A Formal Approach to Fixing Bugs *†

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

12 August 2011

Abstract

Bugs within programs typically arise within well-known motifs, such as complex language features or misunderstood programming interfaces. Some software development tools often detect some of these situations, and some integrated development environments suggest automated fixes for some of the simple cases. However, it is usually difficult to hand-craft and integrate more complex bug-fixing into these environments. We present a language for specifying program transformations which is 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 demonstrate the approach with a prototype implementation.

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, such as FindBugs [8], can detect some of the commonly repeated bugs in particular programming languages, and some extensions to integrated development environments (IDEs), such as the UCDetector plugin [18], 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 unusual or domain-specific bug detection mechanisms that also offers suggested fixes based on the specifications.

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

†This is a post-print of a book chapter to be published by Springer Verlag in Lecture Notes in Computer Science vol 7021. An error in Figure 4 has been corrected in this version.

In this paper we propose a temporal-logic based language 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 use Java as our example platform, though our methodology is applicable to many imperative languages.

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. 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. Therefore, many existing systems for detecting bugs perform analysis at the *bytecode* level, but then have difficulty incorporating fixes to source programs. 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.

The two characteristics of our work are therefore to support extensibility by allowing specification of new bug patterns, and correction of the original high-level programs. In this paper we show how to codify common bug patterns within a formally defined language based on temporal logic. We also simplify the construction of tools for static analysis of bug patterns, through model checking and rewriting.

In Section 2 we describe 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 a starting point the classification of common Java bugs due to Hovemeyer and Pugh [8], used in the description of the FindBugs tool which detects most of them. Many of the bugs identified by Hovemeyer and Pugh are simple

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

Figure 1: Pattern for correct locking

```
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 guard by a null check
       if(in != null) {
           try { in.close();
           } catch (IOException e) {
               e.printStackTrace();  } } }
```

Figure 2: Possibly Unclosed File Handle

and their identification requires merely a syntactic pattern matching system. Some of them, however, don't have obvious fixes. We especially consider some concurrency bugs, since they require more than simple syntactic pattern matching to be identified yet are amenable to temporal analysis.

Because of space limitations, in this paper we consider only three examples:

Method does not release lock on all paths This bug arises in a situation where a method acquires a lock, but there exists a path through the method where the lock isn't released. The `java.util.concurrent` lock, as specified in JSR-166, is considered by the authors of FindBugs. Fig. 1 illustrates the standard solution to this bug.

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. Fig. 2 shows an example of (1) where not to place a close and (2) where to place it correctly.

Failed database transactions may not be rolled back The JDBC library for database connections models the beginning, committing and ending of

```
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

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 Fig. 3. Another common problem is the failure to ensure that all paths end in either a commit or a rollback.

2.2 Placing debugging within software development

In general, a good approach to process 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. But if an automated tool is not designed to consider the specification of the program, there is the risk of introducing new bugs into a currently working system. Bug patterns usually identify scenarios that are *likely* to be buggy, rather than being *guaranteed* to be so. In this context, the conservative approach, which we adopt, is to not alter the program, but simply suggest bug fixes to the user.

Our implementation, described in Section 4 and which we call FixBugs, 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.

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 easier than in traditional static analysis systems since the programmer does not have to implement new detailed analysis and transformation steps. 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 temporal logic side conditions may be considered a difficult notation to grasp, but we believe it is a simpler and more intuitive way of formulating dataflow analysis, than hand writing the code directly.

The FixBugs approach is not 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, often at a much higher cost. 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 into current practice.

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 [20], we developed and extended Lacey's TRANS language [11, 9]. 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.

Side conditions are expressed in an extension of CTL [4], a path-based temporal logic which can capture many properties while still being efficient to model-check. 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. The variant of CTL used includes past temporal operators ($\overleftarrow{E}$ and $\overleftarrow{A}$), the final operators EF and AF, and the henceforth operators EG and AG. The next state operators are extended with information on the kind of edge they operate over: for example, 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.

A logical judgement of the form: ϕ @ n states that the formula ϕ is *satisfied* at node n of the control flow graph. 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 side condition that the predicate should be 'expanded' into.

3.2 From TRANS to TRANS$_{fix}$

We describe a variant of the TRANS language, called TRANS$_{fix}$, suitable for specifying the transformation of Java source code with the aim of correcting bugs that may appear within programs. In contrast to TRANS, where the goal is to produce optimized *low-level* code, TRANS$_{fix}$ 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. So 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 program* itself.

TRANS$_{fix}$ specifications consist of actions and side conditions: if the side condition holds 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.

A BNF for the TRANS$_{fix}$ pattern matching language is provided in Fig. 4. Interesting aspects of TRANS$_{fix}$ are its use of metavariables, the new actions and strategies, and the type system. 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 contains a Java statement grammar, extended with metavariables that can bind to different program structures. For example, the pattern for matching an assignment of a variable by an addition expression, that is later followed by re-assignment to that variable, is shown in Fig. 5(a). The code snippet shown in Fig. 5(b) matches that pattern, via the bindings shown in Fig. 5(c).

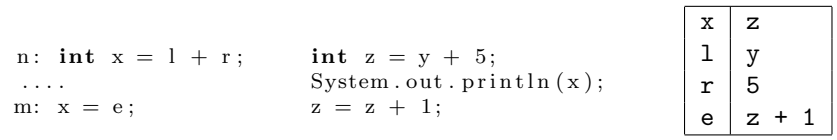
TRANS$_{fix}$ also contains a wildcard operator "`...`" that matches against any statement or (possibly empty) sequence of statements. Since a wildcard statement is a normal pattern matching statement, it can also be bound using a label, allowing the matching of arbitrary blocks of code in strategic 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. Wildcard substitutions are indexed: the nth wildcard block in pattern matching is substituted into the nth wildcard position in the reconstruction pattern.

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, it is a necessary part of the transformation language of TRANS$_{fix}$. Support of scoping allows us to match programming language constructs such as `try` and `catch` blocks.

Java types are also supported in the pattern matching. The pattern `:: m` binds 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

type	::=	:: *metavar* | *primitive-type* | *object-type* | *type* `[]`
expr-pattern	::=	*metavar* (*expression, expression ...*)?
	|	*expression op expression* | *unop expression*
	|	(*type*) *expression* | *expression* `instanceof` *type*
	|	`new` *type expression* | `new` *type* `[]`
statement	::=	*metavar*: *statement* | ‘ *metavar* ‘ | `....` | { *statement** }
	|	*type metavar* = *expression*
	|	`if` *expression statement statement*
	|	`while` *expression statement*
	|	`try` *expression* `catch` *statement* `finally` *statement*
	|	*expression* `;` | `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	::=	*node-condition* $\vee$ *node-condition*
	|	*node-condition* $\wedge$ *node-condition*
	|	$\neg$ *node-condition*
	|	$\exists$ *metavar* . *node-condition*
	|	stmt (*metavar*) | node(*metavar*)
	|	$[EX \mid AX \mid \overleftarrow{EX} \mid \overleftarrow{AX}]_{[metavar]}$ (*node-condition*)
	|	$[EF \mid AF \mid EG \mid AG]$ *node-condition*
	|	$[E \mid A \mid \overleftarrow{E} \mid \overleftarrow{A}]$ (*node-condition* U *node-condition*)
side-condition	::=	*side-condition* $\vee$ *side-condition*
	|	*side-condition* $\wedge$ *side-condition*
	|	$\neg$ *side-condition*
	|	*node-condition* @ *metavar*
	|	*pred* ($metavar_1$,...,$metavar_n$)
action	::=	REPLACE *statement** WITH *statement**
	|	COMPOSE *action* WITH *action*
	|	CHOOSE *action* OR *action*
	|	ADD_METHOD *type metavar*(*type metavar*, ...) *statement* TO *metavar*
transform	::=	*action* WHERE *side-condition*
	|	MATCH *side-condition* IN *transform*
	|	APPLY_ALL *transform*
	|	*transform* $\Box$ *transform*
	|	*transform* THEN *transform*

Figure 4: BNF for $\mathsf{TRANS}_{\mathsf{fix}}$



(a) the pattern (b) matching code (c) bindings

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

arrays. The two `new` calls within the expressions grammar specifically allow pattern matching array initialisers.

3.3 Actions

Simple rewriting merely replaces code fragments with new code, but 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 code 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 passed as arguments and forms a new atomic action. (This is not to be confused with the `THEN` *transformation* (see below) for composing two transformations.)

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 the user must confirm the application of a transformation, and can be given several choices.

3.3.1 Transformationss

are operators for combining different actions. The MATCH ϕ IN T transformation restricts the domain of information in the transformation T by the condition ϕ. The T_1 THEN T_2 transformation 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 transformation uses all of the valuations within transformation T, without this restriction.

3.4 Type System

$\mathsf{TRANS}_{\mathsf{fix}}$ has a simple type system to ensure 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. 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*. There are three types of *kinds*: `Type Kind` for metavariables that bind to Java types, `Expression Kind` for metavariables used for Java expressions, and `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, and
2. that no metavariable may be used in a substitution, if it is not bound beforehand.

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

3.5 Specification Examples

We re-visit the common bugs explained in Subsection 2.1 and show how typical fixes can be expressed in $\mathsf{TRANS}_{\mathsf{fix}}$.

Method does not release lock on all paths The full specification is shown in Fig. 6. Position `l` within the program matches the point at which the lock is locked, and `u` at the position where it is unlocked. The side condition holds 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.

Method may fail to close stream Fig. 7 gives a specification for rearranging 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. Wildcard matching is used to keep the body of the `try` block in place, while moving the `close` call at the end of the method within a `finally` block—therefore ensuring that there is a path where the `close` method throws an exception.

Failed database transactions may not be rolled back A specification for ensuring that transactions are surrounded by the correct catch pattern for

```
REPLACE
        l: m.lock()
        ....
        u: m.unlock()
WITH
    try {
        m.lock()
        ....
    } finally { m.unlock() }
WHERE
        EF (node(u)) ∧ ¬AF (node(u)) @ l
```

Figure 6: Transformation to ensure lock released on all paths

`SQLException` instances is shown in Fig. 8. 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 ensures that `conn` is of the correct type.

4 Prototype Implementation

The approach proposed in this paper 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 the source, outputting a series of programs representing possible bug-fixed variants of the program.

As shown in Fig. 9, the FixBugs system comprises several components. 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 Java programs source code is parsed using the Eclipse [5] project's Java developer tools, which provide a standardised intermediate representation for the programs. This representation is also manipulated by the Generator to produce bug-fixed programs in concrete syntax. The Model Checker relies on the ASM bytecode library [2] in order to generate the control flow graph of the program. ASM allows the manipulation of Bytecode at a programmer-friendly level of abstraction.

4.1 Silhouettes

One line of Java source code is typically compiled into several lines of Java Bytecode. Consequently there is a mismatch in the level of detail when using

REPLACE

```
::streamtype stream = null;
try {
    ....
    thro: 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)) @ thro

Figure 7: Closing File Handles

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 statement of source code is the corresponding set of commands of its bytecode. 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 does not 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 Fig. 10.

We can obtain 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').

Within FixBugs 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

```
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') ∧¬EF(stmt(conn.rollback();))@
                                commit
```

Figure 8: Correction for JDBC Commit and Rollback Pattern

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 use of the ASM Bytecode analysis library makes it easier to extract and coalesce the control flow graph than by writing 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.

4.2 Implementation Details

FixBugs is coded primarily in Scala [15], 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. Disjoint union datatypes, modelled using case classes, provide an intermediate representation for $\mathsf{TRANS}_{\mathsf{fix}}$ specifications. Scala's pattern matching can then be used in order to bind $\mathsf{TRANS}_{\mathsf{fix}}$ metavariables to elements of Java source code, represented using Eclipse's Intermediate Representation. This development approach is described in Fig. 11.

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

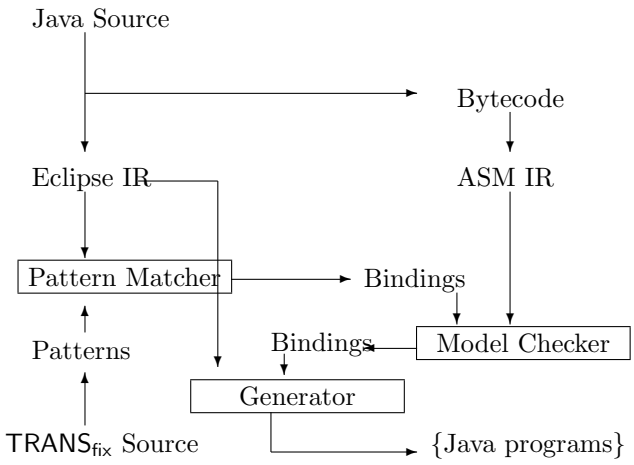


Figure 9: FixBugs Architecture

5 Analysis

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 exploits the same underlying model as the TRANSfix specification language for transformations.

While we are satisfied 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. CTL is polynomial time checkable in the size of the system times the length of the formula [3]. These correspond to the number of statements in the program being transformed, and the side condition of the transformational specification. Our pattern matching and reconstruction implementations are both linear in the size of the pattern plus the size of the method.

Before releasing the software to potential users, we intend to complete the following tasks:

1. Improve performance by making use of some existing symbolic model checker or boolean satisfiability solver.

2. Complete the implementation of language features, for example schematic variables and strategies, and extend to consider inter-procedural analysis.

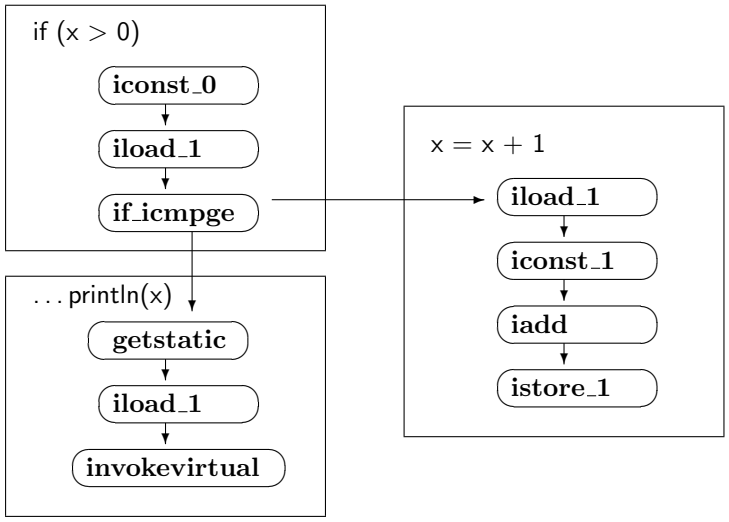


Figure 10: Relating Java statements with the control flow graph

3. Integrate into IDEs, in order to be able to use the tool effectively.

5.1 Related Work

In Section 2 we mentioned FindBugs, a system for detecting bugs within Java programs [8]. Bug patterns are defined as common constructs within programs that often causes errors. FindBugs detects these patterns through static analysis, but does not attempt to fix them. Its bug detection mechanisms are hand written in Java. UCDetector [18] is a plugin for the Eclipse 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 program analysis.

The use of predicates to identify program repair points is the basis of the work of Samanta *et al* [16]. Their approach relies on the use of standard pre- and post-conditions for a Boolean program and using propagation based on Hoare logic. This approach allows them to repair concurrent and recursive programs, and to reason about correctness. However, they haven't yet illustrated the approach on a full programming language, and do not show how language designers could extend the approach themselves by specifying new bug patterns.

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 [1, 14]. Schmidt and Steffen explain the strong link between dataflow analysis and model checking, and show how equations for dataflow analyses can be expressed in modal μ-calculus [17]. Steffen also

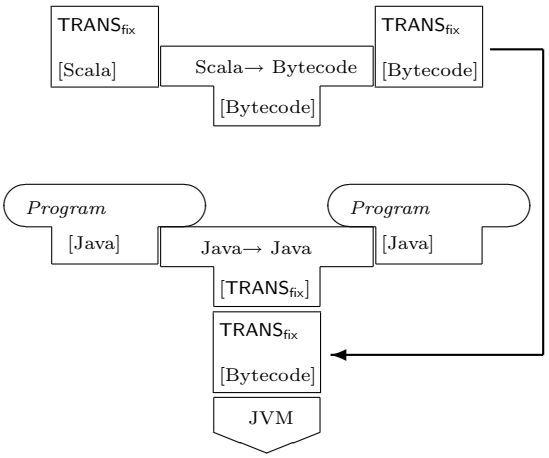


Figure 11: Transformational Diagram for FixBugs

shows how dataflow analysis algorithms can be generated from modal logics [19]. Rewrite rules with temporal conditions have also been used in the Cobalt system [12] which focuses on automated provability and also provides executable specifications, achieved through temporal conditions common to many dataflow analysis approaches. The specific nature of Cobalt's temporal conditions is limited compared to the flexibility provided in TRANS$_{fix}$ from supporting CTL side conditions, even if this may require more expensive model checking. Rhodium is another domain specific language for developing compiler optimisations [13]. 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. The Temporal Transformation Logic (TTL) [10] also uses CTL, but emphasizes verification of the soundness of the transformations themselves, *i.e.* that they are semantics preserving.

5.2 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 should not 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.

We plan to extend our methodology to identify transformations that can be applied soundly, rather than simply leaving the choice of whether to apply these transformations to the user of the tool. The required soundness proper-

ties 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, recently extended by [6]. There are already tools for invariant detection in partially annotated Java programs, [7] 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 conditions.

However, we recall that bug-repairing transformations often have to change the semantics of a program, and the goal of a formal tool should be seen primarily to facilitate the development of correct programs, rather than be constrained by existing specifications. This is the approach supported by the FixBugs tool.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2007.

[2] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, 2002.

[3] 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.

[4] Edmund M. Clarke and E. Allen 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.

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

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

[7] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

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

[9] Sara Kalvala, Richard Warburton, and David Lacey. Program transformations using temporal logic side conditions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(4), 2009.

[10] Aditya Kanade, Amitabha Sanyal, and Uday 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*, Washington, DC, USA, 2006. IEEE Computer Society.

[11] David Lacey. *Program Transformation using Temporal Logic Specifications*. PhD thesis, Oxford University Computing Laboratory, 2003.

[12] 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.

[13] Sorin Lerner, Todd Millstein, Erika Rice, and Craig 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.

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

[15] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, second edition, 2010.

[16] Roopsha Samanta, Jyotirmoy V. Deshmukh, and E. Allen Emerson. Automatic generation of local repairs for boolean programs. In *FMCAD*, 2008.

[17] D.A. Schmidt and B. Steffen. Data-flow analysis as model checking of abstract interpretations. In G. Levi, editor, *5th Static Analysis Symposium*, volume 1503 of *Lecture Notes in Computer Science*, September 1998.

[18] Jorg Spieler. UCDetector: the Unnecessary Code Detector website, 2007. URL: `http://www.ucdetector.org`.

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

[20] Richard Warburton and Sara Kalvala. From specification to optimisation: An architecture for optimisation of Java bytecode. In Oege de Moor and Michael I. Schwartzbach, editors, *Compiler Construction, 18th International Conference*, volume 5501 of *Lecture Notes in Computer Science*. Springer, 2009.