Proving That Programs Eventually Do Something Good

Byron Cook
Microsoft Research
bycook@microsoft.com

Alexey Gotsman
University of Cambridge
Alexey.Gotsman@cl.cam.ac.uk

Andreas Podelski
University of Freiburg
podelski@informatik.uni-freiburg.de

Andrey Rybalchenko
EPFL and MPI-Saarbrücken
rybal@mpi-sb.mpg.de

Moshe Y. Vardi
Rice University
vardi@cs.rice.edu

Abstract

In recent years we have seen great progress made in the area of automatic source-level static analysis tools. However, most of today's program verification tools are limited to properties that guarantee the absence of bad events (*safety properties*). Until now no formal software analysis tool has provided fully automatic support for proving properties that ensure that good events eventually happen (*liveness properties*). In this paper we present such a tool, which handles liveness properties of large systems written in C. Liveness properties are described in an extension of the specification language used in the SDV system. We have used the tool to automatically prove critical liveness properties of Windows device drivers and found several previously unknown liveness bugs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability, Languages

Keywords Formal Verification, Software Model Checking, Liveness, Termination

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. However, most of software verification tools are currently limited to *safety properties* [2, 3] (see Section 5 for discussion). No software analysis tool offers fully automatic scalable support for the remaining set of properties: *liveness properties*.

Consider Static Driver Verifier (SDV) [5, 26] 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 Windows 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`.

This is a safety property for the reason that any counterexample to the property will be a finite execution through the device driver code. We can think of safety properties as guaranteeing that specified bad events will not happen (*i.e.* calling `KeReleaseSpinlock` before calling `KeAcquireSpinlock`). Note that SDV cannot check the equally important related liveness property:

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

A counterexample to this property may not be finite—thus making it a liveness property. More precisely, a counterexample to the property is a program trace 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 `KeReleaseSpinlock` will eventually be called in the case that a call to `KeAcquireSpinlock` occurs).

Liveness properties are much harder to prove than safety properties. Consider, for example, a sequence of calls to functions: "`f(); g(); h();`". It is easy to prove that the function `f` is always called before `h`: in this case we need only to look at the structure of the control-flow graph. It is much harder to prove that `h` is eventually called after `f`: we first have to prove the termination of `g`. In fact, in many cases, we must prove several safety properties in order to prove a single liveness property. Unfortunately, to practitioners liveness is as important as safety. As one co-author learned while spending two 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-experts in formal verification (*i.e.* programmers that write software that needs to be verified) think equally in terms of both liveness and safety.

In this paper we describe a new algorithm which automatically constructs correctness proofs for liveness properties of software systems. The algorithm has been implemented as an extension to the TERMINATOR tool, which is a fully automatic termination prover for software [15]. Properties are described in a new specification language, which is an extension to the safety-property-only language used by SDV. When given a property description

POPL'07 January 17–19, 2007, Nice, France.

and a program, TERMINATOR attempts to construct a correctness proof. If a proof is found, then the property is guaranteed to hold. Conversely, if the proof fails, a potential counterexample is produced. If the counterexample is a non-terminating execution, then it is presented via a finite description, to enable the programmer to analyze it.

Our prototype tool represents the first known liveness prover to handle large systems written in C. The tool is interprocedural, path-sensitive, and context-sensitive. It supports infinite-state programs with arbitrary nesting of loops and recursive functions, pointer-aliasing and side-effects, function-pointers, etc. The tool's scalability leverages recent advances in program termination analysis (*e.g.* [7, 10, 11, 13, 14, 15]).

Following the automata-theoretic framework for program verification [31], our algorithm takes a liveness property and a program and constructs an equivalent *fair termination* problem (termination under a set of fairness constraints, a formal definition will be given later). The novel contributions of the paper include:

- An extension to SDV's language for specifying liveness properties (Section 2).
- A method for checking fair termination of programs (Section 3).
- Experimental results that demonstrate the viability of our approach to industrial software (Section 4).

2. Specifying liveness properties

In this section we describe a language for specifying liveness and safety properties of software systems. The language is an extension of SLIC [6], which is used to specify temporal safety properties in SDV [5]. SLIC 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 [5]). For this reason it was designed such that programmers do not need to modify or annotate source code—the code is typically not available when writing the specification.

Checking liveness can be reduced to checking fair termination [31]. Therefore, we first define a minimal language for specification of fair termination properties, which is essentially a language for defining Streett automata. In Section 3 we describe an algorithm for checking properties in this language. While the language can express all ω-regular properties [32], using it to specify liveness properties of real code is awkward since most of such properties have a response flavor. To overcome this problem, in Section 2.4 we introduce auxiliary statements (`set` and `unset`) that are helpful to concisely specify response requirements.

2.1 Syntax

The syntax of the specification language is defined in Figure 1. A specification describes an automaton that accepts program executions that satisfy the desired property. It consists of three basic parts:

A *state structure* declaration. The state structure defines a set of state variables that are maintained by the automaton representing the specification 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 automaton 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 procedure) 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. The body of a transfer function is written in a simple imperative C-like language. One important control construct is missing from the statements used in specifications: loops. This means that transfer functions always terminate.

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". An example of a fairness constraint is the following: "whenever function `foo` is called infinitely often then it returns a value distinct from 0 infinitely many times". We say that a non-terminating path satisfies a fairness constraint if and only if 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.

Informally, we think of a specification as a monitor that is executed along with the program. Safety properties are expressed using the `error` statement, which explicitly signals that an unsafe state has been reached (*i.e.* a safety property has been violated). A computation of the program does not satisfy the specification if either `error` is called during the computation, or the computation does not terminate and satisfies all the fairness constraints. Note that an empty specification specifies program termination.

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

The expression sub-language ($expr$) 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 `$`int identifiers are used to refer to the function's formal parameters, and the identifier `$return` is used to refer to the return value, which is accessible at the `exit` event.

2.2 Semantics

We treat programs as fair discrete systems [23], where fairness requirements are given in terms of sets of program states. A program $P = (\Sigma, \Theta, \mathcal{T}, \mathcal{C})$ consists of:

- Σ: a set of states;
- Θ: a set of initial states such that $\Theta \subseteq \Sigma$;
- $\mathcal{T}$: a finite set of transitions such that each transition $\tau \in \mathcal{T}$ is associated with a transition relation $\rho_\tau \subseteq \Sigma \times \Sigma$;
- $\mathcal{C} = \{\langle p_1, q_1\rangle, \ldots, \langle p_m, q_m\rangle\}$: a set of compassion requirements, such that $p_i, q_i \subseteq \Sigma$ for each $i \in \{1, \ldots, m\}$.

Transitions in this definition intuitively correspond to program statements. A computation σ is a maximal sequence of states $s_1, s_2, \ldots$ such that s_1 is an initial state, *i.e.* $s_1 \in \Theta$, and for each $i \geq 1$ there exists a transition $\tau \in \mathcal{T}$ such that s_i goes to s_{i+1} under ρ_τ, *i.e.* $(s_i, s_{i+1}) \in \rho_\tau$.

A computation $\sigma = s_1, s_2, \ldots$ satisfies the set of compassion requirements $\mathcal{C}$ when for each $\langle p, q\rangle \in \mathcal{C}$ either σ contains only finitely many positions i such that $s_i \in p$, or σ contains infinitely many positions j such that $s_j \in q$. For example, a computation sat-

Elements of syntax			Description
S	::=	$state$ $transFun^{+}$ $fairness^{+}$	A specification consists of a state structure, and a list of transfer function definitions together with fairness constraints
$state$	::=	`state` $\{$ $fieldDecl^{+}$ $\}$	A state structure is a list of field declarations
$fieldDecl$	::=	$ctype$ id = $expr$;	A field has a C type, an identifier and an initialization expression
$transFun$	::=	$pattern$ $stmt$	A transfer function consists of a pattern and a statement
$pattern$	::=	id . $event$ | `any`	
$event$	::=	`entry` | `exit`	
$stmt$	::=	id = $expr$;	Assignment statement
	|	$stmt$; $stmt$	
	|	`if` ($choose$) $stmt$ [`else` $stmt$]	
	|	`error();`	Safety property violation
	|	`return` [$expr$];	Return from the transfer function
	|	$\{$ $stmt$ $\}$	
$choose$	::=	`nondet()`	Non-deterministic choice
	|	$expr$	
$expr$	::=	id | $expr$ op $expr$ | $\cdots$	Pure expression sub-language of C
id	::=	$C\_identifier$	Refers to state elements or program variables
	|	\$$int$	\$$i$ refers to i^{th} formal parameter
	|	`$return`	Return value of a function
$fairness$	::=	`fairness` $\{$ $pattern$ $\{$ $expr$ $\}$ $pattern$ $\{$ $expr$ $\}$ $\}$	Fairness constraint

Figure 1. Syntax of the specification language.

isfies the compassion requirement (where `pc` denotes the program counter) $\langle$`pc` $=$ `foo.entry`, `pc` $=$ `foo.exit` $\wedge$ `$return` $\neq 0\rangle$ if whenever function `foo` is called infinitely often then it infinitely often returns a value distinct from 0. A computation is fair if it satisfies all compassion requirements of the program. We say that a program is fair terminating if it does not admit any infinite fair computation.

In order to prove that a specification holds of a given program, we *instrument* the specification into the program. That is, we traverse the input program and find all program locations in which the patterns described in the specification can be triggered. A new program—called the *instrumented program*—is produced in which the code found in the transfer functions is inserted to the original program. Formally, this constructs the product of the original program with the automaton representing the specification [31]. In our implementation we also perform a pointer analysis in order to constructs an over-approximation of all the potential pointer aliasing relationships. This allows us to statically find function-call events when functions are called by pointer.

Bodies of transfer function are inserted into the program as they are given in the specification. The variables mentioned in the `state` structure become global variables in the program. For each expression e_i, where $i = 1, 2$, in a fairness constraint from the specification and at each location where the corresponding pattern can be triggered we insert the code "`if (`e_i`) { L: skip; }`", where `L` is a fresh label.

The fairness constraints in the specification are translated to compassion requirements on the instrumented program (that can also be viewed as a fair discrete system). For each fairness constraint we introduce a compassion requirement $\langle$`pc` $=$ `L1` $\vee \ldots \vee$ `L`k, `pc` $=$ `M1` $\vee \ldots \vee$ `M`$l\rangle$ where `L1`$, \ldots,$ `L`k, respectively, `M1`$, \ldots,$ `M`l are the labels introduced during the instrumentation for the first, respectively, second expression of the fairness constraint.

To prove that a program does not violate the specification we need to prove that the instrumented program can never make a call to the `error` function and that it is fair terminating. The former can be done using existing techniques. An algorithm for checking the latter is presented in Section 3.

```
state {}

fairness {
    // First Boolean expression: succeeds
    // on every return from IoCreateDevice
    IoCreateDevice.exit { 1 }

    // Second Boolean expression: succeeds
    // if IoCreateDevice returns
    // something other than
    // STATUS_OBJ_NAME_COLLISION
    IoCreateDevice.exit {
        $return != STATUS_OBJ_NAME_COLLISION
    }
}
```

Figure 2. A specification of termination under a fairness constraint stating that if a program calls `IoCreateDevice` enough times then it will eventually return a value not equal to `STATUS_OBJ_NAME_COLLISION`.

2.3 A simple example

Consider the example specification in Figure 2. As it was noted before, the empty specification "`state {}`" specifies the termination of the program. Figure 2 specifies an instance of fair termination. Recall that fairness constraints in our specification language come as pairs of Boolean expressions guarded by patterns. In this case, the first Boolean expression always succeeds whenever the function `IoCreateDevice` returns. The second Boolean expression does not succeed when `IoCreateDevice` returns `STATUS_OBJ_NAME_COLLISION`. This particular constraint is saying:

> *Ignore non-terminating program executions in which* `IoCreateDevice` *from some point on always returns* `STATUS_OBJ_NAME_COLLISION`.

In other words, a program satisfies this specification if it terminates provided that whenever we continue to call `IoCreateDevice` repeatedly, it will eventually return a value other than `STATUS_OBJ_NAME_COLLISION`.

Functions used during the translation	Fairness constraint	Transfer functions
`void set() { if (q == NONE) { if (nondet()) { q = PENDING; } } }` `void unset() { if (q == PENDING) { q = MATCHED; } }`	`fairness { any { 1 } any { q == PENDING } }`	`main.entry { q = NONE; }` `main.exit { if (q == PENDING) { error(); } }`

Functions used during the translation:

```
void set() {
    if (q == NONE) {
        if (nondet()) {
            q = PENDING;
        }
    }
}

void unset() {
    if (q == PENDING) {
        q = MATCHED;
    }
}
```

Fairness constraint:

```
fairness {
    any { 1 }
    any { q == PENDING }
}
```

Transfer functions:

```
main.entry {
    q = NONE;
}

main.exit {
    if (q == PENDING) {
        error();
    }
}
```

Figure 3. Auxiliary constructs for specifying response-style liveness properties.

2.4 Auxiliary constructs

Most of the frequently specified liveness properties have a response flavor and are awkward to specify with the minimalistic language described in Section 2.1. To make their specification easier we introduce auxiliary constructs—the functions `set` and `unset`, which can be used in the specification's transfer functions. Their intended meaning is that when the property calls `set` then an execution through the property in which `unset` is never called represents a liveness violation. More precisely, a program satisfies a specification with `set` and `unset` if and only if `error` is never called in the instrumented program and there is no computation of the instrumented program that (i) satisfies all compassion requirements, (ii) contains a call to `set`, and (iii) contains no calls to `unset` after the last call to `set`. This corresponds to the validity of the LTL formula

$$\mathbf{G}\left((\mathrm{pc} = \mathtt{set}.\mathrm{entry}) \Rightarrow \mathbf{F}\left(\mathrm{pc} = \mathtt{unset}.\mathrm{entry}\right)\right)$$

under the compassion requirements.

A specification containing `set` and `unset` can be translated to a specification in the language of Section 2.1 using an application of the automata-theoretic framework for program verification [31]. Namely, we translate the negation of the LTL formula above into a Büchi automaton and construct the synchronous product of the program and the automaton. Toward this end we introduce an extra variable `q` into our program denoting the state of the automaton and assume defined three constants representing the states of the automaton—`NONE`, `PENDING`, and `MATCHED`. Initially $\mathtt{q} = \mathtt{NONE}$ and the state `PENDING` is accepting. We then define `set` and `unset` as is shown in Figure 3 and add the fairness condition and transfer functions from Figure 3 to the specification. The fairness constraint excludes infinite computations satisfying conditions (ii) and (iii) above. To exclude finite computations satisfying (ii) and (iii) we call `error` when the program terminates in the case when there is a pending `set` call.

The original program satisfies the specification if and only if it satisfies the transformed specification, *i.e.* if the instrumented program is safe and fair terminating.

2.5 Using response requirements and state

We provide an example that shows how specifications can use `set`/`unset` and 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, p)` raises the IRQL to the value of `x` and writes the old IRQL value to the location in memory pointed to by `p`; `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. Note that it is a fatal (safety) error to call `KeLowerIrql` using `y` that was not returned by the immediately preceding call to `KeRaiseIrql`.

```
state { int irql = -1; }

KeRaiseIrql.entry {
    if (irql == -1) {
        irql = KeGetCurrentIrql();
        set();
    }
}

KeLowerIrql.entry {
    if ($1 == irql && irql > -1) {
        unset();
    }
    irql = -1;
}
```

Figure 4. A liveness property involving the Windows kernel APIs `KeRaiseIrql` and `KeLowerIrql`. The macro `KeGetCurrentIrql` refers to a variable in the OS environment model.

Figure 4 shows how this specification is modeled in our language. This example demonstrates the usage of the 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.

If the code the specification of which we are writing uses the function `IoCreateDevice`, we might add the `fairness` clause from Figure 2 to the specification in Figure 4 to restrict the behavior of this function.

2.6 Combining liveness and safety

Specifications can contain both liveness and safety properties. In the case of Figure 5, we have a safety property mixed together with the liveness property from Figure 4. TERMINATOR will search for at least one violation of the properties, either of safety or liveness.

```
state { int irql = -1; }

KeRaiseIrql.entry {
    if ($1 <= KeGetCurrentIrql()) {
        error();
    }
    if (irql == -1) {
        irql = KeGetCurrentIrql();
        set();
    }
}

KeLowerIrql.entry {
    if ($1 >= KeGetCurrentIrql()) {
        error();
    }
    if ($1 == irql && irql > -1) {
        unset();
    }
    irql = -1;
}
```

Figure 5. A specification defining both liveness and safety properties involving the Windows kernel APIs `KeRaiseIrql` and `KeLowerIrql`.

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.

2.7 Discussion

Temporal properties can be specified using temporal logics such as LTL [27]. However, such properties can also be specified using automata on infinite words [32] (in fact, LTL is less expressive than such automata); see also [24]. To extend the expressive power of LTL to that of automata on infinite words, industrial languages, such as ForSpec [4] and PSL [1], add to LTL 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. Specifications in SLIC, for example, can be viewed as automata on finite words.

In this paper we are taking an automata-based approach to specifying temporal properties—properties in our language can be viewed as automata on infinite words. We note that compilation techniques described in [32] can be used to extend TERMINATOR to logic-based specifications.

`Set` and `unset` make it easier to specify properties being instances of response specification pattern [18]. The same approach as was taken here can be used to add other specification patterns to our language.

3. Verifying fair termination

In this section we describe a novel algorithm for checking fair termination of programs. The approach we take is to use counterexample-guided refinement for building fair termination arguments (*i.e.*, relations justifying that the program is fair terminating). The algorithm is an extension of the TERMINATOR algorithm [14]. It adds support for fair termination using the proof rule proposed in [28], which separates reasoning about fairness and well-foundedness by using transition invariants [30].

Assume that the specification to be checked has already been instrumented into the program and the problem of checking the conformance of the program to the liveness specification has been reduced to a fair termination problem as described in Section 2.2. We fix a program $P = (\Sigma, \Theta, \mathcal{T}, \mathcal{C})$ represented by a fair discrete system with a (finite) set of compassion requirements $\mathcal{C}$. We want to check whether the program P terminates under the compassion requirements $\mathcal{C}$.

3.1 Counterexample-guided refinement for fair termination

First of all, we introduce some auxiliary definitions. A binary relation R is well-founded if it does not admit any infinite chains. We say that a relation T is disjunctively well-founded [30] if it is a finite union $T = T_1 \cup \ldots \cup T_n$ of well-founded relations.

We remind the reader that a computation σ is a maximal sequence of states $s_1, s_2, \ldots$ such that s_1 is an initial state, and for each $i \geq 1$ there exists a transition $\tau \in \mathcal{T}$ such that s_i goes to s_{i+1} under ρ_τ. A finite segment $s_i, s_{i+1}, \ldots, s_j$ of a computation where $i < j$ is called a computation segment. Note that all the states constituting a computation segment must be reachable from initial states. Following [28], we define two auxiliary functions that map a set of states S to a set of compassion requirements:

$$\mathsf{None}_{\mathcal{C}}(S) = \{\langle p, q\rangle \in \mathcal{C} \mid S \cap p = \emptyset\},$$
$$\mathsf{Some}_{\mathcal{C}}(S) = \{\langle p, q\rangle \in \mathcal{C} \mid S \cap q \neq \emptyset\}.$$

Let S be the set of states that appear in a computation segment σ. Then, $\mathsf{None}_{\mathcal{C}}(S)$ and $\mathsf{Some}_{\mathcal{C}}(S)$ record the compassion requirements from $\mathcal{C}$ that are fulfilled on the infinite computation π obtained by repeating σ infinitely many times and prefixing the result with a computation segment from an initial state of the program to the starting state of σ (we know that such a computation segment exists since all the states in σ are the reachable states of the program P). $\mathsf{None}_{\mathcal{C}}(\sigma)$ keeps track of the compassion requirements $\langle p, q\rangle$ that are fulfilled because π contains only finitely many states from p. $\mathsf{Some}_{\mathcal{C}}(\sigma)$ keeps track of the compassion requirements $\langle p, q\rangle$ that are fulfilled because π contains infinitely many states from q.

Finally, let

$$\mathcal{R}_{\mathcal{C}} = \{\langle s_1, s_{n+1}\rangle \mid \exists \text{computation segment } \sigma = s_1, \ldots, s_{n+1}. \\ \mathsf{None}_{\mathcal{C}}(\sigma) \cup \mathsf{Some}_{\mathcal{C}}(\sigma) = \mathcal{C}\}.$$

We call $\mathcal{R}_{\mathcal{C}}$ the *fair binary reachability relation*. $\mathcal{R}_{\mathcal{C}}$ consists of all the pairs of starting and ending states of (finite) computation segments of the program P such that, if repeated as above, will give (infinite) computations satisfying all the compassion requirements from $\mathcal{C}$. We remind the reader that all the states in a computation segment must be reachable from the initial states so the states in $\mathcal{R}_{\mathcal{C}}$ are reachable from the initial states too.

The following adaptation of Theorem 3 from [28] forms the basis for our algorithm.

THEOREM 1. *The program P terminates under the compassion requirements $\mathcal{C}$ if and only if there exists a disjunctively well-founded relation T such that $\mathcal{R}_{\mathcal{C}} \subseteq T$.*

Theorem 1 says that to prove a program P fair terminating we have to cover its fair binary reachability relation by a finite union of well-founded relations. We build such a relation T by iterative refinement extending T each time a spurious counterexample is discovered.

Instead of considering one computation segment at a time we cover the set of computation segments resulting from the execution of a sequence of program statements as a whole. This is formalized in the following notions of path and fair path.

We define a path π to be a finite sequence of program transitions. Given a path $\pi = \tau_1, \ldots, \tau_n$, we say that π is fair with respect to a compassion requirement $\langle p, q\rangle$ if some computation segment $\sigma = s_1, \ldots, s_{n+1}$ obtained by executing the statements

input
  Program P
  Compassion requirements $\mathcal{C}$
begin
  $T := \emptyset$
  repeat
    if $\underline{\text{exists path } \pi \text{ such that } \mathsf{fair}(\pi) \text{ and } \rho_\pi \not\subseteq T}$ **then**
      if ρ_π is well-founded by W **then**
        $T := T \cup W$
      else
        return "Counterexample path π"
    else
      return "Fair termination argument T"
end.

Figure 6. Incremental construction of a fair termination argument. The compassion requirements on the program specify when the predicate $\mathsf{fair}(\pi)$ holds for a path π. The existence of a fair termination argument implies the validity of the given liveness property under fairness constraints. The evaluation of the underlined **if**-expression is explained in Section 3.2. The relation W can be computed using a ranking function synthesis engine, *e.g.*, [29].

of π either does not visit any p-states or traverses some q-state. A path is fair, written as $\mathsf{fair}(\pi)$, if it is fair with respect to every compassion requirement in $\mathcal{C}$.

The algorithm for the construction of a fair termination argument is presented in Figure 6. The algorithm first performs a fair binary reachability analysis to check whether the inclusion $\mathcal{R}_\mathcal{C} \subseteq T$ holds. Fair binary reachability analysis is described in Section 3.2. ρ_π is the path relation, which is also defined in Section 3.2.

If the subset inclusion holds then, by Theorem 1, P terminates under the compassion requirements $\mathcal{C}$ and we report fair termination. In the case that the inclusion does not hold, a fair path π is produced such that $\rho_\pi \not\subseteq T$. The algorithm then checks if there exists a well-founded relation W (called a ranking relation) covering ρ_π. If such a relation does not exist, π represents a potential fair termination bug and the algorithm terminates. Otherwise, the ranking relation W is added to the set T. This ensures that the same path will not be discovered at the subsequent iterations of the algorithm. The ranking relation W can be generated using any tool for ranking function synthesis [10, 11, 17, 29]. In practice W produced by these tools is usually sufficient for ruling out not only π, but also all the paths that are obtained by repeating π.

3.2 Binary reachability for fair termination

The problem of checking fair binary reachability consists of checking the inclusion $\mathcal{R}_\mathcal{C} \subseteq T$ for a program P with a set of compassion requirements $\mathcal{C} = \{\langle p_1, q_1\rangle, \dots, \langle p_m, q_m\rangle\}$ and a relation T. The solution we propose here is based on an extension of the procedure for solving binary (as opposed to ordinary 'unary') reachability. Our extension takes into account the compassion requirements $\mathcal{C}$.

The key idea of the approach is to leverage the techniques from symbolic software model checking for safety properties (*e.g.* [12, 20, 21]). Note that techniques are available for reducing checking of fair termination to safety checking [31]. We use here another reduction which is more amenable to automated abstraction techniques. Fair binary reachability analysis is performed by transforming the program P to a program $\hat{P}^T$, the set of reachable states of which represents the fair binary reachability relation of the original program. The inclusion $\mathcal{R}_\mathcal{C} \subseteq T$ holds if and only if the transformed program satisfies a certain safety property. If the safety property is violated, then the inclusion does not hold and the counterexample provided by the safety checker can be used to construct the path π in the underlined expression in Figure 6.

We transform the program P to the program $\hat{P}^T$ in the following way. Let $V = \{\mathtt{v}_1, \dots, \mathtt{v}_n, \mathtt{pc}\}$ be the set of all program variables in P including the program counter $\mathtt{pc}$. The set of variables of the program $\hat{P}^T$ contains V and the corresponding pre-versions $\mathtt{'v}_1, \dots, \mathtt{'v}_n, \mathtt{'pc}$. Besides, we introduce two Boolean arrays $\mathtt{in\_p}$ and $\mathtt{in\_q}$ indexed by $1, \dots, m$. Therefore, a state of $\hat{P}^T$ can be represented by a tuple $\langle s, r, \mathtt{in\_p}, \mathtt{in\_q}\rangle$, where $s, r \in S$. The state $\langle s, r, \mathtt{in\_p}, \mathtt{in\_q}\rangle$ represents a computation segment starting with s and ending with r. The variable $\mathtt{in\_p}_j$ (respectively, $\mathtt{in\_q}_j$) is true if and only if there is a state in the segment satisfying p_j (respectively, q_j). We assume that initially $\mathtt{'v} = \mathtt{v}$ and all elements of $\mathtt{in\_p}$ and $\mathtt{in\_q}$ are false.

The transformation is shown in Figure 7. To simplify the presentation, consider first the program $\hat{P}$ obtained using the transformation without the assignment to `fair` and the `assert` statement. At each state of the program $\hat{P}$ we update the elements of $\mathtt{in\_p}$ and $\mathtt{in\_q}$ and we can also non-deterministically choose to start recording a new computation segment. In this case we copy all the program variables to the corresponding pre-variables and clear the contents of the arrays $\mathtt{in\_p}$ and $\mathtt{in\_q}$.

Let $\hat{\Theta}$ be the set of initial states of the program $\hat{P}$ defined as above, $\hat{\Sigma}$ the set of states of $\hat{P}$ and $\text{post}^+_{\hat{P}}(\hat{\Theta})$ the set of states of $\hat{P}$ reachable after at least one step. The following theorem formally defines the meaning of the transformed program described above.

THEOREM 2. *Suppose that in the program P there are no transitions to the initial locations. Then*

$$\begin{aligned}\text{post}^+_{\hat{P}}(\hat{\Theta}) = \{ & \langle s_1, s_{n+1}, \mathtt{in\_p}, \mathtt{in\_q}\rangle \mid \\ & \exists \textit{computation segment } \sigma = s_1, \dots, s_{n+1}. \\ & \forall j \in \{1, \dots, m\}. \\ & (\mathtt{in\_p}_j = \mathsf{false} \Leftrightarrow \langle p_j, q_j\rangle \in \mathsf{None}(\{s_1, \dots, s_n\})) \wedge \\ & (\mathtt{in\_q}_j = \mathsf{true} \Leftrightarrow \langle p_j, q_j\rangle \in \mathsf{Some}(\{s_1, \dots, s_n\}))\}.\end{aligned}$$

Proof sketch. "$\subseteq$". For each state from the set on the left-hand side of the equality there exists a sequence of program transitions of $\hat{P}$ leading to it from an initial state. We prove the inclusion by induction on the length of this sequence.

"$\supseteq$". For each computation segment from the set on the right-hand side of the equality there exists a sequence of program transitions of $\hat{P}$ leading from an initial state to the first state in the computation segment. We first prove the inclusion in the case when this sequence is empty. We then prove the general case by induction on the length of the computation segment. $\square$

The technical restriction on the initial locations is due to the fact that we do not transform the initial statement of the program, and is inherited from the binary reachability analysis of [14]. Note that in Theorem 2 and in the rest of the paper we consider the sequence of operations resulting from the transformation of a statement of the original program as one statement (transition) of the transformed program. The states in computation segments are reachable from the initial states of the original program and, therefore, the sets of states s and r in the theorem depend on $\hat{\Theta}$.

To check whether the inclusion $\mathcal{R}_\mathcal{C} \subseteq T$ holds we have to stop the reachability computation as soon as the current computation segment is fair and T is violated on it. The `assert` statement in the program transformation ensures this. Consider now the full transformation shown in Figure 7 and the corresponding program $\hat{P}^T$. The set of states of the program $\hat{P}^T$ resulting from the transformation is $\hat{\Sigma} \cup \{\mathtt{ERROR}\}$ (where `ERROR` is the state to which the program goes when an `assert` is violated) and the transition relation is a

input

P: program over variables $\mathtt{v}_1, \ldots, \mathtt{v}_n$, program counter `pc`, and initial location $\mathtt{L}_0$

$\{\langle p_1, q_1\rangle, \ldots, \langle p_m, q_m\rangle\}$: set of compassion requirements

T: candidate fair termination argument given by an assertion over the program variables and their pre-versions $`\mathtt{v}_1, \ldots, `\mathtt{v}_n$, `` `pc ``

begin

1. Add pre-variables to P: $`\mathtt{v}_1, \ldots, `\mathtt{v}_n$, `` `pc ``,
2. Add auxiliary variables to P: `fair`, $\mathtt{in\_p}_1, \ldots, \mathtt{in\_p}_m$, $\mathtt{in\_q}_1, \ldots, \mathtt{in\_q}_m$
3. Replace each statement (except for the one at the initial location $\mathtt{L}_0$)

```
L: stmt;
```

with

```
L: fair = ((!p1  && !in_p1)  || q1  || in_q1) &&
                 ...
          ((!pm  && !in_pm) || qm  || in_qm);
   assert(!fair || T);
   if (nondet()) {
       `vi = vi;           /* for each i ∈ {1,...,n} */
       `pc = L;
       in_pi = 0;          /* for each i ∈ {1,...,m} */
       in_qi = 0;          /* for each i ∈ {1,...,m} */
   }
   if (pi) in_pi = 1;      /* for each i ∈ {1,...,m} */
   if (qi) in_qi = 1;      /* for each i ∈ {1,...,m} */
   stmt;
```

4. Add initialization statements: `` `pc = L0; `v1 = v1; ... `vn = vn; ``

end.

Figure 7. Program transformation for checking fair binary reachability using a temporal safety checker. `nondet()` represents nondeterministic choice.

subset of the transition relation of $\hat{P}$. We denote with $\mathrm{post}^+_{\hat{P}^T}(\hat{\Theta})$ the set of states of $\hat{P}$ reachable after at least one step.

THEOREM 3. *Suppose that in the program P there are no transitions to the initial locations. Then the inclusion $\mathcal{R}_\mathcal{C} \subseteq T$ holds if and only if the state* `ERROR` *is not reachable in the program $\hat{P}^T$.*

Proof. *"If"*. Suppose the contrary: `ERROR` is unreachable in $\hat{P}^T$ and $\mathcal{R}_\mathcal{C} \not\subseteq T$. Then there exists a computation segment $\sigma = s_1, \ldots, s_n, s_{n+1}$ such that $\mathsf{None}_\mathcal{C}(\sigma) \cup \mathsf{Some}_\mathcal{C}(\sigma) = \mathcal{C}$ and $\langle s_1, s_{n+1}\rangle \notin T$. For each $j \in \{1, \ldots, m\}$ we define

$$\mathtt{in\_p}^0_j = \begin{cases} \mathsf{false}, & \text{if } \langle p_j, q_j\rangle \in \mathsf{None}_\mathcal{C}(\{s_1, \ldots, s_n\}); \\ \mathsf{true}, & \text{otherwise} \end{cases}$$

and

$$\mathtt{in\_q}^0_j = \begin{cases} \mathsf{true}, & \text{if } \langle p_j, q_j\rangle \in \mathsf{Some}_\mathcal{C}(\{s_1, \ldots, s_n\}); \\ \mathsf{false}, & \text{otherwise.} \end{cases}$$

By Theorem 2 we have that $\langle s_1, s_{n+1}, \mathtt{in\_p}^0, \mathtt{in\_q}^0\rangle \in \mathrm{post}^+_{\hat{P}}(\hat{\Theta})$. Since `ERROR` is unreachable in $\hat{P}^T$, it is also the case that $\langle s_1, s_{n+1}, \mathtt{in\_p}^0, \mathtt{in\_q}^0\rangle \in \mathrm{post}^+_{\hat{P}^T}(\hat{\Theta})$. Consider the execution of $\hat{P}^T$ starting from this state. As the program $\hat{P}^T$ has no transitions to the initial location, the program counter in the state s_{n+1} is different from the initial location and so the next statements to execute will be the ones in the auxiliary code shown in Figure 7. Taking into account the definition of $\mathtt{in\_p}^0$ and $\mathtt{in\_q}^0$ above and the fact that $\mathsf{None}_\mathcal{C}(\sigma) \cup \mathsf{Some}_\mathcal{C}(\sigma) = \mathcal{C}$ one can see that `fair` will evaluate to true. But then since $\langle s_1, s_{n+1}\rangle \notin T$ the `assert` will fail and, hence, $\mathtt{ERROR} \in \mathrm{post}^+_{\hat{P}^T}(\hat{\Theta})$, which contradicts our initial assumption.

"Only if". Again, suppose the contrary: $\mathcal{R}_\mathcal{C} \subseteq T$ and `ERROR` is reachable in $\hat{P}^T$. Since we do not apply the transformation in Figure 7 to the initial location, it follows that there exists a state $\langle s_1, s_{n+1}, \mathtt{in\_p}^0, \mathtt{in\_q}^0\rangle \in \mathrm{post}^+_{\hat{P}^T}(\hat{\Theta})$ such that $\langle s_1, s_{n+1}\rangle \notin T$ and on this state `fair` evaluates to `true`. We have that

$$\mathrm{post}^+_{\hat{P}^T}(\hat{\Theta}) \subseteq \mathrm{post}^+_{\hat{P}}(\hat{\Theta}) \cup \{\mathtt{ERROR}\},$$

thus, $\langle s_1, s_{n+1}, \mathtt{in\_p}^0, \mathtt{in\_q}^0\rangle \in \mathrm{post}^+_{\hat{P}}(\hat{\Theta})$.

Then according to Theorem 2 there exists a computation segment $\sigma = s_1, \ldots, s_n, s_{n+1}$ such that for all $j = \{1, \ldots, m\}$

$$\mathtt{in\_p}^0_j = \mathsf{false} \Leftrightarrow \langle p_j, q_j\rangle \in \mathsf{None}(\{s_1, \ldots, s_n\})$$

and

$$\mathtt{in\_q}^0_j = \mathsf{true} \Leftrightarrow \langle p_j, q_j\rangle \in \mathsf{Some}(\{s_1, \ldots, s_n\}).$$

Since `fair` evaluates to `true` on $\langle s_1, s_{n+1}, \mathtt{in\_p}^0, \mathtt{in\_q}^0\rangle$, we have that $\mathsf{None}_\mathcal{C}(\sigma) \cup \mathsf{Some}_\mathcal{C}(\sigma) = \mathcal{C}$ and, hence, $\langle s_1, s_{n+1}\rangle \in \mathcal{R}_\mathcal{C}$. But since $\mathcal{R}_\mathcal{C} \subseteq T$ it follows that $\langle s_1, s_{n+1}\rangle \in T$, which contradicts a previously established fact. $\square$

It follows from Theorem 3 that to check fair binary reachability one can apply a temporal safety checker on the program $\hat{P}^T$ to prove the non-reachability of the location `ERROR` or generate a corresponding counterexample. In the latter case the counterexample returned by the safety checker is a lasso path, *i.e.* a sequence of program statements of the form $\tau_1, \ldots, \tau_n, \ldots, \tau_p, \tau_n$. The path

```
state {}

PPBlockInits.entry {
    set();
}

PPUnblockInits.entry {
    unset();
}
```

Figure 9. An example liveness property for the program fragment in Figure 8.

$\tau_n, \ldots, \tau_p, \tau_n$ becomes then the path π in the algorithm in Figure 6. The path relation corresponding to this path is defined as follows:

$$\rho_\pi = \{\langle s_2, s_3\rangle \mid \exists s_1 \in \Theta. \\ \langle s_1, s_2\rangle \in \rho_{\tau_1} \circ \cdots \circ \rho_{\tau_{n-1}} \wedge \langle s_2, s_3\rangle \in \rho_{\tau_n} \circ \cdots \circ \rho_{\tau_p}\}.$$

Optimizations. The transformation of P into $\hat{P}^T$ was presented above somewhat idealistically. In practice, it is sufficient to instrument the code shown in Figure 7 only on cutpoints [19]; see [14] for details. Additionally, program slicing techniques can be used to eliminate redundant assignments to variables added during the transformation and sometimes the variables themselves.

Example. Consider the code fragment from Figure 8. Imagine that we are trying to prove that whenever `PPBlockInits` is called, `PPUnblockInits` will eventually be called (Figure 9) with the fairness constraint from Figure 2.

Our implementation constructs a disjunctively well-founded relation T for each cutpoint in the program's control-flow graph. Suppose that we are considering the cutpoint at location 3. While performing fair binary reachability analysis, our extension to TERMINATOR would produce the code in Figure 10. We assume the following conditions:

- We have already translated `set` and `unset` away, instrumented the fairness constraints, and constructed the analogous fair termination problem. The compassion requirements on the resulting program are $\langle \mathtt{pc} = 6.1, \mathtt{pc} = 6.3\rangle$ (corresponding to the fairness constraint from Figure 2) and $\langle \mathtt{true}, \mathtt{q} = \mathtt{PENDING}\rangle$ (corresponding to the condition on `set` and `unset`).
- The variables `in_p1` and `in_q1` are used to represent the compassion requirement corresponding to the fairness constraint in Figure 2. The variables `in_p2` and `in_q2` correspond to the conditions on `set` and `unset`, and hence the property in Figure 9 (the variable `in_p2` can be eliminated as explained below).
- TERMINATOR has already has constructed a candidate fair termination argument for program location 3:

$$T(s,t) \triangleq t(\mathtt{i}) > s(\mathtt{i}) \wedge t(\mathtt{i}) < t(\mathtt{Pdolen}) \\ \wedge s(\mathtt{Pdolen}) = t(\mathtt{Pdolen}).$$

The differences between Figure 8 and Figure 10 are as follows:

- Lines INIT.1–INIT.5 initialize the state of the automaton `q`, pre-variables, and variables for keeping track of compassion requirements.
- Lines 0.1–0.5 and 19.1–19.3 come from the property's transfer functions and in this case are just inlining of the code for `set` and `unset` from Figure 3.
- Lines 6.1–6.4 update the auxiliary variables associated with the compassion requirement corresponding to the fairness constraint in Figure 2.
- Lines 2.1–2.3 update the auxiliary variables associated with the compassion requirement obtained from the condition on the `set` and `unset`. In principle the updates should appear at each line in the new program. However, using live variables analysis, we remove many of them—`in_q2` only needs to be evaluated before it is used. We have also removed `in_p2` since after the simplification of the Boolean expression in line 2.4 (see below) its value is not used in the program.
- Line 2.6 executes a non-deterministic decision as to whether or not to take a snapshot of the current state. Since this program is then passed to a temporal safety checker, this means that, given any valuations returned by `nondet` during numerous executions through this loop, if a bad set of valuations exists, the model checker will find it—this gives us full coverage of the property.
- Lines 2.7–2.10 copy the current state into the auxiliary variables and clear the contents of the variables for keeping track of compassion requirements. This has the effect of starting the recording of a new computation segment. As an optimization we copy only variables that are used in the candidate fair termination argument.
- Lines 2.4–2.5 check the termination condition in the case when the compassion requirements are not being violated. We simplified the Boolean expression in line 2.4 using the fact that one of the Boolean expressions in the compassion requirement for `set` and `unset` is just `true`. After this simplification the variable `in_p2` was not used in line 2.4 anymore, which allowed us to eliminate it.
- Lines EXIT.1–EXIT.3 check the absence of terminating computations violating the condition on `set` and `unset`.

TERMINATOR will perform an infinite-state reachability check on the code in Figure 8 to check that the `assert` cannot fail and `error()` cannot be called. If TERMINATOR can prove that this cannot be the case, then the liveness condition is not violated at this cutpoint and the algorithm proceeds to attempt to prove that the fair termination property is not violated at the next cutpoint.

3.3 Lazy treatment of fairness constraints

The number of fairness constraints that appear in properties of programs with complex interaction with the environment can be large. In many cases some of these constraints may not be required to prove the property. We observe that our abstraction-based algorithm naturally exploits this fact, due to the following reasons. First, our encoding of fairness using Boolean variables that keep track of the fulfilment of the constraints does not introduce a significant increase in the program size. Second, by applying a counterexample-guided abstraction refinement procedure based on predicate abstraction to validate fair termination arguments we only consider those fairness constraints that are relevant to the property. This is ensured by predicate abstraction together with a refinement procedure, which only tracks values of those variables that appear in the predicates that define the abstraction.

4. Experimental results

In this section we describe the results from experiments with our implementation of the proposed algorithm on Windows device drivers. In order to perform the experiments we have implemented the algorithm as an extension to the TERMINATOR termination prover [15], which uses SDV [5] as its underlying safety checker. Tables 1 through 4 contain the statistics from these experiments. We used three liveness properties involving the acquiring and releasing of resources together with the fair termination property in Figure 2. The fairness constraint from Figure 2 was also used in the former three experiments (Tables 1 through 3). Note that

```
PPBlockInits();
while (i < Pdolen) {
    DName = PPMakeDeviceName(lptName[i], PdoType, dcId[i], num);
    if (!DName) { break; }
    RtlInitUnicodeString(&deviceName, DName);
    status = IoCreateDevice(fdx->do, PDOSZ, &deviceName, 0, 0, TRUE, Pdo[i]);
    if (STATUS_SUCCESS != status) {
        Pdo[i] = NULL;
        if (STATUS_OBJECT_NAME_COLLISION == status) {
            ExFreePool(DName);
            num++;
            continue;
        }
        break;
    } else {
        i++;
    }
}
num = 0;
PPUnblockInits();
```

Figure 8. Example code from a Windows device driver dispatch routine. The correct behavior of the code depends on the fairness constraint from Figure 2.

SDV's model of the driver's environment 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 the termination of every dispatch routine within the device driver. We used a timeout threshold of 10,000 seconds and a memory limit of one 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 TERMINATOR has found a proof that the driver does not violate the specification. The validity of the liveness properties that we checked on the device drivers did not depend on significant tracking of heap manipulations or bit-level operations, which caused false bugs in experiments with TERMINATOR [14]. This is why we have not obtained any false bugs in our experiments. We note that techniques from [8] can be used to perform termination analysis in cases where accurate tracking of the heap is required for proving fair termination.

The experimental results demonstrate that we have finally obtained a method for checking liveness properties of real systems code. We believe that the experience that we have had with Windows device drivers will match the results that users will have in other similar domains.

5. Related work

Our proposed algorithm builds on a large body of formal foundations, ranging from the formalization of the semantics of programs by fair discrete systems [25] and the automata-theoretic approach to temporal verification [31] to the more recent construction of fixpoint domains for abstract interpretation with fairness [28]. We also use recent advances in the area of automatic termination analysis (e.g. [11, 14]). From these foundations we have developed (to the best of our knowledge) the first known fully automatic verification tool for liveness properties of infinite-state programs.

The key difference between TERMINATOR and finite-state model checkers that support liveness checking, e.g. SPIN [22], Bandera [16], and Java PathFinder [33], is that TERMINATOR employs completely automatic abstraction, while the others either explore the state space as-is (SPIN) or use user-provided and, hence, not automatic abstractions (Bandera). These tools will terminate with "Out-Of-Memory" for programs with infinite or very large state spaces. Automatic abstraction provides effectiveness and efficiency to overcome this limitation.

The idea of using program transformations to convert liveness into safety is known in finite-state model checking [9, 31]. Here we adapt these ideas to the context of infinite-state systems.

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. 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 the developers of SDV do today: they construct a number of `main.exit` transfer functions in SLIC that check that the liveness property is not violated when the driver terminates. In this case SDV will miss any violations to liveness properties that involve non-terminating executions.

6. Conclusion

Since automatic safety property checking has only recently become a reality, automatic liveness proving for real code has been considered impossible. TERMINATOR is the first known tool to break through this liveness checking barrier. We have applied TERMINATOR to device drivers ranging in sizes from 1,000 to 20,000 LOC.

The proposed algorithm takes advantage of recent advances in termination analysis by converting the problem of liveness checking into fair termination checking. The scalability and support for real programming language features comes from the termination analysis. This paper has also presented a language in which liveness properties can be expressed.

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.

Limitations. A few notes about limitations:

- As program termination is an undecidable problem, TERMINATOR's analysis is not guaranteed to terminate.

Initialization

```
INIT.1  q = NONE;
INIT.2  pre_pc = 1;
INIT.3  pre_i = i;
INIT.4  pre_Pdolen = Pdolen;
INIT.5  in_p1 = in_q1 = in_q2 = 0;
```

Body

```
        ...
0.1     if (q == NONE) {    /* set() */
0.2         if (nondet()) {
0.3             q = PENDING;
0.4         }
0.5     }
1       PPBlockInits();
2       while (i < Pdolen) {
2.1         if (q == PENDING) {
2.2             in_q2 = 1;
2.3         }
2.4         fair = (!in_p1 || in_q1) && ((q == PENDING) || in_q2);
2.5         assert(!fair || !(pre_pc == 3) || (i > pre_i && i < Pdolen && pre_Pdolen == Pdolen));
2.6         if (nondet()) {
2.7             pre_pc = 3;
2.8             pre_i = i;
2.9             pre_Pdolen = Pdolen;
2.10            in_p1 = in_q1 = in_q2 = 0;
2.11        }
3           DName = PPMakeDeviceName(lptName[i], PdoType, dcId[i], num);
4           if (!DName) { break; }
5           RtlInitUnicodeString(&deviceName, DName);
6           status = IoCreateDevice(fdx->do, PDOSZ, &deviceName, 0, 0, TRUE, Pdo[i]);
6.1         in_p1 = 1;
6.2         if (status != STATUS_OBJECT_NAME_COLLISION) {
6.3             in_q1 = 1;
6.4         }
7           if (STATUS_SUCCESS != status) {
8               Pdo[i] = NULL;
9               if (STATUS_OBJECT_NAME_COLLISION == status) {
10                  ExFreePool(DName);
11                  num++;
12                  continue;
13              }
14              break;
15          } else {
16              i++;
17          }
18      }
19      num = 0;
19.1    if (q == PENDING) {    /* unset() */
19.2        q = MATCHED;
19.3    }
20      PPUnblockInits();
        ...
```

Exit points

```
EXIT.1  if (q == PENDING) {
EXIT.2      error();
EXIT.3  }
```

Figure 10. Code produced while performing fair binary reachability analysis on the code from Figure 8. `nondet()` represents nondeterministic choice.

- Counterexamples are not guaranteed to be real counterexamples. Our proposed algorithm attempts to prove that the property *holds*, not that it *doesn't hold*.
- The validity of proofs constructed in TERMINATOR relies on the soundness of the underlying safety checker. For example, TERMINATOR may return a "proof" of correctness when the code is not correct due to the fact that TERMINATOR's symbolic safety checker 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.

Driver	Time (seconds)	LOC	Bugs found
1	15	1K	1
2	314	7K	0
3	2344	15K	0
4	3122	20K	1
1R	16	1K	0
4R	3217	20K	0

Table 1. Checking entering and leaving critical regions. The property proved is the property in Figure 9 with `KeEnterCriticalRegion` and `KeLeaveCriticalRegion` substituted for `PPBlockInits` and `PPUnblockInits` respectively. The fairness constraint used is the one from Figure 2. The bug in driver 1 was known. The bug in driver 4 was not known before. Drivers 1R and 4R are repaired versions of driver 1 and 4 respectively.

Driver	Time (seconds)	LOC	Bugs found
1	23	1K	0
2	188	7K	0
3	271	15K	0
4	T/O	20K	T/O

Table 2. Checking acquiring and releasing of spin locks. The property being checked is the property in Figure 9 with `KeAcquireSpinLock` and `KeReleaseSpinLock` substituted for `PPBlockInits` and `PPUnblockInits` respectively. The fairness constraint used is displayed in Figure 2.

Driver	Time (seconds)	LOC	Bugs found
1	62	1K	5
2	N/A	7K	N/A
3	N/A	15K	N/A
4	T/O	20K	T/O
1R	35	1K	0

Table 3. Checking modifications of IRQLs. The property being checked is the one in displayed in Figure 4 together with the fairness constraint from Figure 2. The bugs in driver 1 were 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 property is trivially true.

Driver	Time (seconds)	LOC	Bugs found
1	9	1K	0
2	129	7K	0
3	1463	15K	1
4	T/O	20K	T/O

Table 4. Checking the termination of driver dispatch routines under a fairness constraint. The fair termination property being checked is in Figure 2. The `main` function in SDV's model of the driver's environment considers all possible calls to the driver's dispatch routines— meaning that Figure 2 represents the termination of every dispatch routine within the device driver. The bug in driver 3 was previously unknown.

- As previously described, TERMINATOR uses pointer analysis to over-approximate the pointer aliasing relationships during instrumentation. In some cases this over-approximation may lead to aliasing relationships that do not occur in the program, which may result in false counterexamples being reported. In many cases false-aliasing relationships can be resolved later during binary reachability (as described in [14]), but not always.

Acknowledgments

Andreas Podelski and Andrey Rybalchenko are supported in part by the German Research Foundation (DFG) as a part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS), by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

Moshe Vardi is supported in part by NSF grants CCR-9988322, CCR-0124077, CCR-0311326, and ANI-0216467, by BSF grant 9800096, by Texas ATP grant 003604-0058-2003, and by a Guggenheim Fellowship. Part of this work was done while the author was visiting the Isaac Newton Institute for Mathematical Science, as part of a Special Programme on Logic and Algorithms, as well as Microsoft Research, Cambridge, UK.

References

[1] ALBIN ET AL. Property specification language reference manual. Tech. Rep. Version 1.1, Accellera, 2004.

[2] ALPERN, B., AND SCHNEIDER, F. Defining liveness. *Information processing letters 21* (1985), 181–185.

[3] ALPERN, B., AND SCHNEIDER, F. Recognizing safety and liveness. *Distributed computing 2* (1987), 117–126.

[4] ARMONI, R., FIX, L., FLAISHER, A., GERTH, R., GINSBURG, B., KANZA, T., LANDVER, A., MADOR-HAIM, S., SINGERMAN, E., TIEMEYER, A., VARDI, M., AND ZBAR, Y. The ForSpec temporal logic: A new temporal property-specification logic. In *TACAS'02: Tools and Algorithms for the Construction and Analysis of Systems* (2002), vol. 2280 of *LNCS*, Springer-Verlag, pp. 296–311.

[5] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. Thorough static analysis of device drivers. In *EuroSys'06: European Systems Conference* (2006).

[6] BALL, T., AND RAJAMANI, S. K. SLIC: A specification language for interface checking (of C). Tech. Rep. MSR-TR-2001-21, Microsoft Research, 2001.

[7] BERDINE, J., CHAWDHARY, A., COOK, B., DISTEFANO, D., AND O'HEARN, P. Variance analyses from invariance analyses. In *POPL'07: Principles of Programming Languages* (2007), ACM Press.

[8] BERDINE, J., COOK, B., DISTEFANO, D., AND O'HEARN, P. Automatic termination proofs for programs with shape-shifting heaps. In *CAV'06: Computer-Aided Verification* (2006), vol. 4144 of *LNCS*, Springer-Verlag, pp. 386–400.

[9] BIERE, A., ARTHO, C., AND SCHUPPAN, V. Liveness checking as safety checking. In *FMICS'02: Formal Methods for Industrial Critical Systems* (2002), vol. 66(2) of *ENTCS*.

[10] BRADLEY, A., MANNA, Z., AND SIPMA, H. Linear ranking with reachability. In *CAV'05: Computer-Aided Verification* (2005), vol. 3576 of *LNCS*, Springer-Verlag, pp. 491–504.

[11] BRADLEY, A., MANNA, Z., AND SIPMA, H. Termination of polynomial programs. In *VMCAI'05: Verification, Model Checking, and Abstract Interpretation* (2005), vol. 3385 of *LNCS*, Springer-Verlag, pp. 113–129.

[12] COLÓN, M. A., AND URIBE, T. E. Generating finite-state abstractions of reactive systems using decision procedures. In

CAV 98: Computer-Aided Verification (1998), vol. 1427 of *LNCS*, Springer-Verlag, pp. 293–304.

[13] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Abstraction refinement for termination. In *SAS'05: Static Analysis Symposium* (2005), vol. 3672 of *LNCS*, Springer-Verlag, pp. 87–101.

[14] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Termination proofs for systems code. In *PLDI'06: Programming Language Design and Implementation* (2006), ACM Press, pp. 415–426.

[15] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Terminator: Beyond safety. In *CAV'06: Computer-Aided Verification* (2006), vol. 4144 of *LNCS*, Springer-Verlag, pp. 415–418.

[16] CORBETT, J., DWYER, M., HATCLIFF, J., PASAREANU, C., ROBBY, LAUBACH, S., AND ZHENG, H. Bandera: Extracting finite-state models from Java source code. In *ICSE'00: Int. Conf. on Software Engineering* (2000), IEEE Press, pp. 439–448.

[17] COUSOT, P. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In *VMCAI'05: Verification, Model Checking, and Abstract Interpretation* (2005), vol. 3385 of *LNCS*, Springer-Verlag, pp. 1–24.

[18] DWYER, M. B., AVRUNIN, G. S., AND CORBETT, J. C. Patterns in property specifications for finite-state verification. In *ICSE'99: Int. Conf. on Software Engineering* (1999), IEEE Press, pp. 411–420.

[19] FLOYD, R. W. Assigning meanings to programs. In *Mathematical Aspects of Computer Science* (1967), J. T. Schwartz, Ed., vol. 19 of *Proceedings of Symposia in Applied Mathematics*, American Mathematical Society, pp. 19–32.

[20] GRAF, S., AND SAÏDI, H. Construction of abstract state graphs with PVS. In *CAV 97: Computer-Aided Verification* (1997), vol. 1254 of *LNCS*, Springer-Verlag, pp. 72–83.

[21] HENZINGER, T., JHALA, R., MAJUMDAR, R., AND SUTRE, G. Lazy abstraction. In *POPL'02: Principles of Programming Languages* (2002), ACM Press, pp. 58–70.

[22] HOLZMANN, G. J. The model checker SPIN. *IEEE Transactions on Software Engineering 23*, 5 (1997), 279–295.

[23] KESTEN, Y., PNUELI, A., AND RAVIV, L. Algorithmic verification of linear temporal logic specifications. In *ICALP'98: Int. Colloq. on Automata, Languages and Programming* (1998), vol. 1443 of *LNCS*, Springer-Verlag, pp. 1–16.

[24] KURSHAN, R. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[25] MANNA, Z., AND PNUELI, A. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, 1992.

[26] MICROSOFT CORPORATION. Windows Static Driver Verifier. Available at www.microsoft.com/whdc/devtools/tools/SDV.mspx, July 2006.

[27] PNUELI, A. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundation of Computer Science* (1977), pp. 46–57.

[28] PNUELI, A., PODELSKI, A., AND RYBALCHENKO, A. Separating fairness and well-foundedness for the analysis of fair discrete systems. In *TACAS'05: Tools and Algorithms for the Construction and Analysis of Systems* (2005), vol. 3440 of *LNCS*, Springer-Verlag, pp. 124–139.

[29] PODELSKI, A., AND RYBALCHENKO, A. A complete method for the synthesis of linear ranking functions. In *VMCAI'04: Verification, Model Checking, and Abstract Interpretation* (2004), vol. 2937 of *LNCS*, Springer-Verlag, pp. 239–251.

[30] PODELSKI, A., AND RYBALCHENKO, A. Transition invariants. In *LICS'04: Logic in Computer Science* (2004), LNCS, IEEE Press, pp. 32–41.

[31] VARDI, M. Verification of concurrent programs—the automata-theoretic framework. *Annals of Pure and Applied Logic 51* (1991), 79–98.

[32] VARDI, M., AND WOLPER, P. Reasoning about infinite computations. *Information and Computation 115*, 1 (1994), 1–37.

[33] VISSER, W., HAVELUND, K., BRAT, G., PARK, S., AND LERDA, F. Model checking programs. *Automated Software Engineering Journal 10*, 2 (2003).