Fairness Modulo Theory: A New Approach to LTL Software Model Checking

Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski

University of Freiburg, Germany

Abstract. The construction of a proof for unsatisfiability is less costly than the construction of a ranking function. We present a new approach to LTL software model checking (i.e., to statically analyze a program and verify a temporal property from the full class of LTL including general liveness properties) which aims at exploiting this fact. The idea is to select finite prefixes of a path and check these for infeasibility before considering the full infinite path. We have implemented a tool which demonstrates the practical potential of the approach. In particular, the tool can verify several benchmark programs for a liveness property just with finite prefixes (and thus without the construction of a single ranking function).

1 Introduction

The long line of research on *software model checking*, i.e., on tools that statically analyze a given program in order to automatically verify a given temporal property, was initially restricted to safety properties [2, 3, 11, 20, 37, 45, 51]. It was later extended to termination [9, 21, 26, 27, 36, 40, 49, 50, 52]. The relative maturity of this research is reflected by the fact that software model checking tools successfully participate in the software verification competition SV-Comp [10], for safety [29, 33, 41, 47] as well as for termination [33, 55, 56].

In a more recent trend, approaches to software model checking are emerging for the general class of LTL properties, and in particular general liveness properties [5, 22, 23, 24]. In this paper, we introduce an approach to LTL software model checking which is based on *fairness modulo theory*, an extension of *reachability modulo theory* as introduced by Lal and Qadeer [42].

In the setting of [42], the existence of a program execution that violates a given safety property is proven via the reachability of an error location of the program along a *feasible* path. A path is feasible if the sequence of statements along the path is executable. This condition is checked by checking whether the corresponding logical formula is *satisfiable modulo theory* (i.e., satisfiable in the logical theory of integers, arrays, etc.). Today, quite efficient *SMT solvers* exist

which can not only prove unsatisfiability but also compute *interpolants* [13, 14, 17, 19, 46]. Interpolants can be used to generalize the proof of unsatisfiability in order to show the infeasibility of more and more paths and eventually prove the unreachability of an error location (which is the underlying idea in the approach to program verification of [35, 36]).

We extend the setting of [42] to LTL by defining the construction of a new kind of program (a *Büchi program*) from the input program and the LTL property. The control flow graph of a Büchi program comes with a distinguished set of nodes which is used to define (infinite) *fair* paths (a path is fair if it visits the distinguished set of nodes infinitely often). Now, in our extension of the setting in [42], the existence of a program execution that violates a given LTL property is proven via the existence of a feasible *fair* path.

In general, to show that the infinite sequence of statements along a path is not executable, one needs to construct a *ranking function*. For example, for each of the two infinite sequences of statements below, one may construct the ranking function r defined by $r(x, y) = x - y$.

τ_1 : `x--` `x>y` `x--` `x>y` `x--` `x>y` ...
τ_2 : `x:=y` `x>y` `x--` `x>y` `x--` `x>y` ...

Every finite prefix of τ_1 is executable. In constrast, τ_2 has the prefix `x:=y` `x>y` which already is not executable.

In the case where an infinite sequence of statements has a finite prefix such that already the prefix is not executable, it is not necessary to construct a ranking function. Instead, it is sufficient to consider the prefix and prove the unsatisfiability of the logical formula corresponding to the finite sequence of statements in the prefix.

Tools exist that, given an infinite sequence of statements like (`x>y` `x--`)$^\omega$ or `x:=y` `x>y` (`x>y` `x--`)$^\omega$, can construct a ranking function like r above automatically [7, 12, 48]. Recent efforts go into improving the scope and the scalability of such tools [8, 25, 34, 43]. In comparison with proving unsatisfiability, the task of constructing a ranking function will always be more costly. Hence, substituting the construction of a ranking function by the construction of a proof of unsatisfiability carries an interesting potential for optimization. The goal of the work in this paper is to investigate whether this potential can be exploited practically. We develop a practical method and tool for LTL software model checking that shows that this is indeed the case.

In the remainder of the paper, after discussing an example, we introduce *Büchi programs* (as described above, we reduce the validity of an LTL property for a given program to the absence of a feasible fair path in a Büchi program). We present an algorithm that constructs such a Büchi program and checks whether it has a feasible fair path. The algorithm selects certain finite prefixes of a path for the check of feasibility before the full infinite path is considered. We then present the evaluation of a tool which implements the algorithm. Our evaluation shows the practical potential of our approach. In particular, the tool can verify several benchmark programs—for a liveness property—just with finite prefixes (and thus without the construction of a single ranking function).

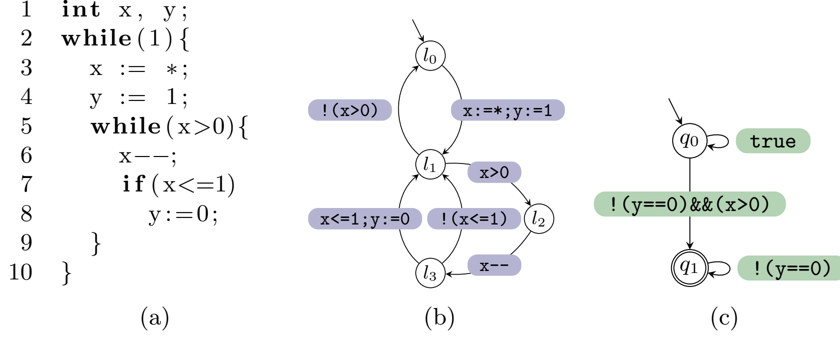


Fig. 1: Program $\mathcal{P}$ is shown in (a) as pseudocode and in (b) as control flow graph. The Büchi automaton $\mathcal{A}_{\neg\varphi}$ that represents the negation of the LTL property $\varphi = \Box(x > 0 \rightarrow \Diamond(y = 0))$ is shown in (c).

2 Example

In this section we demonstrate how we apply our approach to the program $\mathcal{P}$ depicted in Figure 1a and the LTL property $\varphi = \Box(x > 0 \rightarrow \Diamond(y = 0))$.

We represent the program $\mathcal{P}$ by the graph depicted in Figure 1b. The edges of this graph are labeled with program statements. We use the Büchi automaton $\mathcal{A}_{\neg\varphi}$ depicted in Figure 1c as representation of the negation of the LTL property φ.

As a first step we construct the *Büchi program* $\mathcal{B}$ depicted in Figure 2. Afterwards we will show that this Büchi program $\mathcal{B}$ has no path that is fair and feasible, thus proving that $\mathcal{P}$ satisfies the LTL property φ.

A Büchi program is a program together with a fairness constraint: an execution is *fair* if a *fair location* is visited infinitely often. The fair locations of $\mathcal{B}$ are highlighted by double circles. The locations of the Büchi program $\mathcal{B}$ are pairs whose first element is a location of the program $\mathcal{P}$ and whose second element is a state of the Büchi automaton $\mathcal{A}_{\neg\varphi}$. The edges of the Büchi program $\mathcal{B}$ are labeled with sequential compositions of two statements where the first element is a statement of the program. The second element of the sequential composition is an assume statement that represents a letter of the Büchi automaton $\mathcal{A}_{\neg\varphi}$.

A key concept in our analysis is the notion of a *trace*. A trace is an infinite sequence of statements. We call a trace *fair* if it is the labeling of a path that visits some fair location infinitely often. A trace is *feasible* if it corresponds to some program execution. An example for a fair trace is $\tau_1\tau_2^\omega$ where τ_1 and τ_2 are as follows.

τ_1 : `x:=*;y:=1` `!(y==0)&&(x>0)` `!(x>0)` `!(y==0)`
τ_2 : `x:=*;y:=1` `!(y==0)` `!(x>0)` `!(y==0)`

This trace is not feasible because the second statement `!(y==0)&&(x>0)` and the third statement `!(x>0)` are contradicting each other.

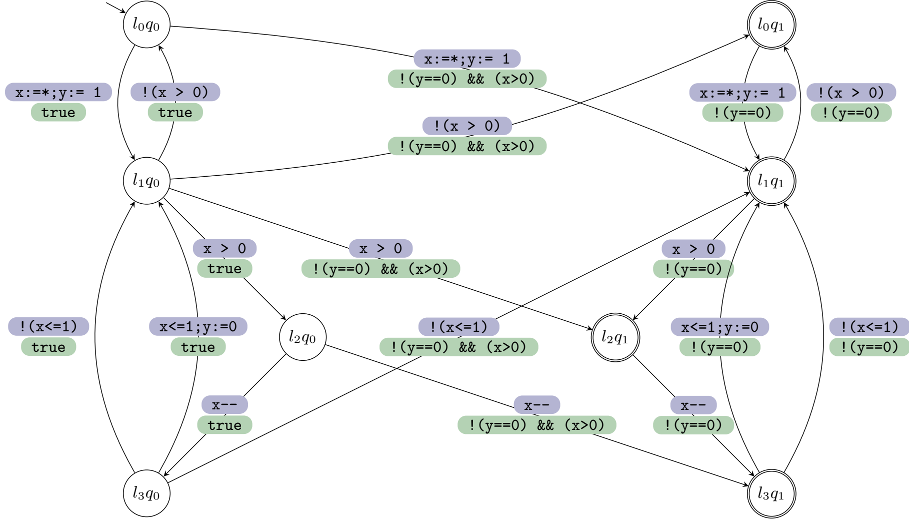


Fig. 2: The Büchi program $\mathcal{B}$ constructed from the program $\mathcal{P}$ (Figure 1b) and the Büchi automaton representing $\neg\varphi$ (Figure 1c). Each edge is labeled with the statements $\texttt{s}_1$ $\texttt{s}_2$, where $\texttt{s}_1$ comes from $\mathcal{P}$ and $\texttt{s}_2$ comes from $\neg\varphi$. The fair locations are l_0q_1,l_1q_1, l_2q_1 and l_3q_1, i.e., all locations that contain the Büchi automaton's accepting state q_1.

Our algorithm constructs Büchi programs such that each fair and feasible trace of the Büchi program corresponds to a feasible trace of the original program that violates the LTL property.

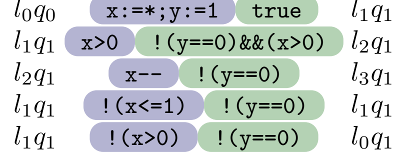
In order to show that $\mathcal{P}$ satisfies φ we show that no fair trace of the Büchi program $\mathcal{B}$ is feasible. Thus, our algorithm tries to find arguments for infeasibility of fair traces in $\mathcal{B}$:

Local infeasiblity. In the Büchi program $\mathcal{B}$ every trace that is the labeling of a path that contains the edge

$$l_3q_1 \; \texttt{x<=1; y:=0} \; \texttt{!(y==0)} \; l_1q_1$$

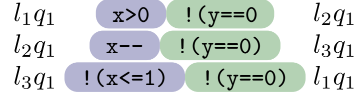
is infeasible, because the statements `y:=0` and `!(y==0)` contradict each other. Another example for *local infeasibiliy* is the edge from l_1q_0 to l_0q_1 which is labeled with the two statements `!(x>0)` and `(x>0)` that contradict each other, too.

Infeasiblity of a finite prefix. Every trace that is the labeling of a path that has the following finite prefix



is infeasible because `!(x<=1)` contradicts `!(x>0)`. Another example for infeasibiliy of a finite prefix is the trace $\tau_1 \tau_2^\omega$ that was discussed before.

ω-Infeasibility. Every trace that is the labeling of an infinite path that eventually loops along the following edges



is infeasible because `x--` infinitely often decreases `x`. Thus, the value of `x` will eventually contradict `!(x<=1)`. The formal termination argument is the ranking function $f(x) = x$.

Each fair trace of $\mathcal{B}$ is infeasible for one of the reasons mentioned above. Hence, we can conclude that program $\mathcal{P}$ indeed satisfies the LTL property φ.

All reasons for infeasiblity that fall into the classes *Local infeasiblity* or *Infeasiblity of a finite prefix* are comparatively cheap to detect. In this example we only needed to synthesize one ranking function, which is in general more expensive.

3 Preliminaries

Programs and Traces. In our formal exposition we consider a simple programming language whose statements are assignment, assume, and sequential composition. We use the syntax that is defined by the following grammar

$$\texttt{s} ::= \texttt{assume bexpr} \mid \texttt{x:=expr} \mid \texttt{s;s}$$

where *Var* is a finite set of program variables, $\texttt{x} \in \mathit{Var}$, `expr` is an expression over *Var* and `bexpr` is a Boolean expression over *Var*. For brevity we use `bexpr` to denote the assume statement `assume bexpr`.

We represent a *program* over a given set of statements *Stmt* as a labeled graph $\mathcal{P} = (\mathit{Loc}, \delta, l_0)$ with a finite set of nodes *Loc* called locations, a set of edges labeled with statements, i.e., $\delta \subseteq \mathit{Loc} \times \mathit{Stmt} \times \mathit{Loc}$, and a distinguished node l_0 which we call the initial location.

In the following we consider only programs where each location has at least one outgoing edge, i.e. $\forall l \in \mathit{Loc},\ \exists s \in \mathit{Stmt},\ \exists l' \in \mathit{Loc} \bullet (l, s, l') \in \delta$. We note

that each program can be transformed into this form by adding to each location without outgoing edges a selfloop that is labeled with `assume true`.

We call an infinite sequence of statements $\tau = s_0 s_1 s_2 \ldots$ a *trace of the program* $\mathcal{P}$ if τ is the edge labeling of an infinite path that starts at the initial location l_0. We define the set of all program traces formally as follows.

$$T(\mathcal{P}) = \{s_0 s_1 \ldots \in \mathit{Stmt}^\omega \mid \exists l_1, l_2, \ldots \bullet (l_i, s_i, l_{i+1}) \in \delta, \text{ for } i = 0, 1, \ldots\}$$

Let $\mathcal{D}$ be the set of values of the program's variables. We denote a program state σ as a function $\sigma : \mathit{Var} \to \mathcal{D}$ that maps program variables to values. We use S to denote the set of all program states. Each statement $\mathtt{s} \in \mathit{Stmt}$ defines a binary relation $\rho_\mathtt{s}$ over program states which we call the *successor relation*. Let Expr be set of all expressions over the program variables Var. We assume a given interpretation function $\mathcal{I} : \mathit{Expr} \times (\mathit{Var} \to \mathcal{D}) \to \mathcal{D}$ and define the relation $\rho_\mathtt{s} \subseteq S \times S$ inductively as follows.

$$\rho_s = \begin{cases} \{(\sigma, \sigma') \mid \mathcal{I}(\mathtt{bexpr})(\sigma) = \mathit{true} \text{ and } \sigma = \sigma'\} & \text{if } s \equiv \mathtt{assume\ bexpr} \\ \{(\sigma, \sigma') \mid \sigma' = \sigma[\mathtt{x} \mapsto \mathcal{I}(\mathtt{expr})(\sigma)]\} & \text{if } s \equiv \mathtt{x\!:=\!expr} \\ \{(\sigma, \sigma') \mid \exists \sigma'' \bullet (\sigma, \sigma'') \in \rho_{\mathtt{s}_1} \text{ and } (\sigma'', \sigma') \in \rho_{\mathtt{s}_2}\} & \text{if } s \equiv \mathtt{s}_1\mathtt{;s}_2 \end{cases}$$

Given a trace $\tau = s_0 s_1 s_2 \ldots$, a sequence of program states $\pi = \sigma_0 \sigma_1 \sigma_2 \ldots$ is called a *program execution of the trace* τ if each successive pair of program states is contained in the successor relation of the corresponding statement of the trace, i.e., $(\sigma_i, \sigma_{i+1}) \in \rho_{s_i}$ for $i \in \{0, 1, \ldots\}$. We call a trace τ *infeasible* if it does not have any program execution, otherwise we call τ *feasible*. We use $\Pi(\tau)$ to denote the set of all program executions of τ. The set of all feasible trace of program $\mathcal{P}$ is denoted by $T_{\mathit{feas}}(\mathcal{P})$, and the set of all program executions of $\mathcal{P}$ is defined as follows.

$$\Pi(\mathcal{P}) = \bigcup_{\tau \in T_{\mathit{feas}}(\mathcal{P})} \Pi(\tau)$$

Büchi automata and LTL properties. We will not formally introduce linear temporal logic (LTL). Every LTL property can be expressed as a Büchi automaton [1]. In our formal presentation we use Büchi automata to represent LTL properties.

A *Büchi automaton* $\mathcal{A} = (\Sigma, Q, q_0, \longrightarrow, F)$ is a five tuple consisting of a finite alphabet Σ, a finite set of states Q, an initial state $q_0 \in Q$, a transition relation $\longrightarrow : Q \times \Sigma \times Q$, and a set of accepting states $F \subseteq Q$. A *word* over the alphabet Σ is an infinite sequence $w = a_0 a_1 a_2 \ldots$ such that $a_i \in \Sigma$ for all $i \geq 0$. A *run* r of a Büchi automaton $\mathcal{A}$ on w is an infinite sequence of states $q_0 q_1 \ldots$, starting in the initial state such that for all $a_i \in w$ there is a transition $(q_i, a_i, q_{i+1}) \in \longrightarrow$. A run r is called *accepting* if r contains infinitely many accepting states. A word w is *accepted* by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. The *language* $\mathcal{L}(\mathcal{A})$ of a Büchi automaton $\mathcal{A}$ is the set of all words that are accepted by $\mathcal{A}$.

An *atomic proposition* is a set of program states. An *LTL property* over a set of atomic propositions AP defines a set of words over the alphabet $\Sigma = 2^{AP}$.

LTL properties are usually denoted by formulas, but several translations from formulas to equivalent Büchi automata are available [31, 32, 54]. We assume that we have given a Büchi automaton $\mathcal{A}_\varphi$ for each LTL property φ.

A program state σ *satisfies* a symbol a of the alphabet 2^{AP} if σ is an element of all atomic propositions in a. A sequence of program states $\sigma_0\sigma_1\ldots$ *satisfies* a word $a_0a_1a_2\ldots \in (2^{AP})^\omega$, if σ_{i+1} satisfies a_i for each $i \geq 0$. A sequence of program states π *satisfies* the LTL property φ if π satisfies some word $w \in \mathcal{A}_\varphi$. A trace $\tau = s_0s_1\ldots$ *satisfies* φ if it has at least one program execution and all program executions of the trace satisfy φ. A program $\mathcal{P}$ *satisfies* φ if all program executions of $\mathcal{P}$ satisfy φ. We will use the $\models$ symbol to denote each of these "satisfies relations", e.g., we will write $\mathcal{P} \models \varphi$ if the program $\mathcal{P}$ satisfies the LTL property φ.

We note that these definitions do not put any restrictions on the initial state σ_0 of a sequence of program states. This accounts for the fact that our programs do not have to start in a given initial program state and allows programs that satisfy the LTL property $\Box(x = 0)$. For example, the program whose first statement sets the variable x to 0 and whose other statements do not modify x.

4 Büchi Program and Büchi Program Product

In this section we introduce the notion of a Büchi program, which is a program which is extended by a fairness constraint. We show that the problem whether a program satisfies an LTL property can be reduced to the problem whether a Büchi program has a fair program execution.

Definition 1 (Büchi program). *A Büchi program* $\mathcal{B} = (\mathit{Stmt}, \mathit{Loc}, \delta, l_0, \mathit{Loc}_{\mathit{fair}})$ *is a program* $\mathcal{P} = (\mathit{Loc}, \delta, l_0)$ *whose set of statements is Stmt, with a distinguished subset of locations* $\mathit{Loc}_{\mathit{fair}} \subseteq \mathit{Loc}$*. We call the locations* $\mathit{Loc}_{\mathit{fair}}$ *the* fair locations *of* $\mathcal{B}$*.*

An example for a Büchi program is the program depicted in Figure 2 which was discussed in Section 2.

Definition 2 (Fair trace). *A trace* $s_0s_1s_2\ldots$ *of a Büchi program* $\mathcal{B}$ *is a* fair trace *if*

- *there exists a sequence of locations* $l_0, l_1, \ldots$ *such that* $l_0 \xrightarrow{s_0} l_1 \xrightarrow{s_1} l_2 \xrightarrow{s_2} \ldots$ *is a path in* $\mathcal{B}$*, i.e.,* $(l_i, s_i, l_{i+1}) \in \delta$ *for* $i = 0, 1, \ldots$*, and*
- *the sequence* $l_0, l_1, \ldots$ *contains infinitely many fair locations.*

We use $T_{\mathit{fair}}(\mathcal{B})$ *to denote the set of* fair traces *of* $\mathcal{B}$*.*

If we consider the Büchi program $\mathcal{B} = (\mathit{Stmt}, \mathit{Loc}, \delta, l_0, \mathit{Loc}_{\mathit{fair}})$ as a Büchi automaton where the alphabet is the set of program statements *Stmt*, the set of states is the set of program locations *Loc*, the transition relation is the labeled edge relation δ the initial state is the initial location l_0 and the set of accepting states is the set of fair locations $\mathit{Loc}_{\mathit{fair}}$, then the language of this Büchi automaton is exactly the set of fair traces of the Büchi program.

Definition 3 (Fair program execution).
A program execution π of a Büchi program $\mathcal{B}$ is a fair program execution *of $\mathcal{B}$ if π is the program execution of some fair trace of $\mathcal{B}$. We use $\Pi_{fair}(\mathcal{B})$ to denote the set of all fair program execution of $\mathcal{B}$.*

We note that traces that are fair and feasible have at least one fair program execution.

Boolean expressions over the set of program variables Var, and atomic propositions both define sets of program states. For a letter $a \in 2^{AP}$, we will use `assume a` to denote the assume statement whose expression evaluates to *true* for each state σ that satisfies a. Hence `assume a` has the following successor relation.

$$\{(\sigma, \sigma') \mid \sigma \models p \text{ for each } p \in a\}$$

Definition 4 (Büchi program product). *Let $\mathcal{P} = (Loc, l_0, \delta_{\mathcal{P}})$ be a program over the set of statements Stmt, AP a set of atomic propositions over the program's variables Var, and let $\mathcal{A} = (\Sigma, Q, q_0, \rightarrow, F)$ be a Büchi automaton whose alphabet is $\Sigma = 2^{AP}$. The* Büchi program product $\mathcal{P} \otimes \mathcal{A}$ *is a Büchi program $\mathcal{B} = (Stmt_{\mathcal{B}}, Loc_{\mathcal{B}}, l_{0_{\mathcal{B}}}, \delta_{\mathcal{B}}, Loc_{F_{\mathcal{B}}})$ such that the set of statements consists of all sequential compositions of two statements where the first element is a statement of $\mathcal{P}$ and the second element is a statement that assumes that a subset of atomic propositions is satisfied, i.e.,*

$$Stmt_{\mathcal{B}} = \{\texttt{s; assume a} \mid s \in Stmt, a \in 2^{AP}\},$$

the set of locations is the Cartesian product of program locations and Büchi automaton states, i.e.,

$$Loc_{\mathcal{B}} = \{(l, q) \mid l \in Loc \text{ and } q \in Q\},$$

the initial location is the pair consisting of the program's initial location and the Büchi automaton's initial state, i.e.,

$$l_{0_{\mathcal{B}}} = (l_0, q_0),$$

the labeled edge relation is a product of the program's edge relation and the transition relation of the Büchi automaton such that an edge is labeled by the statement that is a sequential composition of the program's edge label and an assume statement obtained from the transition's letter, formally defined as follows

$$\delta_{\mathcal{B}} = \{((l, q), s; \texttt{assume a}, (l', q')) \mid (l, s, l') \in \delta_P \text{ and } (q, a, q') \in \rightarrow\},$$

the set of fair locations contains all pairs where the second component is an accepting state of the Büchi automaton, i.e.,

$$Loc_{F_{\mathcal{B}}} = \{(l, q) \mid l \in Loc \text{ and } q \in F\}.$$

The following theorem shows how we can use the Büchi program product to check if a program satisfies an LTL property.

Theorem 1. *The program $\mathcal{P}$ satisfies the LTL property φ if and only if the Büchi program product $\mathcal{B} = \mathcal{P} \otimes \mathcal{A}_{\neg\varphi}$ does not have a trace that is fair and feasible, i.e.,*

$$\mathcal{P} \models \varphi \qquad \textit{iff} \qquad T_{fair}(\mathcal{B}) \cap T_{feas}(\mathcal{B}) = \emptyset$$

Proof. For brevity, we give only a sketch of the proof. A more detailed proof is available in an extended version of this paper [30]. First, we use the definition of the Büchi program product to show the following connection between traces of $\mathcal{B}$, traces of $\mathcal{P}$ and words over 2^{AP}. $s_0; \texttt{assume a}_0 \;\; s_1; \texttt{assume a}_1 \ldots \in T_{fair}(\mathcal{B})$ if and only if $s_0 s_1 s_2 \ldots \in T(\mathcal{P})$ and $a_0 a_1 a_2 \ldots \in \mathcal{L}(\mathcal{A}_{\neg\varphi})$. Next, we use this equivalence to show that for a sequence of program states the following holds. $\pi \in \Pi_{fair}(\mathcal{B})$ if and only if $\pi \in \Pi(\mathcal{P})$ and $\pi \models \mathcal{A}_{\neg\varphi}$. A Büchi program has a fair program execution if and only if it has a fair and feasible trace. We conclude that the intersection $T_{fair}(\mathcal{B}) \cap T_{feas}(\mathcal{B})$ is empty if and only if each program execution of $\mathcal{P}$ satisfies the LTL property φ. □

5 LTL Software Model Checking

In this section we describe our LTL software model checking algorithm. The algorithm is based on counter example guided abstraction refinement (CEGAR) in the fashion of [35] extended by a check for termination of fair traces and a corresponding abstraction refinement.

Figure 3 shows an overview of the algorithm. The general idea is to create and continuously enlarge a Büchi automaton $\mathcal{A}_D$ whose language contains all fair traces of $\mathcal{B}$ that are already known to be infeasible. The algorithm starts by constructing a Büchi program $\mathcal{B}$ with the product construction from Section 4. Initially, $\mathcal{A}_D$ is a Büchi automaton that recognizes the empty language.

We use the similarities between Büchi programs and Büchi automata, i.e., that $\mathcal{L}(\mathcal{B}) = T_{feas}(\mathcal{B})$, throughout the whole algorithm. For example, in the first step of our CEGAR loop we check whether the set of fair traces represented by $\mathcal{A}_D$ is a superset of the fair traces of $\mathcal{B}$ (the first box in Figure 3). This check for *trace inclusion* can be done with only Büchi automata operations.

If the set of fair traces of $\mathcal{A}_D$ is indeed a superset of the set of fair traces of $\mathcal{B}$, we know that there is no fair and feasible trace in $\mathcal{B}$ and our algorithm returns *safe*.

As the trace inclusion check is performed by computing $\mathcal{L}(\mathcal{B}) \setminus \mathcal{L}(\mathcal{A}_D)$, we will receive a fair trace τ of $\mathcal{B}$ that witnesses that the set of fair traces of $\mathcal{A}_D$ is no superset of the set of fair traces of $\mathcal{B}$. In this case, τ is always of the form $\tau_1 \tau_2^\omega$.

Next, our algorithm tries to decide whether τ is feasible or not. This is done by first checking various finite prefixes for feasibility. More precisely, the stem τ_1, the loop τ_2 and then the concatenation $\tau_1 \tau_2$ are checked for feasibility in that order. If none of those finite prefixes is infeasible, our algorithm tries to prove that the full infinite trace terminates. The termination analysis (inner lower box) tries to find a ranking function to prove that the loop will terminate eventually.

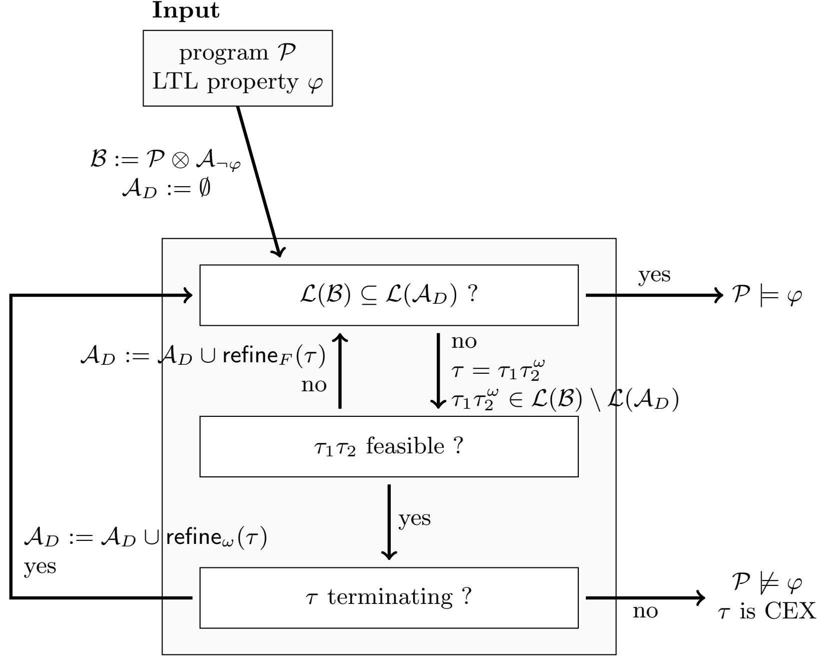


Fig. 3: The model checking algorithm. We use an automata-based approach that collects generalizations of infeasible traces in a Büchi automaton $\mathcal{A}_D$. The three inner boxes represent the three checks, which lead either to a refinement of $\mathcal{A}_D$, a result, or to a timeout (not shown).

When non-termination can be proven, we conclude that τ is feasible. Therefore, τ is a fair and feasible trace in $\mathcal{B}$ and thus a counterexample for the property φ. If instead termination can be shown, we know that τ is infeasible and the algorithm continues to the next step.

Note that the checks for feasibility of τ_1 and τ_2 as well as the termination analysis are based on – in general – undecidable methods. It is possible that they do not terminate. In such cases, our algorithm runs into a timeout and returns *unknown* as answer.

In the last step of the CEGAR loop we want to refine $\mathcal{A}_D$ by adding more fair and infeasible traces. We do this by replacing $\mathcal{A}_D$ with a Büchi automaton that is the union of the old Büchi automaton $\mathcal{A}_D$ and a new Büchi automaton which we create from trace τ. This new Büchi automaton recognizes all fair traces of $\mathcal{B}$ that are infeasible for the same reason for which trace τ is infeasible. Depending on the reason for infeasiblity of trace τ, we use different methods for the construction of this new Büchi automaton: if τ was infeasible because we found an infeasible finite prefix, we use the method refine_F, if τ was infeasible because we found a ranking function, we use refine_ω.

The methods refine_F and refine_ω generalize a single trace to a set of traces. The input of these methods is the trace τ together with an infeasibility proof (resp. termination proof). The output is a Büchi automaton that accepts a set of traces whose infeasibility (resp. termination) can be shown by this infeasibility proof (resp. termination proof). refine_F and refine_ω guarantee that at least the single trace is contained in the language, but usually recognize a much larger set of traces. As the generalization performed by these methods is quite involved, it is not in the scope of this paper. We refer the interested reader to [35, 36] for a detailed description.

6 Implementation and Evaluation

We implemented the algorithm from Section 5 as Ultimate LTLAutomizer in the program analysis framework Ultimate [16]. This allowed us to use different, already available components for our implementation:

- a parser for ANSI C extended with specifications written in ACSL [6],
- various source-to-source transformations that optimize and simplify the input program,
- an implementation of the Trace Abstraction algorithm [35] to determine feasibility of finite trace prefixes,
- an implementation of a ranking function synthesis algorithm based on [34] to prove termination of fair traces in the Büchi program, and
- various automata operations like union, complementation and intersection of Büchi automata.

For the LTL property we use a custom annotation compatible to the ACSL format. After parsing, we transform the LTL property with LTL2BA [32] to a Büchi automaton, which is then together with an initial program the input for the product algorithm.

Our implementation of the product construction already contains some optimizations. For one, we already described that we remove locally infeasible traces by removing infeasible edges during the construction. We also convert the expression `e` of `assume e` statements to disjunctive normal form. If this results in edges labeled with more than one disjuncts, i.e. with `assume` $\mathtt{e_1||e_2||...||e_n}$, we convert them to n edges labeled with `assume` $\mathtt{e_i}$. This improves the performance of the ranking function synthesis algorithm considerably.

Table 1 shows a comparison of our implementation against the benchmarks and the data provided by [23], in which the authors compare their novel LTL-checking approach based on decision predicates (DP) against a Terminator-like procedure with an extension for fairness [21] (Term.). The set of benchmarks contains examples from "[...] the I/O subsystem of the windows kernel, the back-end infrastructure of the PostgreSQL database server, and the Apache web server", as well as "some toy examples". As the tools that were used in [23] are not publicly available, we could not re-run their implementations on our

Program	Lines	φ	Term. [21] Time (s)	Term. [21] Result	DP [23] Time (s)	DP [23] Result	Ultimate LTLAutomizer Time (s)	Ultimate LTLAutomizer Result	$\|\mathtt{r}_F\|$	$\|\mathtt{r}_\omega\|$	Inc. (%)
Ex. Sec. 2 of [23]	5	$\Diamond\Box p$	2.32	✔	1.98	✔	0.51	✔	1	0	122
Ex. Fig. 8 of [21]	34	$\Box(p \Rightarrow \Diamond q)$	209.64	✔	27.94	✔	0.72	✔	2	0	186
Toy acquire/release	14	$\Box(p \Rightarrow \Diamond q)$	103.48	✔	14.18	✔	0.44	✔	1	1	129
Toy linear arith. 1	13	$p \Rightarrow \Diamond q$	126.86	(✔)	34.51	(✔)	1.10	✗	5	1	0.28
Toy linear arith. 2	13	$p \Rightarrow \Diamond q$	**T.O.**	**T.O.**	6.74	✔	0.82	✔	4	2	0.24
PostgreSQL strmsrv	259	$\Box(p \Rightarrow \Diamond\Box q)$	**T.O.**	**T.O.**	9.56	✔	1.04	✔	2	0	216
PostgreSQL strmsrv + bug	259	$\Box(p \Rightarrow \Diamond\Box q)$	87.31	(✗)	47.16	(✗)	0.66	✔	2	0	216
PostgreSQL pgarch	61	$\Diamond\Box p$	31.50	(✔)	15.20	(✔)	0.33	✗	2	0	209
PostgreSQL dropbuf	152	$\Box p$	**T.O.**	**T.O.**	1.14	(✔)	3.57	✗	1	1	148
PostgreSQL dropbuf	152	$\Box(p \Rightarrow \Diamond q)$	53.99	✔	27.54	✔	1.37	✔	2	1	168
Apache `accept()`	314	$\Box p \Rightarrow \Box\Diamond q$	**T.O.**	**T.O.**	197.41	✔	502.15	**OOM**	-	-	209
Apache progress	314	$\Box(p \Rightarrow (\Diamond q_1 \vee \Diamond q_2))$	685.34	✔	684.24	✔	2.01	✔	4	0	209
Windows OS 1	180	$\Box(p \Rightarrow \Diamond q)$	901.81	✔	539.00	✔	43.59	✔	1	1	178
Windows OS 2	158	$\Diamond\Box p$	16.47	✔	52.10	✔	0.11	✔	1	0	176
Windows OS 2 + bug	158	$\Diamond\Box p$	26.15	✗	30.37	✗	0.22	✗	1	0	174
Windows OS 3	14	$\Diamond\Box p$	4.21	✔	15.75	✔	0.08	✔	2	0	220
Windows OS 4	327	$\Box(p \Rightarrow \Diamond q)$	**T.O.**	**T.O.**	1,114.18	✔	1.86	✔	1	3	207
Windows OS 4	327	$(\Diamond p) \vee (\Diamond q)$	1,223.96	✔	100.68	✔	-	**N.R.**	-	-	-
Windows OS 5	648	$\Box(p \Rightarrow \Diamond q)$	**T.O.**	**T.O.**	**T.O.**	**T.O.**	20.76	✔	1	16	190
Windows OS 6	13	$\Diamond\Box p$	149.41	✔	59.56	✔	**T.O.**	**T.O.**	6	8	158
Windows OS 6 + bug	13	$\Diamond\Box p$	6.06	✗	22.12	✗	0.05	✗	0	0	61
Windows OS 7	13	$\Box\Diamond p$	**T.O.**	**T.O.**	55.77	✔	0.91	✔	2	11	161
Windows OS 8	181	$\Diamond\Box p$	**T.O.**	**T.O.**	5.24	✔	53.55	✔	4	55	168

Table 1: The results of the comparison with the benchmarks from [23]. "Program", "Lines", and "φ" contain the name of the benchmark, the lines of code of the program, and the checked property (atomic propositions have been abbreviated). "Result" states whether the tool proved the property (✔), produced a valid counterexample (✗), ran out of time (**T.O.**) or out of memory (**OOM**). **N.R.** shows the instance where we could not use the benchmark because the property was not specified explicitly and could not be guessed from the comments in the file. "Time" contains the runtime of the respective tool in seconds. For Ultimate LTLAutomizer, there are additional statistics columns: "$|\mathtt{r}_F|$" states how many traces were refined using refine_F, and analogous "$|\mathtt{r}_\omega|$" for refine_ω. "Inc." shows how much the product increased in size compared to the original CFG of the program. The timeout for "Term." and "DP" was four hours, our timeout was 20 minutes. Our memory limit was 8GB.

machine. Therefore, the results in the columns "Term." and "DP" are verbatim from the original publication.

We could solve most of the benchmarks in under five seconds. Notable exceptions are "Windows OS 5", where the other tools run into a timeout, and "Windows OS 8" where we performed much slower than DP. We are still unclear about the **OOM** result in "Apache `accept()`", but we suspect a bug in our tool.

In many instances with liveness properties we did not need to provide a ranking function, because the generalization from traces that are infeasible because of infeasible finite prefixes already excluded all fair traces of the Büchi program. For the remainder, the termination arguments were no challenge, except for "Windows OS 8": we had difficulties to generalize from many terminating traces, which also resulted in the slowdown compared to DP.

The expected increase in size of the Büchi program compared to the initial program's CFG (Inc.) was also manageable. Interestingly, in both instances of "Toy linear arith." the product was even smaller than the original CFG, because we could remove many infeasible edges.

On four benchmarks ULTIMATE LTLAUTOMIZER results are different from the data in [23]: we contacted the authors and confirmed that our result for "Toy linear arith. 1" is indeed correct. We also could not run the benchmark "Windows OS 4", because the LTL property contained variables that were not defined in the source file. We did not yet receive a response regarding this issue as well as regarding the correctness of our results in the other three instances.

								Statistics for ✔ and ✗			
Program set	Avg. Lines	|Set|	✔	✗	**T.O.**	**OOM**	★ (**N.R.**)	Avg. Time (s)	Avg. $\|\mathbf{r}_F\|$	Avg. $\|\mathbf{r}_\omega\|$	Inc. (%)
RERS P14	514	50	19	21	2	0	8	107.21	21	< 1	329
RERS P15	1353	50	24	0	11	12	3	103.46	17	< 1	369
RERS P16	1304	50	15	1	16	14	4	297.34	32	< 1	362
RERS P17	2100	50	26	0	9	9	6	56.38	12	< 1	324
RERS P18	3306	50	21	0	17	10	2	262.03	24	< 1	297
RERS P19	8079	50	0	0	28	17	5	-	-	-	-
coolant	65	18	6	10	2	0	0	1.75	2	1	258
Benchmarks from Tab. 1	157	23	15	5	1	1	0 (1)	16.78	2	5	184

Table 2: Results of ULTIMATE LTLAUTOMIZER on other benchmark sets. "RERS" are the online problems from "The RERS Grey-Box Challenge 2012" [39] and "coolant" consists of toy examples modelled after real-world embedded systems with specifications based on the LTL patterns described in [53]. Each program set contains pairs of a file and a property. "Avg. Lines" states the average lines of code in the sample set, and |Set| the number of file-property pairs. In the next five columns we use the same symbols as in Table 1 except for ★, which represents abnormal termination of ULTIMATE LTLAUTOMIZER. The last four columns show the average runtime, the average number of refinements with refine_F and refine_ω, and how much the size of the optimized product increased on average compared to the original CFG. We used the same timeout and memory limits as in Table 1.

We also considered two other benchmark sets (see Table 2). First, we ran the on-site problems from the RERS Grey-Box Challenge 2012 [39] (RERS). RERS is about comparing program verification techniques on a domain of problems comparable to the ones seen in embedded systems engineering. For this, they generate control-flow-intensive programs that contain a so-called ECA-engine (event-condition-action): one non-terminating while loop which first reads an input symbol, then calls a function that based on the current state and the input calculates an output symbol, and finally writes this output symbol. We took all 6 problem classes from the on-site part of the challenge and tried to solve them with our tool. The classification (P14 to P19) encodes the size and assumed difficulty of the problem class: P14 and P15 are small, P16 and P17 are medium, and P18 and P19 are large problems. Inside a size bracket, the larger number means a higher difficulty.

We were able to verify roughly 43% of the RERS benchmarks without any modifications. The RERS set also helped us finding a bug that one of our optimizations on the Büchi program product introduced and which is responsible for all but four of the ★ results. For the remaining four examples, ★ occurred because Ultimate LTLAutomizer was unable to synthesize a ranking function. Interestingly, the RERS benchmarks did seldomly require generalizations with refine_ω. In most cases, the refine_F already excluded all fair traces from the Büchi program. This trend can also be observed in the number of refine_ω applications on the benchmarks that timed out (not shown in Table 2).

Second, we used a small toy example modeled after an embedded system, a coolant facility controller that encompasses two potentially non-terminating loops in succession. The first polls the user for the input of a sane temperature limit (except one example all versions of the coolant controller can loop infinitely in this step if the input is not suitable). The second loop polls the temperature, does some calculations, increments a counter and sets the "spoiled goods" flag if the temperature limit is exceeded. The LTL properties specify that the spoiled variable cannot be reset by the program (safety), that setup stages occur in the correct order (safety and liveness), and that the temperature controlling loop always progresses (safety and liveness). We then introduced various bugs in the original version of the program and checked against the property and its negation. Although the coolant examples are quite small, they contain complex inter-dependencies between traces which lead to timeouts in two cases.

An unexpected result of the evaluation was, that the initial size of the program does not seem to define the performance of the verification, both in time and success rate, as the larger programs from P17 and P18 had more results and were faster than their counterparts from P15 and P16. Also, the effective blow-up due to the product construction is no more than four times, which is still quite manageable.

The benchmark sets together with Ultimate LTLAutomizer are available from [30].

7 Related Work

An earlier approach to LTL software model checking was done in [21]. There, the authors reduced the problem to fair termination checking. Our work can be seen as improvement upon this approach, as we also use fair termination checking, but only when it is necessary. We avoid a large number of (more costly) termination checks due to our previous check for infeasible finite prefixes and the resulting generalizing refinement.

In [23], the authors reduce the LTL model checking problem to the problem of checking ∀CTL by first approximating the LTL formula with a suitable CTL formula, and then refining counterexamples that represent multiple paths by introducing non-deterministic prophecy variables in their program representation. This non-determinism is then removed through a determinization procedure. By using this technique, they try to reduce their dependence on termination proofs, which they identified as the main reason for poor performance of automata-theoretic approaches. Our approach can be seen as another strategy to reduce the reliance on many termination proofs. By iteratively refining the Büchi program with different proof techniques, we often remove complex control structures from loops and thus reduce the strain on the termination proof engine.

There exist various publicly available finite-state model checking tools that support both LTL properties and programs, but are in contrast to ULTIMATE LTLAUTOMIZER limited to finite-state systems: SPIN [38] and Divine [4] are both based on the Vardi-Wolper product [57] for LTL model checking. Divine supports C/C++ via LLVM bytecode, SPIN can be used with different front-ends that translate programs to finite-state models, e.g. with Bandera [28] for Java. NuSMV [18] and Cadence SMV [44] reduce LTL model checking to CTL model checking. NuSMV can use different techniques like BDD symbolic model checking using symbolic fixed point, computation with BDDs, or bounded model checking using MiniSat. Cadence SMV uses Mu-calculus with additional fairness constraints [15].

8 Conclusion and Future Work

The encoding of the LTL program verification problem through the infeasibility of *fair* paths in a Büchi program has allowed us to define a sequence of semi-tests which can be scheduled before the full test of infeasibility of an infinite path. The occurrence of a successful semi-test (the proof of infeasibility for a finite prefix, by the construction of a proof of unsatisfiability) makes the full test redundant and avoids the relatively costly construction of a ranking function. Our experiments indicate that the corresponding approach leads to a practical tool for LTL software model checking.

We see several ways to improve performance. We may try to use alternatives to LTL2BA such as SPOT [31]; see [54]. The technique of large block encoding [11] adapted to Büchi programs, may help to reduce memory consumption.

References

[1] C. Baier, J.-P. Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[2] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213, 2001.

[3] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking C programs. *STTT*, 5(1):49–58, 2003.

[4] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *CAV*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.

[5] P. Bauch, V. Havel, and J. Barnat. LTL Model Checking of LLVM Bitcode with Symbolic Data. In *MEMICS*, pages 47–59. Springer, 2014.

[6] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto, et al. ACSL: ANSI/ISO C Specification Language, Feb. 2015. `http://frama-c.com/download.html`.

[7] A. M. Ben-Amram. Size-change termination, monotonicity constraints and ranking functions. In *CAV*, pages 109–123, 2009.

[8] A. M. Ben-Amram and S. Genaim. On the linear ranking problem for integer linear-constraint loops. In *POPL*, pages 51–62, 2013.

[9] J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV*, pages 386–400, 2006.

[10] D. Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In *TACAS*, pages 401–416, 2015.

[11] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software Model Checking via Large-Block Encoding. In *FMCAD*, pages 25–32. IEEE, 2009.

[12] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *CAV*, pages 491–504, 2005.

[13] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An interpolating sequent calculus for quantifier-free presburger arithmetic. *J. Autom. Reasoning*, 47(4):341–367, 2011.

[14] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The OpenSMT solver. In *TACAS*, pages 150–153, 2010.

[15] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, pages 428–439. IEEE, 1990.

[16] J. Christ, D. Dietsch, E. Ermis, M. Heizmann, J. Hoenicke, V. Langenfeld, J. Leike, B. Musa, A. Nutz, and C. Schilling. The Program Analysis Framework Ultimate, Feb. 2015. `http://ultimate.informatik.uni-freiburg.de`.

[17] J. Christ, J. Hoenicke, and A. Nutz. SMTInterpol: An interpolating smt solver. In *SPIN*, pages 248–254, 2012.

[18] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*, pages 359–364. Springer, 2002.

[19] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In *TACAS*, pages 93–107, 2013.

[20] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169. Springer, 2000.

[21] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving that programs eventually do something good. In *ACM SIGPLAN Notices*, volume 42, pages 265–276. ACM, 2007.

[22] B. Cook, H. Khlaaf, and N. Piterman. Fairness for infinite-state systems. In *TACAS 2015*, pages 384–398, 2015.
[23] B. Cook and E. Koskinen. Making prophecies with decision predicates. In *ACM SIGPLAN Notices*, volume 46, pages 399–410. ACM, 2011.
[24] B. Cook, E. Koskinen, and M. Y. Vardi. Temporal property verification as a program analysis task - extended version. *FMSD*, 41(1):66–82, 2012.
[25] B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger. Ranking function synthesis for bit-vector relations. *FMSD*, 43(1):93–120, 2013.
[26] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426. ACM, 2006.
[27] B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *PLDI*, pages 320–330, 2007.
[28] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, H. Zheng, et al. Bandera: Extracting finite-state models from Java source code. In *ICSE*, pages 439–448. IEEE, 2000.
[29] M. Dangl, S. Löwe, and P. Wendler. Cpachecker with support for recursive programs and floating-point arithmetic - (competition contribution). In *TACAS*, pages 423–425, 2015.
[30] D. Dietsch, M. Heizmann, and V. Langenfeld. Ultimate LTLAutomizer website., Feb. 2015. `http://ultimate.informatik.uni-freiburg.de/ltlautomizer`.
[31] A. Duret-Lutz and D. Poitrenaud. SPOT: an Extensible Model Checking Library using Transition-based Generalized Büchi Automata. In *MASCOTS*, pages 76–83. IEEE, 2004.
[32] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV*, pages 53–65. Springer, 2001.
[33] M. Heizmann, D. Dietsch, J. Leike, B. Musa, and A. Podelski. Ultimate automizer with array interpolation - (competition contribution). In *TACAS*, pages 455–457, 2015.
[34] M. Heizmann, J. Hoenicke, J. Leike, and A. Podelski. Linear ranking for linear lasso programs. In *ATVA*, pages 365–380, 2013.
[35] M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In *CAV*, pages 36–52, 2013.
[36] M. Heizmann, J. Hoenicke, and A. Podelski. Termination analysis by learning terminating programs. In *CAV*, pages 797–813, 2014.
[37] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.
[38] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*, volume 1003. Addison-Wesley Reading, 2004.
[39] F. Howar, M. Isberner, M. Merten, B. Steffen, and D. Beyer. The RERS Grey-Box Challenge 2012: Analysis of Event-Condition-Action Systems. In *ISoLA*, pages 608–614. Springer, 2012.
[40] D. Kroening, N. Sharygina, A. Tsitovich, and C. M. Wintersteiger. Termination analysis with compositional transition invariants. In *CAV*, pages 89–103. Springer, 2010.
[41] D. Kroening and M. Tautschnig. CBMC - C bounded model checker - (competition contribution). In *TACAS*, pages 389–391, 2014.
[42] A. Lal and S. Qadeer. Reachability modulo theories. In *RP 2013*, pages 23–44, 2013.
[43] J. Leike and M. Heizmann. Ranking templates for linear loops. In *TACAS*, pages 172–186, 2014.

[44] K. McMillan. Cadence SMV. *Cadence Berkeley Labs, CA*, 2000. `http://www.kenmcmil.com/smv.html`.
[45] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136. Springer, 2006.
[46] K. L. McMillan. Interpolants from Z3 proofs. In *FMCAD*, pages 19–27, 2011.
[47] A. Nutz, D. Dietsch, M. M. Mohamed, and A. Podelski. ULTIMATE KOJAK with memory safety checks - (competition contribution). In *TACAS*, pages 458–460, 2015.
[48] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.
[49] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, pages 32–41. IEEE Computer Society, 2004.
[50] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL*, pages 132–144. ACM, 2005.
[51] A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, pages 245–259, 2007.
[52] A. Podelski, A. Rybalchenko, and T. Wies. Heap assumptions on demand. In *CAV*, pages 314–327, 2008.
[53] A. C. Post. *Effective Correctness Criteria for Real-time Requirements*. Shaker, 2012.
[54] K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In *Model checking software*, pages 149–167. Springer, 2007.
[55] T. Ströder, C. Aschermann, F. Frohn, J. Hensel, and J. Giesl. Aprove: Termination and memory safety of C programs - (competition contribution). In *TACAS*, pages 417–419, 2015.
[56] C. Urban. Function: An abstract domain functor for termination - (competition contribution). In *TACAS*, pages 464–466, 2015.
[57] M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *LICS*, pages 322–331. IEEE Computer Society, 1986.