Proving Termination by k-Induction

Jianhui Chen
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
chenjian16@mails.tsinghua.edu.cn

Fei He*
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
hefei@tsinghua.edu.cn

ABSTRACT

We propose a novel approach to proving the termination of imperative programs by *k-induction*. By our approach, the termination proving problem can be formalized as a k-inductive invariant synthesis task. On the one hand, k-induction uses weaker invariants than that required by the standard inductive approach. On the other hand, the *base case* of k-induction, which unrolls the program, can provide stronger pre-condition for invariant synthesis. As a result, the termination arguments of our approach can be synthesized more efficiently than the standard method. We implement a prototype of our k-inductive approach. The experimental results show the significant effectiveness and efficiency of our approach.

CCS CONCEPTS

• **Theory of computation → Logic and verification**; • **Software and its engineering → Formal software verification**.

KEYWORDS

Proving Termination, k-Induction, Invariant Synthesis

ACM Reference Format:
Jianhui Chen and Fei He. 2020. Proving Termination by k-Induction. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3324884.3418929

1 INTRODUCTION

Termination is a very important liveness property for software verification. Most of the techniques for proving termination of imperative programs are based on the notion of *ranking function* [1, 2, 5, 16–18]. These methods try to find an expression whose value decreases strictly on each loop iteration. Its value is also bounded by an invariant. As a result, the value of the expression cannot decrease infinitely, and thus the number of the loop iterations must be finite. We name the expression the *explicit ranking function* (*eRF*). The invariant is called the *support invariant*. Sometimes, such an expression can hardly be synthesized by the template-based methods [2, 4, 17], e.g., linear or polynomial templates. For example, the program in Figure 1 is terminating since the integer-valued variable x eventually becomes 0. One of the *eRF*s for this program is $|x|$. But it can not be synthesized by either linear or polynomial templates.

Actually, we can employ another method to prove the termination of this program. This method is based on the so-called *implicit ranking function* (*iRF*) [9]. It does not require an explicit expression but just an invariant as the termination argument. To achieve this, a counter variable i is inserted into the loop body of the program. The counter variable i decreases by 1 after each loop iteration, e.g., line 6 of Figure 1. Suppose the program is terminating, the number of the loop iterations should be finite. The maximal number of the loop iteration is denoted as $m(X)$. We assume that i is greater than $m(X)$ initially. As a result, i should always be greater than 0 in every loop iteration. Once we find an invariant to guarantee $i > 0$ with in the loop iteration, the termination of the program is proved. That because if the loop is infinite, i should become less than 0 eventually, and thus the contradiction arose. This proving procedure can be formalized as a *standard inductive* verification condition presented as follows. We call it the *termination certificate* (*TC*).

$$Pre(X) \wedge i > m(X) \Rightarrow Inv(X, i) \quad (1)$$

$$Inv(X, i) \wedge G(X) \wedge T(X, X') \wedge i' = i - 1 \Rightarrow Inv(X', i') \quad (2)$$

$$Inv(X, i) \wedge G(X) \Rightarrow i > 0 \quad (3)$$

In the above formulas, $Pre(X)$ represents the precondition of the loop. $G(X)$ is the loop guard and $T(X, X')$ represents the loop body. If we find an invariant $Inv(X, i)$ making the *TC* of a program valid, the termination of the program is proved. For example, Figure 2 is an instantiation of the *TC* of the program in Figure 1. The precondition $Pre(X)$ is $\top$. The $m(X)$ is approximated by x and $-x$. We can find that $Inv(x, i) \triangleq i \geq x \wedge i \geq -x$ is a suitable invariant which makes the *TC* valid. As a result, we can prove the termination of this program by synthesizing such an invariant based on linear templates.

```
1 while (x ≠ 0) {
2   if (x > 0)
3     x = x - 1;
4   else
5     x = x + 1;
6 //i = i - 1;
7 }
```

Figure 1: A progarm

$$\top \wedge i > x \wedge i > -x \Rightarrow Inv(x, i) \quad (4)$$

$$Inv(x, i) \wedge x \neq 0 \wedge x' = ite(x > 0, x-1, x + 1) \wedge i' = i - 1 \Rightarrow Inv(x', i') \quad (5)$$

$$Inv(x, i) \wedge x \neq 0 \Rightarrow i > 0 \quad (6)$$

Figure 2: Termination certificate

*Fei He is the corresponding author.

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3418929

```
1 assume (i ≥ 0);
2 assume (i > -b);
3 while (a > 0) {
4   a = a - b;
5   b = a + 2b;
6   i = i - 1;
7 }
```

Figure 3: Example A

```
1 assume (i ≥ 0);
2 assume (i > -b);
3 while (a > 0) {
4   a = a - b;
5   b = a + 2b;
6   assume (a > 0);
7   a = a - b;
8   b = a + 2b;
9   i = i - 1; }
```

Figure 4: Transformed A

```
1 assume (i > x);
2 assume (y ≤ 0);
3 while (x ≥ 0) {
4   x = x + y;
5   y = y - 1;
6   i = i - 1;
7 }
```

Figure 5: Example B

```
1 assume (i > x);
2 assume (y ≤ 0);
3 assume (x ≥ 0);
4 x = x + y;
5 y = y - 1;
6 while (x ≥ 0) {
7   x = x + y;
8   y = y - 1;
9   i = i - 1; }
```

Figure 6: Transformed B

However, the *iRF*-based method heavily relies on invariant synthesis. Sometimes, a strong invariant is required to ensure the validity of the *TC*, but we can hardly synthesize it. k-Induction is an efficient approach to proving inductive properties, which is widely studied in formal verification of reachability problems [3, 6, 7, 10]. We employ k-induction to strengthen the *iRF*-based method. The strengthening has several advantages. First, the *step case* of k-induction uses significantly weaker invariants than that required by the standard inductive approach. Moreover, the *base case* of k-induction, which unrolls the program k times, provides a stronger precondition for invariant synthesis. As a result, compared to the standard inductive method, the invariant can be synthesized more effectively and efficiently if k-induction is integrated. We show these by some examples in Section 2.1.

In this paper, we propose the k-inductive termination certificate to prove the termination of programs and present the k-inductive algorithm of our approach. We implement our approach on top of the tool FreqTerm [9]. It is a powerful tool that uses the standard inductive *iRF*-based method to prove termination. We take it as the baseline and evaluate our approach by the benchmarks mainly from [9]. The experimental results show the significant effectiveness and efficiency of our approach.

2 OUR APPROACH

2.1 Motivation Example

Comparing with the standard inductive approach, the k-inductive approach uses significantly weaker invariants. Moreover, the precondition for synthesizing invariants is stronger than the standard inductive approach. Hence, we can synthesize invariant effectively and efficiently. We show these by some simple examples (Figure 3-6). In these examples, the green codes represent the approximation of $m(X)$ and the blue codes is generated by k-induction unrolling.

Consider the program in Figure 3. It is terminating, although not so obvious. Actually, we can employ the standard inductive approach to prove termination of this program, but a very strong invariant is required, i.e., $(i \geq 0 \vee a \leq 0) \wedge (i > 0 \vee a \leq b) \wedge i > -b$. Synthesizing such a complex invariant is rather difficult. However, if k-induction is used and the loop is unrolled one more time in the loop body, i.e., Figure 4, we can prove termination of this program by a significantly weaker invariant, i.e., $(i \geq 0 \vee a \leq 0) \wedge i > -b$. It can be synthesized more easily. Furthermore, if we unroll the loop three more times, we can even prove termination of this program by a very weak invariant $i \geq 0 \vee a \leq 0$. As we can see, integrating the *iRF*-based method with k-induction can significantly weaken the invariants that it requires, and the invariant synthesis will be more efficient.

Another example program is shown in Figure 5. It is also a terminating program since x eventually decreases to less than 0. However, we can hardly prove termination of this program by the standard inductive approach. That is because the invariants implied by the precondition $y \leq 0 \wedge i > m(x, y)$ are too weak, e.g., $y \leq 0$. Such an invariant cannot guarantee the value of x decreases strictly. However, if we use k-induction and unroll the loop once in front of the loop head, i.e., Figure 6, the precondition becomes $y \leq 0 \wedge x \geq 0 \wedge x' = x + y \wedge y' = y - 1 \wedge i > m(x', y')$. It is strengthened and implies a stronger invariant $y' \leq -1$, which ensures the value of x decreases strictly in the loop body. Then we can find that $Inv(i, x, y) \triangleq y \leq -1 \wedge i \geq x$ is an appropriate invariant to make the *TC* valid. So, k-induction can also strengthen the precondition for synthesizing invariants, and thus make the invariant synthesis of the *iRF*-based method more effective.

2.2 K-Inductive Termination Certificate

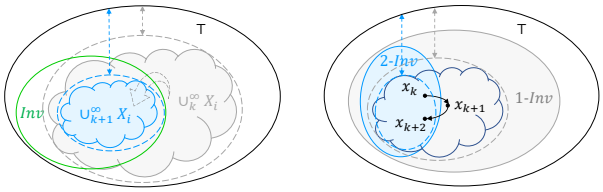
In this section, we present our k-inductive approach formally. We first introduce the k-inductive verification condition of termination as follows. We call it the *k-inductive termination certificate* (k-*TC*). It has a similar form as the standard *TC*.

$$Pre(X_0, X_1) \wedge \bigwedge_{i=1}^{k-1} (G(X_i) \wedge T(X_i, X_{i+1})) \wedge i_k > m(X_k) \Rightarrow \overline{Inv}(X_k, i_k) \quad (7)$$

$$\overline{Inv}(X_0, i_0) \wedge \bigwedge_{i=1}^{k} (G(X_{i-1}) \wedge T(X_{i-1}, X_i)) \wedge i_k = i_0 - 1 \Rightarrow \overline{Inv}(X_k, i_k) \quad (8)$$

$$\overline{Inv}(X, i) \wedge G(X) \Rightarrow i > 0 \quad (9)$$

The formula (7) is the *base case* of the k-*TC*. It means the invariant should cover (be implied by) the program state space after $k-1$ times of loop iterations. The base case of the k-*TC* is a little different from the standard k-inductive principle for reachability problems. It just requires the invariant to be satisfied on the k-th step instead of all of the first k steps. Comparing to formula (4) of standard *TC*, the precondition of formula (7) is strengthened by the k times unrolling of the loop. Figure 7(a) shows the effect of the strengthening. The black ellipse represents the universal set, denoted as $\top$. The gray area represents the program state space after k times loop iterations, i.e., denoted as $\cup_k^\infty X_i$. The blue area represents the program state space after $k+1$ times loop iterations, i.e., denoted as $\cup_{k+1}^\infty X_i$. It is



(a) base case strengthening (b) step case strengthening

Figure 7: Strengthening of k-induction

clear that $\cup_{k+1}^{\infty} X_i \subseteq \cup_{k}^{\infty} X_i$. As a result, the more times of unrolling, the smaller the state space is, and the more easily we can synthesize an invariant $\overline{Inv}$ to cover the state space.

The formula (8) is the *step case* of the k-*TC*. It means that from any state in the invariant, after going through k times of loop iterations, the reached state should also be included in the invariant. The step case of the k-*TC* is also different from that of the standard k-inductive principle. It does not need to assume the invariant to be satisfied on every unrolling step. That is because proving termination property does not need a globally satisfied invariant. Precisely speaking, we just need an infinite often satisfied predicate, which is denoted as $\overline{Inv}$. We show this in our proof of Theorem 2.1. Comparing to formula (5), the loop is unrolled k times in formula (8). It is clear that a 1-inductive invariant is also k-inductive, but the opposite does not hold. As shown in Figure 7(b), the gray ellipse is the 1-invariant, it should cover the whole program state space. But we can synthesize a 2-invariant (the blue ellipse) on the subset of the program state space. As a result, with the step case unrolling, we can use rather weak invariant to make the k-*TC* valid.

We have the following theorem to ensure the correctness of our k-inductive approach.

Theorem 2.1 (Soundness). *An imperative program P is terminating if it has a valid k-inductive termination certificate.*

Proof. We prove its contrapositive proposition, i.e., if the program P is not terminating, it has no valid k-*TC*. We use proof by contradiction. First, we insert the counter variable i into P and assume P has a valid k-*TC*. Because P is not terminating, there exists an infinite execution $\sigma = s_0, s_1, s_2, \cdots$ such that $(s_0, s_1) \models Pre(X_0, X_1)$ and $(s_j, s_{j+1}) \models G(X_j) \wedge T(X_j, X_{j+1})$ for $j \geq 1$. It is clear that the states in σ satisfy formulas (7) and (8). So, for every $j \geq k \wedge j \bmod k \equiv 0$, the state $s_j \models \overline{Inv}(X, i)$. However, the value of i is finite initially, and it decreases by 1 after every k steps. Hence, i eventually becomes negative in the infinite execution σ. As a result, formula (9) is invalid and a contradiction arises. □

Note that the standard *TC* is a special case of the k-*TC*, i.e., let k be 1. Moreover, if there is an invariant for the standard *TC*, it is also an invariant for the k-*TC* ($k > 1$). That is because a 1-inductive invariant is also k-inductive for $k > 1$. As a result, the k-inductive approach is at least as complete as the standard inductive approach. Furthermore, we have shown that there exist some examples that can only be solved by our k-inductive approach. In practice, our k-inductive approach should be more powerful than the standard inductive approach.

Algorithm 1: k-Inductive Algorithm.

input : A program P
output: {P *is terminating, unknown*}

1 **for** $k \leftarrow 1, 2, 3, \ldots$ **do**
2 $P_k \leftarrow transform(P, k)$
3 $TC_k \leftarrow boundApproxi(P_k)$ ▷ TC_k has an unkonwn $\overline{Inv}$
4 $\overline{Inv} \leftarrow synthesizeInv(TC_k)$
5 **if** $\overline{Inv} \neq \emptyset$ **then**
6 **return** P *is terminating*
7 **return** *unknown*

2.3 Algorithm

The algorithm of our approach is presented in Algorithm 1. In this algorithm, we first transform the input program P by unrolling the loop k times, and insert the statement of decreasing the counter variable, i.e., `i = i - 1`, at the end of the loop body (line 2). Then, we try to find out an approximation of the maximal times of loop iterations, i.e., $m(X)$ (line 3). The approximation has the form of $i > expr_1 \wedge i > expr_2 \cdots$, where $expr_i$ can be arbitrary expressions. The syntax guided synthesis (SyGuS) method [8] can be used to help us find out these expressions. Besides, we can also use the template-based method to instantiate the approximation, and synthesize it and the invariant simultaneously. After that, the k-inductive termination certificate TC_k is generated from the transformed program P_k and the approximation. Next, we try to synthesize an invariant to ensure the validity of TC_k (line 4). Since the k-*TC* can be viewed as a set of *constrained Horn clause*s (CHCs). A Horn solver can be used to solve it and get the invariant [13, 14]. Moreover, the template-based method [4] can also be employed to synthesis the invariant. The program is terminating if we find the invariant within the time limit of the synthesis procedure. Otherwise, we increase k by 1 and repeat the above procedure until the maximal bound or the timeout is reached.

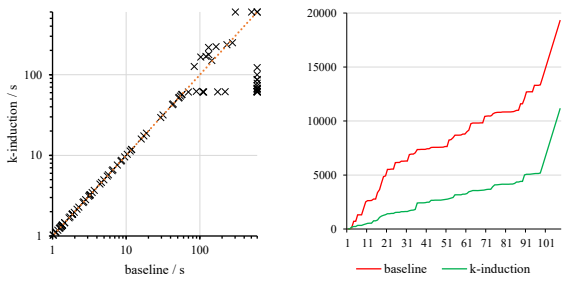
We employ the program in Figure 5 as an example to illustrate our algorithm. The input P is the program in Figure 5 (without the gray and green codes). At first, k is assigned to 1. Unrolling 1 time does not change the input program. So, the *transform* procedure only inserts the counting statement `i = i - 1` into the loop of the input program P. The output of this procedure is P_1. Next, we generate the approximation of $m(X)$ by the *boundApproxi* procedure. We sample some expressions from P_1 by the SyGuS method. For example, the sampled expressions can be x and y, and we get an approximation $i > x \wedge i > y$. The 1-inductive termination certificate TC_1 is also generated in this procedure. It is presented as follows.

$$y_0 \leq 0 \wedge x_1 = x_0 \wedge y_1 = y_0 \wedge i_1 > x_1 \wedge i_1 > y_1 \Rightarrow Inv(x_1, y_1, i_1) \quad (10)$$

$$Inv(x_0, y_0, i_0) \wedge x_0 \geq 0 \wedge x_1 = x_0 + y_0 \wedge y_1 = y_0 - 1 \wedge i_1 = i_0 - 1 \Rightarrow Inv(x_1, y_1, i_1) \quad (11)$$

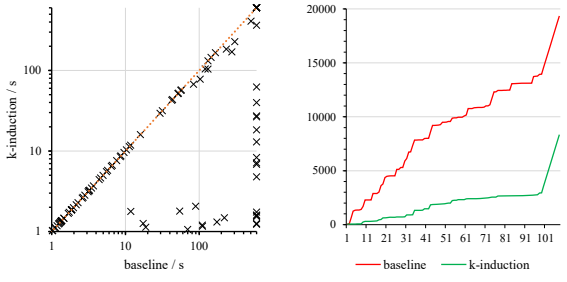
$$Inv(x, y, i) \wedge x \geq 0 \Rightarrow i > 0 \quad (12)$$

The $Pre(X_0, X_1)$ part of the example program is an `assume` statement. It does not change the value of the variables. So we represent it by the equations $x_1 = x_0$ and $y_1 = y_0$. Next, we try to synthesize an invariant to make TC_1 valid. However, the synthesis fails since



(a) Single benchmark (b) Accumulated time

Figure 8: Experiment result



(a) Single benchmark (b) Accumulated time

Figure 9: Best k values

the precondition $y_0 \leq 0$ is not strong enough. As a result, we increase k to 2 and continue our algorithm. The *transform* procedure unrolls the loop of P two times and inserts the counting statement. The output transformed program is P_2. Next, in the *boundApproxi* procedure, the approximation of $m(X)$ is still $i > x \wedge i > y$, and the 2-inductive termination certificate TC_2 is generated as follows.

$$\begin{aligned} y_0 \leq 0 \wedge x_1 = x_0 \wedge y_1 = y_0 \wedge \\ x_1 \geq 0 \wedge x_2 = x_1 + y_1 \wedge y_2 = y_1 - 1 \wedge \\ i_2 > x_2 \wedge i_2 > y_2 \Rightarrow Inv(x_2, y_2, i_2) \quad (13) \\ Inv(x_0, y_0, i_0) \wedge \\ x_0 \geq 0 \wedge x_1 = x_0 + y_0 \wedge y_1 = y_0 - 1 \wedge \\ x_1 \geq 0 \wedge x_2 = x_1 + y_1 \wedge y_2 = y_1 - 1 \wedge \\ i_2 = i_0 - 1 \Rightarrow Inv(x_2, y_2, i_2) \quad (14) \\ Inv(x, y, i) \wedge x \geq 0 \Rightarrow i > 0 \quad (15) \end{aligned}$$

Now, we can find an invariant $Inv(x, y, i) \triangleq y \leq -1 \wedge i > x$ to make TC_2 valid. That is because the strengthened precondition implies $y_2 \leq -1$. It is strong enough to guarantee the termination of the loop. As a result, our algorithm proves the termination of the input program P by a 2-inductive termination certificate.

3 EVALUATION

We implement our k-inductive algorithm on top of the tool FreqTerm [9]. It is an efficient tool to prove termination. It outperforms other tools such as AProVE [11], Ultimate Automizer [12], and HipTNT+ [15] on its termination benchmarks. FreqTerm proves termination by the *iRF* and the standard inductive *TC*. In our experiment, FreqTerm is considered as the *baseline*. The implementation of our k-inductive algorithm is called KIndTerm. The benchmarks are mainly from [9]. There are 171 terminating programs considered by [9]. However, 79 of them require a lexicographic termination argument which is currently not implemented in our approach. As a result, our experiment is conducted on the remaining terminating programs from [9] (92) and some terminating programs crafted by ourselves (16). Totally, we have 108 benchmarks.

In our first experiment, the timeout for each tool is set to 600 seconds. FreqTerm takes all of 600 seconds to solve each benchmark. Our KIndTerm arranges the time as follows. The time limitation for synthesizing 1-*TC* is set to 60 seconds, and the time limitation for synthesizing k-*TC* ($k > 1$) is set to 180 seconds. Figure 8 present the experimental results of FreqTerm and KIndTerm. There are totally 99 benchmarks solved by FreqTerm or KIndTerm. On the other 9 benchmarks, both of the tools run out of the time. The scatterplot in Figure 8(a) presents the time spent on each benchmark by FreqTerm (x-axis) and KIndTerm (y-axis) respectively. Our tool KIndTerm can solve 96 benchmarks (12 timeout) while FreqTerm can solve 82 benchmarks (26 timeout). The line chart in Figure 8(b) shows the accumulated time cost of KIndTerm and FreqTerm to solve all of the benchmarks. FreqTerm totally costs 19347s while KIndTerm just costs 11183s. Our tool KIndTerm is 42.2 percent faster than FreqTerm.

In another experiment, we try to use a series of k-*TC* ($k = 1, 2, 3, \cdots$) to solve each benchmark respectively. Then we find out the best k value for each benchmark. The k-*TC* with the best k value takes the least time to solve the benchmark. We compare our approach with the best k value against the baseline. The result is shown in Figure 9. On about one-third benchmarks, the 2-*TC* is significantly faster than the baseline. A few benchmarks are solved by 3- or 4-*TC*. But some of them are timeout in our first experiment since too much time is spent on previous iterations with smaller k. The incompleteness of the SyGuS method also limits the efficiency of our approach. Because sometimes it can not find the existing invariant for the current k value. Our approach with best k value only costs 8344s on all benchmarks, including 4800s spent on the timeout benchmarks. It is more than 2 times faster than FreqTerm.

4 CONCLUSION AND FUTURE WORK

In this paper, we propose a novel approach to proving the termination of imperative programs by k-induction. It strengthens the standard inductive termination certificate of the *iRF*-based method. Our k-inductive method is more powerful than the standard method both in theory and in practice. The experimental results also show the significant effectiveness and efficiency of our approach. Besides the SyGuS method, the template-based synthesis is also suitable for synthesizing invariant of our k-*TC*. Moreover, the lexicographic k-inductive termination certificate should be a good method to handle the termination problem of programs with more complex control-flow.

ACKNOWLEDGMENTS

This work was partially funded by the National Key R&D Program of China (No. 2018YFB1308601), the NSF of China (No. 61672310 and No. 61527812), and the Guangdong Science and Technology Department (No. 2018B010107004).

REFERENCES

[1] Amir M Ben-Amram and Samir Genaim. 2013. On the linear ranking problem for integer linear-constraint loops. *ACM SIGPLAN Notices* 48, 1 (2013), 51–62.

[2] Aaron R Bradley, Zohar Manna, and Henny B Sipma. 2005. Termination analysis of integer linear loops. In *International Conference on Concurrency Theory*. Springer, 488–502.

[3] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. 2015. Safety verification and refutation by k-invariants and k-induction. In *International Static Analysis Symposium*. Springer, 145–161.

[4] Michael A Colón, Sriram Sankaranarayanan, and Henny B Sipma. 2003. Linear invariant generation using non-linear constraint solving. In *International Conference on Computer Aided Verification*. Springer, 420–432.

[5] Michael A Colóon and Henny B Sipma. 2001. Synthesis of linear ranking functions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 67–81.

[6] Leonardo De Moura, Harald Rueß, and Maria Sorea. 2003. Bounded model checking and induction: From refutation to verification. In *International Conference on Computer Aided Verification*. Springer, 14–26.

[7] Alastair F Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. 2011. Software verification using k-induction. In *International Static Analysis Symposium*. Springer, 351–368.

[8] Grigory Fedyukovich, Samuel J Kaufman, and Rastislav Bodík. 2017. Sampling invariants from frequency distributions. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 100–107.

[9] Grigory Fedyukovich, Yueling Zhang, and Aarti Gupta. 2018. Syntax-guided termination analysis. In *International Conference on Computer Aided Verification*. Springer, 124–143.

[10] Mikhail YR Gadelha, Hussama I Ismail, and Lucas C Cordeiro. 2017. Handling loops in bounded model checking of C programs via k-induction. *International Journal on Software Tools for Technology Transfer* 19, 1 (2017), 97–114.

[11] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, et al. 2014. Proving termination of programs automatically with AProVE. In *International Joint Conference on Automated Reasoning*. Springer, 184–191.

[12] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination analysis by learning terminating programs. In *International Conference on Computer Aided Verification*. Springer, 797–813.

[13] Kryštof Hoder and Nikolaj Bjørner. 2012. Generalized property directed reachability. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 157–171.

[14] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods in System Design* 48, 3 (2016), 175–205.

[15] Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. 2015. Termination and non-termination specification inference. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 489–498.

[16] Jan Leike and Matthias Heizmann. 2014. Ranking templates for linear loops. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 172–186.

[17] Andreas Podelski and Andrey Rybalchenko. 2004. A complete method for the synthesis of linear ranking functions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 239–251.

[18] Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. 2016. Synthesizing ranking functions from bits and pieces. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 54–70.