introduction

There are tools which work on an already built binary, by intercepting calls (such as strace[13]), running the code on a virtual machine (such as valgrind[14]), or by transforming the binary before (such as syzygy[6]) or during (such as orbit profiler[12]) the execution of the program. There are tools which work within the compiler, using transformations on the intermediate language in it - for example, sanitizers[10] in the compilers are usually implemented this way. There are also tools which work by analyzing, and possibly modifying the source code. For example, static analyzers[11] work by performing more detailed checks on the source code, even providing automatic correction options for some cases.

Received by the editors: March 31, 2018.

2010 *Mathematics Subject Classification.* 68N15.

1998 *CR Categories and Descriptors.* D.3.3 [**Software**]: PROGRAMMING LANGUAGES – *Languages Constructs and Features*.

Key words and phrases. C++ programming language, source code transformation, instrumentation.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

All of these have their disadvantages.

When a tool works on an already existing binary, it lacks information. Debug symbols can be generated for any type of builds, but optimizations, such as inlining limit the options available for tools even then. This can be countered by running the tool on binaries compiled with special flags and providing an API that programs can use to share information.

When a tool is integrated into the compiler, it is only available with that compiler – and as these tools are often under active development, possibly even limited to recent compiler versions, limits their use especially for software targeting several compilers, operating systems or platforms.

When a tool works on the source code, it is limited to the capabilities of the language, and it is subject to the differences between the compilers and the complexity of interpreting the source code. These disadvantages are especially crucial in the case of C++: While the C++ standard[7] is detailed, it leaves choices to the compiler, and there are several examples for the most used compilers providing different results for simple looking C++ programs - sometimes even diverging from the standard. Interpreting the language is also challenging because of the preprocessor: as C/C++ programs tend to use many different configuration options[1], there is not a single AST to be analyzed and modified.

Most tools could be implemented using different techniques, and there are examples for implementing the same tool in different ways: Code coverage can be measured with in-compiler instrumentation (the `-fcoverage` option of clang), with transforming the source code (Coco[**Bullseye**]), or with a tool working on a special binary (gcov[5]). On the other hand, it is entirely possible that a tool can not be implemented in all three ways: Uninitialized memory reads can be detected by a binary tool (valgrind) or an in-compiler instrumentation tool (memory sanitizer in clang), but implementing it with a source transformation is not possible within the limits of the language.

In this article, we focus on the problems and possible solutions when implementing AST level source code instrumentation tools based on clang tooling. Source-based tools were chosen because of their generality: tools working on the binary or as part of the compiler are limited to the platforms where the tool runtime or the compiler is supported. This is often a limiting factor even on desktop systems - several tools, such as valgrind, or the clang sanitizers only work on Linux-like systems. On other, especially embedded systems, the problem is even more significant: these targets often have custom compilers, making compiler based techniques unusable, and possibly limited or no support for running external runtime tools along the main program.

Our focus is how these transformations can be integrated into and performed on large-scale projects. We discuss questions like how a source code transformation tool can be included in the build process, or how the amount of available configurations increases the complexity or possible problems[16][15].

It is also important to mention that source code transformations could easily result in behavior changes of the program[4], and ensuring that these are not happening is a similarly important aspect of tool development. In some cases, this is impossible. In this situation, it is important to minimize and document these - as in the case of the previous Coco example, which results in behavior changes with specific operators. While we mention that this is an issue, further analysis of the question is out of scope for of article.

2. Transformation overview

A C++ program is built by transforming every C++ source file separately into an object file, then linking those object files, and dependencies together into a library or executable. This process is also layered: the dependencies used by the linking step are built similarly, but often provided only in the final, binary form. Larger projects usually consist of multiple components, each built this way, depending on each other. Based on this, we can split dependencies into two categories: internal, which are built by the project, and external, which are expected to be found in a compatible binary format.

The first issue with program instrumentation is handling the dependencies: when the instrumentation changes the build process – as in the case of in-compiler or source based instrumentations –, it is possible that changes have to be made in the dependencies. An extreme example for this is the memory sanitizer[10]: it requires every dependency, including the C++ standard library, to be built using the memory sanitizer.

This is, even more, an issue when using source transformation tools: in this case, the task is not only the addition of some compiler flags into the build process of the dependency, but the actual execution of another tool during its build. It is also important to note that some dependencies are only provided in a binary form, making transformations in them impossible.

While the tool itself can not make the task of building everything in the necessary way more manageable, the problem can be mitigated by limiting what parts of the software transform. In case of the memory sanitizer example, there are no better choices because the way it is designed, but most software should be implemented in a way that would allow at least limited usage without rebuilding everything. To achieve this, tools either need a way to decide which files they can safely change, or they should not rely on any change that would change the "interface" of a file.

Clang tools generally use a compilation database for executing the tool: a JSON file containing every compilation command with all of their parameters. This file can be generated by commonly used C++ build systems, and then the tool can look up the specific compilation parameters from it. This process is executed as follows:

(1) Configure the project. For some tools, like CMake[8], this step also generates the compilation database.
(2) Build the project. For some tools, like bear[9] with make, this is the step that generates the compilation database. If it was generated by the compilation step, and the project uses no generated source files, this step could be skipped.
(3) Run the tool on some or all of the source files.

This process works perfectly with read-only tools, that do not change the source code. It is also suitable for some transformation tools, by repeating the second step once more after the transformation and compiling the modified program. Unfortunately, this approach leads to issues in some special cases.

Focusing on a single component of the build process, C++ sources include other files, handled by the preprocessor. A header file can be, and often will be included by more than one source file. While this usually does not result in any issues, it is possible that different source files include the headers in the context of varying preprocessor definitions. It is also possible that a source file includes a header multiple times with a different preprocessor definition context, or simply a build can include a source file multiple time with different compilation flags, providing a different name to the resulting object file.

These all could cause problems when changing the source code:

```
#ifdef SOMETIHNG_DEFINED
int foo(SOMETHING_DEFINED a);
#else
void foo();
#endif
```

With a simple transformation approach, it is possible that this file is transformed twice:

- first, when included with the definition set, only the first part is transformed
- after that, when included with the definition not set, only the second part is transformed

With the transformation process implemented the previously described way, depending on the exact sources, this could cause compile or runtime errors: As the process performs in-place transformations, the execution of the second compilation would work on the source file already transformed by the first execution of the tool. This is often the desired behavior: if the tools result

is permanent – such as when using automatic refactoring tools –, in the ideal outcome the transformation should include every sub-transformation required by any used configuration, and further runs of the tools should not result in additional changes. In this case, if the results conflict, the developer could be expected to look at them and fix the remaining issues manually.

With automatic, temporary transformations, however, user interactions during the build should be avoided, but keeping the same number of source files as initially is not a requirement: different translation units could use different versions of the sources, as long as these provide the same result a correctly implemented single file would. As the changes presented by the preprocessor definitions are limited to the current unit, this statement will hold. Based on this, the previous process can be generalized as follows:

(1) Configure the project.
(2) For every compiler invocation in the source code:
    (a) Invoke the transformation tool with the same parameters as the compiler, providing an out of place transformation in a unique temporary directory: every input file used by the compilation process should be written to a different location
    (b) Invoke the original compiler command, on the modified files
    (c) (Optional) Remove the temporary files

This change in the execution of the tool solves most of the mentioned issues: by doing an out-of-place transformation, always based on the original source codes, different transformation processes will not be based on the previous outputs – the tool will not accidentally transform the same source location multiple times. And by invoking the tool just before the original compilation program, we prevent accidental overrides: each compilation will be executed immediately after the required source codes are transformed. Finally, by requiring a unique temporary directory, we guarantee that parallel builds will not cause issues when the same file is used by multiple translational units transformed at the same time.

While these transformations increase the IO bandwidth required by the compilation commands, a memory file system could be used to avoid actual disk writes.

This approach also gives the advantage that it can be implemented as a wrapper around the compilation command. While the previous version required the generation of a compilation database and a separate run for the tool based on that database, the modified version does not need any change in the build script, except for a change in the compiler executable. This approach is used for example by the Coco coverage tool, which merely changes the system PATH seen by the build process.

The disadvantage is that this approach assumes that at least the transformation tool and the compiler can be run on the same platform. As our goal is constructing tools with clang tooling, which supports most Unix-like systems and windows, this is likely achievable.

We also have to note that with this simple modification, we did not solve the issue when a file is included multiple times, but differently within a single translational unit. Related issues are addressed later.

3. A note on compiler specific preprocessor definitions

The process we described is limited when the source code contains compiler-specific conditional blocks. While these are not common in high-level code, as the transformation process works on the entire source code, it will encounter these: they are commonly used in standard library implementations, and also in several widely used C++ libraries, such as boost[2].

In our experience, most transformations do not require changes in these parts of the code. If for some reason this is required, an AST transformation based tool can not be used. While it is possible to modify the predefined definitions for a clang tool, these conditions are not there without reasons in the source code. While it is possible that the only reason behind them is a compiler specific optimization or a compile-time optimization, the more likely reason is that other compilers can not understand the code within the condition.

An excellent example for this is the boost preprocessor library, which has numerous preprocessor conditions because of the slight differences between the preprocessors in different compilers. Trying to parse a different branch of that library other than what is designed for that compiler will likely result in errors during the early stages of compilation, and the inability of the compiler to produce a valid AST.

If such a macro is in the code base of the project the tool has to modify, and it is an uncommon case, the tool could get away be reporting a diagnostic, and provide developers the ability to manually resolve it. For transformations in third-party libraries and common occurrences, and when the tool has to guarantee that it will not miss any instrumentation, this is not an option.

As an example, the Coco[3] code coverage tool falls into this category: missing covered code would not be acceptable in a code coverage tool. On the other hand, coverage analysis also falls into the category where AST information is not required. While the tool is based on source transformations, it does so based on the token stream.

As it does not have to be able to construct a complete AST from the tokens, only to find the blocks and conditions in the code, it could change the transformation process to the following:

(1) Run the original compiler with an additional flag, which instructs it only to preprocess the source code, and output the result
(2) Instrument this preprocessed source code
(3) Run the original compiler using the modified sources

This process defers the preprocessing of the sources to the original compiler, to avoid any possibility of interpreting preprocessors definitions differently. The result should be a C++ source code, possibly referencing builtins specific to the used compilers. While these builtins will likely prevent another compiler from completely parsing and validating the source code, any compiler should be able to tokenize the result.

4. Mixing C and C++ code

Another issue is presented when C and C++ code is intermixed. While our goal is the instrumentation of C++ programs, C++ projects sometimes also contain C source files, which can not be built by a compiler in C++ mode.

If a C++ program also contains C source files, it adds additional questions when developing a tool. The first is, should the tool also have C support?

Some tools instrument code fragments that are only valid in C++ – in which case, they may safely ignore the question, as they will never have to modify a source code fragment which fails be parsed by a C compiler. Tools however often do not fall into this category.

Some tools could instrument C sources too, but the transformations done by the tool require a C++ compiler – it is possible that instrumenting C code in a similar manner is impossible, or will not be completely reliable. For example, RAII is unavailable in C sources, but it is possible for exceptions to pass through C code if it calls a C++ function.

Tools also have to be aware that header files may be shared between C and C++ source files. For this, the header file has to be compatible with both C and C++ compilation: Every C++ specific language has to be behind a conditional preprocessor directive. While the standard way to do this is the `__cplusplus` definition, some project uses their specific definitions provided by the build system.

The file also has to contain at least one global variable, or one function with extern C linkage when compiled with a C++ compiler, and these have to be in conditional sections which do not contain any not conditionally defined C++ symbols. Otherwise, even if the file is used by both a C and a C++ compiler,

the name mangling in C++ would ensure that the different language object files use different symbol names, preventing any possible issues.

The process described in the previous section ensures that the tool can not break the compilation of a header when it is used by a C or a C++ compiler: if a header is included in multiple translation units, it will be translated multiple times, differently. When used in a C compilation, the transformation process may either ignore it or could provide a C compatible transformation.

However, it also provides no information the way different headers are used: as it only wraps the compiler command and builds no external database about the files, it has no way to check if a source file is used with both languages. Compared to the previous example, where if a source code fragment was not disabled by a preprocessor definition, it was always transformed in the same way, in this case a program could end up with both a modified and an unmodified version of the source code, causing linking errors, or possible runtime problems.

One issue is caused by the linker: if an inline function has definitions in several object files, the linker will choose only one of them. In this scenario, when the C compiler does not instrument a function, but a C++ compiler does, the linker could choose any of the implementations.

```
// some_header.h
#ifdef __cplusplus__
extern "C" {
#endif
inline foo() {
  CPP_ONLY_INSTRUMENTATIN_MACRO;
  printf("bar\n");
}
#ifdef __cplusplus__
}
#endif
```

A possible workaround that during the compilation, the tool could convert global inline functions it has to modify to static functions. The issue with this approach is that with this change, the address of the function will be different in every object file, possibly changing the behavior of the program, if it depends on equality checks of the function addresses. This limitation has no possible automatic fix: While a wrapper function could guarantee that the same object address is used in every translational unit, it would also reintroduce the original issue in a more limited form. A tool also has no reliable way if the function pointer is used in a comparison. For this the best a transformation tool could do is to provide a diagnostic if it encounters the situation, and require the developer to solve it or silence the warning.

Another issue is presented by functions which only have declarations in the source file, and only affects tools that change function signatures: In this case,

it is possible that the tool correctly updates the signature, and changes the implementation of the function, which is in a C++ source file but does not update calls to the function in C code. Similarly, it is possible that the function is implemented in C, in which case the implementation will be unchanged, but the C++ callers will provide an additional parameter to it. As this is an extern C function, none of the above would result in linker errors – but both would result in runtime issues, where the exact results depend on the used calling conventions. While this is a more limited issue compared to the inline functions, it similarly has no automatic solution.

5. Dealing with conditional macro expansions

In the previous sections we discussed several issues presented by conditional preprocessor directives, but only in the context, that preprocessor directives will cause different parts of the source code to be compiled. Another issue presented by the preprocessor is macro expansion: when the transformation code has to modify a source code fragment which is at least partially is a result of the expansion of one or more preprocessor macro.

```
#define FACTORY_FUNCTION(T)       \
  T* create_or_return() {         \
    static T* instance = new T(); \
    return instance;              \
  }

// ...

INLINE_MACRO FACTORY_FUNCTION(TYPE_NAME_MACRO(foo,bar));
```

In a permanent transformation, the goal would be the transformation of the code behind the macro - so the code using it would remain the same, but the macros would expand to a different source. In an automatic tool, however, it could be easier to expand the macros to the actual source code generated by them, and then transform that source code. This could prevent several edge cases which could not be solved by the tool: for example, if the macro is defined in a header file, the tool can not be sure that every single of its use has to be instrumented.

Also, as every file is transformed uniquely by every translation unit, expansion will not cause issues even when a macro is defined differently for different compiler invocations. However it does not solve the previously mentioned issue, where one header, without an include guard, is included multiple times in a translational unit, but with differently expanding macros. In this case, the macros are expanded multiple times differently in the same file and could cause problems during the source code transformation.

As this can not happen when the `pragma once` compiler extension is used – that would prevent the second, different expansion –, we can provide a perfect workaround by cloning the file: when a transformation problem as described earlier is detected – a macro expansion is detected, but at a location where a macro was already expanded previously, but differently –, the file should be duplicated, and the include before, and after the conflicting should be changed to refer to the second file. The downside of this approach is that it assumes that the transformation tool has a detailed data structure about its transformations, and can perform this detection.

Alternatively, a simpler approach can be implemented in two separate phases:

1. The first hooks into the preprocessing phase of clang tooling
   - (a) It collects every source file used by the preprocessor into a list
   - (b) If it detects that a source file was already used, and it again emits non-whitespace tokens, it also marks the location where it happened
2. If it detected a multiply used source file, it creates multiple clones of that file and changes the invocing include directives. This can be implemented using a virtual in-memory filesystem in clang tooling, without writing anything to a file system handled by the operating system. After that, it starts over with using this virtual file system.
3. if it did not detect any source file used multiple times, it runs the second phase, which is the real tool as previously described, after the AST was parsed.

In this process, we can guarantee that the second rerun will not contain any source files used multiple times and that the preprocessor tool should not result in any noticeable change in the behavior of the program. The AST tool also requires no modifications, as if a macro in the source is expanded differently multiple times, it is hidden behind the preprocessor tool.

The disadvantage of this approach is that the tool will duplicate files even when it did not have to, as it can not tell if the tool will modify them. However this is a rarely used possibility in the language, and as such, will not result in any noticeable performance hit for most projects. In the detection step, it also assumes that the process only transforms actual C++ code, not macro definitions - it will not duplicate files which have no header guard, but only change preprocessor symbols.

After these modifications, the tool will be able to safely expand macros that have possibly different results but depend only on information defined by the project. However, macros depending on external information result in different issues:

- There are standard macros, which could be possibly changed by the tool, such as `__FILE__` or `__LINE__`.
- There are nonstandard macros, which in practice work the same way for every compiler, such as `__COUNTER__`
- There are macros which are often different for compilers, such as `__clang__` or `_MSC_VER`.

A transformation tool has to deal two possible situations with these: macros conditionally depending on these expressions – either directly, or indirectly, by a used macro –, or macros using these macros in expansions.

Some answers are clear in the previous list – for example, unless the tool can be sure that the real compiler will do the same steps as the clang tool, it should not expand a macro. In this case, the good answer is most likely only providing a diagnostic – based on that the developer or the tool developer may investigate the issue further, and possibly improve it. As an example, if the tool detects that the real compiler is GCC, and the condition is only based on `GNUC` macro, without depending on a clang specific macro, the expansion can be safely done. However, if the preprocessor did not take a previous condition only because while it allowed GCC, but disallowed clang, it is no longer expandable. This analysis requires a rather complex logic and understanding of different compiler internals. These decisions also require logging how the preprocessor evaluates conditions in the clang tool, making it realistic only for tools that often encounter these special cases.

Other decisions are not easy to decide: expanding macros such as `__COUNTER__` or `__PRETTY_FUNCTION__` could be perfectly safe even if the real compiler would interpret them somewhat differently. Another good example for this is the `__DATE__` macro, which is provided by every compiler, but it is evaluated differently every time.

Instead of expanding them, however, another approach is a more limited, and slower macro expansion: while the mentioned macros are often used in various C/C++ projects, it is unlikely that a tool has to transform the actual tokens generated by these macros. A more likely situation is that these macros are used in another macro - which also generates the source code which has to be transformed. In this case, instead of the full expansion of the source code fragment the tool has to transform, it could try to use a more restrained approach by only expanding macros one level at a time, and stopping as soon as the source locations where the transformation hat to take place are actual tokens.

This also means that if a transformation has to modify two locations – for example, wrapping a function call in another –, only the macros that contain

the location before and after the function call has to be expanded - macros within the function parameter list can stay unexpanded.

While this limitation does not solve the original issue, as there can be still situations the tool can not handle, it will greatly reduce their number. By implementing a similar approach, a tool has to emit fewer diagnostics about unexpandable locations.

6. Conclusion

In this article, we presented a methodology for developing C(++) source transformation tools, which is simple for developers to implement but also reduces the possible compilation or runtime issues caused by it. While we were not able to provide an automatic solution for every possible corner case, we provided workarounds to reduce the times the software has to provide diagnostic and/or require active interaction from the developer using it.

The methods we described are primarily intended for automatic AST based source code transformations, but most of the techniques we described could be adapted for other tools: for example, while most of our used methods are unsuitable for automatic refactoring, or permanent transformations, the described ideas could be used to implement the error detection and recovery capabilities of these tools in large projects.

We also have to note that while our results improve the usability and precision of these source transformation tools, it could be improved – both with the open questions in the mixed language projects and macro expansions. There are interesting improvement possibilities, such as logging the possible transformation conflicts during the build process, using an external code-database for finding how functions or headers are used or reconstructing actual macro expansions based on the differences between the preprocessor output of the actual compiler and clang.

We also only described the methods of these techniques without providing an actual library implementing these features: most of the methods described are generic and could be implemented in a generic reusable way for any tool, but current implementations only exist as a part of actual code instrumentation tools. The development of a ready to be used simple toolset would certainly reduce the cost of writing code transformation tools.

We only focused on the build process and the preprocessor, without mentioning the additional issues caused by accidental semantic changes in the program. With the capabilities of clang tooling, it would be possible to develop a framework which could validate that a given source change would not result in any unintended side effect - that is, apart from adding the additional

instrumentation, logging, or validation, it will not cause changes in the original program flow.

While code transformation and analysis is part of the development process for a long time, the increasing number of available tools for developing such software makes it an interesting research area and makes the development of usable and reliable tools easier.

References

[1] "An Empirical Analysis of C Preprocessor Use". In: *Software Engineering, IEEE Transactions on* 28 (Jan. 2003), pp. 1146–1170.

[2] boost. *Boost C++ libraries*. 2018. URL: https://www.boost.org/ (visited on 03/31/2018).

[3] FrogLogic. *Coco coverage analysis tool*. 2018. URL: http://www.froglogic.com/coco (visited on 03/31/2018).

[4] Alejandra Garrido and Ralph Johnson. "Challenges of Refactoring C Programs". In: *International Workshop on Principles of Software Evolution (IWPSE)* (Jan. 2002), pp. 6–14.

[5] GCC. *gcov - A test coverage plarform*. 2018. URL: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html (visited on 03/31/2018).

[6] Google. *Syzygy Transformation Toolchain*. 2018. URL: http://github.com/google/syzygy/ (visited on 03/31/2018).

[7] ISO. "ISO/IEC 14882:2014 Information technology — Programming languages — C++". In: Geneva, Switzerland: International Organization for Standardization, 2014.

[8] Kitware. *CMake*. 2018. URL: https://cmake.org (visited on 03/31/2018).

[9] Nagy Laszlo. *Build EAR*. 2018. URL: http://github.com/riszotto/Bear/ (visited on 03/31/2018).

[10] LLVM. *Clang Memory Sanitizer*. 2018. URL: https://clang-analyzer.llvm.org/ (visited on 03/31/2018).

[11] LLVM. *Clang Static Analyzer*. 2018. URL: https://clang-analyzer.llvm.org/ (visited on 03/31/2018).

[12] pierricgimmig. *Orbit Profiler*. 2018. URL: http://github.com/pierricgimmig/orbitprofiler/ (visited on 03/31/2018).

[13] strace. *strace*. 2018. URL: https://strace.io/ (visited on 03/31/2018).

[14] Valgrind. *Valgrind instrumentation framework*. 2018. URL: http://valgrind.org/ (visited on 03/31/2018).

[15] Laszlo Vidacs. "ICSOFT 2009 - 4th International Conference on Software and Data Technologies, Proceedings". In: vol. 1. Jan. 2009, pp. 232–237.

[16] Daniel Waddington and Bin Yao. "High-fidelity C/C++ code transformation". In: *Science of Computer Programming* 68.2 (2007). Special Issue on ETAPS 2005 Workshop on Language Descriptions, Tools, and Applications, pp. 64–78. ISSN: 0167-6423. URL: http://www.sciencedirect.com/science/article/pii/S0167642307000718.

Department of Programming Languages and Compilers, Eötvös Loránd University

Email address: zsoltparragi@caesar.elte.hu, gsd@elte.hu

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 2, 2018
DOI: 10.24193/subbi.2018.2.05

INCREMENTAL DECOMPILATION OF LOOP-FREE BINARY CODE: ERLANG

GREGORY MORSE, DÁNIEL LUKÁCS, AND MELINDA TÓTH

Abstract. Decompiling byte code to a human readable format is an important research field. A proper decompiler can be used to recover lost source code, helps in different reverse engineering tasks and also enhances static analyzer tools by refining the calculated static semantic information. In an era with a lot of advancement in areas such as incremental algorithms and boolean satisfiability (SAT) solvers, the question of how to properly structure a decompilation tool to function in a completely incremental manner has remained an interesting problem.

This paper presents a concise algorithm and structuring design pattern for byte code which has a loop-free representation, as is seen in the Erlang language. The algorithms presented in this paper were implemented and verified during the decompilation of the Erlang/OTP library.

1. Introduction

Decompilation of compiled code is the process of transforming a compiled module typically in a machine readable byte code format, back into a human readable source code format. A decompiler, is a tool which automates this process. The practical nature and visible result of a decompiler is an important tool which is useful for source code recovery, reverse engineering of hostile code, or for compiler validation. Decompilers have almost exclusively relied upon an approach which treats the binary as a flat and static block of instruction

Received by the editors: March 31, 2018.

2010 *Mathematics Subject Classification.* 68W01, 68N20.

1998 *CR Categories and Descriptors.* code [**Computing Methodologies - SYMBOLIC AND ALGEBRAIC MANIPULATION**]: Algorithms – *Nonalgebraic algorithms*; code [**Software - PROGRAMMING LANGUAGES**]: Processors – *Incremental compilers*.

Key words and phrases. incremental decompilation, Erlang, dominator tree, post-dominator tree, code duplication.

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

data, and proceeds with various stages also called passes over the byte-code and iteratively through various graph structures until it achieves source code. This approach is well understood, yet it is not a general method, and it does not always produce usable or valid results.

The theoretical limit of decompilation lies at the general computability problem of decidability, famously highlighted by the halting problem. It is proven that there exist cases where it is undecidable whether or not a program continues executing or stops, and this can be generalized to any decision pathway. The first case needing this level of generality is unreachable code. The other is that of self-modifying code, where the binary code is able to modify itself.

The motivation for a study in incremental decompilation theory becomes clear: not only for generality of the decompilation process, but the approach invariably could be used to make better, more flexible compilers. Compilers and decompilers involve the same theory as both do binary transformation of code structures, albeit the programming languages utilize context-free grammars and encourage block structures which is typically far less expressive than byte code which tends to allow any control flow graph (CFG). A CFG can contain sequences, selections and loops. The language of Erlang [1] has been chosen as it is an excellent case study for incremental decompilation for several reasons: no back edges inducing loops in the graph, impossibility of self-modifying code, and an easy to process byte-code which will be pre-structured as instructions by its own libraries.

As a main contribution, a framework for incremental decompilation is presented in this paper based on using a symbolically semantic equivalent representation and meticulous graph structuring. This includes algorithms contributed for scanning methods, processing at merge and exit nodes and variable emission.

Various concepts and tools maintain the incremental cascading of effects. An analysis of semantic equivalence of byte code in a meta-data enhanced abstract syntax tree (AST) representation for any language allows for a cross-language approach. The importance of variable emission scenarios, classifying side effects and nearly inexpressible byte code operations is demonstrated. The incremental maintenance of dominator trees, reachability, and common ancestors are discussed as with minimal processing at merge nodes.

Data interfaces encapsulate the graph structure containing basic blocks, and another is used for the enhanced AST. Algorithms are considered for overall decompilation, handling edges not expressible in the target language, merge nodes with their optimized processing and minimal variable emission. Two scanning algorithms for overall decompilation are studied. The nuances highlight the technical challenge of achieving a consistent incremental algorithm.

Since code copying is a technique which has exponential growth consequences in complexity, simplifications for boolean short circuits are considered. The clean up of the AST is itself a crucial element of the decompiler for a readable and usable decompiled output.

2. Background

"Erlang is a programming language originally developed at the Ericsson Computer Science Laboratory. OTP (Open Telecom Platform) is a collection of middleware and libraries in Erlang." [2].

An abstract syntax tree (AST) is a tree containing information directly corresponding to the grammar of the language. It can be pretty printed to Erlang source code, or compiled to BEAM, or even emulated by the Erlang eval library. In fact the Erlang shell uses this eval emulator to execute commands through evaluating the AST directly without BEAM conversion, albeit with a performance penalty and possibility to execute code which fails the compilers more strict validation on things such as variable bindings in block structures.

A decompiler based on a graph rewriting technique was written which shows that multiple valid approaches to Erlang decompilation are certainly possible [3]. Both follow further a seminal work for decompiler graph structuring [4].

2.1. BEAM code.

The BEAM code, provides a set of opcodes, which operates on a state containing the current instruction location, 1024 registers {x, 0} through {x, 1023} and a stack starting at $\emptyset$ which when initialized has a head always at {y, 0} and can be of any size limited by the memory of the system or any emulator configured limit, and 16 special floating point registers {fr, 0} through {fr, 15}. The current line number is specified in an instruction and is part of the state, although it is only accessible through stack traces which should only be accessed when errors occur per documentation recommendation. The BEAM code is contained in a file with the .beam extension and directly representable by a large tuple containing the whole module, some attribute information, its functions and their BEAM opcodes.

The state of the system is accessible thanks to external code libraries, so potentially unknowable values could find their way into these registers when interacting with the greater system state.

A few special opcodes are used in the emulator itself where it modifies the original BEAM code but these do not concern the decompiler directly.

The correctness of BEAM code has several levels: syntactic valid when compiling it, correctness done by the compiler's validator to prevent situations that would crash the emulator, and finally that which runs on the emulator without crashing. For the sake of generality, it is best to consider the latter correctness as the decompiler could encounter code which is custom crafted with a

modified Erlang source code which disables the validator. The gold standard of correctness is very hard to achieve, namely compilability to identical binary code. Though typically its need is rare, any timing, line number or other minute side effects require it.

2.2. **Incremental Graph Maintenance.** There is inefficiency of constantly recomputing the dominator information which is utilized constantly as soon as any decidable merging happens in the control flow. It is assumed the reader understands the shorthand graph notations for edges and dominators.

Maintaining a graph structure upon edge additions is termed as an incremental algorithm, while further including edge deletions which typically is more complex, is termed as a fully online algorithm.

Algorithms for maintaining a dominator tree incrementally on edge addition, as well as a fully on-line algorithm which also adds edge deletion are known. For this purpose, although G. Ramalingam of IBM Research Laboratories provided the next major break-through [5], the preferred algorithm is the Sreedhar, Ghao, Lee algorithm which uses a data structure called a DJ-graph which is a dominator tree with join edge information about the connectivity of the graph due to the fact that dominator information alone is not enough to easily determine the scope of how much of the tree is effected by addition or deletion. Join edges are all edges which are not between ancestors and descendants in the dominator tree. By introducing the concept of an iterated dominance frontier (IDF) [6], a relatively simple and elegant algorithm emerges to incrementally maintain the DJ graph [7].

Reachability of a given graph node to the return node is also important information to decide how to process when an exception disconnects the control flow for a node or a subset of nodes to the return node. Determining this information cumulatively can be performed with a simple depth or breadth first walk from the return node up the tree to the root where the visited nodes are the set. Incrementally maintaining this information is trivial for edge addition, as adding edges does not reduce reachability and nodes always start as reaching the return node to later not reaching it. For generality, only when a predecessor not reaching is added to a successor which is reaching, then the whole reverse subgraph of the predecessor is added to the set. For edge deletion, the successor if reachable which implies the predecessor is reachable can check if any of its remaining successors is still in the reaching set, otherwise remove itself and continue the process recursively up the graph. The two processes are mirrored as can be seen formally (where REVREACH(X) is the subgraph reachable from the reverse graph rooted at X):

$$\text{AddEdgeReachSet}(U, V, R_s) : \begin{cases} U \notin R_s, V \in R_s & R_s \cup \text{REVREACH}(U) \\ \text{otherwise} & R_s \end{cases}$$

$$\text{NREVREACH}(U, R_s) = [U] \cup \forall Y \in \text{PREDS}(U), \text{NREVREACH}(Y), \nexists X \in \text{SUCCS}(Y), X \in R_s$$

$$\text{RemoveEdgeReachSet}(U, V, R_s) : \begin{cases} V \in R_s, \nexists X \in \text{SUCCS}(U), X \in R_s & R_s \setminus \text{NREVREACH}(U) \\ \text{otherwise} & R_s \end{cases}.$$

It should be noted that actual implementations would likely use breadth first search (BFS) methodology for efficient computation. In fact, this reachability question is more formally known as a transitive closure, and maintaining transitive closure for a directed or undirected graph is a problem which has been studied. Solutions exist such as based on maintaining the order of a linked list, as this problem efficiency-wise is a data structure problem [8].

The depth first search (DFS) tree is a tool used mostly for efficient computation of dominators in this context. Its incremental computation is still an open problem for directed graphs due to the fact that one edge addition or deletion can cause very far reaching changes, along with the fact that multiple valid traversals are possible. In the acylic case however, there are algorithms known [9].

3. Main Contribution: Semantic Equivalence

Various data-flow oriented semantic equivalence of various BEAM opcodes and their corresponding Erlang code equivalent are analyzed. This is limited by control flow structures which are not single instructions but various sequences tied together in certain ways and requires graph analysis. Of importance to the data flow analysis is the concept of purity. Purity of a function is an attribute that the code it contains does not change the state of the system in anyway except that which is returned to the caller.

Three valid approaches to consistent side effect handling are apparent based on the purity analysis and detection. The first is to always emit variables for every state change, but this makes the code unreadable requiring later clean up based on single usage or dead variables, and is expensive as tracing original values for decidability in these variables requires traversing dominators. The second approach is to emit variables for any state change except pure built-in functions (BIFs). This is straightforward and readily implementable. It is the approach which is taken for simplicity. The last approach and more elegant is to emit variables for any state change except functions detected as pure by tools such as PURITY [10]. To not stick to one of these conservative approaches would ultimately to be emitting code which is in fact not reflective of the original code. The discussion does not end with side effects as any values represented as the result of AST emission also must be treated as having a side-effect or the state itself would need to contain AST entries. To avoid this, some situations require variable emissions for the entire block structure of the lambda function creator. The captured variables of the fun expressions also must have variables emitted, not because of side effects, but because injecting

side-effect free representational expressions into the captured variables is certainly different from the original code, where a type of fence exists between the captures and the lambda function since only variables can be captured, and compiler be optimization beyond this is not done. Tables of all instructions with side effects and variable emission fences should be generated and codified.

The whole block of the function has a return value as well which is represented by a single value which must be denoted as a state item. In Erlang, it is always {x, 0}. In practice, this could be a group of values but ultimately most languages allow its expression as a singular value so some type of language grouping element would need to be used to bundle them regardless, such as tuples or lists. The generality could thus be extended.

3.1. **Byte code and Metadata Enhanced Abstract Syntax Tree.** The classical view of byte code running on a system is that of a Fetch, Decode and Execute loop, as per the way the central processing unit (CPU) itself works. The Fetch and Decode step can be considered as one combined unit when not considering timing issues. For a decompiler, this view is changed to a Decide how to Fetch and Decode, and Symbolically Execute loop. The state itself is symbolic of the actual state and does not contain literal values. However the fetch and decode operation when generalized must decide as specifically as it can, the set of bits resultant from the current symbolic state.

To maintain the data flow aspects of decompilation in the AST while it structures itself based on control flow, entries are utilized with a special meta-data key. These contain the values of the x, y and fr registers representing the current state. These are guaranteed at the beginning of every block, and where there are sequences in a block emitted due to side effects including function calls or variable assignment, an updated meta-data entry appears after it.

A table and then code for the state should be compiled for the target language which for Erlang includes the line number and registers as discussed.

3.2. **Scanning and Overall Algorithm.** The sequential scan for a decompiler is not only straightforward to implement, and seems to be a natural choice for scan order, it leads to a number of consequences when dealing with incremental algorithms and decidability aspects. In fact for decidability, it is not sufficient, and cannot be considered as an appropriate algorithm at all, since the decidability algorithm could effectively decide that it needs to know more about another pathway before it can make a decision. This is thereby a dependency and so a different scan ordering addressing these dependencies in decidability must be looked at. However the incremental theory of both approaches will be developed hereby, as some very interesting details are gleaned

from the differences and answering decidability questions is not always a requirement.

The merge nodes are the motivating factor for decisions, and processing of these nodes which is discussed shortly, can be done best when all recursive predecessors which reach the merge point are already scanned, as at this point its post-dominating status is stable. Otherwise if it does not post-dominate, it may later, or if it does post-dominate, it may post-dominate more nodes later, in both cases based on exit scenarios. So the best place to scan at any moment, is any non-merge node, or the un-scanned merge node which is not reached by any other un-scanned merge node and it is processable meaning a colliding edge is confirmed not to reach it. Instead of doing a series of negative reachability checks, a BFS ordering of the graph can provide the topmost node which would reach all the others, so the first un-scanned and processable merge node in the BFS is the best candidate to scan. This greatly simplifies the exit scenarios and even better reduces exponential code copying which occurs as a result of delayed decisions.

The decompiler should maintain the status of each node in a simple structure called the ProcessingState which can have values of: Unprocessed, Processable, Processed, Colliding, which progresses as it is added and then becomes processable then processed, and thereafter possibly marked as a colliding for optimal processing of the node its edge collides, and then it is moved back to Processed. The BFS ordering scan deals with using the processing state to chose the next processable node and is guaranteed to mark all as processing if on jumps it looks ahead to the next node and marks it. The overall algorithm of the incremental decompiler decidably fetches and decodes, and symbolically executes in a loop while structuring based on conditionals, merge points, side effects, or semantic equivalents (see it with example in Appendices B and C).

3.3. **Return, Exit Nodes and Conditionals.** Due to the difficulty of maintaining the post-dominator tree as it could have multiple roots, an exit node is introduced which any node added to the graph maintains connectivity to at all times including the entry node. And also due to the nature of functional languages returning a value upon exit, another placeholder node called a return node representing the emission of a return value is also added. The return node will be permanently connected to the exit node, and all nodes who are being processed or not yet processed will maintaining a successor of the return node. Any exceptions or errors, will cause a node to redirect from the return node to the exit node.

However, the exit node functions differently as it is completely symbolic and no merge occurs. Therefore it does not make sense for it to post-dominate any nodes in the graph, beyond those in the subset of nodes which do not reach

the return node. Therefore the reverse graph needs to be maintained slightly differently than the graph, and a technique for doing this, is that any node which is a predecessor of the exit node has its successor drawn as the successors to its set of predecessors which are part of the return node reaching set R_s. These nodes are chosen by the nearest predecessors reaching the return node or formally as PredSetReach(U, R_s).

The obvious caveat, is that in certain cases, the return node may not be needed, if all code paths go to the exit node. In this case, the return node could either be deleted or more conveniently made to be the sole successor of the exit node. Figure 1 indicates the presumed structuring first, and then one of them becomes the final structure.

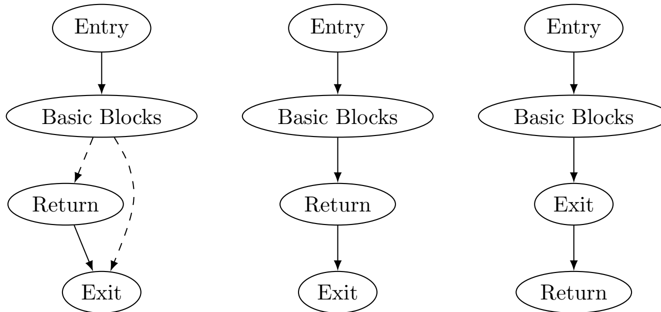


Figure 1. Three control flows: the latter two only for final CFG with no exceptions/exits or no normal returns

For the dominator tree, the prior strategy introduces a problem. The exit node should not dominate any node except the return node, as this is a special allowed control flow transfer which can exit the function regardless of where it is in the AST. To deal with this, when computing the dominator tree, all nodes succeeded by the exit node, have all of their predecessors' successors, replaced with the exit node successor. In effect, this makes the node have no effect on the dominator tree, as if it were deleted.

The exception/exit incidence including the relevant opcodes, their context, semantic equivalence and whether they are singular pathways which effect R_s should be compiled in a table and then coded.

No effect on post-dominator of nearest return reaching node, as the decision node itself is the nearest return reaching node, since by definition its continuation path is unexplored and thus still reaching the return node, and previously the node itself was unexplored thus reaching the return node.

An important routine of the decompiler is conditional structuring which revolves around laying out the edges in the graph for conditionals and exits

which comprise the control flow instructions as well as any block structure instructions which require further analysis due to the fact that they are not single instructions but sequences thereof, and adding various AST emissions as well as meta-data embedded within.

Another table should be compiled of all the control flow instructions which structure each opcode as series of AST modifications and graph changes, while keeping track of the next node for sequential scanning over binary conditionals. Figure 3 demonstrates the equivalent non-block re-structuring of multi-selection conditionals to binary conditionals. Since variable assignment is allowed consistently within conditionals, using binary conditionals, and variable assignment, avoids what is termed a block structure, a language structure requiring a single return value and consistent entry and exit from the structure. The return values can simply be unused as hence ignored by avoiding block structuring. Block structuring has a solution albeit complex and not discussed. The cost is that exponential code duplication can occur.

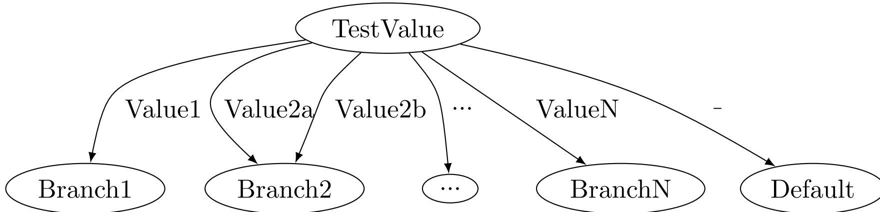


Figure 2. BEAM style select branching structure

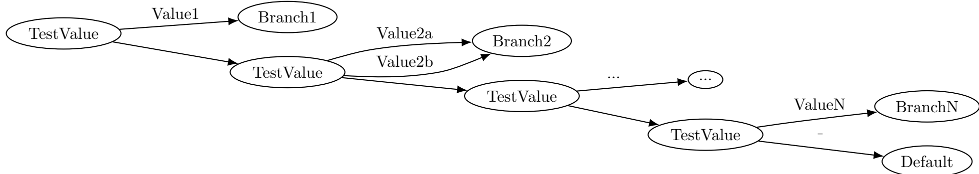


Figure 3. Select semantic equivalent for nested, non-block style structuring

3.4. **Merge Nodes, Cross Edges and AST Mapping.** Anytime two or more edges are incumbent on a node, a merge occurs where the data flows through different paths in the graph must be coalesced. The single static assignment (SSA) form has been used to represent this using a ϕ-function to represent state values at these collision places.

An optimization can occur at this stage when the merge node is reached by an implicit edge referred to hereby as a colliding edge and this edge is

also classified as a cross edge. This is an optimization over merely using the equivalent jump control flow semantic equivalent.

The incremental theory develops by first realizing that the importance of the merge nodes is underscored by the fact that they post-dominate other nodes. Variable assignments at appropriate places can be done in a consistent way such that these merge nodes are not important unless they are also post-dominators. Two events can occur which cause a merge node to become a post-dominator: 1) Most common is that a label is reached where a prior basic block is implicitly added as a colliding edge, or alternatively it could be solely a merge from prior basic block jump targets. 2) An exception or exit occurs. This potentially causes a chain of potential post-dominator merge nodes which must be checked. However these do not merge in the same sense as having a colliding edge. Reprocessing of already processed post-dominators is possible, since an already processed post-dominator can become a post-dominator of an additional node when its exit is realized. In this case, no variable assignments can occur but cross edge processing must proceed. A node which is processed and all that it reaches is immediately post-dominated by a node which is also processed are not further processable and do not risk an exit occurring on them, a stable and decided set of the nodes.

Timing	**Node State**	**Action**
On Reach	Not post-dominating any node	Do nothing
On Reach	Post dominated ≥ 1 nodes	Process fully
Exit	Unprocessed	Do nothing
Exit	Processed and not post-dominating any node	Process fully
Exit	Processed and post-dominating ≥ 1 nodes $\exists N \in$ REVREACH(Node), get_processed(PIDOM(N)) $\neq$ Processed	Process w/o variable assignment

TABLE 1. Post Dominating Node Incidence Classifier for Sequential Scanning

Table 1 and a HandleExit function is thereby a consequence of a sequential scan through the code. This is a convenience and shortcut taking approach for Erlang BEAM code since it can be safely assumed if emitted from the compiler to be all reachable and since it is not able to self-reference and hence self-modify itself, a sequential scan only need know that a given label is reachable from the EntryLabel. An alternative and improved incremental approach could keep a queue of un-scanned locations, and not scan them until they can be processed and hence post-dominate some nodes, while at the same time all nodes reaching it area also processed. In this case, the whole table is unnecessary as there is only a single case that processes fully when post-dominating according to these conditions. This is a more general and more ideal way of

decompiling, but in some contexts, Table 1 can also be a relatively easy to implement and workable methodology.

These are filtered so only the already visited ones are considered. Finally the merge node processing occurs for all of them, as it would normally. Block structures which have only exit paths should also be identified and processed at this point for maximizing incremental effect unless all of their processing would be done at once at the end. This discussion cannot continue without introducing the effect of cross edges as they are the most important aspect towards the resolution before variable assignment occurs.

The DFS-based cross edges and inexpressible forward edges – from the perspective of the AST which reflects valid edges – must be processed to determine where copying of code can resolve these situations. In general, the AST entries are allowed edges in a language when moving: forward to any ancestor, up to a any descendant, to an immediate sibling, or to an exit/exception. Other edges are considered to be a cross edge, and is represented hereby as EdgeClassifier.

A DFS is needed which given that the AST is an ordered tree, is unique and a sorting of the paths of the nodes. Edges which are cross edges will be represented by variable CrossEdges ← E \ EdgeClassifier(E).

The cross edges must be classified based on their significance for processing based on the current node. Therefore the CrossEdges are further mapped by a function to the nearest post-dominator or nearest common ancestor (NCA) which is the longest common suffix (LCS) between dominator paths, except that the node in consideration has included itself in its dominator sequence:

$\text{NCA(PathX, PathY)} = \text{hd(LCS(PathX, PathY))}$

$\text{NearPDom(P, S, Node)} = \text{NCA}(\begin{cases}\text{DOM}(P) & P = \text{Node}\\ \text{SDOM}(P) & \text{otherwise}\end{cases}, \text{DOM(S)})$

$\text{NearCrossPDom(Node)} \leftarrow \forall (P, S) \in \text{CrossEdges, NearPDom(P, S, Node)}$

The cross edges are then filtered so that CopyEdges $\leftarrow \forall (P, S) \in$ CrossEdges, Node = NearPDom(P, S, Node). These are further sorted based on the DFS, where the greater successors or in case of equality, greater predecessors are processed first, hence a bottom up strategy, for convenience and consistency which allows certain data structure optimizations. These values are incrementally computable and as for the NCA, it could be recalculated based on the set of changed nodes in incremental dominator recomputation.

The merge structuring algorithm is divided into three stages where when a colliding edge is a cross edge, a special merge node is added as a place holder for code copying and variable assignments, followed by cross edge code duplication, and then variable assignment. The graph copying going on here removes cross edges, but otherwise has no effect on the post-dominator tree of the original nodes, so intermediate re-computation is not necessary.

Shortcuts are possible here where recognizing the nested, geometrically opposite diamond like shape of andalso as well as orelse is the most prevalent and most useful one as this causes an exponential blow up in the graph and therefore also the AST which makes processing slower even if it can be cleaned up at the end. Otherwise the only illegal edges generally seen are the result of compiler optimizations which reduced copied code, and hence copying the code again becomes necessary. The basic structures are easy patterns to find and can be recursively applied ideally in a bottom to top order.

A bijective mapping between the state nodes in the generated AST and the control flow graph is needed which also must be incrementally maintained. A simple method is to use an indexed path down the tree. The DFS of the tree then is nothing more than a sort operation over of these labels based on their indexed paths. This is incrementally maintained via a two-way indexing scheme for binary search and insertion in sorted order.

The data structure to encapsulate the AST would include not only the actual AST, but the mapping and a sorted two-way indexing for efficiency in lookups and ranking when doing operations on the graph where a DFS ordering based on the AST is necessary. The path of the nodes in the tree would be encapsulated and efficiently implemented in some format. The enhanced AST structure should provide operations for inserting nodes, child nodes, getting or setting the meta-data, getting the DFS of the ordered tree, and copying from a node into another node part of the AST with respect to the NCA.

As for the graph, adding edges, removing edges, getting a BFS, the reverse graph DFS ordering, and the same copy graph operation as for the AST is needed. Symbols for the entry, return, exit nodes and the next node and current node should be a part of this. The graph should be able to answer the various mathematically represented queries for edges, (post-)dominators, reachability, CrossEdges, NearCrossPDom(Node) (see Appendix A).

3.5. **Variable Assignment.** Variable assignment takes all the merge nodes that are post-dominators, after cross edges have been processed, and assigned variables to the minimal consistent set of nodes necessary based on the dominator tree. The merge node itself has its dominator used as a reference point, as variables would not need to be assigned beyond the dominator. But all of its predecessors, and their recursive dominators up to this top dominator need to be considered. If changes or the lack of changes are consistent between all children of a dominator, then the dominator itself can be chosen instead of the child nodes, which is a key reduction in avoiding excessive variable emission. The naive strategy would be to assign variables to all predecessors if there is a change in any one of them.

A very useful optimization is that it is possible to only perform variable assignment on merge nodes that are post-dominators, after cross edges have been processed, but then the algorithm needs to be adjusted to go through the predecessors of these nodes that do not post-dominate.

First the set of changed nodes is determined by recursively going through the dominator tree, and then one more traversal determines the nodes that did not change minimally with respect to the ones that did, so a variable emission always occurs to ensure that a variable is present at the merge node in all pathways. The state information should be maintained therefore by having symbolic current state information for every node as it would be expensive and undesirable to recompute something so easily maintained incrementally.

Algorithm 1 finds the minimal set of changed nodes with regards to the predecessors and their dominators, while Algorithm 2 does this for the nodes which did not change with respect to the nodes which did change in a similar way with this additional exclusion and not needing to query for state changes. Finally Algorithm 3 gives routines used in the merge algorithm for emitting a single value on return, or going through all state item values otherwise. A summary example is provided in Figure 4.

Algorithm 1 Find Minimal Set of Changed Nodes

Require: Candidates is a queue (not a set)
1: **procedure** FindChangedNodes(Candidates, Top, StateItem)
2: Changed $\leftarrow \emptyset$
3: **while** Candidates $\neq \emptyset$ **do**
4: (C, Candidates) $\leftarrow$ (head of Candidates, pop Candidates)
5: **if** C = Top **then** ▷ No Processing
6: **else if** StateChange(IDOM(C), C, StateItem) **then**
7: Changed $\leftarrow \{C\} \cup$ Changed
8: Candidates $\leftarrow$ Candidates $\setminus$IDOM(C)
9: **else**
10: Candidates $\leftarrow$ add IDOM(C) to end of Candidates
11: **end if**
12: **end while**
13: **return** Changed
14: **end procedure**

- EmitVariable(X, Y, S) is a routine which returns a pair (Assignment, Variable) by assigning a variable for the state item S of node set X based on its current state, and then emitting the symbolic representation of the assignment of that variable in the state storage for node Y, the post-dominator, along with symbolic representation of the variable without the assignment.
- StateChange(X, Y, S) is a routine which determines if the stored state has a symbolic difference between nodes X and Y for state item S.

Algorithm 2 Find Minimal Set of Unchanged Nodes Relative to a Set of Changed Nodes

```
1: procedure FindNotChangedNodes(Changed, Candidates, Top, DominatorNodes)
2:     NotChanged ← ∅
3:     while Candidates ≠ ∅ do
4:         NotChanged ← NotChanged ∪∀C ∈ Candidates, C = Top ∨ IDOM(C) ∈ DominatorNodes ∧ IDOM(C) ∉ Changed
5:         Candidates ← {IDOM(C)|C ∈ Candidates, C ≠ Top, IDOM(C) ∉ DominatorNodes}
6:     end while
7:     return NotChanged
8: end procedure
```

Algorithm 3 Variable Assignment Routines

```
1: procedure AssignVariable(X, StateItem)
2:     Nodes ← FindChangedNodes(PREDS(X), IDOM(X) StateItem)
3:     EmitVariable(∀N ∈ Nodes ∪ FindNotChangedNodes(Nodes, PREDS(X) \ Nodes, IDOM(X), ⋃{DOM(C)|C ∈ Nodes}), X, StateItem)
4: end procedure
5: procedure AssignVariables(X)
6:     ∀ StateItem ∈ State, AssignVariable(X, StateItem)
7: end procedure
```

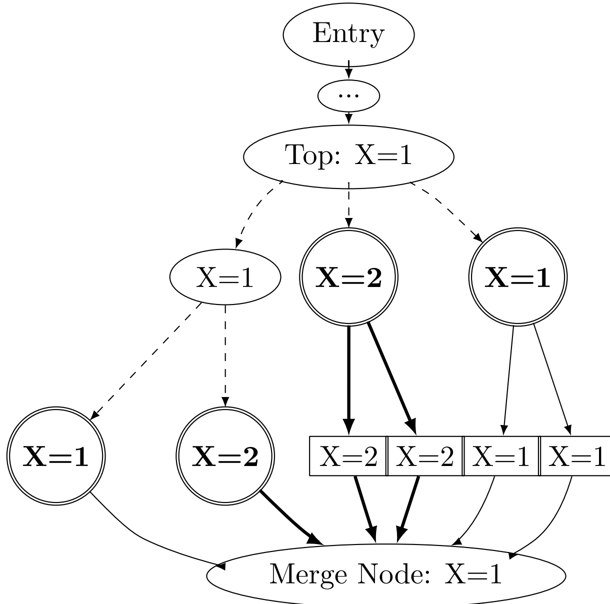


FIGURE 4. Example of the variable assignment algorithm

3.6. **Graph Correction, Clean up and Correctness.** The graph corrections required for a correct AST, which involve removing the meta-data annotations, and the list to tuple transformations of catch blocks and some function

calls for receive which kept a list then tuple nesting structure must be fixed first and manually so a valid AST results. Next comes the various graph corrections required for compilable code, which are the empty values in case structure pathways where a single path went through due to an error in the other path. Any amount of sibling code which comes after can be considered to go into this area. This could be improved but more useful is a totally separate sequence for optimization. Based on this, what emerges is a dependency order for a minimal ambiguity in the cleanup actions.

Line numbers would be very difficult to match, and a novel approach would be needed for true generality.

A proof of correctness of the algorithm has a basis in that variable assignments are processed on merge nodes using a classical dominator-based approach only a single time when its decided the dominator tree for a merge node cannot further change. For the cross edges, they are handled only when they post-dominate some nodes or have an increase therein. The two processing orders will guarantee all predecessor edges recursively are known, so no cross edges will remain when the remaining set to process is empty.

4. Conclusion and Future Work

In this paper we presented a methodology to demonstrate that not only is incremental decompilation possible and feasible, but it can be practically implemented with good results. The technical considerations and details laid forth provide a framework for correct CFG structuring via binary conditionals, and can be extended to block structures like **catch** or **receive**, along with setting forth a code clean-up framework. Due to the close relationship between refactoring of code, and program transformation, this project is to become of the RefactorErl toolkit released by ELTE [11].

We have successfully evaluated and validated our methodology on the source of the Erlang/OTP libraries and the compiler test suite [12]. The details of the evaluation was presented in [13].

In the future, a study should deal with complicated incremental structuring induced by introduction of loops. They have their own theory and ambiguity in determining nesting, along with interference in conditionals studied here where multi-entry/exit loops are concerned. Decidability issues through the use of tools such as boolean satisfiability (SAT) solvers could be incorporated. As for Erlang, the possibility of writing obfuscators based on identified decompilation weaknesses is also an open challenge.

References

[1] Joe Armstrong. *Programming Erlang.* The Pragmatic Bookshelf, 2nd edition, October 2013. ISBN 978-1-93778-553-6.

[2] Ericsson AB. Erlang Programming Language. `http://www.erlang.org`, 2018. [Accessed: 2018.03.14].
[3] Dániel Lukács and Melinda Tóth. Structuring Erlang BEAM Control Flow. In *Proc. of the 16th ACM SIGPLAN International Workshop on Erlang*, Erlang 2017, pages 31–42, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5179-9.
[4] Cristina Cifuentes. *Structuring decompiled graphs*, pages 91–105. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. ISBN 978-3-540-49939-8.
[5] G. Ramalingam and Thomas Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proc. of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 287–296, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0.
[6] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 62–73, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1.
[7] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Incremental computation of dominator trees. *ACM Trans. Program. Lang. Syst.*, 19(2):239–252, March 1997. ISSN 0164-0925.
[8] Paul F. Dietz. Maintaining order in a linked list. In *Proc. of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 122–127, New York, NY, USA, 1982. ACM. ISBN 0-89791-070-2.
[9] Paolo G. Franciosa, Giorgio Gambosi, and Umberto Nanni. The incremental maintenance of a depth-first-search tree in directed acyclic graphs. *Information Processing Letters*, 61(2):113 – 120, 1997. ISSN 0020-0190.
[10] Mihalis Pitidis and Konstantinos Sagonas. Purity in Erlang. In Jurriaan Hage and Marco T. Morazán, editors, *Implementation and Application of Functional Languages*, pages 137–152, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24276-2.
[11] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máté Tejfel, and Melinda Tóth. RefactorErl - Source Code Analysis and Refactoring in Erlang. In *Proc. of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2*, pages 138–148, Tallin, Estonia, October 2011.
[12] Ericsson AB. Erlang/OTP (source code). `https://github.com/erlang/otp`, 2018. [Accessed: 2018.03.14].
[13] Gregory Morse. Towards a General Theory of Incremental Decompilation. TDK Thesis, Budapest, Hungary, May 2018.

Department of Programming Languages and Compilers, Faculty of Informatics, ELTE, Eötvös Loránd University, 1/C Pázmány Péter sétány, Budapest, 1117, Hungary

Email address: `morse@inf.elte.hu` `dlukacs@caesar.elte.hu` `toth_m@inf.elte.hu`

Appendix Appendix A Merge and Exit Incidence Algorithms

Algorithm 4 Algorithms to Process Merge Nodes

```
 1: procedure HandleMerge(Node, IsExit)
Phase 1 - Add place holder node
 2:    if Node = ReturnNode or IsExit then
 3:        InsertNode ← ExitNode, SearchNode ← Node
 4:    else if (CurrentNode, Node) ∈ NearestCrossPdom then
 5:        InsertNode ← NextNode(), SearchNode ← InsertNode,
 6:        insert_ast_node (Node, get_ast_node(Node), InsertNode)
 7:        if CurrentNode ∈ PREDS(Node) then
 8:            add_edge(CurrentNode, InsertNode), remove_edge(CurrentNode, Node)
 9:        end if
10:        add_edge(Node, InsertNode), add_edge(InsertNode, ReturnNode)
11:        remove_edge(Node, ReturnNode)
12:    else
13:        InsertNode ← NextNode(), SearchNode ← Node,
14:    end if
Phase 2 - Resolve cross edges via code duplication
15:    CrossPairs ← ∀(X, Y) ∈ CrossEdges, X ≠ SearchNode ∧ (NearestCrossPDom(X, Y) = Node
     ∨ NearestCrossPDom(X, Y) = InsertNode)
16:    while CrossPairs ≠ ∅ do
17:        (From, To) ← arg max_{(X,Y)∈CrossPairs} ( get_ast_dfs (Y), get_ast_dfs (X))
18:        CrossPairs ← CrossPairs \ {(From, To)}, NodeSet ← REACH(From) \ REACH(Node)
19:        copy_ast_node(X, Y, NodeSet), copy_graph_nodes(X, Y, NodeSet)
20:    end while
Phase 3 - Variable assignment
21:    AfterNode ← { Node          Node = ReturnNode ∨ IsExit
                   { InsertNode    otherwise
22:    if Node = ReturnNode then
23:        insert_ast_node (Node, AssignVariable(AfterNode, ReturnRegister))
24:    else if ¬ IsExit then
25:        set_ast_node({ Node         IsExit      , AssignVariables(AfterNode))
                        { InsertNode   otherwise
26:    end if
27:    return AfterNode
28: end procedure
```

- insert_ast_node (Node, EmitValue, NewValue, NewNode) where EmitValue and NewNode are optional, must insert at the next available AST path after Node, EmitValue if present, and NewValue always, and if NewValue was a meta-data, then NewNode specifies the graph node path into which the meta-data path will be stored for later lookup or removal.
- insert_ast_node_child (Node, Kind, NewValue, NewNode) is identifical to the previous one except Kind gives an additional path information to be traversed based on what type of child is being added such as a conditional or block structure.
- get_ast_node(Node) fetches the node data specified.
- set_ast_node(Node, Data) replaces the data at the node specified.

Algorithm 5 Algorithms to Process Exit Nodes

1: **procedure** HANDLEEXIT
2: NewPDoms ← PotentialNewPDoms
3: **while** N **do**ewPDoms $\neq \emptyset$
4: $C \leftarrow \arg\min_{X \in \text{NewPDoms}}$ get_rev_dfs (X)
5: HandleMerge(C, true), NewPDoms ← NewPDoms $\backslash C$
6: **end while**
7: **end procedure**

- get_ast_dfs (X) returns the simple tree walk ordering of node X to allow a non-conflicting order when copying. In fact this is an approximation ordering that will not always work, without continuing to recompute nearest cross edge post dominators until a fixed point is reached. Technically there is a subgraph induced by FromPath which is copied to ToPath, and any of these subgraphs which contains a ToPath creates a dependency. The optimal scenario is therefore to avoid iterative calculation by adding all the cross edge pairs to a list in dependency order by not adding them until their dependencies are first added which also induces a proper partial ordering. It is a very important point as the algorithm simplifies and hides this fact.
- copy_ast_node(FromNode, ToNode, NodeSet) surgically copies all the contiguous set of the AST tree entries at the same level as FromPath and including it, which contains NodeSet, into ToPath and which must be inserted as new mapped entries for all the meta-data that was copied into ToNode to maintain the integrity and consistency of the AST structure.

As for the graph, the following corresponding operations must be implemented and maintained:

- add_edge(X, Y) adds the edge from node X to node Y.
- remove_edge(X, Y) removes the edge between node X and node Y.
- get_bfs (X) gets a comparable breath first search ordering of node X for the scanning algorithm.
- get_rev_dfs (X) gets a comparable depth first search ordering of node X in the reverse graph rooted at ReturnNode.
- copy_graph_nodes(FromNode, ToNode, NodeSet) copies all the nodes in subgraph NodeSet replacing all edges between nodes in the set with the new nodes, and maintaining all edges which were to outside the subgraph except any nodes preceded by FromNode which are changed to ToNode. This is the corresponding operation to copy_ast_node which deals with the AST on copy.
- EntryNode symbolizes the entry node, ReturnNode symbolizes the return node, ExitNode symbolizes the exit node, and NextNode() symbolizes the next node which is to be newly added to the graph, and CurrentNode is the current node being processed or considered, while CurInst is the current instruction.
- PREDS, SUCCS, DOM, SDOM, PDOM, PSDOM, IDOM, PIDOM, REACH, REVREACH, CrossEdges, NearestCrossPdom(Node).
  PotentialNewPDoms $\leftarrow \forall C \in (\bigcup \forall X \in$ PredSetReach(Node, R_s), SPDOM(X)) $\setminus$ [ReturnNode, ExitNode], get_processed(C)=Processed

Appendix Appendix B Overall and Scanning Algorithms

Algorithm 6 Sequential and Breadth First Oriented Scanning

```
 1: procedure ContinueScan
 2:     if Jumped and IsBFSScan then
 3:         set_processed(GetNextLabel(), Processable)
 4:         Candidates ← ∀C ∈ PREDS(ReturnNode), get_processed(C) = Processable
 5:         if Candidates = ∅ then
 6:             return (∅, ReturnNode)
 7:         else
 8:             NextNode ← arg min_{X∈Candidates} get_bfs(X)
 9:             CollideNode ← ∀C ∈ PREDS(NextNode), get_processed(C) = Colliding
10:             return (GetInstructionAt(NextNode, { hd(CollideNode)   CollideNode ≠ ∅
                                                   { NextNode          otherwise      )
11:         end if
12:     else
13:         return (GetNextInstruction(), CurrentNode)
14:     end if
15: end procedure
```

- get_processed(Node) returns either Unprocessed, Processable, Processed or Colliding.
- set_processed(Node, State) sets Node's processing status to State.
- SemanticEquivalence(C) provides the symbolic net effect on the state of instruction C.
- IsBranching(C) indicates if the C instruction is a branching or exit/exception instruction.
- IsLabel(C) indicates if the C instruction is a label and hence merging point.
- NodeFromLabel(C) returns an already mapped node for the label C, whether pre-existing or requiring a new graph node assignment.
- HasSideEffect(C) classifies if any side effect or other condition arises requires variable assignment.
- UpdateState(Data, StateDifference) updates the state in Data based on the difference.
- Output(C) gets the state.
- GetNextInstruction() gets the next instruction after instruction CurInst unless it is empty and then the entry instruction.
- GetNextLabel() scans forward after a jump for the next label to mark a node as not collided and hence processable.
- GetInstructionAt(Label) gets the instruction at location specified by Label.
- IsBFSScan represents the option for a breath first scan versus sequential.

Algorithm 7 Overall Decompile to AST

```
1: procedure DecompileToAST
2:     AST ← [EntryMetadata], Graph ← [EntryNode, ReturnNode, ExitNode]
3:     CurInst ← ∅, CurrentNode ← EntryPoint, Jumped ← false
4:     while (CurInst, CurrentNode) = ContinueScan(Jumped), CurInst ≠ ∅ do
5:         if IsBranching(CurInst) then
6:             Jumped ← DoBranchingStructuring()
7:         else if IsLabel(CurInst) then
8:             NewNode ← NodeFromLabel(CurInst)
9:             if IsBFSScan ∧ get_processed(CurrentNode) = Processed then
10:                set_processed(CurrentNode, Colliding)
11:            else
12:                add_edge(CurrentNode, NewNode), remove_edge (CurrentNode, ReturnNode), set_processed(CurrentNode, Processed), set_processed(NewNode, Processed), HandleMerge(NewNode, false)
13:            end if
14:        else if HasSideEffect(CurInst) then
15:            (Emit, NewVariable) ← AssignVariable(SemanticEquivalent(CurInst))
16:            insert_ast_node(CurrentNode, Emit, UpdateState(get_ast_node(CurrentNode), NewVariable), CurrentNode)
17:        else
18:            set_ast_node(CurrentNode, UpdateState(get_ast_node (CurrentNode), SemanticEquivalent(CurInst)))
19:        end if
20:    end while
21:    if ExitNode ∉ PREDS(ReturnNode) then
22:        HandleMerge(ReturnNode)
23:    end if
24:    return Changed
25: end procedure
```

Appendix Appendix C Example

An overall example of a program in Figure 5 will highlight several of the key ideas with a special view of the final AST with super-imposed cross edge arrows in Figure 6 corresponding to the graph view of the prior step in Figure 7.

```
1 incstruct(A, B, C, D, E) ->
2   if not A -> A;
3   A andalso B orelse C ->
4     if D + E =:= 1 -> B; true -> error(D + E) end;
5   true -> A end.
```

Figure 5. Example code highlighting cross edge identification, merge node, code copying and variable assignment

```
1:       case Arg1 of  end
4:           true -> case Arg2 =/= true of  end
8:               true -> case Arg3 =:= true of  end
16:                  true -> case Arg4 + Arg5 =:= 1 of  end
18:                      true -> Var1 = Arg2
19:                      false -> error(Arg4 + Arg5)
17:                  false -> Var1 = Arg1
9:               false -> case Arg4 + Arg5 =:= 1 of  end
12:                  true -> Var1 = Arg2
13:                  false -> error(Arg4 + Arg5)
5:           _ -> case Arg1 of  end
6:               false -> Var1 = Arg1
7:               _ -> case Arg3 =:= true of  end
10:                  true -> case Arg4 + Arg5 =:= 1 of  end
14:                      true -> Var1 = Arg2
15:                      false -> error(Arg4 + Arg5)
11:                  false -> Var1 = Arg1
2:       Var1
3:
```

Figure 6. Example program after final structuring step at return node with arrows showing where copying occurred

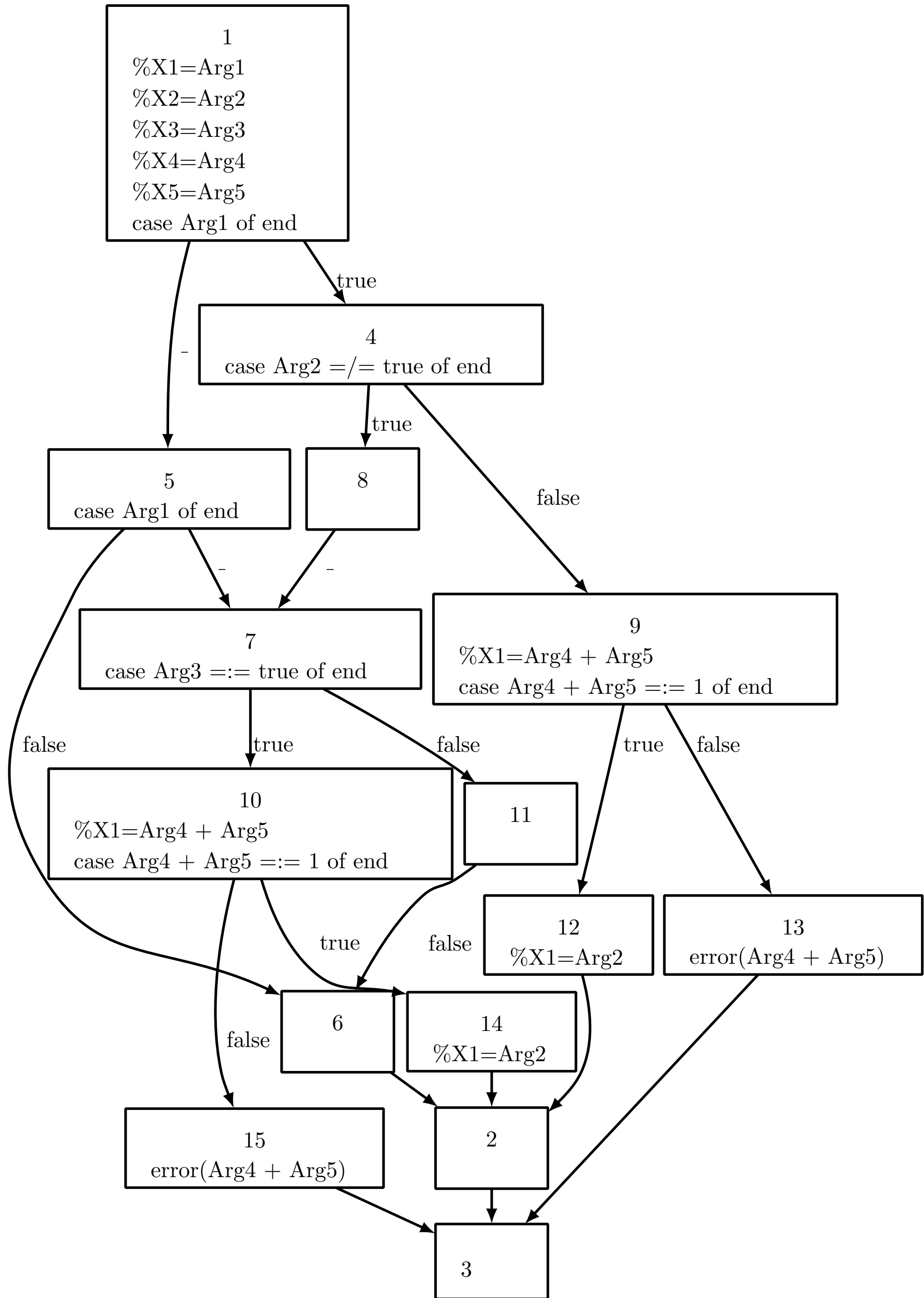


FIGURE 7. Graph derived from example program in final structuring step at return node

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 2, 2018
DOI: 10.24193/subbi.2018.2.06

AN INITIAL PROTOTYPE OF TIERED CONSTRAINT SOLVING IN THE CLANG STATIC ANALYZER

RÉKA KOVÁCS AND GÁBOR HORVÁTH

Abstract. Static analysis is a widely used method for finding bugs in large code bases. One of the most popular static analysis tools used for software written in C/C++ languages is the Clang Static Analyzer [1]. During symbolic execution [2] of the source code, the analyzer models path sensitivity by keeping track of constraints on symbolic variables. The built-in constraint manager module, while granting excellent performance, only handles constraints on certain types of integer expressions, which has a detrimental effect on the quality of the analysis, as the infeasibility of certain execution paths cannot be proved. This often leads to false positive findings, i.e. error reports issued for code parts that are actually correct.

This paper records the first milestone in an effort to integrate the state-of-the-art Z3 theorem prover [3] into the Clang Static Analyzer in order to post-process bug reports. While full integration is hindered by the burden Z3 places on the duration of the analysis, the refutation of false positive reports using information collected by the default pass can improve analysis quality substantially while introducing only a moderate regression in performance. We present an initial prototype of the tiered constraint solving solution that is already capable of filtering out some bogus reports, evaluate it on real-world software projects, and explore possible improvements we plan to accomplish in our future work.

1. Introduction

Static analysis is the analysis of software without actually executing programs, usually performed by an automated tool on the source code. Static analysis tools are widely used in the continuous integration chains of production software as their comprehensive checks can provide rapid feedback on the code's performance, reliability, and safety.

Received by the editors: April 2, 2018.

2010 *Mathematics Subject Classification.* 68N20.

1998 *CR Categories and Descriptors.* F.3.2 [**Logics and meanings of programs**]: Semantics of Programming Languages – *Program analysis*.

Key words and phrases. Static analysis, symbolic execution, Clang, SMT solver.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

One of the main considerations behind the design decisions of static analysis tools is the number of false positive reports produced by the tool. False positives are warnings issued incorrectly for code parts that do not contain erroneous behavior. A high ratio of bogus reports is a much greater problem for an industrial bug finding tool than some number of bugs missed (even so as it is impossible to find all bugs using static analysis [5]). Bug reports need to be reviewed by developers one-by-one in order to correct potential errors in their software. If the tool presents an overwhelming amount of false warnings to the developer, its usability suffers and developers lose their trust in the tool.

The Clang Static Analyzer is a symbolic execution engine built atop of clang [4], a relatively recently developed LLVM compiler front-end for C-family languages. The Static Analyzer is an increasingly popular choice of a static analysis tool despite still being a work-in-progress, as it is free, open-source, but its power already matches that of most closed-source tools widespread in the industry.

Despite heavy developer effort, the Static Analyzer suffers from the problem of false positive reports much like other similar tools. One possible way to improve the quality of the reports is to improve the constraint management of symbolic expressions, which plays an important role in proving the infeasibility of impossible execution paths during symbolic execution. An important intermediary step in this direction is the refutation of false positive reports by re-evaluating constraints by a more powerful constraint solver than the one currently built into the engine.

In the following section, we give a brief overview of the inner workings of the analyzer and explain the role of constraint management during the symbolic execution of a program. In Section 3, we present the results of an experiment highlighting the problem with a straightforward solution and explain the motivation behind the choice of the method described in this paper. In Section 4, we explore some aspects of the problem that need to be considered while constructing the solution. Next, we evaluate our prototype on real-world software projects and then raise some questions for future work in Section 6.

2. The Clang Static Analyzer

2.1. **General overview.** The Clang Static Analyzer is a symbolic execution engine capable of analyzing C, C++, and Objective-C programs. Symbolic execution is a form of abstract interpretation of the source code, where each unknown value encountered is assigned a symbol, on which operations are performed symbolically. Along this process, the analyzer attempts to enumerate all possible execution paths by building a so-called *exploded graph* [6]. Each

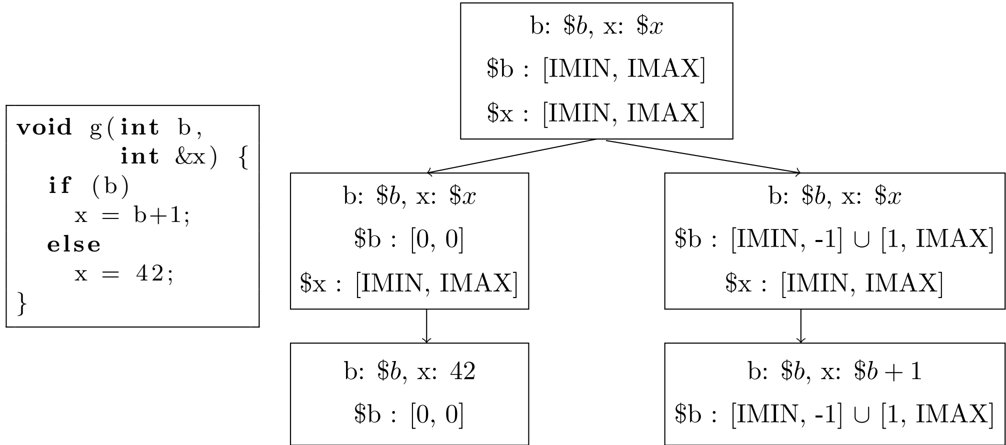


FIGURE 1. An example depicting the representation of symbolic execution in the exploded graph.

vertex of this graph is a *(program state, program point)* pair, where the program point determines the current location in the program, and memory is represented using a hierarchy of memory regions [7]. The program state holds traits of the program such as the *environment*, which records symbolic values of active expressions, and a data structure holding *range constraints*, i.e. ranges that symbolic values may take [8], among others.

During the execution of a path, the analyzer collects constraints on symbolic expressions. The built-in constraint solver module can reason about simpler pointer and integer expressions, by representing constraints on them using closed ranges of integer values. One of the main roles of the constraint manager is to determine whether these constraints become unsatisfiable, in which case the analysis of the current path should be terminated. It is vital for the analyzer to recognize such infeasible program paths in the exploded graph mainly as not to issue error reports for paths that will never be executed.

An example analysis can be seen along with its simplified exploded graph on Figure 1. Function `g` can lead to two execution paths. Since the value of `b` and `x` is initially unknown, these values are represented with symbols `$b` and `$x`, which can take on arbitrary values. As the analysis continues on the execution path corresponding to the `else` branch, the value of `b` is known to be zero, and later we discover that the value of `x` is the constant `42`. Symbol `$x` is no longer needed on this path. On the other path, the value of `b` can be anything but zero. Later, we also discover that the value of `x` is one greater than the original value of `b`. The symbol `$x` is no longer needed on any of the paths, it can be garbage collected.

2.2. **Constraint management in the Static Analyzer.** As mentioned previously, the analyzer collects constraints on symbolic variables encountered in the program to be able to detect if they become unsatisfiable. Solving these constraints is only one side of the coin: generating and managing them is another. Support for constraint management is therefore scattered throughout the analyzer engine. The current solution centers around a solver operating on range-based constraints, which is only capable of handling some common binary operations between symbolic values and concrete integers (called `SymIntExprs`), and some relational operations between two symbols (`SymSymExprs`). Although it is very fast, it lacks support for many other commonly used arithmetic operations even on `SymIntExprs`, such as bitwise operations, multiplication, division, etc.

In 2017, support for an alternative constraint solver backend, the Z3 Theorem Prover, has been added to the engine [9]. Z3 is a state-of-the-art general purpose SMT solver developed by Microsoft Research. Z3 is capable of handling most arithmetic operations unsupported by the current solver, such as those on floating-point values, and it also represents integers more realistically, modeling them with fixed-width bitvectors, granting greater precision in its results.

Unfortunately, the analyzer will not be able to harness the full power of the Z3 Theorem Prover until symbolic expression support is improved in the engine. Namely, the analyzer currently does not build up symbolic expressions consisting of floating-point type values, and subsequently does not generate constraints on them, meaning that information about such expressions never arrives at the constraint manager. Still, without any further effort, Z3 should already be able to improve analysis precision for expressions involving pointers and integers.

Nevertheless, the analyzer still does not employ Z3 as the default constraint solver backend. The reason behind this is its negative impact on the duration of the analysis, with execution times soaring up to and above a factor of 20 times the usual. This slow-down stems from the nature of SMT solvers, which follow complex inner heuristics, and often use up all of the allowed time as limited by the timeout parameter for a single operation. For practical use, an intermediary solution is needed.

One possible compromise is to use the Z3 Theorem Prover for *false positive refutation*. This means to perform the analysis as usual, then post-process the collected bug reports to find those that lie on paths that are found to be infeasible by Z3. This could eliminate a large portion of false positive reports while only introducing a moderate burden on the duration of the analysis.

3. Motivation: an experiment

In an effort to explore how each of the currently available constraint solving backends affect analysis performance and quality, we made the following experiment. For 3 real-world open-source projects, we ran two analyses, each with default settings but differing in the use of the constraint manager backend. We were concerned in the number of reports and execution times in each case. In the table below, the RB keyword denotes the default range-based solver built into the engine, while reports added and removed are meant for the Z3 cases compared to the runs using the range-based solver. Analysis duration is presented in the format `hh:mm:ss`.

Project name	Reports (RB)	Reports (Z3)	Reports removed	Reports added	Duration (RB)	Duration (Z3)
tmux [10]	15	15	0	0	00:01:06	03:09:45
redis [11]	53	20	1	34	00:01:19	03:21:01
xerces-c [12]	69	2	0	67	00:05:40	03:06:22

Table 1. Information about default analyses run using different constraint manager backends on some open-source projects.

It is interesting how the number of bugs seems to drop significantly when using the Z3 backend in the case of *redis* and *xerces-c*. A study of the bug reports could shed light on whether there are already some false positives eliminated, but based on the analyzer statistics shown below for *redis*, it seems more likely that Z3 is timing out and giving up on interesting paths.

Statistic	Range-based	Z3
The # of steps executed.	82 419 176	44 062 305
The # of functions at top level.	19 993	6 834
The # of paths explored by the analyzer.	65 627	33 382
The # of basic blocks in the analyzed functions.	215 656	70 679
The max # of basic blocks in a function.	3 152	1 575
The # of times we reached the max # of steps.	255	163

Table 2. Sample statistics collected by the analyzer about its own operation during the analysis of the *redis* project.

Using the proposed bug post-processing method, the engine will call Z3 significantly fewer times than it would in the case of an ordinary analysis with the Z3 backend. Because of this, it might be reasonable to slightly increase the timeout limit when refutation is switched on, as it could enable the analyzer to explore more paths with Z3, prospectively improving the quality of the analysis.

4. A Prototype of Tiered Constraint Solving

Under tiered constraint solving, we really mean re-solving constraints for paths that lead to bugs. To understand how this can be achieved inside the analyzer technically, we first give a brief overview of the workings of its bug reporting mechanism, and then explain the rationale behind some design decisions of the prototype.

4.1. **Bug reporting in the Static Analyzer.** If the analyzer finds a critical issue during building the exploded graph, such as a division by zero error, it stops the analysis on that execution path, generates an error node, and emits a bug report (less critical problems would generate a non-fatal error node, in which case the analysis continues on that path, but the bug report is still emitted). Bug reports are continuously collected during analysis, and processed after the construction of the exploded graph is finished.

In order to generate a meaningful path diagnostic from a bug report, and to suppress some reports which are likely to be false positives, the analyzer executes *bug reporter visitors* at this late stage of the analysis. Starting from the error node, visitors travel backwards on the bug path and perform arbitrary operations needed to accomplish their task - usually place additional notes to interesting locations along the path. The bug reporter visitor interface therefore offers a convenient way to implement the tiered constraint solving prototype.

4.2. **Building the refutation visitor.** Constraints on symbolic values are collected in the program state during the analysis. Whenever a new constraint appears for a symbol, the constraint manager attempts to add it to those already in the state, and if it can prove them to be unsatisfiable, then the current state is said to be infeasible. While building the exploded graph, the engine uses this information in order not to generate a new node for an impossible state. As branching statements are typical points in a program where constraints are added to expressions, constraint management issues relevant for bug report post-processing can be demonstrated on examples involving branching.

Stage 1: Readily available constraints. Sometimes, the range-based constraint solver can reason about a branching condition, and even gets to the point of generating constraints corresponding to a true and a false assumption on the condition, but fails to prove that one of the created states is infeasible. Consider the following example:

```
void g(int d);
void f(int *a, int *b) {
  int c = 5;
  if ((a - b) == 0)
    c = 0;
  if (a != b)
    g(3 / c); // division by zero: false positive
}
```

Arriving at the second `if`, both conditions are understood and translated to ranged constraints, but the solver is not able to prove that they contradict each other. This can be seen from the exploded graph as the current path splits to two, meaning that the constraint manager found both new states to be feasible. On the path assuming that both conditions are true, the exploded node holds the following constraints:

```
Ranges of symbol values:
(reg_$0<int * a>) - (reg_$1<int * b>): { [0, 0] }
(reg_$1<int * b>) - (reg_$0<int * a>): { [-9223372036854775808, -1],
                                          [1, 9223372036854775807] }
```

Here, the `(a != b)` condition has been rearranged to `((b - a) != 0)` by the engine, and the constraint was generated by substracting zero from the full range of possible pointer values, representing the result with a union of the two intervals below and above zero. These constraints can be modeled by the following small `z3` program:

```
(declare-const a (_ BitVec 32))
(declare-const b (_ BitVec 32))
(assert (= (bvsub a b) #x00000000))
(assert (bvslt (bvsub b a) #x00000000))
(assert (bvsgt (bvsub b a) #x00000000))
(check-sat)
(get-model)
```

which, after execution, gives an `unsat` result, i.e. the problem is proved to be unsatisfiable. This is the simplest case our bug reporter visitor should be able to handle: ranged constraints are readily available in the program state, they only need to be fed to a Z3 solver instance in the proper format. For this, constraints on symbolic values need to be converted to the internal expression type used by Z3, which involves the translation of integer relations into their corresponding correct bitvector operations. If the translation succeeds and the Z3 solver can prove the state to be infeasible, the report is marked invalid, and never shown to the user.

Stage 2: Constraints that need to be extracted. If the constraint manager encounters a symbolic expression that it cannot reason about, it also cannot generate a constraint for it. Consider the following example:

```
void g(int c);
void f(int a, int b) {
  int c = 0;
  if (a > 3)
    if (b < 3)
      if (b > a)
        g(5 / c); // division by zero: false positive
}
```

As the constraint manager gives up while interpreting the third `if` condition, it cannot prove that the state where all three conditions are true is not feasible, hence the false positive report. This example is more problematic than the previous one because whenever the constraint manager cannot generate a constraint for an expression, the constraint will also not appear in the program state. The data structure that comes to our aid is the *control flow graph* (CFG), a representation of all paths that might be traversed during program execution. The CFG is constructed by the compiler in an intermediary step of the analysis, and the analyzer core relies on it heavily while building the exploded graph. While the unrecognized condition does not appear among the constraints directly, it can still be extracted from CFG-related information recorded in the exploded node (the terminator statement of the CFG block):

```
Terminator: if b > a
line=6, col=7
Condition: true
...
Ranges of symbol values:
reg_$0<int a>: { [4, 2147483647] }
reg_$0<int b>: { [-2147483648, 2] }
```

from where it could be extracted and fed to a Z3 solver instance with the method outlined at the previous example. Z3 could prove that the path is infeasible, as demonstrated by the program below, which gives an `unsat` result.

```
(declare-const a (_ BitVec 32))
(declare-const b (_ BitVec 32))
(assert (bvsgt a #x00000003))
(assert (bvslt b #x00000003))
(assert (bvsgt b a))
(check-sat)
(get-model)
```

Other examples where false positives can be eliminated with tricks like this can be discovered through systematically designed experiments.

Stage 3: Constraints that need improved symbolic expression support. Symbolic values are the building blocks of symbolic expressions being created by the *symbolic value builder* module during the analysis of a program. Expressions not supported by the symbolic value builder become `UnknownVals` and never get to the point of being handled by the constraint manager. Because of this, such constraints will never appear in the *environment* (the data structure of the program state that maps expressions to their corresponding symbolic values), meaning that they will also not appear in the exploded graph, on which the visitor is meant to operate.

```
void g(int d);
void f(float c) {
  int a = 2;
  if (c > 42.0)
    return;
  if (c > 0.0)
    a = 0;
  if (- 3.14 * c * c > 0)
    g(3 / a); // divide by zero: false positive
}
```

In the above example, `c` is a floating-point value for which the symbolic valuei builder cannot create a valid symbol at the present. As there is no information in the graph that could help prove that the truthness of the third `if` condition leads to an infeasible state, the path leading to the false positive report is created.

This problem can be mitigated by adding support for currently unhandled symbolic values to the symbolic value builder. After such improvements, information needed for the false positive refutation visitor to work will be present in the graph and the previously described methods can be used.

5. Evaluation

In its present state, the refutation visitor implementation is capable of handling constraints that are understood by the default constraint manager and saved into the *range constraints* data structure of the program state. It implements a so-called bug reporter visitor, that is run for each bug report after the construction of the exploded graph is completed. Starting from the error node, the visitor traverses backwards on each buggy execution path, collecting the appropriate ranged constraints from the visited nodes, and adding them

to a Z3 solver instance. At the end of the path, the solver is asked to solve the constraints, and if it finds them unsatisfiable, the bug report gets invalidated.

Evaluation experiments were conducted by running the appropriate analysis configurations on a collection of open-source software projects listed below, on the same virtual machine and using 12 threads. An attempt was made to select both smaller and larger projects written for different purposes in both C and C++ (apart from the previously cited projects: [13], [14], and [15]).

5.1. **Refutation vs. the Z3 constraint solving backend.** The false positive refutation option was designed to provide a compromise between the speed of the default analysis and the precision of an analysis using the Z3 constraint manager backend. We therefore ran analyses with the refutation option switched on on the open-source projects studied in Section 3, in order to compare their results to those using the Z3 backend.

We do not expect the same results for several reasons. First, analyzing projects using the Z3 backend, the whole process uses the Z3 constraint manager, and the resulting exploded graph may differ from the one built in default mode. This means that constraints stored in the graph may be slightly more realistic or precise than those generated in the default mode. However, its working mechanism also differs from the case in which the default analysis is merely enhanced by the refutation visitor. Because of its independent nature, refutation may eliminate false reports that an analysis with the Z3 backend cannot, e.g. those caused by weaknesses in the engine's general operation. And even though it operates on constraints generated by the default solver, the table presented below shows that its advantage in speed may outweigh its disadvantage in granting report quality.

Project name	Reports (default)	Reports (FPR)	Reports (Z3)	Duration (default)	Duration (FPR)	Duration (Z3)
tmux	15	15	15	00:01:06	00:02:02	03:09:45
redis	53	49	20	00:01:19	00:01:22	03:21:01
xerces-c	69	29	2	00:05:40	00:05:50	03:06:22
libWebM	6	6	0	00:00:56	00:00:58	09:26:28
curl	17	14	10	00:01:16	00:01:15	01:34:04
memchached	17	14	1	00:00:37	00:00:38	00:48:32

TABLE 3. Comparison of analyses run with the default configuration, with refutation enabled and using the Z3 constraint manager backend.

5.2. **Refutation vs. default analysis.** From an industry viewpoint, the study of any performance regression refutation poses on the analysis is essential. The following table contains the number of bug reports for two analysis runs for 6 open-source projects, one with a default configuration, and one with the naive prototype of false positive refutation enabled.

As expected, the tiered constraint solving pass did not create any new reports. Since it begins to operate after the exploded graph is completed, it does not participate in the actual analysis process, and can only remove some of the existing reports, but has no means to add new ones. The number of invalidated reports, on the other hand, depends heavily on the analyzed project. In most cases, only a few bugs were removed by the visitor, which is reasonable considering that it currently handles a narrow range of subtle false positive cases. The *xerces-c* project stands out in this regard, with more than a half of its bugs thrown away. After performing a manual inspection of some of the removed reports, either the falseness of the reports was difficult to determine (because of long bug paths), or we found that the report was truly a mistake on the analyzer's behalf and its removal increased the overall quality of the analysis.

Project name	Reports (default)	Reports (FPR)	Reports removed	Duration (default)	Duration (FPR)
tmux	15	16	0	00:01:01	00:01:18
redis	53	161	4	00:02:15	00:04:01
xerces-c	69	40	40	00:03:22	01:01:22
libWebM	6	28	0	00:01:21	00:02:50
curl	17	36	3	00:01:01	00:01:00
memcached	17	14	3	00:29:30	00:40:17

TABLE 4. Report numbers for analyses with a default configuration and with false positive refutation (FPR) enabled for some open-source projects.

6. FUTURE WORK

Although the tiered constraint solving protoytype is already capable of eliminating some false positive bug reports, its functionality can be extended and its results greatly improved once further enhancements outlined below will be introduced into the analyzer.

6.1. **Symbolic expression support.** The main purpose of the initial introduction of the Z3 backend was to enable the analyzer to reason about floating-point values. For this to work fully, the analyzer needs to generate and handle symbolic floating-point expressions (`SymFloatExprs` and `FloatSymExprs`). Apart from floats, symbolic expression support should generally be improved in the symbolic value builder module, including arithmetic operations involving values other than concrete integers. This could fundamentally improve the precision of the analysis.

The analyzer sometimes makes assumptions about algebraic operations that were written with the integer-based constraint manager in mind, and often do not hold for other types of values (e.g. the `x == x` is `true` assumption does not hold for special floating-point values like `NaNs`). Along with the introduction of new types, these assumptions could be revised and extended to support operations defined for any new types, for example for floats. Additionally, assumptions like these could also be checked for expressions involving an integer and a symbol or two symbols.

Code parts responsible for dropping constraints that will not be digested by the symbolic value builder are scattered around in its current form. This makes it difficult to evaluate how changing the level of detail affects the performance of the engine (and it is also more difficult to determine what is handled). This logic could be collected to one place behind a flag, so that symbolic expression handling could be controlled easily.

6.2. **Packaging.** Although the adoption of the tiered constraint solving solution in the Clang Static Analyzer is already in progress, its usage for a normal user is hindered by packaging issues. Users typically use the analyzer as part of their continuous integration toolchains and are reluctant to make modifications to their command scripts, so Z3 support should be granted just by updating their *clang* to the latest version.

This is however not possible because of the project's licensing policy. Although Microsoft Research open-sourced their Z3 theorem prover, its license is still not compatible with *clang*'s liberal open-source license, and thus cannot be included in the latest *clang* package. On the contrary, the Z3 sources need to be downloaded and installed separately by each user and then *clang* needs to be built with special flags that find the Z3 installation and enable its support. Most of the users would probably abandon the refutation feature because of such inconveniences.

One idea to solve this situation would be to find and integrate another SMT solver like the Z3 theorem prover, but with a compatible license. Alternatively, if no project is found that suits the analyzer's needs, a small SMT solver could

be re-implemented inside the analyzer much like the range-based solver, but a more powerful one.

To facilitate solver comparing experiments and to make solver backend switching more flexible, a general SMT solver interface could be implemented in the analyzer. The current constraint management framework relies heavily on the built-in range-based solver, and only has been extended to support Z3 in a very special manner. Most of the duplicate work involved in adding a new backend at the present could be avoided with a general interface.

7. Conclusion

Minimizing the number of false positive reports is a critical issue for most static analysis tools in order to grant high-quality results to their users. A method taking an important step towards this goal was presented for one of the most widely used open-source static analysis tools for C family languages, the Clang Static Analyzer. The solution works by introducing an additional step towards the end of the analysis, when constraints on symbolic expressions encountered on the buggy execution path are re-evaluated by a powerful external constraint solver engine, invalidating a bogus bug report if the path leading to it is found to be infeasible. This step is needed because the default built-in constraint solver is designed to prefer speed over precision, while using the precise external solver for the whole process would result in unacceptably long execution times. In order for the analyzer to retain its industrial-strength performance, a practical intermediary solution was needed.

The tiered constraint solving solution described in this paper is careful to preserve close-to-usual execution times while eliminating many of the false positive reports, as found by an evaluation on a set of real-world software projects. Additionally, an agenda of possible enhancements was outlined that might be useful to study and implement to further improve the results. The prototype is currently under review by the open-source community.

8. Acknowledgement

We owe our special thanks to Artem Dergachev and George Karpenkov, core developers of the Clang Static Analyzer, for the discussion and advice regarding the proposed changes.

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

References

[1] Clang Static Analyzer. https://clang-analyzer.llvm.org

[2] HAMPAPURAM, Hari; YANG, Yue; DAS, Manuvir. Symbolic path simulation in path-sensitive dataflow analysis. In: ACM SIGSOFT Software Engineering Notes. ACM, 2005. p. 52-58.

[3] DE MOURA, Leonardo; BJØRNER, Nikolaj. Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2008. p. 337-340.

[4] "clang" C language family frontend for LLVM. https://clang.llvm.org/

[5] RICE, Henry Gordon. Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society, 1953, 74.2: 358-366.

[6] REPS, Thomas; HORWITZ, Susan; SAGIV, Mooly. Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1995. p. 49-61.

[7] XU, Zhongxing; KREMENEK, Ted; ZHANG, Jian. A memory model for static analysis of C programs. In: International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Springer, Berlin, Heidelberg, 2010. p. 535-548.

[8] Artem Dergachev: Clang Static Analyzer - A Checker Developer's Guide. 2016. https://github.com/haoNoQ/clang-analyzer-guide

[9] Dominic Chen: Add new Z3 constraint manager backend. Differential Review. 2017. https://reviews.llvm.org/D28952

[10] Tmux, a terminal multiplexer. https://github.com/tmux/tmux/

[11] Redis, an open source, in-memory data structure store. https://redis.io/

[12] Xerces-C++ XML Parser. https://xerces.apache.org/xerces-c/

[13] WebM, an open web media project. https://www.webmproject.org/

[14] Curl, a command line tool for transferring data with URLs. https://curl.haxx.se/

[15] Memcached, a distributed memory object caching system. https://memcached.org/

Email address: `rekanikolett@caesar.elte.hu`

Eötvös Loránd University, Department of Programming Languages and Compilers, Pázmány Péter st. 1/C., Budapest, Hungary

Email address: `xazax@caesar.elte.hu`

Eötvös Loránd University, Department of Programming Languages and Compilers, Pázmány Péter st. 1/C., Budapest, Hungary

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 2, 2018
DOI: 10.24193/subbi.2018.2.07

INCREMENTAL RELATIONAL ASSOCIATION RULE MINING OF EDUCATIONAL DATA SETS

LIANA MARIA CRIVEI

Abstract. *Educational Data Mining* is an attractive research field in which the underlying idea is that of bringing the *data mining* perspective into *educational environments*. The main focus is to better understand the educational related phenomena by extracting, through data mining techniques, meaningful hidden patterns from educational data sets. *Incremental Relational Association Rule Mining* ($IRARM$) has been introduced as an effective *online data mining* method for dynamically mining interesting *relational association rules* (RARs) in a dynamic data set which is extended with new data instances. The study conducted in this paper is aimed to emphasize the effectiveness of both RAR and $IRARM$ mining methods in *educational data mining* settings. Experiments performed on various academic data sets highlight the potential of using *relational association rules* for uncovering relevant knowledge from educational related data.

1. Introduction

Data mining (DM) techniques are extensively applied nowadays in various domains including medicine, bioinformatics, software engineering, to discover relevant patterns in large databases, especially due to their potential of uncovering hidden information from data.

Applying DM techniques in education [5] has attracted researchers from both DM and educational research and thus a new interdisciplinary research discipline known as *educational data mining* (EDM) emerged. The main focus in EDM is to develop methods for extracting knowledge from data that come from various educational information systems and educational environments.

Received by the editors: April 15, 2018.

2010 *Mathematics Subject Classification.* 68T05, 68P15.

1998 *CR Categories and Descriptors.* H.2.8[**Database management**]: Database Applications – *Data Mining*; I.2.6[**Computing Methodologies**]: Artificial Intelligence – *Learning*.

Key words and phrases. Data mining, Educational data mining, Relational association rule, Incremental algorithm.

Through mining educational data sets, EDM's purpose is to better understand the students' learning process and thus to offer additional insights into educational related phenomena.

Within the DM domain, *association rule* (AR) mining represents an important data analysis and mining technique [9] applied in various *supervised* and *unsupervised* learning scenarios for extracting rule based patterns from data sets. *Ordinal association rules* (OARs) [7] were proposed as a particular class of ARs which express ordinal relationships between the attributes characterizing a data set. Afterwards, *relational association rules* (RARs) [6, 11] have been introduced as an extension of OARs capable to capture various type of non-ordinal relations between data attributes.

We are approaching in this paper the problem of *incremental relational association rule mining* ($IRARM$) in the context of EDM. The process of incremental RAR mining is appropriate specifically for *online* DM scenarios, where the data set to be mined is dynamic and thus continuously extended with real-time arriving data streams. In such situations, $IRARM$ approach aims to progressively adapt the interesting RARs identified in a data set, when it is enlarged with new instances. Since the learning processes within educational environments are by nature online processes, the idea of investigating the $IRARM$ perspective in EDM comes naturally. The EDM literature also reveals that DM is very useful in the educational field particularly when exploring the online learning environment [18].

The contribution of the paper is summarized as follows. First, we are emphasizing the relevance of RAR mining in the field of *educational data mining* (EDM) with the goal of uncovering meaningful patterns within educational data sets. Secondly, we extend the experimental evaluation of our previously proposed *incremental relational association rule* mining approach ($IRARM$) [17] on several EDM case studies. The effectiveness of $IRARM$ is emphasized through the reduction in mining time achieved when using $IRARM$ against RAR mining from scratch when a data set is extended with new instances. The study conducted in this paper is novel in the EDM literature, since neither the classical nor the incremental RAR mining approaches have been applied on academic data sets, so far.

The rest of the paper is structured as follows. Section 2 introduces the EDM domain and emphasizes its relevance within the larger DM field. A background on RAR mining and its incremental extension $IRARM$ is presented in Section 3, together with an example of RAR mining in EDM. Section 4 describes the experiments performed for highlighting the performance of $IRARM$ on four academic data sets and discusses upon the obtained experimental results. The

conclusions of the paper and directions for future improvements are highlighted in Section 5.

2. Educational Data Mining

EDM is an attractive research field in which the underlying idea is that of bringing the *data mining* perspective into *educational environments*. The main focus is to better understand the educational related phenomena by extracting, through data mining techniques, meaningful hidden patterns from educational data sets.

Extracting relevant patterns from the educational processes would also be useful for understanding students and how they learn, as well as improving the educational outcomes (e.g. learning outcomes). EDM has received lately considerable attention from the research community since extracting hidden knowledge from educational data is of particular interest for the academic institutions and also useful for improving their teaching methodologies and learning processes [18].

Various applications using data mining techniques have been developed, so far, in the EDM field. *Machine learning* methods are intensively investigated, both from a *supervised* and *unsupervised* perspective, as data mining techniques for building course planning systems, detecting what type of learners are the students, grouping students according to their similarity, predicting the students' performance for courses, assisting instructors in the educational process [15].

We briefly review, in the following, several recent approaches which have been developed for assessing the performance of students in educational environments.

Ayers et al. applied in [3] several clustering algorithms such as hierarchical agglomerative clustering, K-means and model based clustering for grouping students according to their skill sets.

Bharadwaj and Pal conducted in [4] a study towards identifying features which are strongly correlated with the academic students' performances. The authors found out that characteristics such as the living location, medium of teaching, mother's qualification, the family annual income, and student's family status highly influence the performance of the prediction task. Pal and Pal conducted in [21] a study using decision tree based classification algorithms to identify the students needing special advising and counseling from the teachers.

Supervised classification models such as *Naive Bayes*, *decision trees*, *neural networks* have been applied in [15] together with *Synthetic Minority Over-Sampling* (SMOTE) method to improve the accuracy of a machine learning

model for predicting the students' final grade for a particular course. An analysis of the performance of the previous mentioned machine learning models, including *support vector* classification was performed by Shahiri et al. in [23]. Additionally, a study was conducted upon the effectiveness of the attributes involved in the classification process.

Ahmed et al. focused in [2] on predicting the performance of instructors and analyzed the factors that affect students' academic achievements, with the purpose of improving the quality of the educational system. Several classifiers such as J48 Decision Tree, Multilayer Perceptron, Naïve Bayes, and Sequential Minimal Optimization were applied and compared to identify the best performing classification algorithm. Among all considered classifiers, J48 provided the best classification accuracy of 84.8%.

The problem of predicting the students performance (PSP) has been considered in [24] as regression problem and a hybrid method combining a collaborative filtering-based system and a regression-based one has been proposed.

3. Background on relational association rules

In Section 3 the fundamental concepts related to *relational association rule* (RAR) mining [11] are reviewed. Then, the relevance and importance of RAR mining in the context of EDM is emphasized through an example on an educational data set. Section 3.2 briefly presents the *incremental relational association rule* mining (*IRARM*) approach [17].

3.1. Relational association rule mining. *Association rule* (AR) mining represents an important data analysis and mining technique [9] useful in multiple *machine learning* tasks for uncovering meaningful rule based patterns in data sets. *Ordinal association rules* (OARs) [7] were proposed as a particular class of ARs which express ordinal relationships between the attributes characterizing a data set. RARs [6, 11] have been introduced as an extension of OARs able to express different type of non-ordinal relations between data attributes.

The *Relational Association Rules* (*RARs*) notion is defined in the following paragraphs.

We consider $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$ a set of *instances* or *records*. Let $\Omega = (a_1, \ldots, a_m)$ be a sequence of m attributes characterizing each instance from the data set $\mathcal{D}$. Each attribute a_i takes values from a non-empty and non-fuzzy domain Δ_i, which also contains a *null* (*empty*) value. We denote by $\Psi(d_j, a_i)$ the value of attribute a_i for an instance d_j.

We denote by $\mathcal{T}$ the set of all possible relations that are not necessarily ordinal which can be defined between two domains Δ_i and Δ_j.

Definition 3.1. *A* relational association rule [11] *is an expression*

$$(a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_h}) \Rightarrow (a_{i_1}\tau_1 a_{i_2}\tau_2 a_{i_3}\ldots\tau_{h-1}a_{i_h}),$$

where $\{a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_h}\} \subseteq \Omega$, $a_{i_k} \neq a_{i_p}$, $k, p = 1, \ldots, h$, $k \neq p$ *and* $\tau_k \in \mathcal{T}$ *is a relation over* $\Delta_{i_k} \times \Delta_{i_{k+1}}$, Δ_{i_k} *being the domain of the attribute* a_{i_k}.

a) *If* $a_{i_1}, a_{i_2}, \ldots, a_{i_h}$ *are non-missing in* m *instances from the data set then we call* $s = \frac{m}{n}$ *the* support *of the rule.*
b) *If we denote by* $D' \subseteq \mathcal{D}$ *the set of instances where* $a_{i_1}, a_{i_2}, a_{i_3}, \ldots, a_{i_h}$ *are non-missing and all relations* $\Psi(d_j, a_{i_1})\tau_1\Psi(d_j, a_{i_2})$, $\Psi(d_j, a_{i_2})\tau_2 \Psi(d_j, a_{i_3})$, $\ldots$, $\Psi(d_j, a_{i_{h-1}})\tau_{h-1}\Psi(d_j, a_{i_h})$ *hold for each instance* d *from* D' *then we call* $c = \frac{|D'|}{n}$ *the* confidence *of the rule.*

Interesting RAR's were defined in [11] as those rules which have both their *support* and *confidence* greater than or equal to specified minimum thresholds. For mining interesting RARs an Apriori-like algorithm named *DRAR* (Discovery of Relational Association Rules) was proposed in [12] as an extension of the *DOAR* algorithm introduced in [7] for uncovering OARs.

3.1.1. *Example.* For a better understanding of the concept of RAR, an example on an EDM related data set is considered. The aim is to highlight the relevance of applying RAR mining in the context of educational data sets.

The data set used in our example is a real data set, containing the grades obtained by students at a Computer Science undergraduate course offered at Babeş-Bolyai University in a time frame of four academic years (2014-2018). The complete data set is available at [1]. There are a total of 867 instances characterized by 6 attributes, denoted by $a_1, a_2, \ldots, a_6$. These attributes represent the following: written exam score (a_1), seminar score (a_2), laboratory score (a_3), first practical test score (a_4), second practical test score (a_5) and final grade (a_6). Considering the minimum support threshold at $s_{min} = 1$ and the minimum confidence threshold at $c_{min} = 0.6$, we applied *DRAR* mining algorithm. Since all the attributes in our experiment have integer values, two possible binary relations between integer valued attributes were used: $\leq$ and $>$. The discovered maximal interesting RARs are illustrated in Table 1.

Each line from Table 1 describes a RAR of a certain length (depicted in the first column), which has the confidence illustrated in the third column. For example, the first line in Table 1 refers to the RAR $a_1 \leq a_3$ of length **2** (i.e. the rule contains two attributes) having a confidence of **0.739**. This rule has the following interpretation: the value of the attribute a_1 is less or equal than the value of the attribute a_3 in 73.9% of instances from the analyzed data.

Analyzing the last rule depicted in Table 1 one observes that for 61.3 % of the students the grade for the written exam is less or equal than the first test

Length	Rule	Confidence
2	$a_1 \leq a_3$	0.739
2	$a_1 \leq a_5$	0.751
2	$a_1 \leq a_6$	0.825
2	$a_2 \leq a_3$	0.751
2	$a_2 \leq a_4$	0.828
2	$a_2 \leq a_5$	0.819
2	$a_2 \leq a_6$	0.669
2	$a_3 \leq a_4$	0.722
2	$a_3 \leq a_5$	0.711
2	$a_3 > a_6$	0.605
2	$a_4 \leq a_5$	0.713
2	$a_5 > a_6$	0.604
3	$a_1 \leq a_4 > a_6$	0.613

TABLE 1. Interesting maximal relational association rules mined for $s_{min} = 1$ and $c_{min} = 0.6$.

grade, which is greater than the final grade. This suggests that the grades for the practical test are greater than those for the written exam, which could be considered typical because the written exam requires wider knowledge. Analyzing other interesting rules depicted in the table 1: $a_3 \leq a_4$ and $a_3 \leq a_5$ we observe that the grade obtained for the laboratory is less than the both practical test scores. This is an indication that some of the laboratory assignments are more difficult or complex than the actual practical test. The complexity of the practical test could be increased. We also observe that $a_3 > a_6$ meaning the laboratory score is less than the final grade score.

The RARs mined from the academic data may be relevant for the professor and can provide indications about the complexity of the laboratory assignments or the written exams.

3.2. **Incremental relational association rule mining.** We have previously introduced in [17] an *incremental relational association rule mining* approach, called *IRARM*, which is useful when a data set to be mined is extended with new objects. In such situations, for uncovering the interesting RARs from the extended data set, *IRARM* will efficiently adapt the RARs discovered in the data set before the extension. This incremental process will be more effective than running *DRAR* from scratch on the extended data set.

We consider in the following that the data set $\mathcal{D}$ to be mined is dynamic, being extended at a certain time with a non-empty set of instances $\{d_{n+1}, d_{n+2},$

$\ldots, d_s\}$. We denote the enlarged set of instances by $\mathcal{D}^{ext} = \{d_1, d_2, \ldots, d_s\}$, while the set of newly added instances is $\mathcal{D}^{new} = \mathcal{D}^{ext} \setminus \mathcal{D}$. For pre-specified minimum support (s_{min}) and confidence thresholds (c_{min}), we analyze the problem of incrementally identifying all *interesting* RARs in the extended data set $\mathcal{D}^{ext}$ by adapting the set of interesting RARs mined in $\mathcal{D}$ before its extension. Through the *IRARM* method we aim to reduce the running time required to mine the set $\mathcal{R}ules^{ext}$ of interesting RARs from $\mathcal{D}^{ext}$.

Certainly, new interesting RARs could be produced by the newly added instances, but also RARs which were interesting enough in the data set before extension may become uninteresting on the extended data set. The set $\mathcal{R}ules^{ext}$ of all interesting RARs may be discovered by applying the *DRAR* method from scratch, on the extended set of objects. But this process can be computationally expensive. That is why our goal is to replace it by a more efficient algorithm *IRARM* (*Incremental Relational Association Rule Mining*), which preserves the completeness of the RARs generation procedure. Considering the newly added instances, *IRARM* adjusts the set $\mathcal{R}ules$ of all interesting RARs in the initial data set $\mathcal{D}$ to produce the set of all interesting RARs in the extended data set $\mathcal{D}^{ext}$.

The idea behind determining the set $\mathcal{R}ules^{ext}$ will be further described. Two main *stages* characterize the *IRARM* method. The **first stage** is **filtering** the set $\mathcal{R}ules$ of interesting RARs from the initial data set $\mathcal{D}$ in order to maintain only the rules which are interesting in the extended data set $\mathcal{D}^{ext}$ as well. The **second stage** consists of **extending** the subset previously obtained with new rules which were not interesting in $\mathcal{D}$, but become interesting on the extended data set $\mathcal{D}^{ext}$. After the second phase is completed the set $\mathcal{R}ules^{ext}$ of interesting RARs from the extended data set $\mathcal{D}^{ext}$ will have been mined.

More details about the description of *IRARM* algorithm can be found in [17].

The educational process is essentially dynamic therefore the results of the evaluation for new students are available in an incremental manner. While new information is accessible the existent academic data set is continuously updated. In such situations the discovery of interesting relational association rules in academic data sets is an incremental process. Consequently it is more efficient from a computational viewpoint to apply the *IRARM* incremental method (by adapting the set of rules identified before updating the data) rather than applying *DRAR* from the scratch on the entire data set.

4. Results and discussion

We provide in the following an experimental evaluation of *IRARM* on two academic data sets, as well as a discussion upon the obtained results.

4.1. **Case studies and data sets.** In order to evaluate the performance of *IRARM*, two case studies will be conducted on four educational data sets.

The first case study is performed starting from a real academic data set collected from the Babeş-Bolyai University.

4.1.1. *First case study.* The *first data set* used in our study is the real data set described in Section 3.1.1 and available at [1].

The *second data set* considered in our evaluation is synthetically generated from the first data set and is available at [1]. The number of instances from this data set is 867, as in our first data set. Since our first data set contained a relatively small number of attributes, namely 6, we extended the set of attributes to 10 attributes, $a_1, a_2, \ldots, a_{10}$. The first 5 attributes from this data set have the same meaning as described in Section 3.1.1. Attributes a_6, a_7, a_8 and a_9 represent the scores for four additional practical tests, while the last attribute a_{10} represents the final grade. We mention that the values for the added attributes a_6–a_9 were randomly generated, using a uniform distribution, within the interval determined by the attributes a_2, a_3, a_4, a_5.

4.1.2. *Second case study.* The second case study used for evaluating *IRARM* contains the *Turkiye Student Evaluation* data sets publicly available at [14]. There are two data sets (*Turkey Student Evaluation Generic* and *Turkey Student Evaluation Specific*) each containing a total of 5820 evaluation scores provided by students from Gazi University in Ankara (Turkey). Each data set contains a total of 28 course specific questions and additional 5 specific attributes. Details about the attributes can be found at [14].

4.2. **Experimental results.** Let us denote by s the number of instances from the analyzed data set. For both data sets from our first case study $s = 867$, while for the data sets from the second case study $s = 5820$.

In the performed experiments, for all data sets considered for evaluation, the following experimental methodology was applied. We have started with n instances in the data set (for various values for n) and afterwards the data set was extended with $s - n$ entities. Different values were used for the minimum confidence threshold c_{min}, while s_{min} was set to 1 since our data sets do not contain missing values.

For each experiment, the set of interesting RARs on the extended data set containing s instances was obtained in two ways:

(1) by adapting using *IRARM* the set of RARs obtained on the data set before its extension;
(2) by applying the *DRAR* method from scratch on the extended data set.

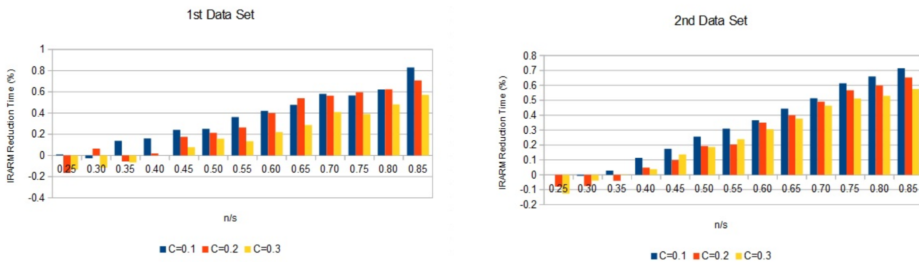
We mention that, using method (1) or (2), the set of interesting RARs discovered in data is the same, but we expect the total mining time for $IRARM$ to be lower than the total mining time of $DRAR$ applied from scratch. The experiments presented in this section were performed on a PC with an Intel Core i7 Processor at 2.30 GHz, with 4 GB of RAM.

In the mining process, we used the following binary relations between the integer valued attributes: $>, <, =$.

For all four data sets from our case studies, we have repeatedly run $DRAR$ and $IRARM$ for different values for c_{min} and different values for $\frac{n}{s}$. Tables 2 and 3 present the results obtained when considering $c_{min} = 0.1$ and varying $\frac{n}{s}$ from 0.25 to 0.85 with a step size of 0.05. For a certain combination of parameters (n, s, c_{min}), the mining method ($DRAR$ or $IRARM$) was executed 20 times and the results were averaged over these executions. The fifth column from the tables gives the reduction in total running time achieved by $IRARM$ computed as $\frac{DRAR\ time - IRARM\ time}{DRAR\ time}$.

From Tables 2 and 3 we observe that, when the percentage of initial instances $\frac{n}{s}$ is larger than 0.35, the running time of $IRARM$ is increasingly reduced with respect to the running time of $DRAR$, as the number of instances added to the data set decreases. The maximum reduction in mining time obtained by $IRARM$ is achieved when $\frac{n}{s}$ is 0.85 and is higher than 70%.

Figure 1 depicts, for the data sets from our first case study, how the percentage of $IRARM$'s running time reduction increases when increasing $\frac{n}{s}$, for three different minimum confidence thresholds: 0.1, 0.2 and 0.3.



(A) *First data set.* (B) *Second data set.*

FIGURE 1. $IRARM$'s reduction in total mining time for the data sets from our first case study, using different minimum confidence thresholds.

Figures 2 and 3 illustrate, for the data sets from the first case study, how the running time for the main operations of $IRARM$ algorithm (*Filter* function, candidates generation process, *Select* function, support and confidence

Data set	**n**	**s-n**	**Time DRAR** (ms)	**Time *IRARM*** (ms)	***IRARM* time** reduction (%)
First data set	217	650	5.6	5.55	**0.0089**
	260	607	5.65	5.8	**-0.0265**
	303	564	5.8	5	**0.1379**
	347	520	5.65	4.75	**0.1593**
	390	477	5.8	4.4	**0.2414**
	433	434	5.8	4.35	**0.25**
	477	390	5.8	3.7	**0.3621**
	520	347	5.7	3.3	**0.4211**
	564	303	5.45	2.85	**0.4771**
	607	260	5.6	2.35	**0.5804**
	650	217	5.3	2.3	**0.5660**
	694	173	5.55	2.1	**0.6216**
	737	130	5.85	1	**0.8291**
Second data set	217	650	15.05	15.05	**0**
	260	607	14.9	15	**-0.0067**
	303	564	14.8	14.4	**0.0270**
	347	520	15.05	13.35	**0.1130**
	390	477	14.95	12.35	**0.1739**
	433	434	15.05	11.2	**0.2558**
	477	390	15	10.35	**0.31**
	520	347	15.15	9.6	**0.3663**
	564	303	15	8.35	**0.4433**
	607	260	14.9	7.25	**0.5134**
	650	217	15.15	5.85	**0.6139**
	694	173	15.15	5.15	**0.6601**
	737	130	15.1	4.3	**0.7152**

TABLE 2. Experimental results on the data sets from our first case study for $s_{min} = 1$ and $c_{min} = 0.1$.

computation) evolved when varying $\frac{n}{s}$ for $c_{min} = 0.1$. From the figures we observe that running times for the *Filter* and *Select* operations decrease while $\frac{n}{s}$ increases.

Figure 4 illustrates for the data sets from the second case study, how the percentage of *IRARM*'s running time reduction increases when increasing $\frac{n}{s}$, for two different minimum confidence thresholds: 0.8 and 0.85.

The experimental results presented in this section highlighted the effectiveness of *IRARM* method, which reduces the mining time against the time achieved by applying DRAR mining from scratch when a data set is extended with new instances.

4.3. **Comparison to related work.** The *incremental relational association rule mining* approach previously introduced in [17] and applied in this paper on educational data sets is new both in the DM and EDM literature. Existing incremental approaches from the DM literature handle only *non-relational*

Data set	n	s-n	Time DRAR (ms)	Time *IRARM* (ms)	*IRARM* time reduction (%)
Turkiye Student Evaluation Generic data set [14]	1455	4365	814	875.2	**-0.0751**
	1746	4074	552.2	564.75	**-0.0227**
	2037	3783	782.7	703.25	**0.1015**
	2328	3492	794.35	664.55	**0.1634**
	2619	3201	834.65	678.35	**0.1873**
	2910	2910	765	564.95	**0.2615**
	3201	2619	816.85	525.15	**0.3571**
	3492	2328	692.4	386.1	**0.4424**
	3783	2037	739.95	358.7	**0.5152**
	4074	1746	570	238.25	**0.5820**
	4365	1455	809.65	312.35	**0.6142**
	4656	1164	628.55	228.3	**0.6368**
	4947	873	501.65	156.45	**0.6881**
Turkiye Student Evaluation Specific data set [14]	1455	4365	686.7	751.65	**-0.0946**
	1746	4074	684.35	695.65	**-0.0165**
	2037	3783	742.4	676.55	**0.0887**
	2328	3492	845.95	704.05	**0.1677**
	2619	3201	967.55	779.25	**0.1946**
	2910	2910	1010.95	637.9	**0.3690**
	3201	2619	980.75	540.4	**0.4490**
	3492	2328	936.45	475.7	**0.4920**
	3783	2037	730.25	383.85	**0.4744**
	4074	1746	565.1	252.7	**0.5528**
	4365	1455	748.35	291.85	**0.6100**
	4656	1164	952.9	360.8	**0.6214**
	4947	873	961.85	288.2	**0.7004**

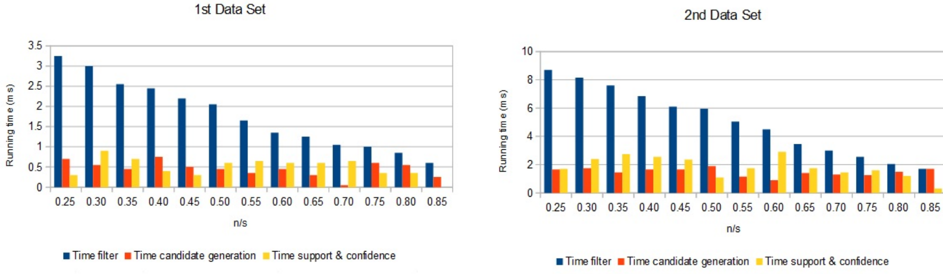
TABLE 3. Experimental results on the data sets from our second case study for $s_{min} = 1$ and $c_{min} = 0.8$.

association rules. In the EDM literature we have not found, so far, approaches using *relational association rule mining* or *incremental relational association rule mining* on EDM scenarios.

We present in the following several data mining methods which deal with the *incremental mining* perspective on *non-relational* association rules.

Sarda and Srinivas introduced in [22] an algorithm for incremental association rule mining, in which the data set is extended with new instances. The proposed adaptive algorithm was able to identify new rules for the updated database, avoiding multiple scans of it. Yafi *et al.* proposed in [25] an incremental association rules mining algorithm named YAMI based on the Apriori model on evolving databases. The authors also introduced the concept of *shocking interesting rule*, as a rule which surpass all user's expectations.

The incremental association rule mining on dynamic transactional databases was investigated by Chandraker and Sao in [8]. Nath *et al.* present in [19] a



(A) *First data set.* (B) *Second data set.*

FIGURE 2. Running time (ms) for the main operations of $IRARM$, for both data sets from the first case study, for $c_{min} = 0.1$.

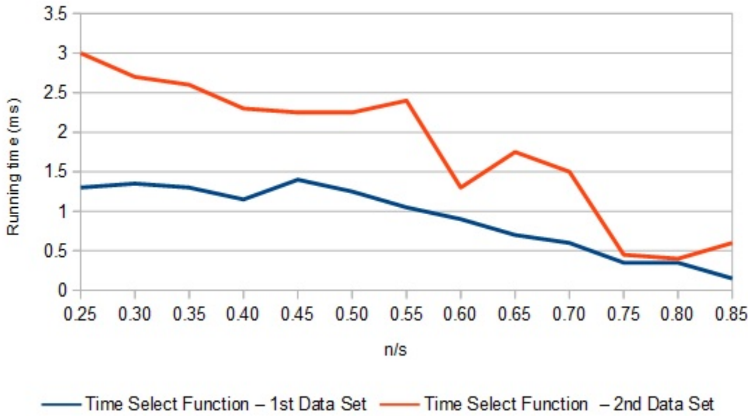
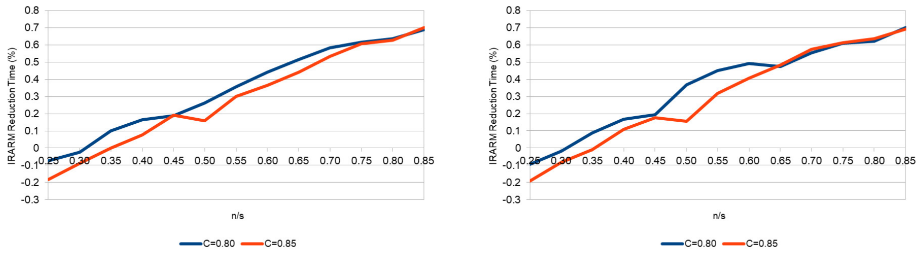


FIGURE 3. Running time (ms) for the *Select* function, for both data sets from the first case study, for $c_{min} = 0.1$.

survey on incremental association rule mining. They review frequent itemset generation techniques, rule generation techniques and incremental association rule mining techniques. The authors emphasize several research issues and challenges, such as the incremental behaviour of the data set, the number of data set scans and the number of generated candidate itemsets. Dhanabhakyam and Punithavalli [13] propose An Adaptive Association Rule Mining with Faster Rule Generation Algorithm (FRG-AARM) with the intent of acquiring a more efficient Market Basket Analysis.



(A) *Turkiye Student Evaluation Generic* data set.

(B) *Turkiye Student Evaluation Specific* data set.

FIGURE 4. *IRARM*'s reduction in total mining time for the data sets from the second case study, using different minimum confidence thresholds.

Ogunde *et al.* [20] introduced an Adaptive Incremental Mining Algorithm (AIMA) aimed to adapt to the trend of constant data updates in distributed databases.

An incremental association rule mining algorithm has been proposed by Yu-Dong *et al.* [26]. This was named VSIFP-Growth (Improved FP-Growth) and used together with parallel computing techniques with the purpose of developing the PVSIFP-Growth algorithm for frequent itemsets generation. Li et al. [16] developed *TDUP*, a *three-way decision update pattern approach* together with a synchronization mechanism in order to reduce the number of scans of the initial data set.

The above presented methods deal with *incremental* AR mining, but from a *non-relational* perspective. Unlike the classical *association rules*, RARs are capable to express relationships between data attributes. Thus, RARs may be more powerful than classical ARs in various machine learning scenarios, including those related to EDM tasks.

5. Conclusions and future work

We investigated in this paper the application of classical and incremental RAR mining for knowledge discovery in data sets from educational environments, with the goal of uncovering meaningful patterns within educational data sets. The relevance of uncovering RARs in academic data sets has been emphasized in the context of the students' learning process, as offering additional insights into educational related phenomena. Additionally, the effectiveness of *incremental* RAR mining in online EDM scenarios was highlighted through several case studies.

Future work will be done in order to extend the experimental evaluation of *IRARM* on other EDM tasks, to further test its performance. An *incremental adaptive* RAR mining will be also investigated for academic data sets, when both new instances and new features are added to the data set. Furthermore, we plan to apply RAR, gradual RARs [10] and *IRARM* mining algorithm in supervised learning EDM scenarios, such as predicting student's academic performance.

Acknowledgements

The author acknowledges the assistance received by using the UCI Machine Learning Repository.

References

[1] Academic data set, 2018. http://www.cs.ubbcluj.ro/∼liana.crivei/AcademicDataSets.

[2] Ahmed Mohamed Ahmed, Ahmet Rizaner, and Ali Hakan Ulusoy. Using data mining to predict instructor performance. *Procedia Computer Science*, 102:137 – 142, 2016. 12th International Conference on Application of Fuzzy Systems and Soft Computing, ICAFS 2016, 29-30 August 2016, Vienna, Austria.

[3] Elizabeth Ayers, Rebecca Nugent, and Nema Dean. A comparison of student skill knowledge estimates. In *Educational Data Mining - EDM 2009, Cordoba, Spain, July 1-3, 2009. Proceedings of the 2nd International Conference on Educational Data Mining.*, pages 1–10, 2009.

[4] Brijesh Kumar Baradwaj and Saurabh Pal. Mining educational data to analyze students' performance. *CoRR*, abs/1201.3417, 2012.

[5] Alejandro Bogarín, Rebeca Cerezo, and Cristóbal Romero. A survey on educational process mining. *Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery*, 8(1), 2018.

[6] Alina Câmpan, Gabriela Şerban, and Andrian Marcus. Relational association rules and error detection. *Studia Universitatis Babes-Bolyai Informatica*, LI(1):31–36, 2006.

[7] Alina Campan, Gabriela Şerban, Traian Marius Truta, and Andrian Marcus. An algorithm for the discovery of arbitrary length ordinal association rules. In *DMIN*, pages 107–113, 2006.

[8] Toshi Chandraker and Neelabh Sao. Incremental mining on association rules. *International Jurnal of Engineering and Science*, 1(11):31–33, 2012.

[9] H. Y. Chang, J. C. Lin, M. L. Cheng, and S. C. Huang. A novel incremental data mining algorithm based on fp-growth for big data. In *2016 International Conference on Networking and Network Applications (NaNA)*, pages 375–378, July 2016.

[10] I. G. Czibula, G. Czibula, D.-L. Miholca. Enhancing relational association rules with gradualness. *International Journal of Innovative Computing, Information & Control*, 13(1):289-305, 2017.

[11] Gabriela Şerban, Alina Câmpan, and Istvan Gergely Czibula. A programming interface for finding relational association rules. *International Journal of Computers, Communications & Control*, I(S.):439–444, June 2006.

[12] Gabriela Czibula, Maria-Iuliana Bocicor, and Istvan Gergely Czibula. Promoter sequences prediction using relational association rule mining. *Evolutionary Bioinformatics*, 8:181–196, 04 2012.

[13] M. Dhanabhakyam and M. Punithavalli. An efficient market basket analysis based on adaptive association rule mining with faster rule generation algorithm. *The Standard International Journals on Computer Science Engineering and its Applications (CSEA)*, 1(3):105–110, 2013.

[14] N. Gunduz and E. Fokoue. UCI machine learning repository, 2013.

[15] Syed Tanveer Jishan, Raisul Islam Rashu, Naheena Haque, and Rashedur M. Rahman. Improving accuracy of students' final grade prediction model using optimal equal width binning and synthetic minority over-sampling technique. *Decision Analytics*, 2(1):1, Mar 2015.

[16] Yao Li, Zhi-Heng Zhang, Wen-Bin Chen, and Fan Min. Tdup: an approach to incremental mining of frequent itemsets with three-way-decision pattern updating. *International Journal of Machine Learning and Cybernetics*, 8(2):441–453, Apr 2017.

[17] Diana-Lucia Miholca, Gabriela Czibula, and Liana Maria Crivei. A new incremental relational association rules mining approach. In *22nd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems*, KES2018, page to be published. Procedia Computer Science, 2018.

[18] Siti Khadijah Mohamad and Zaidatun Tasir. Educational data mining: A review. *Procedia - Social and Behavioral Sciences*, 97:320 – 324, 2013. The 9th International Conference on Cognitive Science.

[19] B. Nath, D. K. Bhattacharyya, and A. Ghosh. Incremental association rule mining: A survey. *Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 3(3):157–169, 2013.

[20] Adewale O. Ogunde, Olusegun Folorunso, and Adesina S. Sodiya. The design of an adaptive incremental association rule mining system. In *Proceedings of the World Congress on Engineering 2015 - Volume I*, London, UK, 2015.

[21] Kumar Ajay Pal and Saurabh Pal. Analysis and mining ofeducational data forpredictingthe performance of students. *International Journal of ElectronicsCommunication and Computer Engineering*, 4(5):278—-4209, 2013.

[22] N. L. Sarda and N. V. Srinivas. An adaptive algorithm for incremental mining of association rules. In *Proceedings of the 9th International Workshop on Database and Expert Systems Applications*, DEXA '98, pages 240–, Washington, DC, USA, 1998. IEEE Computer Society.

[23] Amirah Mohamed Shahiri, Wahidah Husain, and Nur'aini Abdul Rashid. A review on predicting student's performance using data mining techniques. *Procedia Computer Science*, 72:414 – 422, 2015. The Third Information Systems International Conference 2015.

[24] Thi-Oanh Tran, Hai-Trieu Dang, Viet-Thuong Dinh, Thi-Minh-Ngoc Truong, Thi-Phuong-Thao Vuong, and Xuan-Hieu Phan. Performance prediction for students: A multi-strategy approach. *CYBERNETICS AND INFORMATION TECHNOLOGIES*, 17(2):164 – 182, 2017.

[25] Eiad Yafi, Ahmed Al-Hegami, Afshar Alam, and Ranjit Biswas. YAMI: Incremental mining of interesting association patterns. *The International Arab Jurnal of Information Technology*, 9(6):504–510, 2012.

[26] Guo Yu-Dong, Li Sheng-Lin, Li Yong-Zhi, Wang Zhao-Xia, and Zeng Li. Large-scale dataset incremental association rules mining model and optimization algorithm. *International Journal of Database Theory and Application*, 9(4):195–208, 2016.

Department of Computer Science,, Faculty of Mathematics and Computer Science,, Babeş-Bolyai University, Kogălniceanu 1, Cluj-Napoca, 400084, Romania

Email address: liana.crivei@cs.ubbcluj.ro