It's Doomed; We Can Prove It

Jochen Hoenicke[1], K. Rustan M. Leino[2], Andreas Podelski[1], Martin Schäf[1], and Thomas Wies[1,3]

[1] University of Freiburg
[2] Microsoft Research, Redmond
[3] EPFL, Switzerland

Abstract. Programming errors found early are the cheapest. Tools applying to the early stage of code development exist but either they suffer from false positives ("noise") or they require strong user interaction. We propose to avoid this deficiency by defining a new class of errors. A program fragment is *doomed* if its execution will inevitably fail, in whatever state it is started. We use a formal verification method to identify such errors fully automatically and, most significantly, without producing noise. We report on preliminary experiments with a prototype tool.

1 Introduction

Software engineers agree on that bugs found early are the cheapest. Tools applying to this stage of development, however, usually suffer from false positives or require strong user interaction. Perhaps the only "cheap" bugs are those found by the compiler. Fixing them is cheap since they are fixed by the programmer as they appear. We note that no programmer would doubt the relevance of a compiling error in a program fragment because this is an error regardless of the intended use of the program fragment, i.e., there is no way it can be dismissed (there is no "noise").

In this paper, we propose the definition of a class of program errors that can be detected as early (i.e., for a possibly isolated program fragment), as automatically (i.e., by a tool, without user input and without user interaction) and as precisely (no noise) as, e.g., a missing semicolon.

We define that a program fragment is *doomed* if an execution that reaches it will inevitably fail, i.e., executing the program fragment will never lead to a normal termination of the program.

We present a formal verification method (on top of an existing static checker) to identify such errors fully automatically and, most significantly, without producing noise. We report on a prototype implementation on top of Boogie [2, 4] that can be used in combination with Spec# or VCC [6] to either analyze C# or C programs. Preliminary experimental results indicate its practical potential.

J. Hoenicke, K.R.M. Leino, A. Podelski, M. Schäf, and T. Wies. It's doomed; we can prove it. In *FM 2009*, number 5850 in LNCS, pages 338-353. Springer, 2009.
The orgininal publication is available at www.springerlink.com

Related Work. We first want to point out that the class of errors our approach finds is subsumed by almost every bug detection tool and that most tools will find even more real bugs. However, the increased error detection rate comes at a price: these tools either produce a lot of noise or they require heavy user interaction. For instance, a set of unit tests that executes every statement in the program at least once will detect all errors related to doomed program points but one has to write or generate the test cases.

Our work can best be compared to Findbugs [1], which tries to find a reasonable amount of bugs using different control and dataflow analysis approaches while having in mind that flooding the user with false positives would ruin everything. With Findbugs, we share the idea of searching for contradictions in the dataflow. For this purpose, Findbugs uses a special pattern detection mechanism which is very fast but can miss errors and produce false positives. We give an experimental comparison of our approach and Findbugs in Section 7.

Other static analysis tools like e.g., Splint [9] are less comparable since they focus on finding as many bugs as possible and therefore produce noise or require special code annotations.

The results produced by our tool could be reproduced using full fledged automatic verifiers such as Blast [12] by first trying to prove the program, collecting all unverified assertions, negating them and rerunning the verification. If the verifier is able to prove such a negated assertion then, the corresponding statement will fail under any circumstances. However, this would be a rather convoluted and costly way to find doomed program points. Also, tools such as Blast are meant to be applied to the whole program, i.e., at a rather late stage of development when the errors we are targeting have probably already been fixed.

From the algorithmic point of view, our approach is strongly related to extended static checkers such as ESC/Java [10] and modular program verifiers such as Spec# [2, 4]. While these tools issue warnings whenever they cannot prove the absence of an error, as opposed to issuing warnings only for definite errors, we share their approach of transforming the program and the idea of using predicate transformers to obtain a representation that can be checked by a theorem prover [17]. These tools use special annotations such as invariants to prove certain properties. For the purpose of error detection, though, these annotations are not required. As we show later on, the user can still provide such information in order to increase the error detection rate.

Proving the existence of bad states, as done by Rümmer and Shah [21], differs from our approach in that they prove the existence of inputs for which something bad might happen while a doomed program point guarantees that nothing good can happen when reaching it. Their approach will find more errors, but requires a specification of the desired inputs or will otherwise be imprecise. Doomed points are errors regardless of the desired program behavior.

There is also previous work on refinement and noise reduction techniques for existing error detection and verification approaches. That work has the effect of reducing false positives, but we instead take a new approach.

2 Examples

In the following, we present a collection of examples that demonstrate what kinds of errors our approach is able to find and, more importantly, what kinds of vulnerabilities it does not report.

Example 1. Our first example is given in Figure 1. It demonstrates a trivial, yet common error that can happen during development. In fact, the example is inspired by an error that was found in an old version of Eclipse [14].

If our algorithm identifies an error in a program, then it will report not just the statement that crashes, but also the statements that actually lead to the crash. This provides additional hints to the developer that help him to fix the error. If we apply our algorithm to the example program, then it will report lines 5 and 6 as a guaranteed error. It reports line 6 because whenever the expression `*ptr` is evaluated, this will cause a null pointer dereference. It further reports line 5 because if the else branch of the conditional is taken, `ptr==0` has been evaluated to true, which guarantees the error in line 6.

```
1 int access(int *ptr)
2 {
3   if (ptr)
4     *ptr = 0;
5   else
6     printf ("%d", *ptr);
7
8   return 0;
9 }
```

Fig. 1: Trivial

```
1  int getMin(int *a, int x) {
2    int i, j, temp;
3    for (i= x-1; i >= 0; i--) {
4      for (j= 1; j <= i; j++) {
5        if (a[j-1] > a[j]) {
6          temp = a[j-1];
7          a[j-1] = a[j];
8          a[j] = temp;
9        }
10     }
11   }
12   return a[i];
13 }
```

Fig. 2: Loop

Example 2. Our second example is less trivial, yet contains a common error. The function `getMin` in Figure 2 returns the minimal element of an array. For this purpose, it first sorts the array and then returns the first element. However, there is a mistake in the loop bound of the *for* loop in line 3. The loop will decrease the variable `i` until it has a negative value. This leads to an out-of-bounds array access in line 12. Our algorithm detects that the out-of-bounds access is inevitable. It reports lines 3 and 12 as what leads to the error. This is the only warning emitted by our algorithm. Since there is no precondition saying that array `a` is allocated and its size is given by `x`, any attempt to verify that the procedure is safe without taking into account its calling context would generate additional warnings of potential boundary errors.

```
1  /* Sorted tree */
2
3  typedef void* T;
4  typedef  struct
5    entry *Entry;
6  struct entry
7  {
8      Entry left ;
9      Entry right ;
10     int key ;
11     T data ;
12 };
```

```
12 void update(Entry root,
13              int key, T dat) {
14   Entry x = root;
15   while (x->key != key) {
16     if (key < x->key)
17       x = x->left;
18     else
19       x = x->right;
20   }
21   x->data = dat;
22 }
```

Fig. 3: Complex

Example 3. Our last example demonstrates how the user of our tool benefits from the fact that it detects guaranteed errors rather than arbitrary errors. The program fragment in Figure 3 is taken from a library that implements a map data structure using a sorted binary tree. The function `update` takes three parameters: the root of the data structure, a key to an entry in the data structure, and a data value. It then traverses the tree to find the entry for the given key and updates the data value associated with this key. The function works correctly if the calling context guarantees that there is already an entry for the given key in the data structure. If this assumption is violated, the function crashes. Note that there is no null pointer check that guards the dereference of variable `x` in the while condition at line 16. The fact that there is an entry for the given key guarantees that `x` is not null.

It is a real challenge for any bug finding tool to prove that line 15 does not cause a null pointer dereference and, thus, not report this line as a potential error. For extended static checking or a modular program verifier, the user needs to specify the precondition saying that there exists an entry in the tree for the given key. However, this is not sufficient to prove the absence of a null pointer dereference. The user further needs to specify a data structure invariant that expresses the fact that the tree is sorted. This information is required in the loop invariant of the while loop. Even if all necessary specifications are given, automatically proving that the loop invariant implies the absence of null pointer dereferences is still tricky. Extended static checkers use theorem provers to automate this task. The theorem prover needs to conclude from the sortedness property and the existence of an entry for the given key that this entry is located in the subtree that the while loop traverses into. Modern theorem provers still require proof hints from the user to accomplish such proofs. All these tasks are time consuming and require the expertise of a verification engineer.

If, on the other hand, one attempts to use abstraction based program analyses to automatically infer the necessary preconditions, then only the use of a very sophisticated shape analysis would leave any hope for success. However, such

analyses are expensive and do not yet scale well to large programs. Using them online while coding is unrealistic.

In contrast, our algorithm will not report any errors, simply because there exist executions that never dereference any null pointers.

3 Doomed Program Points

We now formally define the new class of errors that we consider in this paper.

In order to abstract away from the details of a concrete programming language, we only assume that a program defines a set of possibly infinite *executions* (sequences of states).

We assume that executions are divided into two types: admissible executions and inadmissible executions. An execution is inadmissible if it causes some undesirable behavior; in particular, it is inadmissible if it diverges or violates an assertion. Syntactically, we only assume that a program comes with a finite set of *program points*.
Each state in an execution belongs to a unique *program point*. We say that an execution passes through a program point ℓ if one of its states belongs to ℓ.

Definition 1. *A program point ℓ is called* doomed *if all executions that pass through ℓ are inadmissible.*

In particular, a program containing a doomed program point has an inadmissible execution, or no execution passes through it (i.e., it is part of *dead code*). Once a doomed program point is reached in an execution, this execution is guaranteed to fail. In this sense, doomed program points are the witnesses of guaranteed errors.

We define the problem of *error verification* as the problem of identifying all doomed program points in a given program. We say that an algorithm for this problem is sound if, for any given program, it identifies only doomed program points. We say that it is complete if it identifies all doomed program points.

4 Preliminaries

We define our algorithm with respect to a subset of the Boogie language [2,18]. Boogie is an intermediate verification language designed for program analysis. It provides a small set of control constructs that, yet, allows the encoding of full-fledged programming languages such as C, C#, and Java (see, e.g., [2,5,6]).

The syntax of our simple language is defined in Figure 4. A program consists of a sequence of blocks. Each block consists of a unique program point, a sequential statement, and a goto statement that connects the block with a non-empty set of successor blocks. The atomic statements of our language are assignments, non-deterministic assignments (**havoc**) of program variables, assert statements, and assume statements. We do not specify the concrete syntax of expressions that are used in these statements. In principle, they can be arbitrarily complex

$$
\begin{aligned}
\mathit{Program} &::= \mathit{Block}^+ \\
\mathit{Block} &::= \mathit{PPId}: \ \mathit{Stmt}; \ \textbf{goto}\ \mathit{PPId}^+ \\
\mathit{Stmt} &::= \mathit{VarId} := \mathit{Expr} \mid \textbf{havoc}\ \mathit{VarId}^+ \\
&\quad \mid \textbf{assert}\ \mathit{Expr} \mid \textbf{assume}\ \mathit{Expr} \\
&\quad \mid \mathit{Stmt}; \ \mathit{Stmt}
\end{aligned}
$$

Fig. 4: Simple Language

logical formulae. Each block either has a transition to other blocks or goes to a unique program point called *Term* which means that the program has terminated normally.

A program state is a valuation of the program variables and a program counter that evaluates to a program point *id*. A program gives rise to a set of executions. An execution consists of a sequence of states describing the successive execution of the program blocks starting from some block in the program. An execution terminates normally if it reaches the block *Term*, it ends in an error if an **assert** in some block evaluates to *false*, and it is infinite if the program does not terminate. A sequence of states where an **assume** in a block evaluates to *false* models an infeasible computation. The admissible executions are the feasible executions that terminate normally.

If we translate a real program into our simple language, we can model arrays and the program's heap using function-valued program variables that map indices or memory addresses to values. The concrete representation of the heap depends on the semantics of the translated language. For example, one way to model a Java-like language is to use a function-valued program variable per field in a class; other possible memory models are discussed, e.g., in [6,16,18,19]. For brevity of exposition, our simple language does not support procedures (although Boogie does).

5 Error Verification Algorithm

Outline. We now give the outline of our algorithm for error verification. It is implemented by the procedure Exorcise given in Figure 5. Procedure Exorcise takes a program as input and returns a set of doomed program points. The procedure first transforms the input program P into a program P' in loop-free passive form. This means, that (1) program P' has no cycle in the graph formed by its blocks and goto statements and (2) blocks in P' consist only of assume and assert statements. The transformation is such that the set of doomed program points in P' is a subset of the set of doomed program points of P.

After the transformation, procedure Exorcise iterates over all program points in program P'. For each program point ℓ, it generates a logical formula $\mathsf{EVC}(\ell, P')$.

```
proc Exorcise(P : program)
  var Doomed : set of doomed program points
  var P' : program
  var φ : formula

  begin
    P' := Transform(P)
    for each program point ℓ in P' do
      φ := EVC(ℓ, P')
      if Valid(φ) then
        add ℓ to Doomed
    od
    return Doomed
  end
```

Fig. 5: Error verification algorithm

We call $\mathsf{EVC}(\ell, P')$ an *error verification condition*. The error verification condition is valid if and only if program point ℓ is doomed in P'. The procedure then calls the subroutine Valid, which checks whether the error verification condition is valid. We assume that Valid is a sound test for logical validity, e.g., implemented by a theorem prover. If the error verification condition is valid, then the program point ℓ is added to the set of doomed program points.

For exposition purposes, we will present simplified versions of subroutines Transform and EVC, and argue that procedure Exorcise is sound. Afterwards, we discuss improvements of these subroutines that are crucial for scalability of the algorithm and increased error detection rate.

Program transformation. In the following, we describe a simple version of subroutine Transform that transforms a program into loop-free passive form [3,11]. Note that the transformation described below is by now standard and is used in several extended static checkers and program verifiers (e.g.,, [2,10]). We therefore provide only a brief description. For a more detailed discussion, see [3].

The first step in $\mathsf{Transform}(P)$ is to transform program P into a loop-free program. We now think of our program P as a control flow graph where each block is a single node labeled with the program point associated with the block. We assume that each cycle in the graph has a unique entry point, the *loop header* (if not, one can first apply node splitting, see e.g., [15]). Edges from nodes inside a cycle back to the loop header are called *back edges*. We assume without loss of generality that a loop header is a block that consists of just one goto statement that either goes to the first block of the loop body or skips the loop, jumping to a block that we call the *loop exit*. The variables that are modified by a statement in the loop are called *loop targets*. We can now over-approximate any number of loop iterations as follows: first, wipe out all information about the loop targets by inserting appropriate havoc statements on entry to the loop body; then, replace each back edge of the loop with an edge to the loop exit. We can think

of this transformation as eliminating loops using trivial loop invariants. In fact, if the user or some preceding analysis provides loop invariants, they can be incorporated into the transformation to increase precision (see [3]).

Once our program is loop free, procedure $\mathsf{Transform}(P)$ transforms it to passive form. First, we apply a *single assignment* transformation [7] where auxiliary variables are introduced to ensure that each program variable is assigned at most once per execution path [11]. The general idea is to replace each read of a variable by the auxiliary variable that represents its value at that point in the program, and to introduce a new auxiliary variable for every write. For example, an assignment $x := x + 1$ may be transformed into $x_{k+1} := x_k + 1$, where k is some sequence number (see [3, 11] for details). Second, since no assignment of an auxiliary variable is preceded by a use of that variable, we can replace each assignment $x_k := e$ by an assume statement $\mathbf{assume}(x_k = e)$.

The following proposition states soundness of the transformation into loop-free passive form.

Proposition 1. *For any program P, the set of doomed program points of program $\mathsf{Transform}(P)$ is a subset of the doomed program points of program P.*

The proof relies on the fact that the program obtained from loop elimination preserves all admissible executions of the original program. Furthermore, there is a mapping from executions of the loop-free program to executions of the passive program that preserves admissibility.

Error Verification Conditions. We now describe how we generate an error verification condition for a given program point in a loop-free passive program.

Recall that the *weakest precondition* $wp.S.Q$ of a statement S with respect to predicate Q describes the pre-states of S from which every execution of S terminates normally in a state satisfying Q [8]. Thus, if the weakest precondition $wp.S.true$ is universally valid, then all executions of statement S are admissible. Therefore, weakest preconditions are used for generating verification conditions that prove program correctness.

We can use a similar approach to check for doomed program points. The *weakest liberal precondition* of a statement S with respect to a predicate Q describes the pre-states of S from which every terminating execution of S ends in Q [8]. Thus, $wlp.S.false$ is the set of all states such that any normally terminating execution of S ends in a state satisfying *false*, which means that there are no executions of S that terminate normally. Thus, weakest liberal preconditions allow us to precisely characterize statements with guaranteed errors.

Proposition 2. *Let S be a passive loop-free program. If $wlp.S.false$ is universally valid, then all executions of S are inadmissible.*

The proof of Proposition 2 goes by structural induction over S using the predicate transformer semantics of passive loop-free programs from [20] that is given in Table 1. Hereby, the statement $S \,\square\, T$ stands for non-deterministic choice between statements S and T. Since our program is in passive form, the

Stmt	*wp.Stmt.Q*	*wlp.Stmt.Q*
assert E	$E \wedge Q$	$E \implies Q$
assume E	$E \implies Q$	$E \implies Q$
$S; T$	$wp.S.(wp.T.Q)$	$wlp.S.(wlp.T.Q)$
$S \,\square\, T$	$wp.S.Q \wedge wp.T.Q$	$wlp.S.Q \wedge wlp.T.Q$

Table 1: Semantics of predicate transformers

statements of the program do not affect the program state. The only effect of a passive statement is to choose whether the execution is admissible. As described in [11], this allows us to capture the semantics of a passive program in terms of so called *outcome predicates*. Given a statement S, the predicate $N.S$ denotes the pre-states of S from which the execution of S may be admissible, while predicate $W.S$ denotes the pre-states from which the execution of S may be inadmissible. The formal semantics of these two outcome predicates is given in Table 2. Using

Stmt	*N.Stmt*	*W.Stmt*
assert E	E	$\neg E$
assume E	E	*false*
$S; T$	$N.S \wedge N.T$	$W.S \vee (N.S \wedge W.T)$
$S \,\square\, T$	$N.S \vee N.T$	$W.S \vee W.T$

Table 2: Semantics of outcome predicates

these outcome predicates, it is shown in [17] that weakest preconditions can be characterized as

$$wp.S.Q \equiv \neg(W.S) \wedge (N.S \implies Q) \ .$$

Similarly, we can characterize weakest liberal preconditions as follows.

Proposition 3. *Let S be a program in passive form and Q a predicate. Then the following equivalence holds:*

$$wlp.S.Q \equiv (N.S \implies Q) \ .$$

The size of predicate $N.S$ is linear in the size of statement S. We can, thus, conclude that the size of the weakest liberal precondition $wlp.S.false$ is also linear in S. In contrast, the weakest precondition is worst-case quadratic.

For a program point ℓ in a loop-free passive program P we denote by $st(\ell, P)$ the statement that corresponds to the subprogram following this program point. The statement $st(\ell, P)$ can be computed from the control-flow structure of the program as follows: given the block

$$\ell : \ S; \textbf{goto}\, \ell_1, \ldots, \ell_n$$

for a program point ℓ in P, the corresponding statement $st(\ell, P)$ is defined recursively as

$$st(\ell, P) \stackrel{\text{def}}{=} S; (st(\ell_1, P) \,\square \cdots \square\, st(\ell_n, P)) \ .$$

For the terminating program point *Term*, the statement $st(Term, P)$ is the empty statement. Since program P is loop-free, statement $st(\ell, P)$ is well-defined for all program points.

We can now define the error verification condition $\mathsf{EVC}(\ell, P)$ as follows:

$$\mathsf{EVC}(\ell, P) \stackrel{\text{def}}{=} wlp.st(\ell, P).false \ .$$

Hereby, $wlp.st(\ell, P)$ is computed according to Proposition 3. From Proposition 2 we conclude that error verification condition generation is sound.

Proposition 4. *Let P be a program in loop-free passive form and ℓ a program point in P. Then, ℓ is doomed if the error verification condition $\mathsf{EVC}(\ell, P)$ is valid.*

Soundness. From Proposition 1 and 4 we can now conclude the soundness of our algorithm.

Theorem 1. *Procedure* Exorcise *is a sound algorithm for the error verification problem.*

Avoiding exponential blow-up. The weakest liberal precondition for statement $st(\ell, P)$ and a predicate Q can be computed recursively as follows:

$$wlp.st(\ell, P).Q = wlp.S. \begin{pmatrix} wlp.st(\ell_1, P).Q \\ \wedge \ \ldots \\ \wedge \ wlp.st(\ell_n, P).Q \end{pmatrix} \tag{1}$$

However, there is a crucial problem when one computes $wlp.st(\ell, P).false$ using Equation 1. If a program point ℓ' is reachable from ℓ then $wlp.st(\ell', P).false$ occurs as a subformula in $wlp.st(\ell, P).false$ as many times as there are paths in the control-flow graph from ℓ to ℓ'. This can lead to an exponential blow-up in the size of the resulting verification condition. We follow the idea of [3] and [17] and avoid this blow-up by defining Equation 1 in the underlying logic. For this purpose, we introduce auxiliary Boolean variables B_ℓ for $wlp.st(\ell, P).false$ and build the formula

$$\begin{aligned} F_{Bdef} : \bigwedge_{\ell \in P} & \left(B_\ell \equiv wlp.S.(B_{\ell_1} \wedge \cdots \wedge B_{\ell_n})\right) \\ & \wedge (B_{Term} \equiv false) \end{aligned}$$

Using this definition we redefine our EVC as follows.

$$\mathsf{EVC}(\ell, P) \stackrel{\text{def}}{=} F_{Bdef} \implies \neg B_\ell \, .$$

6 Extended EVC Generation

Until now, our algorithm only detects errors that occur in every path that starts in a program point ℓ. Code that precedes the program point is not taken into account when checking $\mathsf{EVC}(\ell, P)$. An example is given in Figure 6. The program point in line 3 is doomed, since the value of i is never a valid pointer at this program point. To prove this, one needs to consider the assignment in line 1. It is not enough to just consider every conditional block by itself. We detect program points in conditional blocks by introducing a new variable R_ℓ that indicates that the block B_ℓ was reached. The variable is initially false; an assignment that sets the variable to true is added to the block and we change the post-condition from *false* to $R_\ell \implies$ *false*.

```
1  int *i = 0;
2  if (k != 0)
3      *i = 3;
```

Fig. 6: PathProg

We introduce all reachability variables at the same time but set only one R_ℓ to false in the precondition. Thus, we do the following transformation of the program (before passifying the program). The Block $\ell : S; \mathbf{goto}\, \ell_1, \ldots, \ell_n$ is transformed to

$$\ell : R_\ell := \mathit{true}; S; \mathbf{goto}\, \ell_1, \ldots, \ell_n$$

and in the block Term we add the assertion

$$\mathbf{assert}(\bigwedge_{\ell \in P} R_\ell)$$

We compute F_{Bdef} for this annotated program as described in the previous section.

We now redefine our EVC. To check if there is a doomed program point in block B_ℓ we check the validity of

$$\mathsf{EVC}(\ell, P) \stackrel{\text{def}}{=} \neg R_\ell \wedge F_{Bdef} \implies \neg B_{start}\,.$$

Proposition 5. *Let P be a program in loop-free passive form and ℓ a program point in P. Then, ℓ is doomed if and only if the error verification condition $\mathsf{EVC}(\ell, P)$ is valid.*

As for the weaker Proposition 4, we can conclude from Proposition 1 and 5, that our algorithm is sound.

Completeness. Our algorithm is not complete for unrestricted programs because the error verification problem is in general undecidable[4]. However, if we start

[4] An instance of the error verification problem is to decide whether a given program never terminates.

from a loop-free program then the algorithm is complete under the assumption that the generated verification conditions are expressible in some logical theory for which validity checking is decidable.

Faster Theorem Prover Interaction. Instead of checking validity of EVC, we check the unsatisfiability of its negation

$$\neg R_\ell \wedge F_{Bdef} \wedge B_{start}\,.$$

Since the part $F_{Bdef} \wedge B_{start}$ does not change, we can push it as axiom and then just check unsatisfiability of $\neg R_\ell$ for each Block B_ℓ. This way the theorem prover has to parse the main part of the verification condition only once and can reuse lemmas that are derived from this formula.

7 Implementation and Experiments

We have built a prototype implementation of algorithm Exorcise on top of Boogie [2] and applied it to a C# version of the Findbugs Null Pointer Microbenchmark [13] and the examples in Figures 1, 2, and 3. For our prototype, we have used heuristics for dealing with loops and function calls. We describe these heuristics in the following.

Loops: In order to increase detection rate we unroll each loop body three times. One unrolling for the first, the last, and an arbitrary iteration. For the arbitrary iteration, we set all variables modified inside the loop body to havoc at the beginning and at the end of the unrolled iteration. The back edges of the original loop are replaced. For the first iteration, the back edges are changed to the first block of the arbitrary and the last iteration. For the arbitrary iteration, the back edge is changed to the last iteration. The last iteration will always leave the loop. This simple unrolling allows us e.g., to find doomed program points caused by iteration across array bounds as in Figure 2 as well as simple cases of non-termination where an iterator is not iterated inside the loop body.

By unrolling the first and last iteration, we might have introduced unreachable control flow (e.g., there is a condition in the loop body that is satisfied only in the third iteration). We are not allowed to check these program points since they might be false positives. Thus we only check the first block of the unrolled iterations. Most guaranteed errors inside the loop body will propagate to this point.

Function Calls: We handle functions calls by simple inlining. As for loops, we have to be careful that we do not introduce additional control flow paths. Thus, we check only the first block of an inlined function.

Obviously, inlining will not scale, since we still have to check all functions separately. Therefore, we inline only up to a certain depth and use trivial contracts for any further calls. So far, we have experienced that this is not as bad as it would seem, since doomed program points tend to have a local scope, i.e., in practice there are only few guaranteed errors that involve multiple function calls.

Experiments: The results of our experiments are shown in Table 3. The result columns show whether the respective tool detects an existing error (true positive), a non-existing error (false positive), misses an error (false negative), or does not produce a warning on correct code (true negative). The benchmark contains

		Exorcise		Findbugs [13]	Spec# [2]	
program	incorrect?	time (in ms)	result	result	time (in ms)	result
fp1	no	156	true neg	true neg	39	true neg
tp1	yes	171	true pos	false neg	27	true pos
fp2	no	160	true neg	true neg	35	true neg
tp2	yes	160	true pos	false neg	27	true pos
fp3	no	175	true neg	true neg	50	true neg
tp3	yes	187.5	true pos	true pos	58.5	true pos
tp4	yes	109.2	true pos	true pos	35	true pos
fp4	no	171	true neg	true neg	43	true neg
tp5	yes	152	true pos	true pos	15.5	true pos
tp6	yes	144	true pos	true pos	31	true pos
itp1	yes	109.2	true pos	false neg	15.6	false neg
ifp1	no	93.6	true neg	true neg	46.8	true neg
itp2	yes	15.6	true pos	true pos	0	false neg
itp3	yes	46.8	true pos	true pos	15.6	false neg
TRIVIAL	yes	179.5	true pos	true pos	54.5	true pos
LOOP	yes	699	true pos	false neg	129	true pos, 3 false pos
COMPLEX	no	246	true neg	true neg	43	2 false pos
Total Time		3.2 s		4.53 s	0.67 s	

Table 3: Comparison of Exorcise, Findbugs and Spec# on the Findbugs Null Pointer Micro Benchmark and the example from Figures 1, 2, and 3. The columns list the analyzed function, whether it contains a bug, the running time and result of Exorcise, the result of Findbugs, and running time and result of Spec#. Results can either be true positives if an error is found, true negatives if no error is reported on correct programs, false positives if a non-existing error is reported, or false negatives if an existing error is overlooked.

nine functions with one null pointer error and five without. Our algorithm is able to detect all nine null pointer errors without producing false positives. Findbugs misses three errors, but does not produce false positives either. Spec# misses true positives and produces false positives if no further information is provided. More benchmarks on the Findbugs Null Pointer Micro Benchmark can be found in [13].

All benchmarks were executed several times on a 2.4 GHz machine with 2 GB of RAM running Windows XP. Our approach is slower than Spec#. While Spec# checks each function once, our tool has to check each block of a function

separately in the worst case. We are working on optimizations concerning the size and construction of the formula and the interaction with the theorem prover, but for the worst case our algorithm will always be slower than Spec#.

The times measured for Findbugs are not directly comparable to those for our analysis, since Findbugs computes many pieces of additional information. For larger programs, Findbugs should be faster than Spec# and Exorcise, but so far we have not found a good benchmark that is available in both C# and Java.

The big bottleneck of this approach is that our algorithm has to check for each block if it contains a doomed program point. Using the insight that a block that has only one successor will be doomed if the successor is doomed, we can reduce the number of checked blocks. Furthermore, using the optimization from the previous section, we observe that the theorem prover, after checking the first block, can reuse large parts of its work for the remaining blocks. Table 4 shows how much time our implementation spends on constructing the EVC, checking the first block, and checking all further blocks. Looking at e.g., the function Loop, we observe that the time spent on checking all blocks but the first one is less then checking the first block.

program	# queries	EVC construction (ms)	1st block (ms)	avg ms/block
tp1	5	17	134	2
Trivial	3	17	150	2
Loop	7	62	391	23
Complex	5	18	166	5

Table 4: Number of total blocks checked and the time (in ms) consumed for constructing the EVC, checking the first block, and the average time for all further blocks.

8 Conclusion

The main contribution of this work is the idea of error verification and the demonstration that this idea can be realized in practice. We have shown that error verification can easily be integrated in extended static checkers or program verifiers that provide the infrastructure for generating verification conditions and automatic theorem provers to check them. We therefore believe that this idea can now be adopted and extended by many others. We see a huge potential in this work as this can be a formal method which is applicable by every programmer. Using the fact that it can be built on top of e.g., Spec#, it also allows the programmer to annotate his program using e.g., pre- and postconditions to see if certain properties are always violated. This allows a smooth learning curve towards the use of full program verification.

We see much room for further improvements of our method. For instance, we want to optimize error verification by developing specialized techniques for finding *correct* executions, so that error verification conditions are quickly recognized as invalid. Doomed program points are sparse; i.e., almost all generated error verification conditions are not valid in practice (this is in contrast with the usual verification conditions, for correctness). Every programmer's experience confirms the intuition that it is easier to find a correct execution (for a program fragment that has no guaranteed error) than to find an incorrect one (for a program fragment that may lead to an error). This gives an interesting potential for optimization.

Maybe the best reason to use our approach is that there is no argument against it: our method is fully automatic and it remains invisible to the user as long as no doomed program point is found. If a warning is emitted, then this is a definite indication that the program is incorrect.

References

1. Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Workshop on Program Analysis for Software Tools and Engineering, PASTE*, pages 1–8. ACM, 2007.
2. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.
3. Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Workshop on Program Analysis for Software Tools and Engineering, PASTE*, pages 82–87. ACM, 2005.
4. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
5. Shaunak Chatterjee, Shuvendu Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. A reachability predicate for analyzing low-level software. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 4424 of *LNCS*, pages 19–33. Springer, 2007.
6. Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research, February 2009.
7. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 13(4):451–490, 1991.
8. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
9. David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

10. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation, PLDI*, pages 234–245. ACM, 2002.
11. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Annual ACM Symposium on the Principles of Programming Languages, POPL*, pages 193–205. ACM, 2001.
12. Thomas A. Henzinger, Ranjit Jhala, Rubak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *Model Checking Software, 10th International SPIN Workshop*, volume 2648 of *LNCS*, pages 235–239. Springer, 2003.
13. David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *Workshop on Program Analysis for Software Tools and Engineering, PASTE*, pages 9–14. ACM, 2007.
14. David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. *ACM SIGSOFT Software Engineering Notes*, 31(1):13–19, 2006.
15. Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 19(6):1031–1052, 1997.
16. Viktor Kuncak. *Modular Data Structure Verification.* PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.
17. K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters, IPL*, 93(6):281–288, 2005.
18. K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, June 2008. Available at http://research.microsoft.com/~leino/papers.html.
19. David C. Luckham and Norihisa Suzuki. Verification of array, record, and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 1(2):226–244, 1979.
20. Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 11(4):517–561, 1989.
21. Philipp Rümmer and Muhammad Ali Shah. Proving programs incorrect using a sequent calculus for Java Dynamic Logic. In *Tests and Proofs, First International Conference, TAP 2007*, volume 4454 of *LNCS*, pages 41–60. Springer, 2007.