T2: Temporal Property Verification

Marc Brockschmidt[1], Byron Cook[2], Samin Ishtiaq[1], Heidy Khlaaf[2], and Nir Piterman[3]

[1]Microsoft Research Cambridge, [2]University College London, [3]University of Leicester

Abstract. We present the open-source tool T2, the first public release from the TERMINATOR project [9]. T2 has been extended over the past decade to support automatic temporal-logic proving techniques and to handle a general class of user-provided liveness and safety properties. Input can be provided in a native format and in C, via the support of the LLVM compiler framework. We briefly discuss T2's architecture, its underlying techniques, and conclude with an experimental illustration of its competitiveness and directions for future extensions.

1 Introduction

We present T2 (TERMINATOR 2), an open-source framework that implements, combines, and extends techniques developed over the past decade aimed towards the verification of temporal properties of programs. T2 operates on an input format that can be automatically extracted from the LLVM compiler framework's intermediate representation, allowing T2 to analyze programs in a wide range of programming languages (*e.g.* C, C++, Objective C, ...). T2 allows users to (dis)prove *CTL*, *Fair-CTL*, and *CTL** specifications via a reduction to its *safety*, *termination* and *nontermination* analysis techniques. Furthermore, *LTL* specifications can be checked using the automata-theoretic approach for LTL verification [26] via a reduction to fair termination, which is subsumed by Fair-CTL.

In this paper we describe T2's capabilities and demonstrate its effectiveness by an experimental evaluation against competing tools. T2 is implemented in F# and makes heavy use of the Z3 SMT solver [11]. T2 runs on Windows, MacOS, and Linux. It is available under the MIT license at `github.com/mmjb/T2`.

Related work. We focus on tool features of T2 and consider only related publicly released tools. Note that, with the exception of KITTeL [13], T2 is the only open-source termination prover and is the first open-source temporal property prover. Similar to T2, ARMC [23] and CProver [19], implement a TERMINATOR-style incremental reduction to safety proving. T2 is distinguished from these tools by its use of lexicographic ranking functions instead of disjunctive termination arguments [10]. Other termination proving tools include FuncTion [25], KITTeL [13], and Ultimate [16], which synthesize termination arguments, but have weak support for inferring supporting invariants in long programs with many loops. AProVE [14] is a closed-source portfolio solver implementing many successful techniques, including T2's methods. We know of only one other tool able to automatically prove CTL properties of infinite-state programs:[4] Q'ARMC [2],

[4] We do not discuss tools that only support finite-state systems or pushdown automata.

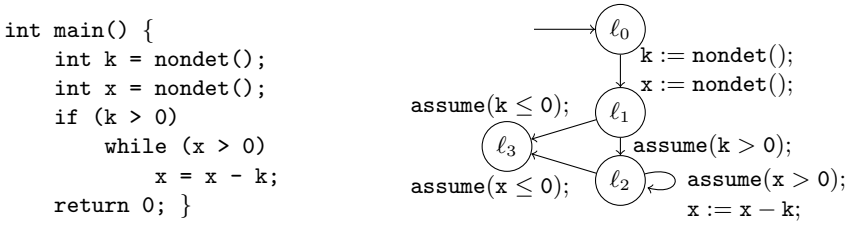


Fig. 1: **(a)** C input program. **(b)** T2 control-flow graph of the program in (a).

however Q'ARMC does not provide an automated front-end to its native input and requires a manual instantiation of the structure of the invariants. We do not know tools other than T2 that can verify Fair-CTL and CTL* for such programs.

Limitations. T2 only supports linear integer arithmetic fragments of C. An extension of T2 that handles heap program directly is presented in [1].[5] As in many other tools, numbers are treated as mathematical integers, not machine integers. However, our C front-end provides a transformation [12] that handles machine integers correctly by inserting explicit normalization steps at possible overflows.

2 Front-end

T2 improves on TERMINATOR by supporting a native input format as well as replacing the SLAM-based C interface by one based on LLVM.

Native Format. T2 allows input in its internal program representation to facilitate use from other tools. T2 represents programs as graphs of program locations $\mathcal{L}$ connected by transition rules with conditions and assignments to a set of integer variables $\mathcal{V}$. The location $\ell_0 \in \mathcal{L}$ is the canonical start state. An example is shown in Fig. 1(b). We assume that variables to which we do not assign values remain unchanged. For precise semantics of program evaluations, we refer to [3].

C via LLVM. In recent years, LLVM has become the standard basis of program analysis tools for C. We have thus chosen to extend llvm2kittel [13], which automatically translates C programs into integer term rewriting systems using LLVM, to also generate T2's native format. Our implementation uses the existing dead code elimination, constant propagation, and control-flow simplifications to simplify the input program. Fig. 1(a) shows the C program from which we generate the T2 native input in Fig. 1(b). Further details can be found in [4].

3 Back-end

In T2, we have replaced the safety, termination, and non-termination procedures implemented in TERMINATOR by more efficient versions. In addition, we added support for temporal-logic model checking.

Proving Safety. To prove temporal properties, T2 repeatedly calls to a safety proving procedure on instrumented programs. For this, T2 implements the Impact [21] safety proving algorithm, and furthermore can use safety proving techniquesimplemented in Z3, *e.g.* generalized property directed reachability

[5] Alternatively, the heap-to-integer abstractions implemented in Thor [20] for C or the one implemented in AProVE [14] for C and Java can be used as a pre-processing step.

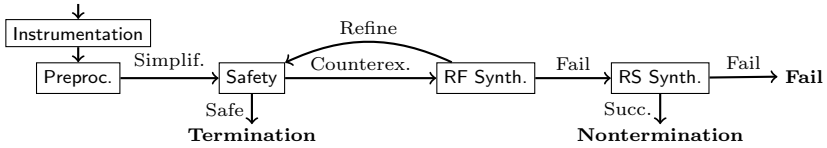


Fig. 2: Flowchart of the T2 termination proving procedure

(GPDR) [17] and Spacer [18]. For this, we convert our transition systems into sets of linear Horn clauses with constraints in linear arithmetic, in which one predicate p_ℓ is introduced per program location ℓ. For example, the transition from ℓ_2 to ℓ_2 in Fig. 1(b) is represented as $\forall \mathtt{x}, \mathtt{k}, \mathtt{x}' : \mathsf{p}_{\ell_2}(\mathtt{x}', \mathtt{k}) \leftarrow \mathsf{p}_{\ell_2}(\mathtt{x}, \mathtt{k}) \wedge \mathtt{x}' = \mathtt{x} - \mathtt{k} \wedge \mathtt{x} > 0$.

Proving Termination. A schematic overview of our termination proving procedure is displayed in Fig. 2. In the initial Instrumentation phase (described in [3]), the input program is modified so that a termination proof can be constructed by a sequence of alternating safety queries and rank function synthesis steps. This reduces the check of a speculated (possibly lexicographic) rank function f for a loop to asserting that the value of f after one loop iteration is smaller than before that iteration. If the speculated termination argument is insufficient, our Safety check fails, and the termination argument is refined using the found counterexample in RF Synth. We follow the strategy presented in [10] to construct a lexicographic termination argument, extending a standard linear rank function synthesis procedure [22],[6] implemented as constraint solving via Z3. The overall procedure is independent of the used safety prover and rank function synthesis.

In our Preprocessing phase, a number of standard program analysis techniques are used to simplify the remaining proof. Most prominently, this includes the termination proving pre-processing technique presented in [3] to remove loop transitions that we can directly prove terminating, without needing further supporting invariants. In our termination benchmarks, about 80% of program loops (*e.g.* encodings of `for i in 1 .. n do`-style loops) are eliminated at this stage.

Disproving Termination. When T2 cannot refine a termination argument based on a given counterexample, it tries to prove existence of a recurrent set [15] witnessing non-termination in the RS Synth. step. A recurrent set S is a set of program states whose execution can eventually lead back to a state from S. T2 uses a variation of the techniques from [5], restricted to only take a counterexample execution into account and implemented as constraint solving via Z3.

Proving CTL. CTL subsumes reasoning about safety, termination, and nontermination, in addition to all state-based properties. T2 implements the bottom-up strategy for CTL verification from [7]. Given a CTL property φ, T2 first computes quantifier-free preconditions $\mathsf{precond}_i$ for the subformulas of φ, and then verifies the formula obtained from φ by replacing the subformulas by their preconditions. Property preconditions are computed using a counterexample-guided strategy where several preconditions for each location are computed simultaneously through the natural decomposition of the counterexample's state space.

[6] T2 can optionally also synthesize disjunctive termination arguments [24] as implemented in the original TERMINATOR [9].

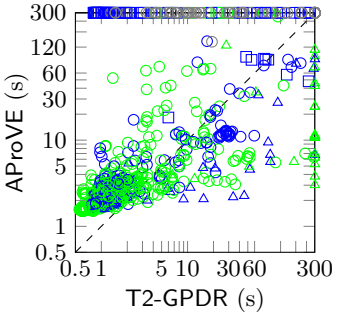
Proving Fair-CTL. T2 implements the approach for verification of CTL with fairness as presented in [6]. This method reduces Fair-CTL to fairness-free CTL using prophecy variables to encode a partition of fair from unfair paths. Although CTL can express a system's interaction with inputs and nondeterminism, which linear-time temporal logics (LTL) are inadequate to express, it cannot model trace-based assumptions about the environment in sequential and concurrent settings (e.g. schedulers) that LTL can express. Fairness allows us to bridge said gap between linear-time and branching-time reasoning, in addition to allowing us to employ the automata-theoretic technique for LTL verification [26] in T2.
Proving CTL.* Finally, T2 is the sole tool which supports the verification of CTL* properties of infinite-state programs as presented in [8]. A precondition synthesis strategy is used with a program transformation that trades nondeterminism in the transition relation for nondeterminism explicit in variables predicting future outcomes when necessary. Note that Fair-CTL disallows the arbitrary interplay between linear-time and branching-time operators beyond the scope of fairness. For example, a property stating that "along *some* future an event occurs *infinitely often*" cannot be expressed in either LTL, CTL nor Fair-CTL, yet it is crucial when expressing "possibility" properties, such as the viability of a system, stating that every reachable state can spawn a fair computation. Contrarily, CTL* is capable of expressing CTL, LTL, Fair-CTL, and the aforementioned property. Additionally, CTL* allows us to express existential system stabilization, stating that an event can eventually become true and stay true from every reachable state. Note that for properties expressible in Fair-CTL, our Fair-CTL prover is relatively (to safety and termination subprocedures) complete, whereas our CTL* prover is incomplete.

4 Experimental Evaluation & Future Work

We demonstrate T2's effectiveness compared to competing tools. We do not know of other tools supporting Fair-CTL and CTL* for infinite-state systems, thus we do not present such experiments and instead refer to [6] and [8]. Note that T2's performance has significantly improved since then through improvements in our back-end (e.g. by using Spacer instead of Impact). We refer to [4] for a detailed discussion of the properties and programs that these logics allowed us to verify.
Termination Experiments. We compare T2 as termination prover with the participants of the Termination Competition 2014 and 2015 using the collection of 1222 termination proving benchmarks used at the Termination Competition 2015 for integer transition systems. These benchmarks include manually crafted programs from the literature on termination proving, as well as many examples obtained from automatic translations from programs in higher languages such as Java (*e.g.* from `java.util.HashSet`) or C (*e.g.* reduced versions of Windows kernel drivers). The experiments were performed on the StarExec platform with a timeout of 300 seconds. Our version of T2 uses the GPDR implementation in Z3 as safety prover. Furthermore, we also consider three further versions of T2, using the three different supported safety provers. For these configurations, we use no termination proving pre-processing (NoP) step and only use our safety proving-based strategy, to better evaluate the effect of different safety back-ends. The

Tool	Term	Nonterm	Fail	Avg. (s)
AProVE	641	393	188	49.1
CppInv	566	374	282	65.5
Ctrl	445	0	777	80.0
T2-GPDR	627	442	153	23.6
T2-GPDR-NoP	589	438	195	31.4
T2-Spacer-NoP	591	429	202	33.5
T2-Impact-NoP	529	452	241	37.2

(a)

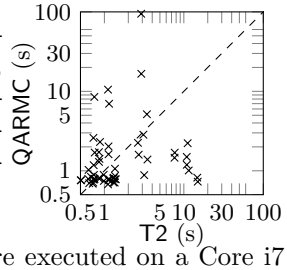


(b)

Fig. 3: Termination evaluation results. (a) Overview table. (b) Comparison of T2 and AProVE. Green (resp. blue) marks correspond to terminating (resp. non-terminating) examples, and gray marks examples on which both provers failed. A □ (resp. a △) indicates an example in which only T2 (resp. AProVE) succeeded, and ∘ indicates an example on which both provers return the same result.

overall number of solved instances and average runtimes are displayed in Fig. 3(a), and a detailed comparison of AProVE and T2-GPDR is shown in Fig. 3(b).[7] All provers are assumed to be sound, and no provers returned conflicting results.

The results show that T2's simple architecture competes well with the portfolio approach implemented in AProVE (which subsumes T2's techniques), and is more effective than other tools. Comparing the different safety proving back-ends of T2 shows that our F# implementation of Impact is nearly as efficient as the optimized C++ implementations of GPDR and Spacer. The different exploration strategies of our safety provers yield different counterexamples, leading to differences in the resulting (non)termination proofs. The impact of our pre-processing technique is visible when comparing T2-GPDR and T2-GPDR-NoP.



CTL Experiments. We evaluate T2's CTL verification techniques against the only other available tool, Q'ARMC [2] on the 56 benchmarks from its evaluation. These benchmarks are drawn from the I/O subsystem of the Windows OS kernel, the back-end infrastructure of the PostgreSQL database server, and the SoftUpdates patch system. They can be found at `http://www.cims.nyu.edu/~ejk/ctl/`. The tools were executed on a Core i7 950 CPU with a timeout of 100 seconds. Both tools are able to successfully verify all examples. T2 needs 2.7 seconds on average, whereas Q'ARMC takes 3.6 seconds. The scatterplot above compares proof times on individual examples.

Future work. We wish to integrate and improve techniques for conditional termination, which will improve the strength of our property verification. We also intend to support reasoning about the heap, recursion, and concurrency in T2.

[7] All experimental data can be viewed on `https://www.starexec.org/starexec/secure/details/job.jsp?id=11121`.

References

1. A. Albarghouthi, J. Berdine, B. Cook, and Z. Kincaid. Spatial interpolants. In *ESOP'15*.
2. T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In *CAV'13*.
3. M. Brockschmidt, B. Cook, and C. Fuhs. Better termination proving through cooperation. In *CAV'13*.
4. M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, and N. Piterman. T2: Temporal property verification. 2015. `http://arxiv.org/abs/1512.08689`.
5. M. Brockschmidt, T. Ströder, C. Otto, and Jürgen Giesl. Automated detection of non-termination and `NullPointerExceptions` for Java Bytecode. In *FOVEOOS'11*.
6. B. Cook, H. Khlaaf, and N. Piterman. Fairness for infinite-state systems. In *TACAS'15*.
7. B. Cook, H. Khlaaf, and N. Piterman. Faster temporal reasoning for infinite-state programs. In *FMCAD'14*.
8. B. Cook, H. Khlaaf, and N. Piterman. On automation of CTL* verification for infinite-state systems. In *CAV'15*.
9. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI'06*.
10. B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In *TACAS'13*.
11. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*.
12. S. Falke, D. Kapur, and C. Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *VSTTE'12*.
13. S. Falke, D. Kapur, and C. Sinz. Termination analysis of C programs using compiler intermediate languages. In *RTA'11*.
14. J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with AProVE. In *IJCAR'14*.
15. A. Gupta, T. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu. Proving non-termination. In *POPL'08*.
16. M. Heizmann, J. Hoenicke, and A. Podelski. Termination analysis by learning terminating programs. In *CAV'14*.
17. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT'12*.
18. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. In *CAV'14*.
19. D. Kroening, N. Sharygina, A. Tsitovich, and C. Wintersteiger. Termination analysis with compositional transition invariants. In *CAV'10*.
20. S. Magill, M. Tsai, P. Lee, and Y. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL'10*.
21. K. McMillan. Lazy abstraction with interpolants. In *CAV'06*.
22. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI'04*.
23. A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *PADL'07*.
24. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS'04*.
25. C. Urban. The abstract domain of segmented ranking functions. In *SAS'13*.
26. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.