A Post-Placement Side-Effect Removal Algorithm

Mark Harman
Lin Hu
Rob Hierons
Brunel University
Uxbridge
Middlesex
UB8 3PH, UK

Malcolm Munro
Xingyuan Zhang
University of Durham
South Road, Durham
DH1 3LE, UK.

Jose Javier Dolado
Mari Carmen Otero
Facultad de Informática
University of the Basque Country
20009 San Sebastián
Spain

Joachim Wegener
DaimlerChrysler AG
Research and Technology
Alt-Moabit 96a
D-10559 Berlin
Germany

Keywords: Program Transformation, Program Comprehension, Side-Effects

Abstract

Side-effects are widely believed to impede program comprehension and have a detrimental effect upon software maintenance.

This paper introduces an algorithm for side-effect removal which splits the side-effects into their pure expression meaning and their state-changing meaning. Symbolic execution is used to determine the expression meaning, while transformation is used to place the state-changing part in a suitable location in a transformed version of the program. This creates a program which is semantically equivalent to the original but guaranteed to be free from side-effects.

The paper also reports the results of an empirical study which demonstrates that the application of the algorithm causes a significant improvement in program comprehension.

1 Introduction

A side-effect is any change of state which occurs when an expression is evaluated. In this paper we are concerned with the side-effect problem for C programs and will only[1] consider those side-effects which consist of the assignments to variables that occur when an expression is evaluated. Such side-effects are created using the assignment operator in an expression and the pre- and post-increment and decrement operators. In C the behaviour of side-effects is further complicated by the ability to sequence side-effects with the comma operator and to express conditional side-effect computation, both through short circuit evaluation of boolean operators and through the use of the conditional operator.

Side-effects are almost universally deprecated, particularly in the maintenance community, because of their harmful effect upon program comprehension and maintenance [5, 9, 16, 17, 22]. Despite this, reliance upon side-effect remains prevalent in production code.

Our approach allows the programmer to construct a side-effect free equivalent program from an original which contains side-effects. Programmers often rely upon side-effects (despite the advice against using them) because of perceived performance gains. Our approach allows us to have the best of both worlds. The programmer can execute the version of the program with side-effects, but use the side-effect free version for comprehension. Thus, the human is presented with a human-optimised version of the program, while the machine is presented with a machine-optimised equivalent.

In addition to psychological concerns about their impact upon program comprehension, side-effects also create many problems for tools and techniques commonly used as part of the maintenance and evolution process.

For example, the Maintainers' Assistant [25] transformation tool has been applied to reverse engineering problems [24], but the language upon which it operates, WSL, is side-effect free. In order to reverse engineer programs with side-effects, the maintainer therefore needs converters to transform a side-affecting program into a side-effect free equivalent.

In software testing, one of the hardest problems is the automated construction of systematic test data which satisfies some adequacy criterion, such as branch or statement coverage. Recent work [15, 18, 19, 20, 23, 26, 27] has concerned the application of evolutionary algorithms to the problem of searching for good quality test data. This work represents part of a more general move within the software engineering community towards consider-

[1] The C standard [1], gives a slightly more general description, encompassing file-updates and other forms of state change, but in this paper we will concentrate purely on assignments in expressions.

ing search-based techniques as solutions to hard problems where there are a wide variety of potential solutions and where analytic algorithms are infeasible or incomplete [13]. Unfortunately, side-effects reduce the applicability of evolutionary testing, because they prevent the definition of fitness functions which optimise sub-terms of boolean expressions in parallel.

The principle contribution of this paper is the introduction of our side-effect removal algorithm, but the paper also contains the initial results of an empirical study on the impact of side-effects on program comprehension. This study shows that the performance of program compressions tasks is significantly improved when the subjects are dealing with the side-effect free versions of the programs produced by our algorithm. Despite the large body of work on the harmful nature of side-effects, this paper represents (to the author's knowledge) the first paper to present either empirical evidence for these harmful effects or an algorithm which ameliorates them.

Currently, our work is concerned with applications to program comprehension and in using side-effect removal to pave the way for application of our amorphous slicing algorithm [11], but we also believe our approach may circumvent some of the difficulties side-effects present for testing and reverse engineering.

The rest of this paper is organised as follows. Section 2 presents some preliminary definitions. Section 3 presents our side-effect removal algorithm. Section 4 provides a worked example. Section 5 presents the initial results of an empirical study which confirms that side-effects are harmful and that our algorithm helps to improve comprehension.

2 Preliminary Definitions

In previous work we suggested that transformation might be an answer to the problem of side-effects and gave three strategies for side-effect removal transformation [12]. The strategies concern what to do with the statements which denote the side-effects of an expression. These can be placed before the side-effect free version of the expression (pre-placement), after it (post-placement) or temporary variables can be introduced to avoid the concern about where to place them.

In this paper we introduce an algorithm for side-effect removal, using the post-placement algorithm. This is chosen because pre-placement is not possible in all cases [12] and because one of the applications we have in mind is slicing [28, 8, 14, 7, 10], for which the introduction of additional (temporary) variables is counter-productive.

The algorithm used in this paper relies upon the concept of referenced and defined variable sets [2]. The functions $\mathbf{REF}$ and $\mathbf{DEF}$ will be used to denote the referenced and defined variables of an expression, respectively. The $\mathbf{DEF}$ function will also be applied to statements.

We shall adopt the definitions and terminology introduced in previous work [12]. That is, a side-effect free program will be a program in which the only way in which a variable may change its value is as the result of an assignment statement of the form `x = e;` where e is a side-effect free expression (that is one which has only referenced variables and for which the defined variables are empty). A side-effect removal algorithm takes a program (which possibly contains side-effects) and produces a functionally equivalent program which is guaranteed to be side-effect free.

3 The Side-Effect Removal Algorithm

The side-effect removal algorithm uses a top level transformation step $\mathcal{T}$. This is a syntax directed transformation algorithm, which walks over the abstract syntax tree of the side-affecting version of the program and replaces statements which contain side-affecting expression with side-effect free equivalents. The top level transformation uses an auxiliary function $\mathbf{SPLIT}$ to split the expression meaning of a side-affecting expression from the 'state-changing part' which captures the meaning of the side-effects. The $\mathbf{SPLIT}$ function uses a symbolic executor, $\mathcal{E}$ to perform the split. The components **VALID**, $\mathcal{E}$ and $\mathcal{T}$ are formally defined in Figures 1, 2 and 3. The following subsections describe how they are combined to implement side-effect removal. A prototype implementation is available for non-commercial use at

`http://www.brunel.ac.uk/`
`~csstmmh2/linsert`

3.1 Symbolic Execution

The transformation function $\mathcal{T}$ uses a symbolic executor $\mathcal{E}$, to split expressions with side-effects. $\mathcal{E}$ takes an expression, e, and returns a function from symbolic states to a triple, containing a side-effect free version of e, a statement sequence which performs the side-effects of e and the new symbolic state after the evaluation of e.

It will be assumed that $\mathcal{E}$ is applied to a valid expression (one for which the semantics are not 'undefined' according to the C standard). This can be guaranteed by first checking an expression with the function **VALID** (defined in Figure 1).

The symbolic executor $\mathcal{E}$ makes use of an auxiliary function, $\mathbf{Combine}$, which combines the effect of two modifications of a base state σ. The $\mathbf{Combine}$ function takes as parameters the base state, σ and the two modified states, σ' and σ''. The algorithm is so-constructed that, in all calls to $\mathbf{Combine}$, the states σ' and σ'' are produced by the evaluation of two sub-expressions, which occur between the same two sequence points. Therefore (since the expression is valid according to **VALID**), it is known that the defined variables of the two expressions cannot non-trivially intersect. This is exploited in the definition of $\mathbf{Combine}$, which can only be applied to states σ, σ' and σ'' such that at most one of σ' and σ'' update the value of each variable defined in σ. The $\mathbf{Combine}$ function is defined as follows:

$$\mathbf{Combine}(\sigma, \sigma', \sigma'') = ((\sigma' \cup \sigma'') - \sigma) \cup (\sigma' \cap \sigma'')$$

The type of symbolic states, S, is a mapping from variable names to expressions:

$$S \stackrel{\mathsf{def}}{=} I \to E$$

The distfix function $\cdot[\cdot \leftarrow \cdot]$ denotes an update to a symbolic state. That is, $\sigma[y \leftarrow x]$ is identical to the state σ except that the variable name y is mapped to the expression x, more formally:

$$\sigma[y \leftarrow x]z = \begin{cases} \sigma z & z \neq y \\ x & z = y \end{cases}$$

The algorithm uses a simple approach to symbolic execution to maintain the current value of the symbolic state. Symbolic execution is typically problematic in the presence of loops, but is relatively straightforward for loop-free programs [6]. Fortunately, although the algorithm does handle loops, it will not be necessary to symbolically execute them to remove side-effects.

The remainder of this subsection explains the effect of each case in the definition of $\mathcal{E}$.

The rules for pre- and post- increment and decrement operators are straightforward: Each returns the side-effect free version of the expression and updates the symbolic state accordingly. The rule for negation simply propagates side-effect removal into the negated expression.

The rule for arithmetic operators is, by definition, considering two expressions which occur between sequence points (because `AOp` is defined such that it contains no operators which denote sequence points). Since all expressions supplied to $\mathcal{E}$ are deemed to be valid according to **VALID**, each of the operands can be evaluated in the same symbolic state and the resulting states combined using $\mathbf{Combine}$.

The rules for binary logical operators must take account of short-circuit evaluation and therefore, convert the operators into conditional expressions, the semantics of which the original logical operators implicitly denote.

The rule for assignment is relatively straightforward. It simply returns the side-effect free expression and updates the symbolic state accordingly. The right-hand-side of the assignment statement is also transformed using a recursive call to the $\mathcal{E}$ function.

The rules for identifiers and numeric constants are trivial as these are side-effect free, by definition.

The rule for array lookup, simply recurses into the index expression.

The rule for the comma operator takes account of the sequencing of the evaluation of each operand by passing the resultant symbolic state from the symbolic execution of the first expression to the symbolic execution of the second expression. The expression meaning of the first expression is 'thrown away' as it can have no effect.

In the rule for conditional expressions, observe that the equality test $\sigma_2 i = \sigma_3 i$ is not computationally simple. In general it would require a theorem prover, though for simple expressions resulting from side-effects a less demanding algorithm might be possible. The rule is expressed in this way to indicate the possibility of determining when the condition of a conditional expression is redundant, however it is safe to always adopt the 'otherwise' case in the application of this rule. In our current implementation, we use only syntactic equality.

3.2 Splitting Statement and Expression Meanings

The function $\mathbf{SPLIT}$ (defined in Figure 3), takes an expression, e, which may contain side-effects, and returns an expression e' and a statement sequence c. It uses the symbolic executor, $\mathcal{E}$ to do this. The side-effects produced when e is evaluated are mimicked by the execution of the statement sequence c. The expression e' is guaranteed to behave identically to e when it is evaluated in an identical state. The expression e' is thus the 'expression meaning' of the original expression e, while c is the 'state-changing' meaning of e.

More formally, the behaviour of the $\mathbf{SPLIT}$ function can be described in terms of the denotational meaning of the language [21] as follows. Let S be the state, which is a mapping from identifiers (I) to the values they denote (in some value domain V, which we shall assume contains at least numbers in some number set N). Let the meaning of expressions be described by the function $\mathcal{V}$:

$$\mathcal{V} \stackrel{\mathsf{def}}{=} E \to (S \to (S \times V))$$

$\mathcal{V}$ takes an expression in E and returns a function which maps a state to a new state and a value. The new state is the state produced by any side-effects in E. The value is the outcome of expression evaluation. It is useful to define two projection functions which return the value and state components of a state-to-state meaning function. These are the functions $\mathbf{val}$ and $\mathbf{sta}$ defined below[2]:

$$\mathbf{val} \stackrel{\text{def}}{=} (S \to (S \times V)) \to (S \to V)$$
$$\mathbf{sta} \stackrel{\text{def}}{=} (S \to (S \times V)) \to (S \to S)$$
$$\mathbf{val}(m) = \lambda\sigma.(m\sigma) \downarrow 2$$
$$\mathbf{sta}(m) = \lambda\sigma.(m\sigma) \downarrow 1$$

Let the meaning of statements be described by the function $\mathcal{C}$:

$$\mathcal{C} \stackrel{\text{def}}{=} C \to (S \to S)$$

$\mathcal{C}$ maps a statement in C into a state-to-state transformer which denotes the effect of executing C. Let $\mathbf{SPLIT}[\![E]\!] = (E', C)$. $\mathbf{SPLIT}$ has the following properties:

1. $\mathbf{val}(\mathcal{V}[\![E']\!]) = \mathbf{val}(\mathcal{V}[\![E]\!])$
   This requires that the expression produced by $\mathbf{SPLIT}$ preserves the meaning of the original expression with respect to the values it yields.

2. $\mathbf{sta}(\mathcal{V}[\![E']\!]) = \lambda\sigma.\sigma$
   This requires that the expression produced by $\mathbf{SPLIT}$ is side-effect free, by demanding that it has no effect upon the state in which it is evaluated.

3. $\mathcal{C}[\![C]\!] = \mathbf{sta}(\mathcal{V}[\![E]\!])$
   This requires the side-effects of the original expression to be correctly replicated by the execution of the replacement statement sequence produced by $\mathbf{SPLIT}$.

3.3 The Top Level Transformation Algorithm

The transformation function $\mathcal{T}$ defined in Figure 3 implements the post-placement side-effect removal strategy. Each rule uses the $\mathbf{SPLIT}$ function to separate the side-effects from expression meaning. The side-effects are placed at a point at which they are guaranteed to be executed just after the evaluation of the side-effect free version of the expression and before the evaluation of any other expressions. This mimics the behaviour of the original.

The rule for expression statements, 'throws away' the expression meaning of the side-affecting version, because, by definition, it has no effect. The rule for conditionals is forced to create an **else** branch if one is not already present and has to replicate the side-effects in both branches to ensure correct semantics. The rule for **while** loops places a copy of the side-effect after the loop to take account of the fact that the body is not executed on the last test of the side-effect free version of the controlling expression. The **do** . . . **while** rule has to take account of the fact that the side-effects do not take place until after the side-effect free expression has been evaluated on the first iteration of the loop body. This involves copying of the loop body and the side-effects.

The rule which handles **for** loops simply converts the **for** loop into a **while** loop. This happens naturally because the side-effect free versions of the 'initialisation expression' and the 'incremental expression' can, by definition, have no effect. This behaviour is in keeping with the ISO standard which states that the two are equivalent "Except for the behaviour of a `continue` statement in the loop body" [1].

4 An Example

Post placement of side-effects creates longer programs than the original side-affecting programs which it takes as input. This is, in part, a reflection of the fact that side-effects are syntactically compact. However, it also arises from the way in which post-placement requires copying of the side-effects, where multiple execution paths are possible (as is the case with conditionals and loops). Fortunately, the side-effect free version of the program can be transformed using standard transformation rules to reduce its size. Typically these standard transformations would not be applicable to the original side-affecting program, but become applicable as a result of side-effect removal.

Example 1 below shows how side-effect removal can be combined with well–known transformations to produce more readable programs. The side-affecting version of the program fragment, vividly illustrates the way in which even the tiniest fragment of code can be hard to understand in the presence of side-effects.

Example 1
Consider the (apparently) simple example program fragment below:
if *(++i && i −−) x=1;*

The result of applying the symbolic executor $\mathcal{E}$ to this example is illustrated in Figure 4. Thus $\mathcal{T}[\![\mathbf{if}(++i \&\& i--)x = 1;]\!]$
produces:

$\mathbf{if}(i + 1 ? i + 1 : 0)$
$\quad\{\mathbf{if}(i + 1)\ \{i = i + 1; i = i - 1;\}$
$\quad\quad\mathbf{else}\ \{i = i + 1;\}\ x = 1;\}$
$\quad\mathbf{else}$
$\quad\{\mathbf{if}(i + 1)\ \{i = i + 1; i = i - 1;\}$

[2]The operator $\downarrow$ is the tuple selection operator, $(a, b) \downarrow 1 = a, (a, b) \downarrow 2 = b$.

$$
\begin{array}{rcl}
\mathbf{VALID} & \stackrel{\mathsf{def}}{=} & E \to \{T, F\} \\
\mathbf{VALID}[\![++I]\!] & & \\
\mathbf{VALID}[\![--I]\!] & & \\
\mathbf{VALID}[\![I{++}]\!] & & \\
\mathbf{VALID}[\![I{--}]\!] & & \\
\mathbf{VALID}[\![!E]\!] & \text{iff} & \mathbf{VALID}[\![E]\!] \\
\mathbf{VALID}[\![E_1 \ \texttt{AOp} \ E_2]\!] & \text{iff} & \mathbf{VALID}[\![E_1]\!] \wedge \mathbf{VALID}[\![E_2]\!] \ \wedge \\
 & & \mathbf{REF}[\![E_1]\!] \cap \mathbf{DEF}[\![E_2]\!] = \emptyset \ \wedge \\
 & & \mathbf{DEF}[\![E_1]\!] \cap \mathbf{REF}[\![E_2]\!] = \emptyset \ \wedge \\
 & & \mathbf{DEF}[\![E_1]\!] \cap \mathbf{DEF}[\![E_2]\!] = \emptyset \\
\mathbf{VALID}[\![E_1 \,\&\&\, E_2]\!] & \text{iff} & \mathbf{VALID}[\![E_1]\!] \wedge \mathbf{VALID}[\![E_2]\!] \\
\mathbf{VALID}[\![E_1 || E_2]\!] & \text{iff} & \mathbf{VALID}[\![E_1]\!] \wedge \mathbf{VALID}[\![E_2]\!] \\
\mathbf{VALID}[\![I = E]\!] & \text{iff} & \mathbf{VALID}[\![E]\!] \wedge I \notin \mathbf{DEF}[\![E]\!] \\
\mathbf{VALID}[\![I]\!] & & \\
\mathbf{VALID}[\![\mathrm{N}]\!] & & \\
\mathbf{VALID}[\![I[E]]\!] & \text{iff} & \mathbf{VALID}[\![E]\!] \\
\mathbf{VALID}[\![E_1, E_2]\!] & \text{iff} & \mathbf{VALID}[\![E_1]\!] \wedge \mathbf{VALID}[\![E_2]\!] \\
\mathbf{VALID}[\![E_1 ? E_2 : E_3]\!] & \text{iff} & \mathbf{VALID}[\![E_1]\!] \wedge \mathbf{VALID}[\![E_2]\!] \wedge \mathbf{VALID}[\![E_3]\!]
\end{array}
$$

Figure 1. Valid Expressions: Those Not Undefined According to the C Standard

$$
\begin{array}{rcl}
\mathcal{E} & \stackrel{\mathsf{def}}{=} & E \to (I \to E) \to E \times C \times (I \to E) \\
\mathcal{E}[\![++I]\!]\sigma & = & ([\![\sigma I + 1]\!], [\![I = I + 1;]\!], \sigma[I \leftarrow [\![\sigma I + 1]\!]]) \\
\mathcal{E}[\![--I]\!]\sigma & = & ([\![\sigma I - 1]\!], [\![I = I - 1;]\!], \sigma[I \leftarrow [\![\sigma I - 1]\!]]) \\
\mathcal{E}[\![I{++}]\!]\sigma & = & ([\![\sigma I]\!], [\![I = I + 1;]\!], \sigma[I \leftarrow [\![\sigma I + 1]\!]]) \\
\mathcal{E}[\![I{--}]\!]\sigma & = & ([\![\sigma I]\!], [\![I = I - 1;]\!], \sigma[I \leftarrow [\![\sigma I - 1]\!]]) \\
\mathcal{E}[\![!E]\!]\sigma & = & ([\![!E']\!], C, \sigma') \\
 & & \textit{where } (E', C, \sigma') = \mathcal{E}[\![E]\!]\sigma \\
\mathcal{E}[\![E_1 \ \texttt{AOp} \ E_2]\!]\sigma & = & ([\![E_1' \ \texttt{AOp} \ E_2']\!], [\![\{C_1 \ C_2\}]\!], \mathbf{Combine}(\sigma, \sigma', \sigma'')) \\
 & & \textit{where } (E_1', C_1, \sigma') = \mathcal{E}[\![E_1]\!]\sigma \\
 & & \ \textit{and } (E_2', C_2, \sigma'') = \mathcal{E}[\![E_2]\!]\sigma \\
\mathcal{E}[\![E_1 \,\&\&\, E_2]\!]\sigma & = & \mathcal{E}[\![E_1 ? E_2 : 0]\!]\sigma \\
\mathcal{E}[\![E_1 || E_2]\!]\sigma & = & \mathcal{E}[\![E_1 ? 1 : E_2]\!]\sigma \\
\mathcal{E}[\![I = E]\!]\sigma & = & ([\![E']\!], [\![\{I = E'; \ C\}]\!], \sigma'[I \leftarrow E']) \\
 & & \textit{where } (E', C, \sigma') = \mathcal{E}[\![E]\!]\sigma \\
\mathcal{E}[\![I]\!]\sigma & = & (\sigma I, [\![]\!], \sigma) \\
\mathcal{E}[\![\mathrm{N}]\!]\sigma & = & ([\![\mathrm{N}]\!], [\![]\!], \sigma) \\
\mathcal{E}[\![E_1, E_2]\!]\sigma & = & ([\![E_2']\!], [\![\{C_1 \ C_2\}]\!], \sigma'') \\
 & & \textit{where } (E_1', C_1, \sigma') = \mathcal{E}[\![E_1]\!]\sigma \\
 & & \ \textit{and } (E_2', C_2, \sigma'') = \mathcal{E}[\![E_2]\!]\sigma' \\
\mathcal{E}[\![E_1 ? E_2 : E_3]\!]\sigma & = & ([\![E_1' ? E_2' : E_3']\!], [\![\{C_1 \ \mathbf{if}(E_1') \ \{C_2\} \ \mathbf{else} \ \{C_3\}]\!], \sigma') \\
 & & \textit{where} (E_1', C_1, \sigma_1) = \mathcal{E}[\![E_1]\!]\sigma \\
 & & \ \textit{and } (E_2', C_2, \sigma_2) = \mathcal{E}[\![E_2]\!]\sigma_1 \\
 & & \ \textit{and } (E_3', C_3, \sigma_3) = \mathcal{E}[\![E_3]\!]\sigma_1 \\
 & & \ \textit{and } \sigma' i = \begin{cases} \sigma_2 i & \text{if } \sigma_2 i = \sigma_3 i \\ [\![E_1' ? \sigma_2 i : \sigma_3 i]\!] & \text{otherwise} \end{cases}
\end{array}
$$

Figure 2. Symbolically Evaluating Expressions

$$
\begin{array}{rcl}
\mathbf{SPLIT} & \stackrel{\mathsf{def}}{=} & E \rightarrow (E \times C) \\
\mathbf{SPLIT}[\![E]\!] & = & (E', C) \\
 & & \quad \textit{where } (E', C, s) = \mathcal{E}[\![E]\!](\lambda i.i) \\
\mathcal{T}[\![E;]\!] & = & C \\
 & & \quad \textit{where } (E', C) = \mathbf{SPLIT}[\![E]\!] \\
\mathcal{T}[\![\mathbf{if}(E)\ C]\!] & = & [\![\mathbf{if}(E')\ \{C'\ \mathcal{T}C\}\ \mathbf{else}\ \{C'\}]\!] \\
 & & \quad \textit{where } (E', C') = \mathbf{SPLIT}[\![E]\!] \\
\mathcal{T}[\![\mathbf{if}(E)\ C_1\ \mathbf{else}\ C_2]\!] & = & [\![\mathbf{if}(E')\ \{C'\ \mathcal{T}C_1\}\ \mathbf{else}\ \{C'\ \mathcal{T}C_2\}]\!] \\
 & & \quad \textit{where } (E', C') = \mathbf{SPLIT}[\![E]\!] \\
\mathcal{T}[\![\mathbf{while}(E)\ C]\!] & = & [\![\mathbf{while}(E')\ \{\ C'\mathcal{T}C\ \}\ C']\!] \\
 & & \quad \textit{where } (E', C') = \mathbf{SPLIT}[\![E]\!] \\
\mathcal{T}[\![\mathbf{do}\ C\ \mathbf{while}\ (E)]\!] & = & [\![\mathcal{T}C\ \mathbf{while}\ (E')\{C'\ \mathcal{T}C\}\ C']\!] \\
 & & \quad \textit{where } (E', C') = \mathbf{SPLIT}[\![E]\!] \\
\mathcal{T}[\![\mathbf{for}\ (E_1; E_2; E_3)\ C\ ]\!] & = & [\![C_1' \mathbf{for}\ (; E_2';)\ \mathcal{T}\{C_2'\ \mathcal{T}C\ C_3'\}\ C_2' \\
 & & \quad \textit{where } (E_1', C_1') = \mathbf{SPLIT}[\![E_1]\!] \\
 & & \quad \textit{where } (E_2', C_2') = \mathbf{SPLIT}[\![E_2]\!] \\
 & & \quad \textit{where } (E_3', C_3') = \mathbf{SPLIT}[\![E_3]\!] \\
\mathcal{T}[\![\{C_1\ C_2\}]\!] & = & [\![\{\mathcal{T}C_1\ \ \mathcal{T}C_2\}]\!]
\end{array}
$$

Figure 3. The Side-Effect Removal Transformation $\mathcal{T}$ for Post-placement of side-effects

$$
\begin{array}{rl}
 & \mathcal{E}[\![(\ {+}{+}i \,\&\&\, i\ {-}{-})]\!](\lambda x.x) \\
= & \mathcal{E}[\![(\ {+}{+}i ? i\ {-}{-}: 0)]\!](\lambda x.x) \\
= & ([\![E_1' ? E_2' : E_3']\!], [\![\mathbf{if}(E_1')\ \{C_1\ C_2\}\ \mathbf{else}\ \{C_1\ C_3\}]\!], s') \\
 & \quad \text{where } (E_1', C_1, s_1) \ = \ \mathcal{E}[\![{+}{+}i]\!](\lambda x.x) \ = \ ([\![i+1]\!], [\![i = i+1;]\!], (\lambda x.x)[i \leftarrow [\![i+1]\!]]) \\
 & \quad\quad \text{and } (E_2', C_2, s_2) \ = \ \mathcal{E}[\![i{-}{-}]\!]s_1 \ = \ ([\![i+1]\!], [\![i = i-1;]\!], s_1[i \leftarrow [\![i-1]\!]]) \\
 & \quad\quad \text{and } (E_3', C_3, s_3) \ = \ \mathcal{E}[\![0]\!]s_1 \ = \ (0, [\![]\!], s_1) \\
 & \text{and} s'i = \begin{cases} s_2 i & if s_2 i = s_3 i \\ [\![E_1' ? s_2 i : s_3 i]\!] & \text{otherwise} \end{cases} \\
= & ([\![i+1 ? i+1 : 0]\!], [\![\mathbf{if}(i+1)\ \{i = i+1; i = i-1;\}\ \mathbf{else}\ i = i+1;]\!])
\end{array}
$$

Figure 4. Application of $\mathcal{E}$ to Example 1

$\mathbf{else}\ i = i + 1;$

This is the raw output of the side-effect removal transformation. It could hardly be argued to be an improvement upon the original. It might be said that the apparent complexity of the result of side-effect removal indicates the level of implicit sophistication in the original. However, the side-effect removal transformation adopts a simple minded approach, producing a side-effect free result which can be significantly 'cleaned up' using existing transformation.

In what follows the output will be transformed using a set of standard transformations [25, 4]. In the Maintainers' Assistant [25], there are a large number (approximately 500) of transformations. Figure 5 presents a small set of rules which can be used to 'clean up' the result of side-effect removal.

Transformation axiom 1 can be used to simplify the boolean expression $i + 1?i + 1 : 0$ to $i + 1$. Also, using transformation axiom 2 the assignments $\{i = i + 1; i = i - 1;\}$ can be removed. These two transformations yield the program below:

$\mathbf{if}(i + 1)$
  $\{\mathbf{if}(i + 1)\ \{\}\ \mathbf{else}\ \{i = i + 1;\}\ x = 1;\}$
  $\mathbf{else}$
  $\{\mathbf{if}(i + 1)\ \{\}\ \mathbf{else}\ i = i + 1;$

Propagating the truth value of the boolean expression in the outermost conditional statement to its consequent and alternative branches (using axioms 6 and 7) allows for the simplification of each of the two branches:

$\mathbf{if}(i + 1)$
  $\{\{\}x = 1;\}$
  $\mathbf{else}$
  $i = i + 1;$

Using transformation axiom 3, the empty statement sequence $\{\}$ can be removed. Also, transformation rule 1 can be used to re-write the conditional.

$\mathbf{if}((i + 1)! =0)\ x = 1;\ \mathbf{else}\ i = i + 1;$

Using transformation axiom 5, gives:

$\mathbf{if}((i + 1){=}{=}0)\ i = i + 1;\ \mathbf{else}\ x = 1;$

A final transformation step would be to simplify the side-effect free expression to give:

$\mathbf{if}(i{=}{=} - 1)\ i = i + 1;\ \mathbf{else}\ x = 1;$

Work is in progress to extend the LinSERT implementation to support these facilities.

Observe that this example is one in which the preplacement strategy could not be used. The original value of `i` is required to define the outcome of expression evaluation, but, unfortunately the original value is not available because `i` is assigned a new value as a side-effect. This alone is not enough to prevent application of the preplacement approach. However, the assignment function is not an injection, and so its inverse is not a function. The assignment to `i` is captured by the state-to-value function

$$\lambda\sigma. \begin{cases} 0 & \text{if } \sigma\mathtt{i} = -1 \\ \sigma\mathtt{i} & \text{otherwise} \end{cases}$$

the inverse of which is the relation which maps 0 to both 0 and -1 and which maps all other values to themselves. This means that after the side-effects have taken place, the meaning of the side-effect free expression cannot be captured by an expression: The expression requires the original value of `i`, but this value is potentially destroyed by the side-effects.

5 Empirical Evaluation

The empirical study explored the effect of side-effect removal on program comprehension by 18 students from the University of the Basque Country. The students were allocated to two groups, according to ability, so that a similar distribution of ability was present in each group. A general C programming test was used to determine this allocation. Each of the two groups were presented with program comprehension questions, where each question asked for the final values of certain variables used in fragments of C code. In one version of the test, the fragments of code contained side-effects. In the other version of the test, the questions remained the same, but the side-effect free versions of the code fragments were presented to the subject.

In order to reduce the impact of other possible effects which might affect the performance of the student and to remove possible bias in the choice of questions, we produced two versions of the test and used a 'cross-over' design for the experiment.

5.1 Two Versions of the Test

Two versions of the test were created. In both cases the side-effect free programs were produced by applying the algorithms so that there was no chance of bias in the way side-effect free versions were selected, once the side-affecting versions had been chosen. However, this still leaves a possible source of bias in the selection of questions to ask. We addressed this concern by formulating two different tests. In one test the questions were chosen

Axiom 1 (Reduce `false`) $[\![e?e:0]\!] \Rightarrow [\![e]\!]$

Axiom 2 (Fold Inverse) $[\![I = I+1; I = I-1;]\!] \Rightarrow [\![]\!]$

Axiom 3 (Remove Empty Sequence) $[\![\{\}]\!] \Rightarrow [\![]\!]$

Axiom 4 (Idempotence of negation) $[\![!!E]\!] \Rightarrow [\![E]\!]$

Axiom 5 (Permute Conditional) $[\![\mathbf{if}(e)\ c_1\ \mathbf{else}\ c_2]\!] \Rightarrow [\![\mathbf{if}(!e)\ c_2\ \mathbf{else}\ c_1]\!]$

Axiom 6 (Collapse then) $[\![\mathbf{if}(e)\mathbf{if}(e)c_1\mathbf{else}c_2]\!] \Rightarrow [\![\mathbf{if}(e)c_1]\!]$

Axiom 7 (Collapse else) $[\![\mathbf{if}(e)c_1\ \mathbf{else}\ \mathbf{if}(e)c_2\mathbf{else}c_3]\!] \Rightarrow [\![\mathbf{if}(e)c_1\ \mathbf{else}\ c_3]\!]$

Rule 1 (Boolean Expression Meaning)

$$\frac{E \text{ is an arithmetic expression used as a boolean expression}}{[\![E]\!] \Rightarrow [\![E != 0]\!]}$$

Figure 5. General Transformation Axioms and Rules

by academics familiar with the side-effect removal algorithm. We shall call this the 'possibly biased test'. In the other test, the questions were chosen by academics familiar with C programming, but who had not been exposed to the side-effect removal algorithm. We shall call this the 'unbiased test'.

In the 'possibly biased' test, no attempt was made to optimise the questions to illustrate that the algorithm performed well. Rather, we attempted to define simple and commonly accruing side-affecting code fragments. The reason for comparing results with a guaranteed 'unbiased' test was thus to ensure that there was no *subconscious bias* in the selection of questions.

In the 'possibly biased' test there were six questions, chosen to exhibit a range of difficulties. In the unbiased test there were four questions. In both the 'possibly biased' and 'unbiased' tests, each question presents a fragment of code and contains several sub-questions, which ask the student what the final values of variables will be when the code fragment is executed.

5.2 Cross–over of Treatments

All the students were required to attempt all the questions (and all their parts) in both the 'possibly biased' and 'unbiased' tests, both for the side-affecting versions and the side-effect free versions. This gives us the maximum amount of data to analyse, but it raises the possibility of unwanted 'order effects'. That is, if we suppose that the student performed the side-affecting test first and then the side-effect free test second. It is possible that the experience gained from answering the questions in the first version of the test could, perhaps, help (or indeed hinder) the ability to answer those in the second.

We could have split the group of subjects in half, giving one half the treatment (i.e. side-effect free programs), using the other as a 'control', but this would have presented us with the problem of selecting the groups and would have reduced by half the amount of data we could collect. Therefore, instead of removing the possibility of order effects, we chose to measure whether such effects had occurred. To achieve this, we used a cross-over design. For the 'possibly biased' test, one group sat the side-effect free version of the test first, while the other sat the side-affecting version of the test first. For the unbiased test the order in which the groups sat side-effect free and side-affecting versions was reversed. This allowed us to test whether order had an impact upon the scores obtained.

5.3 Results

Figure 6 shows the average score for each question in both its Side-Effect Free (SEF) and Side-Affecting (SE) versions, for the 'possibly biased test', while Figure 7 shows the average score for both versions of the four questions for the unbiased test. Certainly, it can be seen from the figures, that the results for all of the side-effect free versions of the program are noticeably better than those for the corresponding side-affecting versions.

It is also interesting to note that performance (SCORE) appears to be consistently affected across the range of questions asked; the results for the questions containing the side-affecting versions of the code fragments are essentially similar to those obtained for the side-effect free equivalents; they are merely shifted down

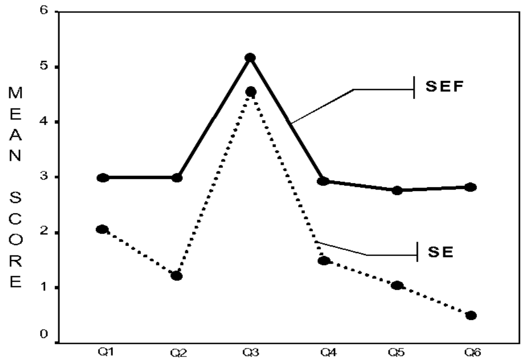

Figure 6. Results: Possibly Biased Test

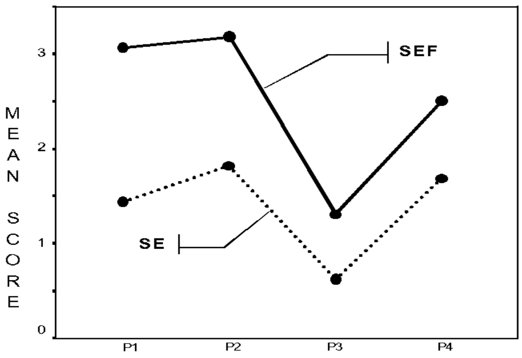

Figure 7. Results: Unbiased Test

by a relatively consistent amount. We might call this shift the result of a 'coefficient of inherent side-effect complexity'.

These informal visual conclusions drawn from the presentation of the results in the figures can be substantiated by performing t–tests and by calculating the F statistic [3] on the data collected. Specifically, by performing a t–test, we found that we cannot reject the null hypothesis relating to order effects on the number of correct answers (with $t = 0.854$ and $p = 0.406$). That is, there is no evidence for differences between the two groups, divided according to the order in which the test were taken. However, by computing the F statistic for the two tests we found that the performance was significantly affected by the treatment (whether the side-affecting or side-effect free version of the test was used). For the 'possibly biased' test the results were $(F = 181.226, p << 0.0001)$, and for the unbiased test they were $(F = 34.89, p << 0.0001)$.

This allows us to state that there is no statistically significant difference between the scores for students who sat the tests in each of the two orders. Also, we can state that the scores were significantly better for both the 'possibly biased' and the unbiased tests when using the side-effect free versions of the program. This gives us good reason to believe that the removal of side-effects is the principal cause of improved performance, and that this improved performance is real. We therefore conclude that side-effect removal using this algorithm improves program comprehension.

6 Conclusion and Future Work

This paper has introduced an algorithm for side-effect removal that uses syntax directed transformation and symbolic execution to transform programs with side-effects into equivalent side-effect free programs.

The algorithm has been shown to significantly improve program comprehension and may also have several other applications in software maintenance such as improving evolutionary testing and removing the barriers to the application of tools such as the Maintainers' Assistant, which works with a side-effect free language, WSL.

Future work will combine the side-effect removal system with the DaimlerChrysler Evolutionary Testing system to explore the possibility of increasing the coverage achieved by existing techniques for automated testing and to reduce the effort required to generate good quality test data.

References

[1] International standards organisation: Programming languages — C. international standard, ISO/IEC 9899: 1990 (E), Dec. 1990.

[2] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, techniques and tools.* Addison Wesley, 1986.

[3] ARMITAGE, P., AND BERRY, G. *Statistical methods in medical research.* Macmillan, London, 1994.

[4] BULL, T. *Software maintenance by program transformation in a wide spectrum language.* PhD thesis, University of Durham, UK, School of Engineering and Computer Science, 1994.

[5] CANNON, L., ELLIOTT, R., KIRCHHOFF, L., MILLER, J., MILNER, J., MITZE, R., SCHAN, E., WHITTINGTON, N., SPENCER, H., KEPPEL, D., AND BRADER, M. Recommended C style and coding standards, 2000. http://www.cs.umd.edu/users/cml/cstyle/indhill-cstyle.html.

[6] COWARD, P. D. Symbolic execution systems - a review. *Software Engineering Journal 3*, 6 (Nov. 1988), 229–239.

[7] DANICIC, S., FOX, C., HARMAN, M., AND HIERONS, R. M. ConSIT: A conditioned program slicer. In *IEEE International Conference on Software Maintenance (ICSM'00)* (San Jose, California, USA, Oct. 2000), IEEE

Computer Society Press, Los Alamitos, California, USA, pp. 216–226.

[8] DE LUCIA, A. Program slicing: Methods and applications. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation* (Florence, Italy, 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 142–149.

[9] HAAHR, P. A programming style for java, Oct. 1999. http://www.webcom.com/˜haahr/essays/java-style/.

[10] HARMAN, M., HIERONS, R. M., DANICIC, S., HOWROYD, J., AND FOX, C. Pre/post conditioned slicing. In *IEEE International Conference on Software Maintenance (ICSM'01)* (Florence, Italy, Nov. 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 138–147.

[11] HARMAN, M., HU, L., ZHANG, X., AND MUNRO, M. GUSTT: An amorphous slicing system which combines slicing and transformation. In 1^{st} *Workshop on Analysis, Slicing, and Transformation (AST 2001)* (Stuttgart, Oct. 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 271–280.

[12] HARMAN, M., HU, L., ZHANG, X., AND MUNRO, M. Side-effect removal transformation. In 9^{th} *IEEE International Workshop on Program Comprehension (IWPC'01)* (Toronto, Canada, May 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 310–319.

[13] HARMAN, M., AND JONES, B. F. Search based software engineering. *Information and Software Technology 43*, 14 (Dec. 2001), 833–839.

[14] HARMAN, M., SIVAGURUNATHAN, Y., AND DANICIC, S. Analysis of dynamic memory access using amorphous slicing. In *IEEE International Conference on Software Maintenance (ICSM'98)* (Bethesda, Maryland, USA, Nov. 1998), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 336–345.

[15] JONES, B., STHAMER, H.-H., AND EYRES, D. Automatic structural testing using genetic algorithms. *The Software Engineering Journal 11* (1996), 299–306.

[16] KERNIGHAN, B. W., AND PIKE, R. *The practice of programming*. Addison-Wesley Longman, Reading, Massachusetts, 1999.

[17] MEYER, B. *Object-oriented Software Construction*, second ed. Prentice Hall, New York, NY, 1997.

[18] MICHAEL, C., MCGRAW, G., AND SCHATZ, M. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 12 (Dec. 2001), 1085–1110.

[19] MUELLER, F., AND WEGENER, J. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)* (Washington - Brussels - Tokyo, June 1998), IEEE, pp. 144–154.

[20] PARGAS, R. P., HARROLD, M. J., AND PECK, R. R. Test-data generation using genetic algorithms. *The Journal of Software Testing, Verification and Reliability 9* (1999), 263–282.

[21] STOY, J. E. *Denotational semantics: The Scott–Strachey approach to programming language theory*. MIT Press, 1985. Third edition.

[22] THOMAS, P. *Learning to program in C*, 2 ed. Plum Hall, Inc., 1989.

[23] TRACEY, N., CLARK, J., AND MANDER, K. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)* (January 1998), IFIP, pp. 169–180.

[24] WARD, M. Reverse engineering through formal transformation. *The Computer Journal 37*, 5 (1994).

[25] WARD, M., CALLISS, F. W., AND MUNRO, M. The maintainer's assistant. In *Proceedings of the International Conference on Software Maintenance 1989* (1989), IEEE Computer Society Press, Los Alamitos, California, USA, p. 307.

[26] WEGENER, J., GRIMM, K., GROCHTMANN, M., STHAMER, H., AND JONES, B. F. Systematic testing of real-time systems. In *4th International Conference on Software Testing Analysis and Review (EuroSTAR 96)* (1996).

[27] WEGENER, J., STHAMER, H., JONES, B. F., AND EYRES, D. E. Testing real-time systems using genetic algorithms. *Software Quality 6* (1997), 127–135.

[28] WEISER, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.