Towards the Automated Correction of Bugs[*]

Richard Warburton and Sara Kalvala
Department of Computer Science,
University of Warwick, UK
{R.L.M.Warburton, S.Kalvala}@warwick.ac.uk

December 14, 2009

Abstract

Bugs within Java programs often fall within well-known motifs, usually arising from misunderstood APIs or language features that encourage buggy corner cases. Existing software development tools can detect some of these situations, and integrated development environments may attempt to suggest automated fixes for some of the simple cases. We present a language for specifying program transformations paired with a novel methodology for identifying and fixing bug patterns within Java source code. We propose a combination of source code and bytecode analyses: this allows for using the control flow in the *bytecode* to help identify the bugs while generating corrected *source code*. The specification language uses a combination of syntactic rewrite rules and dataflow analysis generated from temporal logic based conditions. We introduce a prototype implementation that allows application of these transformations automatically to programs, and discuss correctness issues within the context of such program transformations. Finally we discuss other possible areas of application for this methodology, including generating refactoring operations from specifications and application to other imperative languages.

1 Introduction

Debugging existing programs, while maintaining the *intent* of the programmer, is an unavoidable but difficult task, which can take significant effort in the software development lifecycle. Some existing tools can detect some of the commonly repeated bugs in particular programming languages, and some integrated development environments (IDEs) may attempt to suggest automated fixes for some of the simple cases. However, as far as we are aware, there is no general tool for specifying bug detection mechanisms that also offers suggested fixes based on the specifications.

[*]This work is supported by the EPSRC under grant EP/DO32466/1 "Verification of the optimising phase of a compiler".

In this paper we propose a domain specific language, with side conditions derived from temporal logic, that offers a solution for this difficult problem of finding and fixing subtle bugs. Traditional application of abstract interpretation and static analysis is focused around checking a specified property of a specified program. In this work we seek to find bugs in large families of programs by facilitating the coding of common bug patterns and then detecting instances of those bug patterns. Each instance of a bug pattern is a potential bug and each pattern has one or more resolutions associated with it, that can be instantiated for a given potential bug.

We consider some concurrency bugs, since they require more than simple syntactic pattern matching to be identified yet are amenable to temporal analysis. We use `Java` as our example platform, though our methodology is applicable to many imperative languages. Our approach to considering temporal control flow properties is to syntactically match specific threading library calls, as one would with normal literals.

Our contributions in this paper are as follows:

1. We simplify the construction of tools for static analysis of bug patterns.
2. We propose a method to automatically fix a larger class of bugs than previous tools.
3. We show how to codify common bug patterns within a formally defined language.

In Section 2 we place our work in context, by identifying the kind of bugs which we consider and also the approach to software development for which our approach is particularly suited. We then describe, in Section 3, the language $\mathsf{TRANS}_{\mathsf{fix}}$ which can be used for both identifying bugs and implementing the transformations which correct the bugs. The prototype implementation FixBugs which applies bug fixes written in $\mathsf{TRANS}_{\mathsf{fix}}$ to `Java` programs is described in Section 4.

2 Methodology and Application

2.1 Example Bug Patterns and Categories

We use as starting point the classification of common `Java` bugs due to Hovemeyer and Pugh [Hovemeyer and Pugh(2004)] and which are used in the description of the FindBugs tool which detects most of them.

Many of the bugs identified by Hovemeyer and Pugh are simple and their identification requires merely a syntactic pattern matching system. Many of them, however, don't have obvious fixes.

2.1.1 Method does not release lock on all paths

This bug arises when a method acquires a lock, but there exists a path through the method where the lock isn't released. The FindBugs implementation focuses

```
Lock l = ...;
l.lock();
try {
    // do something
} finally {
    l.unlock();
}
```

Figure 1: Pattern for correct locking

on the `java.util.concurrent` lock, as specified in JSR-166. Figure 1 illustrates the solution to this bug.

2.1.2 Method may fail to close stream

This bug occurs when a method creates an IO stream object but does not assign it to any fields, pass it to other methods that might close it, or return it, and does not appear to close the stream on all paths out of the method. This may result in a file descriptor leak. Good programming discipline requires the use of a `finally` block to ensure that streams are closed. Figure 2 shows an example of (1) where not to place a close and (2) where to place it correctly.

2.1.3 Failed database transactions may not be rolled back

JDBC, a `Java` library for database connections, models the begin, committing and ending of transactions through explicit calls to methods. A common bug pattern is a failure to check whether a transaction needs to be rolled back if its commit fails. The correct pattern is illustrated in Figure 3. Another common problem is the failure to ensure that all paths either end in a commit or a rollback.

2.2 Placing debugging within software development

In general, a good approach to tooling the fixing of bugs is to not entirely automate the application of transformations to the users' programs, since fixes may not always be semantics preserving, as they may change not only the way in which a program operates, but also its overall input/output function. Since the automated tool may not be designed to consider the specification of the program, there is the rirk of introducing new bugs into a currently working system. Bug patterns usually identify scenarios that are likely to be a bugs, rather than being guaranteed to be so. In this context, the conservative approach is to not alter the program, but simply suggest bug fixes to the user.

It may be at times difficult to instrument a bug-finding/fixing tool, and ideally potential users should be assumed to have little experience or understanding of the system in order to productively use it. Their existing development tools may incorporate some way of reporting suggested improvements to code, and

```
BufferedReader in = null;
try {
    in = new BufferedReader(
        new FileReader(''foo''));
    String s;
    while((s=in.readLine()) != null) {
        System.out.println(s);
    }
    // (1) close mistakenly placed
    in.close();
} catch (Exception e) {
    e.printStackTrace();
} finally {
// (2) the close should be placed with
// guarded by a null check
    if(in != null) {
        try {
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 2: Possibly Unclosed File Handle

```
try {
    conn.setAutoCommit(false):
    ....
    conn.commit();
} catch(java.sql.SQLException e) {
    if(conn != null) {
        try {
            conn.rollback();
        } catch (java.sql.SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 3: JDBC Commit and Rollback Pattern

these should still be supported. When using a more sophisticated bug-fixing tool, the user could simply see contextual and appropriate descriptions of the transformations, rather than their formal specification. In this context, the tailoring and deployment of bug-fixing techniques would be an activity undertaken by a few key team members, rather than necessarily every developer.

The inclusion within the development cycle of phases dedicated to improving code quality, such as the refactoring phases promoted by some agile methodologies, provides bug fixing program transformations with a suitable hook on which to integrate themselves to existing methodologies. Within a more traditional, waterfall, development model such an approach could be useful during a testing phase, after the program has been mainly written, but before it is shipped to customers.

The bug-fixing methodology described in this paper fits in particularly well with modern agile software engineering methodologies, such as Extreme Programming, which have increased focus on the quality of the code itself. Application of best practices, unit testing, many eyes reading code through pair programming, etc. all attempt to reduce bugs cropping up within the program being developed by reducing the likelihood of the programmer writing bugs. Whilst these developments have been of positive benefit to programmers, experience shows that bugs still occur.

Our implementation, described in Section 4, uses the Eclipse toolkit's intermediate representation to perform program transformation. This enables the production of source code that is formatted according to users' preferred style guidelines and integrates into the context in which programs are being developed, and ensures that the generated code requires no further formatting.

While we have incorporated a few common bugs into FixBugs, the aim is to provide a *framework* in which more bugs can be accounted for. The designing of new transformations is eased compared to traditional static analysis systems since the programmer doesn't have to implement a detailed static analysis and transformation toolkit in order to achieve their specific goal. Since the program transformations themselves are merely syntactic substitutions, it should be relatively natural for any experienced programmer to tailor the system to common bugs in their application area.

The FixBugs approach isn't intended to subsume traditional debugging techniques such as testing, or traditional formal analysis techniques such as static analysis and model checking. Its integration into existing tools and techniques should complement their usage, allowing automated FixBugs sweeps of the code to be made in order to offer potential improvements to the code base. Bugs can be found as early as possible through these automated tools, rather than being identified later through failing test cases, at a much higher cost.

3 A Language for Detecting and Fixing Bugs

3.1 Basis: the TRANS language

In previous work concerned with the application of formally specified optimizations on bytecode programs [Warburton and Kalvala(2009)], we developed and extended Lacey's TRANS language [Lacey(2003), Kalvala et al.(2009)Kalvala, Warburton, and Lacey]. In TRANS, compiler optimisations are represented through two components: a rewrite rule and a side condition which indicates the situations in which the rewrite can be applied safely.

Temporal logic is used for specifying side conditions under which a transformation may apply. Temporal logics traditionally describe properties of a system relative to a point in time, but in TRANS the points of interest are nodes (or program points) in a *control flow graph* (or CFG) representing a program. A logical judgement of the form: ϕ @ n states that the formula ϕ is *satisfied* at node n of the control flow graph.

The language for expressing conditions is based on CTL [Clarke and Emerson(1982)], a path-based logic which can express many optimisations while still being efficient to model-check. More specifically, a variant of CTL including past temporal operators ($\overleftarrow{E}$ and $\overleftarrow{A}$) is used, to make it easier to specify properties of programs, and the next state operators (EX and AX) are extended with information on the kind of edge they operate over. For example, the operators EX_{seq} and AX_{branch} stand for "there exists a next state via a *seq* edge" and "for all next states reached via a *branch* edge" respectively.

Two types of these basic predicates can be used to obtain information about a node in the control flow graph. The formula $node(x)$ holds at a node n in a valuation that maps n to x. The formula $stmt(s)$ holds at a node n where the valuation makes the pattern s match the statement at node n. As well as judgements about states, the language can make "global" judgements. For example, the formula ϕ @ $n \ \wedge conlit(c)$ states that ϕ holds at n and c is a constant literal throughout the program.

User defined predicates can be incorporated via a simple macro system. These can be used in the same way as core language predicates, and are defined by an equality between a named binding and the temporal logic condition that the predicate should be 'expanded' into.

3.2 From TRANS to TRANS$_{\text{fix}}$

We describe a variant of the TRANS language, called TRANS$_{\text{fix}}$, suitable for specifying the transformation of Java source code with the aim of correcting bugs that may appear within programs. In contrast to the TRANS language for optimisations, where the goal is to produce optimized *low-level* code, TRANS$_{\text{fix}}$ is used to produce source code, since the goal of debugging is usually to maintain reusable and readable source code, for the developers of the software to continue working on. Rather than operating on the low-level code which is used as input for the temporal logic side conditions, rewrite rules must operate on the *source*

$$\begin{aligned}\text{let nullable} \quad = \quad & \neg\mu Z.\ \exists x.\ \mathit{use(x)}\ \wedge \\ & \overleftarrow{\mathtt{AX}}(\mathtt{A}(\neg \mathit{def(c)}\,\overleftarrow{\mathtt{U}}\,\mathit{def(c)} \wedge \\ & \mathit{not\_assigned\_null(x)} \vee Z))\end{aligned}$$

Figure 4: Macro Within Nullable Checking

program itself.

$\mathsf{TRANS_{fix}}$ specifications consist of actions and conditions: if the condition holds true then the action is applied. Many actions consist of replacing statements with other statements, although they can also include adding new methods to classes. Actions are applied if side conditions hold true.

We also allow the use of μ-calculus [Kozen(1983)] to describe properties about nodes. Modal μ-calculus is a fix point based temporal logic with least (μ) and greatest (ν) fix point operators. It is interpreted over labelled transition systems, and in our approach we rely on the same correspondence between Kripke Structures and Control Flow Graphs. In other words the CFG is the model for our μ-calculus. An example use of μ-calculus is given in Figure 4. Steffen uses this logic to generate dataflow analyses from temporal logic [Steffen(1993)]. Our work continues this line of work and applies such ideas to the field of static program analysis and bug fixing.

A BNF for the $\mathsf{TRANS_{fix}}$ pattern matching language is provided in Figure 8. Interesting aspects of $\mathsf{TRANS_{fix}}$ are its use of metavariables, the new actions and strategies, and the type system.

3.2.1 Metavariables and wildcards

The core syntax of the rewrite rules is based on standard programming constructs (assignment statements, while statements, if statements, etc) which we assume are well understood. The syntax is expanded with constructs to support meta-variables, representing either syntactic fragments of the program or nodes of the CFG.

The language for transformations is a `Java` statement grammar, extended with metavariables that can bind to different program structures, and wildcards that can match any statement or sequence of statements. For example, the pattern for matching an integer assignment to an addition expression, that is later followed by re-assignment to that variable, is shown in Figure 5. Figure 6 gives a code snippet which matches to that pattern, and metavariable bindings that show how the pattern is matched.

The language for code reconstruction is the same as pattern matching. Its application is fundamentally different, however. In reconstruction metavariables are substituted with a statement, expression or type that has been bound to the metavariable during pattern matching, and model checking. Each statement in the syntax tree isomorphically corresponds to a node within the CFG, which

```
n: int x = l + r;
....
m: x = e;
```

Figure 5: TRANS$_{\text{fix}}$ Pattern Matching

```
int z = y + 5;
System.out.println(x);
z = z + 1;
```

Metavariable	Binding
`x`	`z`
`l`, `r`	`y`, `5`
`e`	`z + 1`

Figure 6: Sample `Java` Code Listing

enables the use of the results of model checking the side conditions in the code reconstruction.

A consequence of the desire to produce source code is the necessity of incorporating *scoping*; while scoping doesn't exist within methods at a bytecode level, is a necessary part of the transformation language of TRANS$_{\text{fix}}$. This allows us to match programming language constructs such as `try` and `catch` blocks.

The TRANS$_{\text{fix}}$ language contains a wildcard operator "`....`" that matches against any statement or sequence of statements, including no statements. Since a wildcard statement is a normal pattern matching statement, it can also be bound using a label, allowing the matching or arbitrary blocks of code in strategic locations. In order to facilitate the writing of specifications that are intuitive to programmers, we also allow wildcards to be used in the reconstruction of statements. This is syntactic sugar for binding the wildcard statements to metavariables using labels, and then substituting in metavariable references within the reconstruction pattern. Figure 7 gives an example translation. Wildcard substitutions are indexed, so the nth wildcard block in pattern matching is substituted into the nth wildcard position in reconstruction.

3.2.2 `Java` Types

We provide pattern matching for `Java` types as well. The pattern `::m` will bind any type to the metavariable `m`. One can explicitly refer to primitive types, such as `int` or object types, such as `java.util.Vector`. One can also match arrays. The two `new` calls within the expressions grammar allow pattern matching array initialisers specifically.

```
REPLACE
    l.lock()
    ....
    l.unlock()
WITH
    try {
        l.lock()
        ....
    } finally {
        l.unlock()
    }
```

Before

```
REPLACE
        obj.lock()
    _1:    ....
        obj.unlock()
WITH
    try {
        obj.lock()
        `_1`
    } finally {
        obj.unlock()
    }
```

After

Figure 7: Removing Wildcard Reconstruction Sugar

3.3 Actions

A simple rewrite merely replaces code snippets with new code; however, many transformations must actually change the structure of the `class` or apply rewrites at multiple places. These structural changes are supported by additional actions.

The `ADD_METHOD` action takes the return type of the method, its name, arguments and a statement to act as the body. This is then added to a class, specified through a *metavar*. This is our primary method of transforming classes.

The `COMPOSE` action performs sequential composition on the two actions that it is passed as arguments and forms a new atomic action. (This is not to be confused with the `THEN` *strategy* (see below) for composing two *transformations*.) Note that these actions are both disabled if the side condition doesn't hold true for a given set of `metavar` bindings.

Combining uses of actions has many applications, for example one could rewrite a block of code into a method, and replace it with a call to this method, by using a `REPLACE` composed with an `ADD_METHOD`.

A non-deterministic choice action, called `CHOOSE ... OR`, is used when the same analysis might suggest more than one possible fix. This fits in with the methodology of debugging we propose since user must confirm the application of a transformation, thus they may be in a better position to make that choice.

3.3.1 Strategies

As in the TRANS language, *strategies* are operators for combining different transformations. The MATCH ϕ IN T strategy restricts the domain of information in the transformation T by the condition ϕ. The T_1 THEN T_2 strategy applies the sequential composition of T_1 and T_2. When actions are applied normally, ambiguity with respect to what node actions and rewrites are applied to are automatically resolved. In other words, if there are several bindings that have the same value for a node attribute that is being used in a rewrite rule then only one of them is non-deterministically selected. The APPLY_ALL T strategy

```
type            ::=  :: metavar
                 |   primitive-type
                 |   object-type
                 |   type []

expr-pattern    ::=  metavar ( expression, expression ...  )?
                 |   expression op expression
                 |   unop expression
                 |   (type) expression
                 |   new type expression
                 |   expression instanceof type
                 |   new type []

statement       ::=  metavar: statement
                 |   ....
                 |   ;
                 |   ‘ metavar ‘
                 |   type metavar = expression
                 |   if  expression statement statement
                 |   while  expression statement
                 |   try  expression catch statement
                         finally statement
                 |   return expression ;
                 |   expression ;
                 |   { statement* }
                 |   return expression ;
                 |   throw expression ;
                 |   synchronized (expression) { statement  }
                 |   for (expression*, expression, expression*)
                         { statement  }
                 |   switch (expression) { statement*  }
                 |   case expression: statement ;
                 |   default ;
                 |   assert expression ;
                 |   continue metavar ;
                 |   break metavar? ;
                 |   this ( expression, expression ... );
                 |   super ( expression, expression ... );

node-condition  ::=  μ condition-var. node-condition
                 |   ν condition-var. node-condition
                 |   node-condition ∨ node-condition
                 |   node-condition ∧ node-condition
                 |   ¬ node-condition
                 |   ∃ metavar . node-condition
                 |   $[EX \mid AX \mid \overleftarrow{EX} \mid \overleftarrow{AX}]_{[metavar]}$
                         (node-condition)
                 |   $[E \mid A \mid \overleftarrow{E} \mid \overleftarrow{A}]$(node-condition
                          U  node-condition)
                 |   node(metavar)

side-condition  ::=  side-condition ∨ side-condition
                 |   side-condition ∧ side-condition
                 |   ¬ side-condition
                 |   node-condition @ metavar
                 |   pred (metavar_1      metavar_n )
```

uses all of the valuations within transformation T, without this restriction.

3.4 Type System

$\mathsf{TRANS}_{\mathsf{fix}}$ is endowed with a simple type system that ensures that programs transformed by a $\mathsf{TRANS}_{\mathsf{fix}}$ specification are syntactically valid `Java` programs. For example, anything nested at an expression level is an expression. It doesn't guarantee that the programs output are well typed `Java` programs. We cannot ensure output programs are correctly typed because strategies (transformational combinators), such as `THEN`, may be used to combine a transformation that fixes an incorrect program.

In order to differentiate types of meta-variables being used in transformations from the types of `Java` variables, we refer to the former types as *kinds*. The kind system provides guarantees that can be used in our implementation, see Section 4. There are three Kinds within the kind system:

`Type Kind` for metavariables that bind to `Java` types

`Expression Kind` for metavariables used for `Java` expressions

`Statement Kind` for statements and blocks.

The kind system guarantees two important properties:

1. that no metavariable may bind to, or substitute into a position that requires more than one Kind.
2. that no metavariable may be used in a substitution, if it is not bound before hand.

A relatively simple algorithm is used to check these properties. A pass is made of the syntactic replacement rules and side conditions, keeping note of what context a metavariable is used in. If a metavariable is used in a context which implies it would need to be of more than one Kind, then kind checking fails. If there exist metavariables referred to in the substitution part of replacement that isn't bound by either the pattern matching, or the side condition then it also fails.

3.5 Specification Examples

3.5.1 Method does not release lock on all paths

The full specification is provided in Figure 9. Position `l` within the program matches the point at which the lock is locked, and `u` at the position where its unlocked. The side condition holds true where you can sometimes unlock if you have locked, but not on every path. The replacement rule moves the unlock statement within a finally clause, ensuring that the lock gets executed on all paths through the method.

```
REPLACE
        l.lock()
        ....
        l.unlock()
WITH
    try {
        l.lock()
        ....
    } finally {
        l.unlock()
    }
WHERE
```

$$\mathtt{EF}(u) \wedge \neg \mathtt{AF}(u) @ \ \mathtt{l}$$

Figure 9: Transformation to ensure lock released on all paths

3.5.2 Database Transactions

Figure 10 shows a specification for ensuring that transactions are surrounded by the correct catch pattern for `SQLException` instances. The pattern matching of a call to the `setAutoCommit` method, matches the beginning of the transaction. The wildcard binds to anything between that and the `commit` call, i.e. a whole transaction. This block of code is then replaced with another block, surrounded by a catch statement. The catch statement rolls back the transaction in case of a database failure.

The side condition checks to ensure that the `commit` call can never be followed by a `rollback`. It also ensure thats `conn` is of the correct type.

3.5.3 Unclosed File Handles

Figure 11 gives a specification that rearranges the closing mechanism for file handles. It matches the type of the stream object into the metavariable `streamtype` and ensures this is a stream in the side condition. The other component of the side condition ensures that the close method throws an exception.

It uses wildcard matching to keep the body of the try block in place, whilst moving the `close` call at the end of the method within a `finally` block - ensuring that it always gets called.

4 Prototype Implementation

The approach proposed in this paper, based on specifying bug fixes with $\mathsf{TRANS}_{\mathsf{fix}}$ and matching the specifications against low-level program representations, has been prototyped in the implementation we call FixBugs. This implementation takes a `Java` program in both source and `Bytecode` form and applies transformations to it, outputting a series of programs representing possible bug-fixed variants of the program.

```
REPLACE
          conn.setAutoCommit(false):
          ....
commit:   conn.commit();
WITH
     try {
          conn.setAutoCommit(false):
          ....
          conn.commit();
     } catch(java.sql.SQLException e) {
          if(conn != null) {
               try {
                    conn.rollback();
               } catch (java.sql.SQLException e) {
                    e.printStackTrace();
               }
          }
     }
WHERE
  type(conn,'java.sql.Connection') ∧¬AF(stmt(conn.rollback();))@
                                   commit
```

Figure 10: Correction for JDBC Commit and Rollback Pattern

4.1 Architecture

As shown in Figure 12, the FixBugs system comprises several components:

- the Core parses $\text{TRANS}_{\text{fix}}$ specifications, and calls into various components as required;
- the Pattern Matcher produces bindings to metavariables from source code and a pattern;
- the Model Checker produces bindings to metavariables that satisfy the side condition formulae; and
- the Generator alters the program itself, given bound metavariables, according to the actions.

The Model Checker relies on the ASM bytecode library ['Eric Bruneton et al.(2002)'Eric Bruneton, Lenglet, in order to generate the control flow graph of the program, as explained in Section 4.3. The Java programs source code is parsed using the Eclipse [Eclipse Foundation(2009)] project's Java developer tools. These provide a standardised intermediate representation for the programs. This representation is also used by the Generator, which manipulates this representation directly, and concrete syntax is generated from this abstract syntax.

```
REPLACE
    ::streamtype stream = null;
    try {
        ....
        throw: stream.close();
    } catch (ex e) {
        c: ....
    }
WITH
    ::streamtype stream = null;
    try {
        ....
    } catch (ex e) {
        ....
    } finally {
        if(stream != null) {
            try {
                stream.close();
            } catch('IOException' e) {
                e.printStackTrace();
            }
        }
    }
WHERE
          subtype(streamtype,'java.io.OutputStream') ∧
                    EF (node(c)) @ throw
```

Figure 11: Closing File Handles

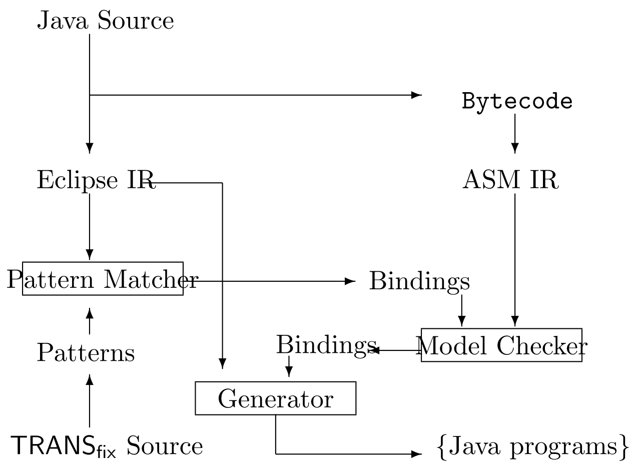

Figure 12: FixBugs Architecture

4.2 Representation

An important issue in writing static analysis systems is the representation over which the analysis is performed, notably whether at source code level, object code level or some intermediate representation. In order to bug-fix the programs themselves (rather than a low-level representation) it is necessary to perform the transformation at the *source code* level. Many existing systems for detecting bugs perform analysis at the *bytecode* level, and thus have difficulty incorporating automatic fixes to programs. There are many advantages, however, to performing analysis at a lower level: for example, it is easier to extract the control flow graph from a language whose control flow is represented by conditional goto statements, rather than loops.

We attempt to blend the best of both worlds with our approach to analysis. We perform syntactic analysis against the source code of the program, whilst performing semantic analysis on a bytecode representation. We use the standard debugging information from the `Java Bytecode` format in order to correlate the results from the source and `Bytecode` analyses.

4.3 Silhouettes

One line of `Java` source code is compiled into one or more lines of `Java Bytecode`. Consequently there is a subtle impedance mismatch between two systems when using the debugging information to bridge the analysis results of these two representational levels. We unify these levels within FixBugs through the concept of a *silhouette*. The silhouette of a line of source code is the corresponding set of lines of its bytecode. This concept is reflected within all aspects of the analysis. For example the control flow graph silhouette of a source code line is the subgraph within the control flow graph that corresponds to that source code line. Every edge within the control flow graph of the program's source code has a corresponding edge within the bytecode control flow graph, but the inverse relation doesn't hold.

Silhouettes consequently partition the `Bytecode` control flow graph into several overlapping subgraphs. The edges between these subgraphs fall into two categories. An edge *(from,to)* is *inbound* with respect to some silhouette S if the *to* node, but not the *from* node is a member of S, it is *outbound* if the *from* node is a member of S, but not *to*. If both *from* and *to* are within S we say that the edge is *contained* within S. The relation between source code and bytecode CFGs is illustrated in Figure 13.

We can minimise the `Java` control flow graph from the `Bytecode` representation very simply with the following steps:

1. extract `Bytecode` control flow graph (G) using ASM.
2. compute line numbering function (L) using ASM.
3. coalesce (G) to form (G').

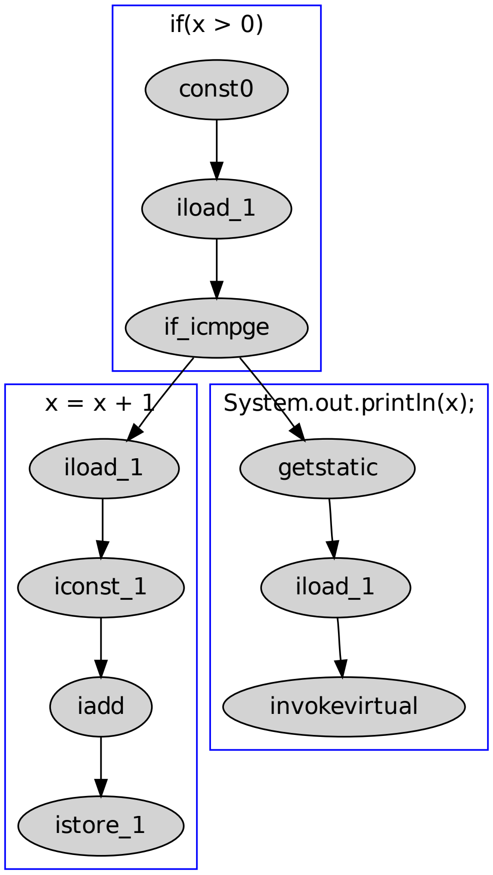

Figure 13: CFG Coalescing

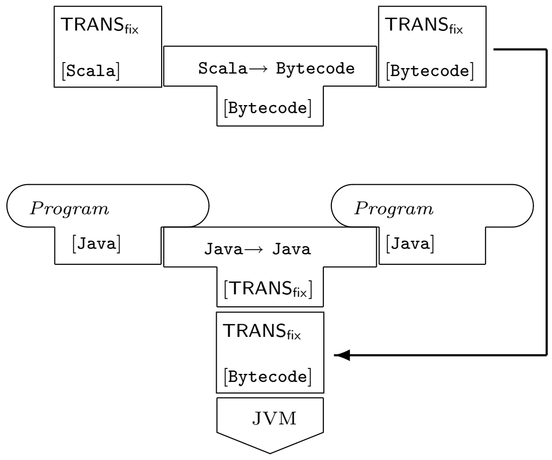

Figure 14: Transformational Diagram for FixBugs

Within the FixBugs implementation we represent the successor function of G as a map from integers onto sets of integers, and L as an array of integers. In order to calculate G' we therefore replace every edge *(from,to)* in G with an edge *(L(from),L(to))*. This ensures all inbound and outbound edges are replaced accordingly. We then remove all edges whose *from* and *to* nodes are identical, since they represent contained edges that don't exist within the source code control flow graph G'.

The existence of `Bytecode` analysis libraries, such as ASM makes it easier to extract the control flow graph and coalesce than to write a custom source code analysis. It also allows us to integrate other information more easily extracted at a `Bytecode` level, and then relabel it onto the `Java` control flow graph accordingly.

4.4 Implementation Details

Most of the software is written primarily in `Scala`, chosen because of its support for a functional style of programming, combined with the plentiful libraries that are available on the `Java` platform. Specification files are parsed using the parser combinators in `Scala`'s standard library, and disjoint union datatypes, modelled using case classes provide an intermediate representation for TRANS$_{\text{fix}}$ specifications. `Scala`'s pattern matching can then be used in order to bind TRANS$_{\text{fix}}$ metavariables to elements of `Java` source code, represented using Eclipse's Intermediate Representation. This development approach is described in Figure 14.

Being a prototype, the current implementation doesn't provide support for all the features of the TRANS$_{\text{fix}}$ language, such as strategies, μ-calculus and class-level actions. The gist of the approach, however, should map directly to these concepts, albeit with some programming effort.

4.5 Performance

While we are happy with the performance of this prototype implementation in practice (applying the bug fixing transformations usually takes in the order of seconds) we have yet to complete an analysis of its computational complexity.

Computational Tree Logic is polynomial time checkable in the size of the system times the length of the formula [Clarke et al.(1996)Clarke, Emerson, and Sistla]. These correspond to the number of statements in the program being transformed, and the side condition of the transformational specification. Modal μ-calculus is exponential time checkable, however the worse case scenario is only reached in the case of nesting ν and μ operators within each other. Such a temporal side condition is only really necessary in complex analyses, which may require exponential time dataflow analysis algorithms themselves. The alternation free subset of modal μ-calculus is linear time checkable in the size of the model times the size of the specification [Cleaveland and Steffen(1993)]. Our pattern matching, and reconstruction implementations are both linear in the size of the pattern plus the size of the method.

We provided a thorough investigation of the performance of several common compiler optimisations specified in TRANS and compared it to existing hand written dataflow analyses in [Warburton and Kalvala(2009)]. In general, generated optimisations are 2x slower than hand written optimisations to apply to `Java` bytecode programs, however, some pathological cases exist that cause worse performance.

There are several differences between the TRANS implementation in [Warburton and Kalvala(2009)] and the $\mathsf{TRANS}_{\mathsf{fix}}$ implementation in this paper. The TRANS implementation compiles specifications, rather than interpreting them, and it also uses Binary Decision Diagrams in order to symbolically represent the state space of the analysis, rather than the explicit model checking we perform. These differences reflect the prototype nature of the implementation described here compared to the relative completeness of the TRANS implementation. The use of TRANS in optimization must consider more states than the proposed use of $\mathsf{TRANS}_{\mathsf{fix}}$, since reducing silhouettes to a source code CFG reduces the number of nodes within the graph, as several bytecode statements may correspond to one source code statement.

5 Analysis

5.1 Related Work

FindBugs is a system for detecting bugs within `Java` programs [Hovemeyer and Pugh(2004)]. It defines a concept of a bug pattern, which is a common construct within a program that commonly causes errors. Misunderstood API features, and difficult language features are good examples of bug patterns. Findbugs detects these patterns through static analysis, but does not attempt to fix them. Its bug detection mechanisms are hand written in Java.

UCDetector[1] is a plugin for the commonly used Eclipse `Java` IDE that finds unecessary code within a project. Its detection mechanism is a custom dead code static analysis. It can also detect when the visibility of a method can be restricted, for example from `public` to `private`. It can automatically fix the dead code issues that it detects, but only performs limited analysis of the programs.

The Netbeans IDE will in future have a system for migrating users away from deprecated method calls, by automatically applying a source code transformation that rewrites a call to a deprecated method into a different method call. These transformations are specified in an anotation to the method definition. The transformation language is simplified, allowing constraints on argument types, and a few specialised metavariables for substitution, for example `$0` represents the object that is calling the method.

Dataflow analysis has long been employed within the compiler optimisation community to iteratively compute the nodes within a program at which optimisations can be soundly applied [Aho et al.(2007)Aho, Lam, Sethi, and Ullman, Muchnick(1997)]. Model checking is a technique in which a decision is made as to whether a given model satisfies some specification. David Schmidt and Bernhard Steffen recognised that there is a strong link between these two research areas. Equations for dataflow analyses have been shown to be expressible in modal μ-calculus [Schmidt and Steffen(1998)], and dataflow analysis algorithms have been generated from modal logics [Steffen(1993)]. This approach is implemented in DFA & OPT-Metaframe [Klein et al.(1996)Klein, Knoop, Koschutzki, and Steffen], a toolkit designed to aid compiler construction by generating analyses and transformations from specifications. Transformations within this system are implemented imperatively, rather than using declarative style rewrite rules, however, the temporal logic specification is converted into a model checker and then optimised. In our case, we found CTL to be sufficient to model the side conditions of transformations.

Rewrite rules with temporal conditions have also been used in the Cobalt system [Lerner et al.(2003)Lerner, Millstein, and Chambers] which focuses on automated provability and also provides executable specifications, achieved through temporal conditions common to many dataflow analysis approaches. Since Cobalt is designed for generating compiler analyses and transformation it has a focus on automatically discharging proof obligations for program equivalence. The specific nature of Cobalt's temporal conditions, while facilitating automatic discharging of proof obligations, is limited compared to the flexibility provided in $\mathsf{TRANS}_{\mathsf{fix}}$ from supporting CTL side conditions, even if this may require more expensive model checking.

Rhodium is another domain specific language for developing compiler optimisations [Lerner et al.(2005)Lerner, Millstein, Rice, and Chambers]. Rhodium consists of local rules that manipulate dataflow facts. This is a significant departure in approach from TRANS, since it uses more traditional, dataflow analysis based specifications rather than temporal side conditions.

[1] `http://www.ucdetector.org/`

The Temporal Transformation Logic (TTL) [Kanade et al.(2006)Kanade, Sanyal, and Khedker] also uses CTL, but emphasizes verification of the soundness of the transformations themselves, ie that they are semantics preserving. Accordingly, instead of approaching optimisation as rewriting, TTL has a set of transformational primitives, each representing a common element used within compiler optimisations, for example replacing an expression with a variable. Each of the transformational primitives has an associated soundness condition that, if satisfied, implies the soundness of the transformation.

5.2 Optimisations

Some existing compiler optimisations can be used to remove potential bugs, or unclear code within programs, for example dead assignment removal, or unreachable code elimination. These can be specified within the existing TRANS system, of which $\mathsf{TRANS}_{\mathsf{fix}}$ is an extension.

Since these would be semantics preserving optimisations, there is less concern about applying them automatically. However user feedback might still be beneficial here, since a user may have written a method, and wish to keep it within their codebase, but may not have started to use it within their code. Consequently, removal of dead code, even if semantics preserving, should be applied with care.

Other optimisations specified in TRANS include lazy code motion, constant propagation, strength reduction, branch elimination, skip elimination, loop fusion, and lazy strength reduction; further details of these can be found in [Kalvala et al.(2009)Kalvala, Warburton, and Lacey].

5.3 Correctness Issues

Unlike compiler optimisations, transformations applied to fix bugs are not semantics preserving. The very aim of the transformation is to alter the program semantics in order to remove a bug. Consequently one is assuming that the program itself is incorrect according to some specification, but can be corrected to match this specification.

It is possible that the program itself might be correct, and accordingly the transformations shouldn't be applied automatically. Additionally the bug finding patterns that we focus on correspond to behaviours that are generally considered bugs within a program, for example deadlocks.

5.4 Further Applications

Other elements of IDE and language analysis tool construction can also benefit from the source transformation language we outline here. Refactoring operations are an interesting example of such a transformation.

A refactoring operation attempts to change the structure of a program internally, in order to improve readability or maintainability, without altering the observable functional behaviour of the program, for example by extracting some block of code into a named method. In this context existing formal analysis of TRANS-like languages could be useful for ensuring observational equivalence.

Refactoring operations require further information from the user of the IDE that $\mathsf{TRANS}_{\mathsf{fix}}$ doesn't provide, for example in the *Extract Method* refactoring operation one would need to know what the name of the method is. In order to automatically apply such transformations the concept of schematic variables is introduced.

A schematic variable is a variable that isn't bound in the pattern matching or temporal constraints, we syntactically differentiate schematic variables by prefixing them with a ? symbol, such as `?x`. We can use the type system described in Section 3.4 and underlying syntactic structure of the transformation to infer the type of any schematic variable. When applying the transformations we can use the schematic variables to display appropriate user interface dialogs in a given IDE.

5.5 Further Work

We intend to work further on the implementation:

1. Improve the performance by implementing a symbolic model checker, or backing onto a SAT solver.

2. Complete the implementation of language features, for example schematic variables and strategies.

3. Integrate into IDEs, in order to be able to use the tool effectively, rather than to simply experiment with $\mathsf{TRANS}_{\mathsf{fix}}$.

We would like to extend our methodology in order to be able to ensure that we are soundly applying transformations, rather than simply leaving the choice of whether to apply these transformations to the user of the tool. The required soundness properties could be annotated onto the program. For example our specification for ensuring that locks are released on all paths is sound *iff* the user of the system wishes a lock to be in a released state as a post-condition of the method. Information of this nature can already be added to `Java` programs using the existing annotations framework, that has been recently extended by [Ernst(2009)]. There are already existing tools for invariant detection about partially annotated `Java` programs, [Ernst et al.(2007)Ernst, Perkins, Guo, McCamant, Pacheco, Tschantz, and X
infers properties about nullness of variables. Another element of such an extension would be the ability to automatically infer the soundness of transformations with respect to given pre and post condtions, progress has been made towards the inverse goal, for example [Scherpelz et al.(2007)Scherpelz, Lerner, and Chambers] provides a system for automatically inferring dataflow analyses from facts.

5.6 Conclusions

We have introduced an approach that allows one to specify static analyses that can be applied to programs, and transformations that can be used to debug the programs. We describe a tool that allows the automated application of these transformations to programs and how its use can be integrated into existing development methodologies. Our implementation uses a novel technique for combining source code and object code analysis through *silhouettes*—a technique for unifying information annotated onto a control flow graph. This provides the same underlying model as the $\mathsf{TRANS}_{\mathsf{fix}}$ specification language for transformations.

We hope that in future this approach can be combined with partial program specifications in order to place the automated fixing of bugs through formally specified program transformation on a sound semantic footing. The codifying of common bug patterns in itself helps programmers to understand and appreciate the art of computer programming through the subtleties of its science.

References

[Aho et al.(2007)Aho, Lam, Sethi, and Ullman] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education;, 2nd edition, 2007.

[Clarke and Emerson(1982)] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag. ISBN 3-540-11212-X.

[Clarke et al.(1996)Clarke, Emerson, and Sistla] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1996.

[Cleaveland and Steffen(1993)] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2(2):121–147, 1993.

[Eclipse Foundation(2009)] Eclipse Foundation. Eclipse website, 2009. `http://www.eclipse.org`.

['Eric Bruneton et al.(2002)'Eric Bruneton, Lenglet, and Coupaye] 'Eric Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Proceedings of the ASF (ACM SIGOPS France) Journ'ees Composants 2002 : Syst'emes 'a composants adaptables et extensibles (Adaptable and extensible component systems)*, 2002.

[Ernst(2009)] M. D. Ernst. Type Annotations Specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, October 5, 2009.

[Ernst et al.(2007)Ernst, Perkins, Guo, McCamant, Pacheco, Tschantz, and Xiao] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S.

Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

[Hovemeyer and Pugh(2004)] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[Kalvala et al.(2009)Kalvala, Warburton, and Lacey] S. Kalvala, R. Warburton, and D. Lacey. Program transformations using temporal logic side conditions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(4), 2009.

[Kanade et al.(2006)Kanade, Sanyal, and Khedker] A. Kanade, A. Sanyal, and U. Khedker. A PVS based framework for validating compiler optimizations. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 108–117, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2678-0. doi: http://dx.doi.org/10.1109/SEFM.2006.4.

[Klein et al.(1996)Klein, Knoop, Koschutzki, and Steffen] M. Klein, D. Knoop, D. Koschutzki, and B. Steffen. DFA & OPT-METAFrame: A toolkit for program analysis and optimization. In *Procs. of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 422–426. Springer, 1996.

[Kozen(1983)] D. Kozen. Results on the proposition mu-calculus. *Theoretical Computer Science*, 27, 1983.

[Lacey(2003)] D. Lacey. *Program Transformation using Temporal Logic Specifications.* PhD thesis, Oxford University Computing Laboratory, 2003.

[Lerner et al.(2003)Lerner, Millstein, and Chambers] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation.* ACM Press, 2003. citeseer.ist.psu.edu/lerner03automatically.html.

[Lerner et al.(2005)Lerner, Millstein, Rice, and Chambers] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 364–377, New York, NY, USA, 2005. ACM Press. ISBN 1-58113-830-X. doi: http://doi.acm.org/10.1145/1040305.1040335.

[Muchnick(1997)] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[Scherpelz et al.(2007)Scherpelz, Lerner, and Chambers] E. R. Scherpelz, S. Lerner, and C. Chambers. Automatic inference of optimizer flow functions from semantic meanings. In *PLDI*, pages 135–145, 2007.

[Schmidt and Steffen(1998)] D. Schmidt and B. Steffen. Data-flow analysis as model checking of abstract interpretations. In G. Levi, editor, *5th Static Analysis Symposium*, volume 1503 of *LNCS*, September 1998.

[Steffen(1993)] B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21:115–139, 1993.

[Warburton and Kalvala(2009)] R. Warburton and S. Kalvala. From specification to optimisation: An architecture for optimisation of java bytecode. In O. de Moor and M. I. Schwartzbach, editors, *Compiler Construction, 18th International Conference*, volume 5501 of *Lecture Notes in Computer Science*. Springer, 2009.