Proving that systems code eventually does something good

Byron Cook
Microsoft Research
Alexey Gotsman
Cambridge University
Andreas Podelski
MPI-Saarbrücken
Andrey Rybalchenko
EPFL
Moshe Vardi
Rice University

Abstract
In recent years we have seen great progress made in the area of automatic source-level static analysis tools. However, all of today’s program verification tools are limited to properties that guarantee the absence of bad events (called safety properties). Until now no formal software analysis tool has provided support for automatically proving properties that ensure that good events eventually happen (called liveness properties), i.e. no tool has supported proving the absence of liveness bugs. In this paper we present such an automatic tool, called VIVO, which handles both safety and liveness properties of large systems written in C. The tool attempts to prove properties of programs; failed proofs result in counterexamples which are presented back to the user. We have used the tool to automatically prove critical liveness properties of Windows device drivers. Furthermore, we have found several previously unknown liveness bugs.

1 Introduction
As computer systems become ubiquitous, expectations of system dependability are rising. To address the need for improved software quality, practitioners are now beginning to use static analysis and automatic formal verification tools. Tools like Static Driver Verifier (SDV) [4], COVERITY [17] and PREFIX [10], for example, are now regularly being applied to industrial-grade systems code.

It should be noted, however, that all of the proof-based software verification tools\(^1\) are currently limited to safety properties [1, 2]. No software analysis tool supports the other remaining set of properties: liveness properties. Consider SDV as an example. SDV is packaged with 60 safety specifications that are automatically proved of the device driver to which SDV is being applied. Many of these properties specify temporal connections between kernel APIs that acquire resources and APIs that release resources. For example:

A device driver should never call KeReleaseSpinlock unless it has already called KeAcquireSpinlock.

Why is this a safety property? Because any counterexample to the property will be a finite execution through the device driver code. If counterexamples are always finite, then the property is a safety property. We can think of safety properties as a guarantee that specified bad events will not happen (i.e. calling KeReleaseSpinlock before calling KeAcquireSpinlock).

SDV cannot check the equally important dual property:

If a driver calls KeAcquireSpinlock then it must eventually make a call to KeReleaseSpinlock.

This is a liveness property, as the counterexample may not be finite. More precisely, a violating trace is one in which KeAcquireSpinlock is called but it is not followed by a call to KeReleaseSpinlock. This trace may be finite (reaching termination) or infinite. We can think of liveness properties as ensuring that certain good things will eventually happen (i.e. that each call to KeAcquireSpinlock will eventually be followed by a call to KeReleaseSpinlock)—words like eventually are usually used in the English-level descriptions of liveness properties. When systems hang, a not uncommon event, they hang due to violations of liveness properties.

In this paper we describe a new tool, called VIVO, which automatically constructs correctness proofs for both liveness and safety properties for software systems. Properties are described in a new specification language called SLIC\(^+\), which is an extension to the safety-property-only language used by SDV. When given a

\(^1\)i.e. tools that only report that code does not violate a property when a mathematical proof of such can be found.
property description and a program, VIVO attempts to construct a correctness proof. If a proof is constructed then the property is guaranteed to hold. Conversely, the proof fails and a potential counterexample is produced. If the counterexample is an non-terminating execution, then it is presented via a finite description, to enable the programmer to analyze it.

VIVO is the first known liveness prover that can handle large systems written in C. It supports infinite-state programs with arbitrary nesting of loops and recursive functions, pointer-aliasing and side-effects, function-pointers, etc. VIVO’s scalability leverages recent advances in program termination analysis (e.g. TERMINATOR [13, 14, 7] and POLYRANK [9, 8]). Using an approach described in [28], VIVO takes a liveness property and a program and constructs an equivalent fair termination problem—fair termination is a generalization of termination and is defined later. VIVO’s implementation of fair termination analysis is an extension to the algorithm used in TERMINATOR.

While the theory of liveness verification was established almost 20 years ago, it has not yet been reduced to practice due to lack of practical algorithms. This paper reports on the first such reduction to practice and its application to systems code. We make several novel contribution to the state-of-the-art:

- SLIC+, a simple and easy-to-use language for specifying both liveness and safety properties (Section 2). We also provide a set of example specifications that demonstrate how SLIC+ can be used to check properties of systems-level code.

- A method of extending a TERMINATOR-like algorithm to fair termination (Section 3),

- Most importantly, we present experimental results that demonstrate the viability of the approach to systems-level code (Section 4). In this paper we describe experiments with VIVO on Windows device drivers ranging in size from 1K to 10K lines of code. We believe that the experience that we have had with Windows device driver will match the results that users will have in other systems-level domains (networking code, Linux device drivers, etc).

The remaining sections introduce the SLIC+ language (Section 2), describe the algorithmic background of VIVO (Section 3), which can be skipped upon first reading, present the experimental results (Section 4), provide an overview of the related work (Section 5), and conclude the paper (Section 6).

2 Writing liveness properties

In this section we describe VIVO’s language for specifying properties for software systems, called SLIC+. Note that this section is organized in the following manner. We first provide information on SLIC+’s language definition (Section 2.1). Then, we present a set of example properties (starting at Section 2.2). Some readers may prefer to look at the examples before reading the more precise language definition.

2.1 A language for writing safety and liveness properties

SLIC+ is an extension of SLIC [6], which is used to specify temporal safety properties in SDV [4]. The language is designed to specify API-usage rules for client-code (like Windows device drivers and their use of the Windows Driver Model API as described in [4]). For this reason SLIC+ is designed such that programmers do not need to modify or annotate source code—the code is typically not available when writing the specification.

In essence SLIC+ allows us to write simple automata that specify how APIs should and should not be used.2 The syntax of the SLIC+ language is defined in Figure 1. A SLIC+ specification consists of three basic parts:

A state structure declaration. The state structure defines a set of state variables that are maintained by the SLIC+ automaton during its execution. The variables can be of any scalar C type or pointer.

A list of transfer functions. Transfer functions define transitions taken by the automata as API operations are invoked and return. Each transfer function has two parts: a pattern specification and a statement block that defines the transfer function body. A pattern specification usually has two parts: a procedure identifier id (i.e. the name of the API) and one of two basic event types (event): entry, exit. These events identify the program points in the named procedure immediately before its first statement and immediately before it returns control to the caller. The any pattern can be used to trigger the event throughout the code.

A list of fairness constraints. The fairness constraints are given as pairs of Boolean expressions inside of the scope of the fairness keyword. Each Boolean expression is guarded by a pattern.

Fairness constraints are an extension of SLIC, and can be used to rule out counterexamples in which the environment is not “fair”. A fairness constraint

2Formally, SLIC+ defines automata on infinite words together with strong fairness, also known as compassion, requirements [23].
restricts attention to non-terminating paths in which either the first Boolean expression succeeds, i.e. it is invoked and evaluates to true, only finitely often or the second Boolean expression succeeds infinitely often. A non-terminating execution can be a counterexample only if it satisfies all of the fairness constraints given. The body of a transfer function is written in a simple imperative C-like language. One important control construct is missing from the language of SLIC\textsuperscript{+} statements: loops. This means that transfer functions always terminate. Safety and liveness properties are expressed using error, set or unset. The error statement is used to explicitly signal that an unsafe state has been reached (i.e. a safety property has been violated). The set and unset statements are SLIC\textsuperscript{+} liveness extensions wrt. SLIC. When the property calls set then a non-terminating execution through the SLIC\textsuperscript{+} property in which unset is never called represents a liveness violation. If such a path exists, then the message passed to set is given to the user. VIVO also checks the related safety property: calls to set must be answered by calls to unset in every terminating execution.

The function nondet() is used to specify non-deterministic value introduction. That is, nondet() returns an arbitrary value. A proof of the specification should then take any valuation into account.

The expression sub-language (expr) of SLIC\textsuperscript{+} is the pure expression language of C, without state update operators (++, --, etc.), pointer arithmetic, or the address-of operator (&). Dereferencing pointers via * and -> is allowed. The identifiers in this language are of several forms: regular C-style identifiers behave as expected, the $i$ refers to $i^{th}$ formal parameter, and the identifier $\$return$ is used to refer to the return value, which is accessible at the exit event.

**Syntax**

```plaintext
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Comment</th>
</tr>
</thead>
</table>
| $\text{state}$ | $\text{transFun}^+$,
|                 | $\text{fairness}^+$,
| $\text{fieldDecl}$ | $\text{ctype id = expr}$ |
| $\text{transFun}$ | $\text{pattern stmt}$ |
| $\text{pattern}$ | $\text{id . event | any}$ |
| $\text{event}$ | $\text{entry | exit}$ |
| $\text{stmt}$ | $\text{id = expr}$,
|                 | $\text{stmt ; stmt}$ |
|                 | $\text{if (choose) stmt [else stmt]}$ |
|                 | $\text{error (message);}$ |
|                 | $\text{unset();}$ |
|                 | $\text{return [ expr ];}$ |
|                 | $\text{stmt}$ |
| $\text{choose}$ | $\text{nondet()}$ |
| $\text{expr}$ | $\text{id | expr op expr | ...}$ |
| $\text{id}$ | $\text{identifier}$ |
|                 | $\$int$ |
|                 | $\$return$ |
| $\text{fairness}$ | $\text{fairness (pattern (expr) pattern (expr))}$ |
```

Figure 1: Syntax of the SLIC\textsuperscript{+} language.

**Instrumentation.** In order to prove that a specification holds of a given program, VIVO instruments the specification into the program. That is, VIVO walks through the input program and finds all of the program locations in which the patterns described in the specification could be triggered. (Formally, this constructs the product of the original program with the SLIC\textsuperscript{+} automaton [28].) A new program—called the instrumented program—is produced in which the code found in the transfer functions is grafted into the original program. During this phase VIVO also performs the pointer analysis described in [16], which constructs an overapproximation of all the potential pointer aliasing relationships. This allows VIVO to statically find function-call events when functions are called by pointer.

To prove that a program does not violate a safety condition we need to prove that the instrumented program can never make a call to the error function. If the proof fails then VIVO outputs a counterexample, which is a trace though the program that starts at the beginning
Figure 2: A SLIC+ specification for whole program termination for C programs. It requires that every program execution, which starts by entering main, must eventually stop by either leaving main or calling exit.

Figure 3: A small example program. The program does not terminate if the scanf function from some point on always assigns the value 25 to the variable x.

2.2 A simple example

Figure 2 gives a simple example of a SLIC+ specification, which specifies program termination in C. This example does not require state variables, hence we do not declare any variables in the state structure declaration. The example has three transfer functions: one for the case where the program starts execution (i.e. the start of the main function), and two others for the cases when the program ceases an execution via exiting the main procedure or making a call to the exit function. A violation of this property is an execution through the program that starts at the entry of main (i.e. \texttt{pc = main.entry}), but never calls exit (\texttt{pc = exit.entry}) or reaches the exit-point of main (\texttt{pc = main.exit}).

This is because program location 1.1 is at the beginning of the first basic-block on main's control-flow graph. The main.exit transfer function causes two snippets of code to be instrumented: one at program location 1.1 and one at 12.1. The code in Figure 3 does not call exit(), meaning that the exit.entry transfer function is not instrumented in Figure 4. VIVO then performs reasoning on this new instrumented program. VIVO will attempt to prove that there does not exist an execution in the instrumented program such that set is called and unset is never called afterwards. Note that this is possible precisely when there exists an execution that enters main, but never returns from it and never calls exit.

In the case of Figure 4, the proof will fail because the program will loop forever if the user always enters 1.1. VIVO will then produce a counterexample in the form of a lasso trace:

\[
\text{Stem} = 1 \rightarrow 1.1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5,
\]
\[
\text{Cycle} = 6 \rightarrow 7 \rightarrow 9 \rightarrow 12 \rightarrow 5.
\]

The numbers in this trace represent the line numbers in the program. This trace is called a lasso trace or a pant-handle trace because it is the composition of a stem and a cycle: the counterexample is a trace in which Stem is executed once followed by an infinite repetition of Cycle. Note that this counterexample can easily be mapped back
to the original program by removing the line-numbers that point to instrumented code.

The following modification of the program makes it terminating for any sequence of user inputs. We replace line 9 in Figure 3:

9 } else if (x>25) {

with

9 } else if {

Then, when trying to prove the termination property (Figure 2) of the fixed program, ViVo would report that it has found a proof.

2.3 Using state in SLIC+ rules

Our second example shows how a SLIC+ definition can maintain internal state. It is based on a specification of how a device driver is supposed to modify the processor’s interrupt request level (IRQL) that controls which kinds of interrupts are to be delivered. Two functions are involved: KeRaiseIrql(x) raises the IRQL to the value of x and KeLowerIrql(y) lowers the IRQL to y. A driver must match these operations correctly: if it raises the IRQL then it must subsequently lower it back to the original value.

Figure 5 shows how this specification is modelled in SLIC+. This example demonstrates SLIC+’s state structure, which in this case contains an integer variable irql that stores the IRQL-value at the time of the call to KeRaiseIrql. Two transfer functions are included in the specification: one calling set if KeRaiseIrql is called (with a few side conditions), the other calling unset only if KeLowerIrql is called appropriately.

2.4 Combining liveness and safety

SLIC+ specifications can contain both liveness and safety properties. In the case of Figure 6, we have a safety property mixed together with the liveness property from Figure 5. ViVo will search for at least one violation of the properties, either of safety or liveness. The safety property in this case specifies that KeRaiseIrql should not be used to lower the IRQL, and KeLowerIrql should not be used to raise IRQL.

2.5 Using fairness constraints

Consider Figure 7, which specifies a fairness constraint. Fairness constraints in SLIC+ come as pairs of Boolean expressions guarded by patterns. In this

Figure 5: A SLIC+ liveness property involving the Windows kernel APIs KeRaiseIrql and KeLowerIrql. The SLIC+ macro KeGetCurrentIrql refers to a variable in the OS environment model.

Figure 6: A SLIC+ specification defining both liveness and safety properties involving the Windows kernel APIs KeRaiseIrql and KeLowerIrql.
3 Implementation

VIVO proves liveness properties by first translating them into a fair termination problem, and then applying a TERMINATOR-style termination prover that we modified (i) to account for the semantics of set/unset calls and (ii) to handle fair termination. The soundness of VIVO’s translation is based on theory developed in [28]. The modifications to the termination prover use the theoretical foundations developed in [25]. This section describes the first working and scalable implementation of these two ideas.

Our translation benefits from a uniform treatment of safety and liveness parts of a given SLIC specification. We do not require separate tools for handling safety and liveness.

The strategy of converting to fair termination allows us to use recent advances in termination proving. Unfortunately it is not possible to convert any liveness problem into termination. In some cases we must convert into fair termination, because some liveness properties only hold in conjunction with fairness constraints.

VIVO’s translation from liveness properties to fair termination ensures that infinite executions in which a call (or calls) to set is eventually followed by a call to unset are excluded from the consideration. Additionally, the second part of our translation guarantees that fairness constraints from SLIC specification are taken into account. That is: liveness properties are checked for fair infinite executions only. We present the translation in Section 3.2 after the example below.

3.1 Example: Proving the correct usage of KeRaiseIrql and KeLowerIrql

Before describing VIVO’s algorithm, we illustrate its application for proving a liveness property under fairness constraints.

Consider the program in Figure 8. Imagine that we would like to prove the that the program respects the property specified in Figure 5. VIVO first produces a new program in which the property has been instrumented into the code, see Figure 9.

In this path, we have a call to the function set is not answered by a call to unset because of the non-terminating loop between them. VIVO found out that if
```c
int irql = -1;

void main()
{
    int t_irql;
    int x;
    int i = 0;
    t_irql = KeGetCurrentIrql();
    if (irql == -1) {
        irql = KeGetCurrentIrql();
        set("......");
    }
    KeRaiseIrql(10);
    while (i < t_irql)
    {
        GetFlag(&x);
        if (x) {
            i = i + 1;
        }
    }
    if (t_irql == irql && irql > -1) {
        unset();
    }
    irql = -1;
    KeLowerIrql(t_irql);
}
```

Figure 9: The program constructed by instrumenting the property from Figure 5 into the program from Figure 8.

GetFlag eventually stops assigning non-zero values to the variable x then the loop will keep the value of i unchanged and thus will stop making progress towards termination and the subsequent call of unset.

We can classify such behavior of GetFlag as unrealistic, and require that it produces infinitely many non-zeros if called infinitely often. This requirement is formalized in SLIC+ as a fairness constraint:

```c
fairness {
    GetFlag.exit { 1; }
    GetFlag.exit { x; }
}
```

which is similar to the one for IoCreateDevice described in Section 2.5.

If we add this fairness constraint on GetFlag to our specification, then VIVO will ignore the above counterexample. For the fair iterations of the loop it will find a termination proof based on the counter variable i and thus will succeed in the verification of the liveness property.

### 3.2 Implementing fair termination

As we have seen from the example, VIVO implements a form of termination analysis that treats set and unset in a special way. Furthermore, in order to guarantee that VIVO can support any liveness property it provides an extension for proving fair termination. This section provides an informal description of how VIVO extends a TERMINATOR-like termination-analysis for these two features. A more mathematically rigorous treatment can be found in the preprint of our coming technical report [11].

#### 3.2.1 TERMINATOR algorithm overview

TERMINATOR implements a counterexample-driven method of searching for and then proving the validity of termination arguments, which are described below. Essentially, for each program location that resides at the beginning of a loop body, recursive function call, or goto statement, TERMINATOR attempts to find a termination argument that covers all possible executions through that program location. These special program locations are called cut-points [18, 22]. During TERMINATOR’s execution the termination argument is refined based on failed proof attempts. This is described extensively in [12] and [13]—however, for our purposes we only need to know two facts:

**Termination arguments** are binary relations on program states (following the transition invariants approach to temporal verification [26]). For example:

\[
T(s, t) \triangleq (t(x) \leq s(x) + 1 \land t(x) > 100) \\
\land (t(y) \leq s(y) + 1 \land t(y) > 100)
\]

(1)

The variables \(s\) and \(t\) represent program states. The idea is that \(s\) and \(t\) are mathematical functions from program expressions to values. In this case the binary relation \(T\) is taking in any two states (named \(s\) and \(t\)) and using the valuations of program expressions (in this case \(x\) and \(y\)) in the states to construct Boolean expressions.

**Binary reachability analysis** is the process of automatically attempting to prove that a binary relation \(T\) holds for every pair of states \(s\) and \(t\) with an important restriction: \(s\) is a reachable state via zero or more transitions from the beginning of the program’s main function and \(t\) is a state reachable from one or more transitions from the state \(s\).

TERMINATOR performs binary reachability analysis numerous times during a termination proof; it is executed at least once for each cut-point, and usually 2-3 times as the termination argument is refined based on failed proof attempts.

It is important to note that all of the changes to the TERMINATOR-style algorithm in order to support set, unset, and fairness can be made in the implementation.
Consider the case where we have the termination argument (1) from above. Imagine that we are proving that location 2 cannot be visited infinitely often in the loop from Figure 10. TERMINATOR checks the validity of $T$ via a transformation on the program: the program that TERMINATOR would produce in this case is shown in Figure 11. If the call to error is unreachable in this new program, then the termination argument $T$ is valid. Programs like the one in Figure 11 are a standard safety checking problem. This means that we can use a software model checker for safety such as SDV’s SLAM [5] or CMC [31] to validate the termination argument $T$.

The intuition behind this transition is that we have constructed a program that computes both a state $s$ and then for every state $t$ reachable from $s$ it checks if $T(s,t)$ holds. Note that $PC(s) = 2$ and $PC(t) = 2$ in the program from Figure 10. The program in Figure 11 first constructs $s$ by running the loop any number of iterations. The non-deterministic choice at line 2.6 means that a proof-based tool will consider all possible cases—having the effect of the full coverage. This means that we can use a software model checker for safety such as SDV’s SLAM [5] or CMC [31] to validate the termination argument $T$.

We observe that since the checking of liveness property is reduced to the verification of a termination argument by a reachability query, VIVO can verify SLIC+ specifications that contain both safety and liveness properties. This means when proving the non-reachability of error in the instrumented program, we do not make a distinction if a particular call to error originates from the safety or liveness part of the specification.

### 3.2.2 Binary reachability in VIVO

As mentioned earlier, in order to support VIVO’s form of fair termination analysis, we need only to modify the binary reachability analysis.

**Supporting set and unset.** In order to support set and unset we introduce two new global variables into the program being constructed for reachability analysis: $S$ is a variable representing whether or not set or unset was the last function called, and still_S is a variable used to represent the history of the valuations of $S$ during certain executions. The functions set and unset are then defined accordingly:
void set(string * msg) { S = 1; }
void unset() { S = 0; still_S = 0; }

We will discuss another operation on S and still_S momentarily.

Supporting fairness. For each fairness constraint, VIVO adds two auxiliary global variables. Let us suppose that we have a single constraint, resulting in two new fairness constraint variables, p1 and q1. During the execution of the program, if the first transfer function’s pattern is triggered then we assign the output of the function into p1, if the second transfer function’s pattern is triggered, we assign the output into q1. The new code that is placed at the cut-point in the fairness-based translation is shown in Figure 12.

If additional fairness constraints are included then the conditional statement is extended using additional conjunctions shown in Figure 13.

Example: Binary reachability in VIVO. Consider the code fragment from Figure 14. Imagine that we are trying to prove that whenever PPBlockInits is called that PPUnblockInits will eventually be called with the fairness constraint from Figure 7. Furthermore, assume the following conditions:

- VIVO has already instrumented the liveness property and constructed the analogous termination problem,
- the variables p1 and q1 are being used to represent the fairness constraint in Figure 7,
- VIVO is working to prove a termination condition at location 3,
- VIVO has already has constructed a candidate termination argument:

\[
T(s, t) \triangleq t(i) > s(i) \land t(i) < t(Pdolen) \\
\land s(Pdolen) = t(Pdolen).
\]

While performing binary reachability analysis, VIVO would produce the code in Figure 15 and then use a safety checker like SLAM on it. The differences between Figure 14 and Figure 15 are as follows:

- Lines 0.1 and 20.1 come from the property’s transfer functions.
- Line 3.1 in Figure 15 implements the first transfer function from the fairness constraint. In principle the update should appear at each line in the new

```
if (phase_change==1) {
  if (!!p1 || q1 && still_S) {
    if (!T(............)) {
      error();
    }
  } else {
    if (nondet() && S) {
      old_x = x;
      old_y = y;
      . = .
      . = .
      . = .
      . = .
      phase_change = 1;
      p1 = 0;
      q1 = 0;
      still_S = 1;
    }
  }
}
```

Figure 12: The program transformation that accounts for a fairness constraint in VIVO’s binary reachability analysis. The additional statement at each cut-point initializes the auxiliary variables p1 and q1 that track the satisfaction of the fairness constraint and the variables still_S and S that track the validity of the liveness property.

```
if ( (!p1 || q1) \\
&& (!p2 || q2) \\
. . \\
&& (!pN || qN) \\
&& still_S \\
) {
  if (! T(............)) {
    error();
  }
}
```

Figure 13: In case of multiple fairness constraints, the program transformation accounts for each of them by adding a pair of auxiliary variables p_i and q_i.
```c
1 PPBlockInits();
2 while (i<Pdolen)
3 {
4     DName = PPMakeDeviceName(lptName[i], PdoType, dcId[i], num);
5     if (!DName) { break; }
6     RtlInitUnicodeString(&deviceName, DName);
7     status = IoCreateDevice(fdx->do,PDOSZ,&deviceName,0,0,TRUE,Pdo[i]);
8     if (STATUS_SUCCESS != status) {
9         Pdo[i] = NULL;
10        if (STATUS_OBJECT_NAME_COLLISION == status) {
11           ExFreePool(DName);
12           num++;
13           continue;
14        }
15        break;
16    } else {
17        i++;
18    }
19 }
20 num = 0;
21 PPUnblockInits();
```

Figure 14: Example code from a Windows device driver dispatch routine. The correct behavior of the code depends on the fairness assumption from Figure 7.

<table>
<thead>
<tr>
<th>Driver</th>
<th>Time (seconds)</th>
<th>LOC</th>
<th>Bugs found</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>1K</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>314</td>
<td>7K</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2344</td>
<td>15K</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3122</td>
<td>20K</td>
<td>1</td>
</tr>
<tr>
<td>1R</td>
<td>16</td>
<td>1K</td>
<td>0</td>
</tr>
<tr>
<td>4R</td>
<td>3217</td>
<td>20K</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: Experimental results (Part 1): VIVO on 6 Windows device drivers. The property proved is displayed in Figure 16. The bug in driver 1 was known. The bug in driver 5 was not known before. Drivers 1R and 4R are repaired versions of driver 1 and 4 respectively.

<table>
<thead>
<tr>
<th>Driver</th>
<th>Time (seconds)</th>
<th>LOC</th>
<th>Bugs found</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>1K</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>188</td>
<td>7K</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>271</td>
<td>15K</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>T/O</td>
<td>20K</td>
<td>T/O</td>
</tr>
</tbody>
</table>

Table 2: Experimental results (Part 2): VIVO on 4 Windows device drivers. The property being checked is displayed in Figure 17.
0.1 set("........");
1 PPBlockInits();
2 while (i<Pdolen)
3 {
3.1 pl = 1;
3.2 if (phase_change==1) {
3.3 if (!(i>old_i && i<Pdolen)) { error(); }
3.4 }
3.5 }
3.7 } else {
3.8 if (nondet() && S) {
3.9 old_i = i;
3.10 phase_change = 1;
3.11 pl = 0;
3.12 q1 = 0;
3.13 still_S = 1;
3.14 }
3.15 }
3.16 DName = PPMakeDeviceName(lptName[i], PdoType, dcId[i], num);
5 if (!DName) { break; }
6 RtlInitUnicodeString(&deviceName, DName);
7 status = IoCreateDevice(fdx->do,PDOSZ,&deviceName,0,0,TRUE,Pdo[i]);
7.1 q1 = (status != STATUS_OBJ_NAME_COLLISION);
8 if (STATUS_SUCCESS != status) {
9 Pdo[i] = NULL;
10 if (STATUS_OBJECT_NAME_COLLISION == status) {
11 ExFreePool(DName);
12 num++;
13 continue;
14 }
15 break;
16 } else {
17 i++;
18 }
19 } num = 0;
20 unset();
21 PPUnblockInits();

Figure 15: Code produced while performing binary reachability analysis on the code from Figure 14.

<table>
<thead>
<tr>
<th>Driver</th>
<th>Time (seconds)</th>
<th>LOC</th>
<th>Bugs found</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>62</td>
<td>1K</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>N/A</td>
<td>7K</td>
<td>N/A</td>
</tr>
<tr>
<td>3</td>
<td>N/A</td>
<td>15K</td>
<td>N/A</td>
</tr>
<tr>
<td>4</td>
<td>T/O</td>
<td>20K</td>
<td>T/O</td>
</tr>
<tr>
<td>1R</td>
<td>35</td>
<td>1K</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3: Experimental results (Part 3): VIVO on 5 Windows device drivers. The property being checked is from Figure 5. The bugs in driver 1 are known. Driver 1R is a repaired version of driver 1. Drivers 2 and 3 are marked as N/A because they call neither KeRaiseIrql(...) nor KeLowerIrql(...), the proof is trivially true.

<table>
<thead>
<tr>
<th>Driver</th>
<th>Time (seconds)</th>
<th>LOC</th>
<th>Bugs found</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>1K</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td>7K</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1463</td>
<td>15K</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>T/O</td>
<td>20K</td>
<td>T/O</td>
</tr>
</tbody>
</table>

Table 4: Experimental results (Part 4): VIVO on 4 Windows device drivers. The property being checked is in Figure 2, together with the fairness condition in Figure 7. The main function in SDV’s OS model considers all possible calls to the driver’s dispatch routines—meaning that Figure 2 implements dispatch routine termination within in SDV. VIVO exceeded its 10,000 second timeout when applied to driver 4. The bug in driver 3 was previously unknown.
4 Experimental Results

In this section we describe the results from experiments on Windows device drivers. In order to perform the experiments we have integrated VIVO into SDV [4] by replacing SLAM with VIVO. Tables 1 through 4 contain the statistics from these experiments. We used three liveness properties involving the acquiring and releasing of resources (Figures 16, 17, and 5) together with the termination property in Figure 2. Note that SDV’s OS model has a main function that non-deterministically decides to call one of the driver’s dispatch routines—meaning that, in the case of SDV, Figure 2 represents dispatch routine termination for the entire device driver. The fairness constraint in Figure 7 was used throughout these experiments. We used a timeout threshold of 10,000 seconds and a memory limit of 1 gigabyte. T/O in the tables means that timeout limit was exceeded. LOC denotes “Lines of code”.

During these experiments we found several previously unknown bugs. Note that, if the number of “Bugs found” is 0, then this means that VIVO found a proof that the driver does not violate the specification.

5 Related work

VIVO builds on a large body of formal foundations, ranging from the formalization of the semantics of programs by fair discrete systems [23] and the automata-theoretic approach to temporal verification [28] to the more recent construction of fixpoint domains for abstract interpretation with fairness [25]. VIVO also uses recent advances in the area of automatic termination analysis (e.g. [9, 13]). From these foundations we have developed (to the best of our knowledge) the first known practical and automatic software verification tool for liveness properties.

The key difference between VIVO and finite-state model checkers for safety and liveness, e.g. SPIN [20], Bandera [15], and Java PathFinder [30], is that VIVO employs abstraction, while the others explore the state space as-is. Such an exploration will terminate with “Out-Of-Memory” for programs with infinite or very large state spaces. Abstraction provides effectiveness and efficiency to overcome this limitation.

It is possible to approximate a liveness property by a stronger safety property. One strategy is to bound the number of steps in which the eventually-event must occur. This does not scale well to large numbers of events, and it is often difficult to decide which finite number of steps should be taken (time is abstracted away in program analysis, so we can not say “in 10 seconds”). Another approach is to write a safety property that at least specifies that the liveness property will not be violated by any terminating executions. This is, in fact, what SDV does today: it constructs a number of main.exit transfer functions in SLIC that check that the liveness property is not violated. In this case SDV will miss any violations to liveness properties that involve non-terminating executions.

Temporal properties can be specified using temporal logics such as LTL [24]. It is well known, however, that such properties can also be specified using automata on infinite words [29] (in fact, LTL is less expressive
than such automata); see also [21]. SLIC+ properties can be viewed as automata on infinite words, where we combine fairness constraints with response requirements (set/unset). Such automata can define all \( \omega \)-regular properties [29]. To extend the expressive power of LTL to that of automata on infinite words, industrial languages such as ForSpec [3] extend LTL with a layer of regular expressions. Logic-based specifications have the advantage that they are easily combined, composed, and can be used to express deeper properties of code. Automata-based specifications have the advantage they are more like computer programs and are therefore easier for programmers to use. Compilation techniques described in [29] can be used to extend VIVO to logic-based specifications.

As described in [13], TERMINATOR has already been used to prove the termination of dispatch routines in Windows device drivers ([13] presents results on 23 device drivers). Termination, however, is just one liveness property. VIVO represents a generalization of this work to arbitrary liveness properties under fairness constraints. Verification of most busy-waiting loops in systems code typically requires dealing with fairness.

6 Conclusion

Liveness properties are much harder to prove than safety properties.\(^3\) Since automatic safety property checking has only recently become a reality, automatic liveness proving for real code has been considered impossible (see [27] for detailed explanation).

VIVO is the first known tool to break through this liveness checking barrier. VIVO is an interprocedural, path-sensitive, and context-sensitive liveness prover. It supports large programs, arbitrary nesting of loops and recursive functions pointer-aliasing and side-effects, function-pointers, etc. It is fully automatic and produces counterexamples in cases of failed proof attempts. It can handle systems code with its full complexity. We have applied VIVO to device drivers ranging in sizes from 1,000 to 20,000 LOC. These experiments were carried out using an integration of VIVO and the Windows SDV [4] product.

VIVO takes advantage of recent advances in termination analysis by converting the problem of liveness checking into termination checking. The scalability and support for real programming language features comes from the termination analysis. This paper has presented a language in which practical liveness properties can be expressed. Furthermore, we have adapted TERMINATOR’s algorithm for the checking of fair termination.

Through the use of examples we have also demonstrated a set of liveness properties that should be checked on Windows device drivers. In fact: over 1/3 of the safety specifications included in the today’s SDV distribution have analogous and equally important liveness properties that should be checked. Similar properties will exist in other programming domains, such as Linux device drivers, embedded software, real-time systems, etc. As one co-author learned while spending 2 years with the Windows kernel team:

- Formal verification experts have been taught to think only in terms of safety properties: liveness properties are considered too hard.
- Non formal verification experts (i.e. the people usually writing the code that needs to be verified) think equally in terms of both liveness and safety.

Limitations. A few notes about VIVO’s limitations:

- As program termination is an undecidable problem, VIVO’s analysis is not guaranteed to terminate.
- Counterexamples provided by VIVO are not guaranteed to be real counterexamples. VIVO attempts to prove that the property holds, not that it doesn’t hold. This is something that we are working on independently using ideas from [19]. Like termination, proving non-termination is undecidable.
- The validity of proofs constructed in VIVO rely on the soundness of the underlying safety checker. For example, VIVO may return a “proof” of correctness when the code is not correct due to the fact that VIVO’s safety checker (SLAM) assumes that integers are not bounded and that code is always being executed in a sequential setting. For this reason the proof is restricted to sequential code in which overflow cannot occur. If we were to use CMC [31] then (in principle) this restriction would be lifted.
- As previously described, VIVO uses pointer analysis to overapproximate the pointer aliasing relationships during instrumentation. In some cases this overapproximation may lead to aliasing relationships that do not occur in the program. In many cases false-aliasing relationships can be resolved later during binary reachability (as described in [13]), but not always. This can lead to false counterexamples.

\(^3\)Consider the small program \texttt{void main() \{ f(); g(); h(); \}}. To prove that the function \( f \) is always called before \( h \) is easy: in this case we need only to look at the structure of the control-flow graph. To prove that \( h \) is eventually called after \( f \) is hard: we first have to prove the termination of \( g \). In fact, in many cases, we must prove many safety properties in order to prove a single liveness property.
Availability. VIVO is currently not available. We are hoping to make a binary executable release in the summer. See the TERMINATOR website for more information:

http://research.microsoft.com/TERMINATOR.

References


