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 [8]. 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 [25] 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 [10]. 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 [12], T2 is the only open-source termination prover and is the first open-source temporal property prover. Similar to T2, ARMC [22] and CProver [18], 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 [9]. Other termination proving tools include FuncTion [24], KITTeL [12], and Ultimate [15], which synthesize termination arguments, but have weak support for inferring supporting invariants in long programs with many loops. AProVE [13] 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], however Q'ARMC does

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

```
int main() {
    int k = nondet();
    int x = nondet();
    if (k > 0)
        while (x > 0)
            x = x - k;
    return 0; }
```

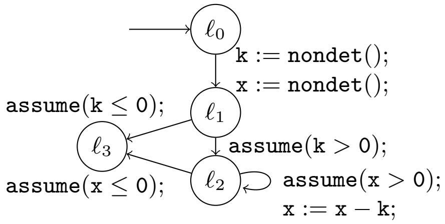


Fig. 1: **(a)** C input program. **(b)** T2 control-flow graph of the program in (a).

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 [11] 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 [12], 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 the Appendix.

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 [20] 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 [19] for C or the one implemented in AProVE [13] for C and Java can be used as a pre-processing step.

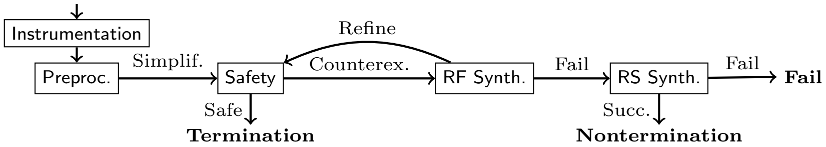


Fig. 2: Flowchart of the T2 termination proving procedure

(GPDR) [16] and Spacer [17]. 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 [9] to construct a lexicographic termination argument, extending a standard linear rank function synthesis procedure [21],[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 [14] 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 [4], 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 [6]. 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 [23] as implemented in the original TERMINATOR [8].

Proving Fair-CTL. T2 implements the approach for verification of CTL with fairness as presented in [5]. 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 [25] in T2.

Proving CTL.* Finally, T2 is the sole tool which supports the verification of CTL* properties of infinite-state programs as presented in [7]. 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.

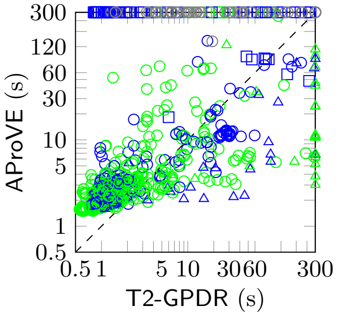
4 Experimental Evaluation & Future Work

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 [5] and [7]. 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 to the Appendix 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

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)

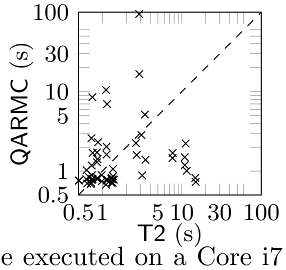


(b)

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.

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

Appendix.

This appendix contains several examples on how to use T2. For the ease of this demonstration, we include easy to follow programs alongside corresponding simple properties. Additional examples of T2 operating on realistic programs with expressive properties are available in the papers relating to the respective technical results [9, 6, 5]. Installation instructions for T2, additional runtime options, and an overview of the program source code can be found alongside its source in `https://github.com/mmjb/T2/blob/master/README.txt`.

A Front-end Pre-processing via LLVM

```
define i32 @main() #0 {
main_bb0:
  %"0" = call i32 (...)* @nondet()
  %"1" = call i32 (...)* @nondet()
  %"2" = icmp sgt i32 %"0", 0
  br i1 %"2", label %main_bb1,
      label %main_bb3

main_bb1:
  %x = phi i32 [ %"4", %main_bb2],
      [ %"1", %main_bb0 ]
  %"3" = icmp sgt i32 %x, 0
  br i1 %"3", label %main_bb2,
      label %main_bb3

main_bb2:
  %"4" = sub nsw i32 %x, %"0"
  br label %main_bb1

main_bb3:
  ret i32 0
}
```

(a)

```
START: main_bb0;

FROM: main_bb0;
  v0 := nondet();
  v1 := nondet();
  x := v1;
TO: main_bb0_end;

FROM: main_bb0_end;
  assume(v0 > 0);
TO: main_bb1;

FROM: main_bb0_end;
  assume(v0 <= 0);
TO: main_bb3;
```

```
FROM: main_bb1;
  assume(x > 0);
TO: main_bb2;

FROM: main_bb1;
  assume(x <= 0);
TO: main_bb3;

FROM: main_bb2;
  v4 := x - v0;
  x := v4;
TO: main_bb1;

FROM: main_bb3;
TO: main_bb3;
```

(b)

Fig. 4: **(a)** Compiled LLMV-IR post llvm2kittel optimizations corresponding to Fig. 1(a). **(b)** T2 input file corresponding to Fig. 1(b), generated from(a).

Our LLVM front-end builds upon and extends llvm2kittel [12]. Our version of llvm2kittel tailored for T2 can be found at `https://github.com/hkhlaaf/llvm2kittel`. llvm2kittel provides multiple optimizations that are helpful for our transformation into the native T2 file format, as it performs function inlining, dead code elimination, constant propagation, and control-flow simplification. Below we provide a very basic notion of how the LLVM intermediate representation (LLVM-IR) corresponds to the T2 format.

The LLVM-IR generated by `clang` for our example C program from Fig. 1(a) is shown in Fig. 4(a). The T2 input file generated from this by our llvm2kittel front-end is displayed in Fig. 4(b). In our translation, basic blocks in the LLVM-IR (main_bb0, main_bb1, . . .) are translated as transition rules labeled with corresponding arithmetic instructions. These instructions are trivially obtained from the LLVM-IR, but all heap memory reads are implemented as **nondet()**, and heap writes are dropped.

A basic block's entry point is represented by a location of the same name, i.e., a transition to the location main_bb2 corresponds to entering the basic block main_bb2. The targets of the generated transitions are extracted from the `br` ("branch") instructions. Sequences of `phi` instructions at the beginning of a basic block *b*, which are needed for LLVM-IR's single static assignment syntax, are encoded on the transitions leading to *b*. For example, in Fig. 4(a), the basic block main_bb0 contains a sequence of instructions before a `br` instruction determines whether to branch to main_bb1 or main_bb3, depending on the value of %0. This is reflected in Fig. 4(b) in the first column, where the comparison of the value %0 (v0 in the T2 file), is done from the main_bb0_end node. If `v0 > 0` we transition to the main_bb1 node, otherwise we transition to the main_bb3 node.

Using our version of llvm2kittel as a front-end, we now show how it can be used to generate native T2 files from C programs. Assume that the C program from Fig. 1(a) is stored as `ex0.c`. We generate a T2 native input file as follows:

```
$ clang -Wall -Wextra -c -emit-llvm -O0 ex0.c -o ex0.bc
$ ./llvm2kittel --eager-inline --t2 ex0.bc > ex0.t2
```

B T2 as Termination Prover

B.1 Native Input

We first demonstrate using T2 to prove termination of the example from Fig. 1, whose textual representation is displayed in Fig. 4 Assume that the example is saved as file `ex0.t2`. Then, the most simple T2 call looks like this:

```
$ ./T2 -termination -input_t2 ex0.t2
Termination proof succeeded
```

To obtain more information about the termination argument, T2 provides the `-print_proof` option:

```
$ ./T2 -termination -input_t2 ex0.t2 -print_proof
Termination proof succeeded
Used the following cutpoint-specific lexicographic rank functions:
 * For cutpoint 7, used the following rank functions/bounds (in descending priority order):
    - RF x, bound 1
```

We see that the proof was done using a (one-element) lexicographic rank function. However, this output is hard to connect to the input program, which had no location 7.[8] To understand the connection better, T2 allows to output all intermediate program representations as DOT graphs:

```
$ ./T2 -termination -input_t2 ex0.t2 -dottify_input_pgms
Created input.dot
Created input__instrumented.dot
Created input__instrumented_cleaned.dot
Created input__instrumented_lex_RF.dot
Termination proof succeeded
```

In general, input.dot corresponds to the parsed program (with renamed locations and numbered transitions), input__instrumented.dot shows it after instrumentation for a termination proof, and input__instrumented_cleaned.dot is the program after the initial Preprocessing step (cf. Fig. 2). A rendering of the input__instrumented.dot file is shown in Fig. 5. Location are circular nodes in the graph, and the labels "loc_i" indicate which node corresponds to location i in the input program.

B.2 Java Input

Using AProVE [13] as frontend, T2 can be used to prove termination of Java programs. As an example, consider the small Java program Ex1 in Fig. 6. As AProVE only supports reading JAR files (i.e., compiled Java code), we will assume that the example was compiled to Ex1.jar, and that AProVE is available as aprove.jar.[9] We can then use AProVE to obtain a T2 file, which we then prove terminating:

```
$ java -cp aprove.jar aprove.CommandLineInterface.JBCFrontendMain --t2 yes Ex1.jar
Dumped to ./Ex1.jar-obl-8.t2
$ ./T2 -termination -input_t2 Ex1.jar-obl-8.t2
Termination proof succeeded
```

We note that AProVE cannot prove this example terminating on its own, as it cannot infer the needed invariant $n < m$. AProVE also supports heap-manipulating programs, and can translate these into integer transition systems, which can then be handled by T2. As example, consider the example program Ex2 in Fig. 6, in which a list is first constructed and its length is subsequently computed. We can prove termination of it as follows:

```
$ java -cp aprove.jar aprove.CommandLineInterface.JBCFrontendMain --t2 yes Ex2.jar
Dumped to ./Ex2.jar-obl-9.t2
$ ./T2 -termination -input_t2 Ex2.jar-obl-9.t2
Termination proof succeeded
```

[8] The reason for the location number is that T2 stores locations as integers, but also allows strings to identify locations in the input (e.g. "START: start;"), and thus renumbers all locations on parsing the input file.

[9] This is downloadable from http://aprove.informatik.rwth-aachen.de/.

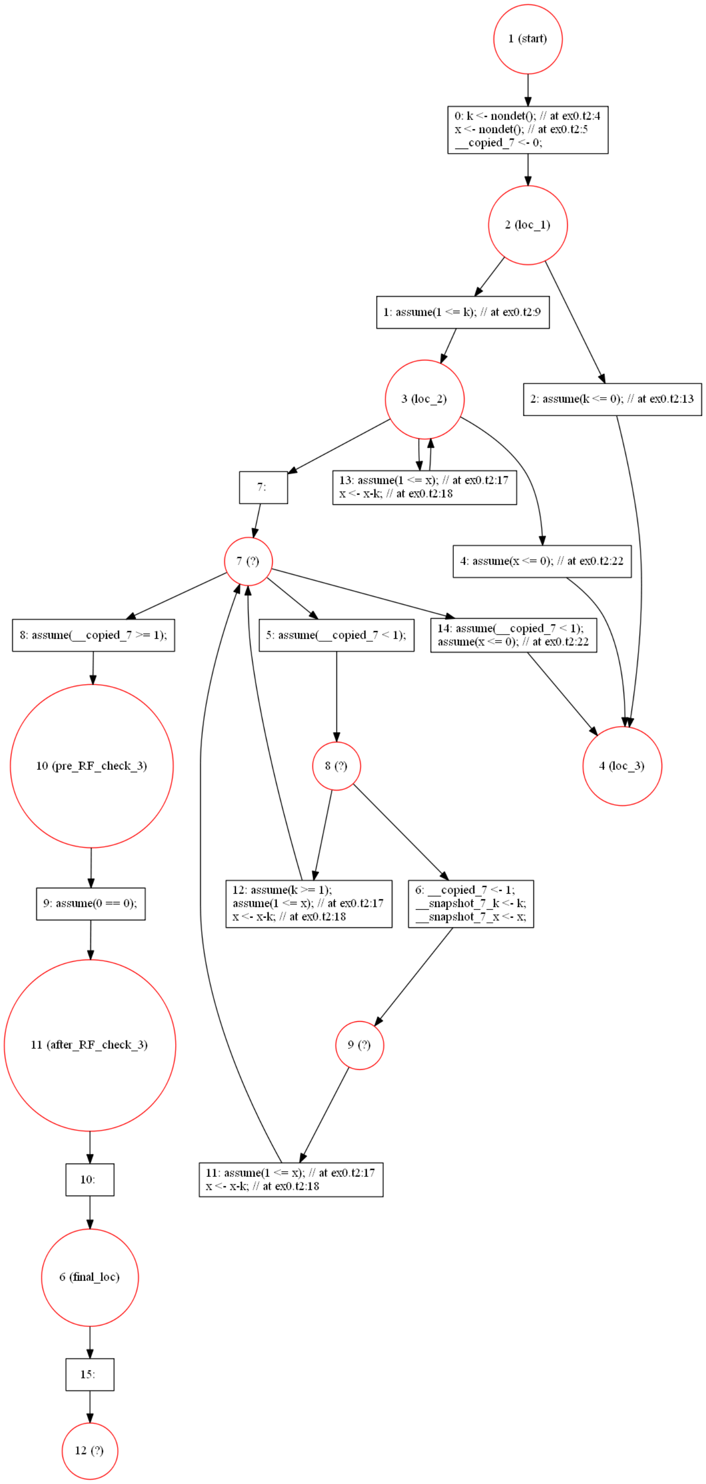

1 (start)
0: k <- nondet(); // at ex0.t2:4
x <- nondet(); // at ex0.t2:5
__copied_7 <- 0;
2 (loc_1)
1: assume(1 <= k); // at ex0.t2:9
2: assume(k <= 0); // at ex0.t2:13
3 (loc_2)
7:
13: assume(1 <= x); // at ex0.t2:17
x <- x-k; // at ex0.t2:18
4: assume(x <= 0); // at ex0.t2:22
7 (?)
8: assume(__copied_7 >= 1);
5: assume(__copied_7 < 1);
14: assume(__copied_7 < 1);
assume(x <= 0); // at ex0.t2:22
10 (pre_RF_check_3)
8 (?)
4 (loc_3)
9: assume(0 == 0);
12: assume(k >= 1);
assume(1 <= x); // at ex0.t2:17
x <- x-k; // at ex0.t2:18
6: __copied_7 <- 1;
__snapshot_7_k <- k;
__snapshot_7_x <- x;
11 (after_RF_check_3)
9 (?)
10:
11: assume(1 <= x); // at ex0.t2:17
x <- x-k; // at ex0.t2:18
6 (final_loc)
15:
12 (?)

Fig. 5: CFG for Fig. 1 after instrumentation for termination

```
public class Ex1 {
  public static void
   main(String... args) {
    int n = args.length;
    int m = 2 * (n + 1);
    while (n > 0) {
      if (m <= 0) {
        n++; m++;
      } else {
        n--; m--; }}}}
```

```
public class Ex2 {
  private Ex2 next = null;
  public Ex2(Ex2 n) { next = n; }
  public static void
   main(String... args) {
    int n = args.length;
    Ex2 list = null;
    while (--n > 0)
      list = new Ex2(list);
    int length = 0;
    while (list != null) {
      length++;
      list = list.next; }}}
```

Fig. 6: Two Java example programs

C Temporal Property Verification

C.1 T2 as a CTL Prover

In this section, we demonstrate how one can verify C programs using the CTL option in T2. In the following demonstration, we will show how we can verify the property EFAG $\mathtt{x} \leq 0$. As demonstrated above, we use llvm2kittel to generate a T2 input file for the program from Fig. 1(a), stored as `ex0.t2`. Note that the LLVM compilation process may slightly modify program variable names. Thus, the variables used to specify the CTL property must be changed accordingly as well. We now run T2 as follows:

```
$ ./T2 -input_t2 ctl-ex.t2 -CTL "[EF]([AG](x <= 0))"
T2 program prover/analysis tool.
Temporal proof succeeded
```

One can additionally specify the `-print_proof` option, which outputs the location-specific preconditions generated for each sub-formula. The precondition is a tuple with the first argument being a program location, and the second being the precondition. That is, a precondition a_φ for a CTL sub-formula φ takes the form $\bigwedge_i(\mathsf{pc} = i \Rightarrow a_{\mathsf{pc}_i})$ where i denotes elements of the program locations.

C.2 T2 as a Fair-CTL and CTL* Prover

Below we show properties which can be expressed in Fair-CTL and CTL*, but not CTL nor LTL. We write these properties in CTL*, a superset of CTL and LTL.

Properties expressible in Fair-CTL. For brevity, when expressing Fair-CTL properties we write Ω for $\mathsf{GF}p \rightarrow \mathsf{GF}q$. A state property is indicated by φ and p and q are subsets of program states, constituting our fairness requirement (infinitely often p implies infinitely often q).

The property $\mathsf{E}[\Omega \wedge \mathsf{G}\varphi]$ generalizes fair non-termination, that is, there exists an infinite fair computation all of whose states satisfy the property φ. The property $\mathsf{A}\big[\Omega \to \mathsf{G}[\varphi_1 \to \mathsf{A}(\Omega \to \mathsf{F}\varphi_2)]\big]$ indicates that on every fair path, every φ_1 state is later followed by a φ_2 state. In [5], we verify said property for a Windows device driver, indicating that a lock will always eventually be released in the case that a call to a lock occurs, provided that whenever we continue to call a Windows API repeatedly, it will eventually return a desired value (fairness). Similarly, $\mathsf{A}\big[\Omega \to \mathsf{G}[\varphi_1 \to \mathsf{A}(\Omega \to \mathsf{FE}(\Omega \wedge \mathsf{G}\varphi_2))]\big]$ dictates that on every fair path whenever a φ_1 state is reached, on all possible futures there is a state which is a possible fair future and φ_2 is always satisfied. For example, one may wish to verify that there will be a possible active fair continuation of a server, and that it will continue to effectively serve if sockets are successfully opened. Below we demonstrate how we can verify our Bakery algorithm benchmark from [5] with a CTL property and a fairness constraint Ω for $\mathsf{GF}p \to \mathsf{GF}q$:

```
$ ./T2 -input_t2 test/bakery.t2
         -CTL "[AG](NONCRITICAL <= 0 || ([AF](CRITICAL > 0)))"
         -fairness "(P == 1, Q == 1)"
T2 program prover/analysis tool.
Temporal proof succeeded
```

Properties expressible in CTL^*. Below are properties that can only be afforded by the extra expressive power of CTL^*, which subsumes $\mathsf{Fair\text{-}CTL}$. These liveness properties are utilized in [7] to verify systems such as Windows kernel APIs that acquire resources and APIs that release resources.

The property $\mathsf{EFG}(\neg x \wedge (\mathsf{EGF}\ x))$ conveys the divergence of paths. That is, there is a path in which a system stabilizes to $\neg x$, but every point on said path has a diverging path in which x holds infinitely often. This property is not expressible in CTL or in LTL, yet is crucial when expressing the existence of fair paths spawning from every reachable state in a system. In CTL, one can only examine sets of states, disallowing us to convey properties regarding paths. In LTL, one cannot approximate a solution by trying to *disprove* either $\mathsf{FG}\ \neg x$ or $\mathsf{GF}\ x$, as one cannot characterize these proofs within a path quantifier.

Another CTL^* property $\mathsf{AG}\big[(\mathsf{EG}\ \neg x) \vee (\mathsf{EFG}\ y)\big]$ dictates that from every state of a program, there exists either a computation in which x never holds or a computation in which y eventually always holds. The linear time property $\mathsf{G}(\mathsf{F}x \to \mathsf{FG}\ y)$ is significantly stricter as it requires that on every computation either the first disjunct or the second disjunct hold. Finally, the property $\mathsf{EFG}\big[(x \vee (\mathsf{AF}\ \neg y))\big]$ asserts that there exists a computation in which whenever x does not hold, all possible futures of a system lead to the falsification of y. This assertion is impossible to express in LTL. Below we demonstrate how we can verify one of our benchmarks from [7]:

```
$ ./T2 -input_t2 1394-succeed-2.t2
         -ctlstar "E F(G (((keA <= 0) || (E F (keR == 1)))))
T2 program prover/analysis tool.
Temporal proof succeeded
```

D T2 options

T2 provides a `--help` command line switch. However, the following switches are noteworthy:

- `--log` turns on live logging, so that T2 reports every attempted proof step in detail (e.g., expansion of leaves in the Impact safety procedure, found counterexamples, program refinements, ...).
- `--safety_implementation` allows to pick the used back-end safety solver. Currently, this supports the internal `impact`, and Z3's `spacer` (the default) and `pdr`. There is also an experimental mode that runs `spacer` and `bmc` (bounded model checking) in parallel.
- `--lexicographic off` forces T2 to use the original TERMINATOR method based on disjunctively well-founded transition invariants.
- `--try_nonterm false` turns off the non-termination prover, useful for when such proofs would be unsound due to over-approximating pre-processing.