Termination Analysis with Algorithmic Learning*

Wonchan Lee[1], Bow-Yaw Wang[2], and Kwangkeun Yi[1]

[1] Seoul National University, Korea
[2] Academia Sinica, Taiwan

Abstract. An algorithmic-learning-based termination analysis technique is presented. The new technique combines transition predicate abstraction, algorithmic learning, and decision procedures to compute transition invariants as proofs of program termination. Compared to the previous approaches that mostly aim to find a particular form of transition invariants, our technique does not commit to any particular one. For the examples that the previous approaches simply give up and report failure our technique can still prove the termination. We compare our technique with others on several benchmarks from literature including PolyRank examples, SNU realtime benchmark, and Windows device driver examples. The result shows that our technique outperforms others both in efficiency and effectiveness.

1 Introduction

Termination is a critical property of functions in program libraries. Invoking a non-terminating library function may result in system lagging or even freezing. Because of its importance, termination analysis has been studied extensively [2–4, 8, 10–14, 17, 22, 24–27] for the last decade and advanced to the level of industrial uses [1, 13].

Among various strategies for proving termination, we are most interested in the transition invariant-based technique. A transition invariant for a transition relation is an over-approximation to the reachable transitive closure of the transition relation [13, 25, 26]. Podelski and Rybalchenko [25] have shown that the termination of a program amounts to the existence of a disjunctively well-founded transition invariant for its transition relation. We therefore aim to find a disjunctively well-founded transition invariant for the transition relation of a program.

Though transition invariants can be defined as a fixpoint, they are not necessarily computed by costly fixpoint iterations. Observe that it suffices to find one disjunctively well-founded *over-approximation* to the least reachable transition

* This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grant 2012-0000468) and National Science Council of Taiwan Grant Numbers 99-2218-E-001-002-MY3 and 100-2221-E-002-116-.

P. Madhusudan and S.A. Seshia (Eds.): CAV 2012, LNCS 7358, pp. 88–104, 2012.
© Springer-Verlag Berlin Heidelberg 2012

invariant. If there are lots of such over-approximations, we have only to design an efficient algorithm to compute one. Indeed, several such algorithms have been proposed to compute reachable transition invariants efficiently [2, 8, 22, 27].

In this paper, we report the first algorithmic-learning-based technique for termination analysis. Recently, algorithmic learning is successfully applied to avoid the costly fixpoint iteration in the context of loop invariant generation [19–21]. In the same spirit, the technique we propose in this paper finds disjunctively well-founded transition invariants without the excessive cost of fixpoint iterations by combining algorithmic learning, transition predicate abstraction, decision procedures, and well-foundedness checkers. Through transition predicate abstraction, we adopt a learning algorithm for Boolean formulae to infer transition invariants over given atomic predicates. Using an SMT solver and well-foundedness checker, we design a mechanical teacher to guide the learning algorithm to find a disjunctively well-founded transition invariant. Randomness is moreover employed to exploit the multitude of transition invariants.

The advantage of our technique is that it can be both efficient and effective, compared to the previous works [2, 8, 22, 27]. The key innovation of our technique is that we decouple the construction of transition invariants from the transition predicate generation. In previous works, the transition predicate generation is tightly coupled with the transition invariant inference and the whole process is optimized by committing to a particular form of transition invariants, which might hurt the effectiveness. However, the following intuition from the years of research on termination analysis teaches us that this is not necessarily the case; termination arguments, or transition predicates, are evident in most cases [27], but it is not so obvious how to combine those predicates to get a disjunctively well-founded transition invariant [2]. To solve the "not-so-obvious" problem efficiently, we use algorithmic learning which was proven to work well in a different domain, inferring loop invariants out of atomic predicates. Being trained by mechanical teachers, learning algorithms become an efficient engine for exploring possible combinations of predicates. For the atomic transition predicate generation, we employ a simple heuristic, which is turned out to be effective for most of examples in the experiments. We can further improve the effectiveness with additional predicates.

Example. Consider the following nested loop. We found that this simple nested loop cannot be proven by any of the existing termination analysis tools [8, 22, 27]:

$$\texttt{while}\ i < 10\ \texttt{do}\ \{j \leftarrow 0; \texttt{while}\ j < 10\ \texttt{do}\ \{i', j' \leftarrow i + 1, j + 1\}\}$$

Our simple heuristic finds the set $\{i < 10, j < 10, i < i', j < j'\}$ of atomic transition predicates. Then, our randomized technique first computes a transition invariant for the inner loop, say, $j < 10 \wedge j < j'$. Since the transition invariant is well-founded, it proves the termination of the inner loop. Next, we replace the inner loop by its transition invariant and proceed to find a transition invariant for the following simple loop:

$$\texttt{while}\ i < 10\ \texttt{do}\ \{j \leftarrow 0; \texttt{assume}(j < 10 \wedge j < j');\}$$

Since its loop body does not update the variable i, it is impossible to prove the termination of the loop. This is exactly what happens in some of the existing tools [8, 27]; after they compute $j < 10 \wedge j < j'$ as a transition invariant of the inner loop, they simply report possible non-termination of the outer loop. The other tool [22] fails because it uses a even more imprecise transition invariant, *true*, as the summary of the inner loop; the tool unrolls and unions the transition relation of a loop body until it reaches a transition invariant and when it unrolls the outer loop, the tool can only assume that the inner loop can change variables arbitrarily. However, there exists another transition invariant $j < 10 \wedge j < j' \wedge i < i'$ which are both expressible with the given predicates and precise enough to prove the termination of the outer loop. As long as a transition invariant is expressible with the given predicates, our randomized algorithm for termination analysis can find it. Let us say our technique returns $j < 10 \wedge j < j' \wedge i < i'$ this time. The new transition invariant is again well-founded. We proceed to replace the inner loop by the new transition invariant:

$$\texttt{while}\ i < 10\ \texttt{do}\ \{j \leftarrow 0; \texttt{assume}(j < 10 \wedge j < j' \wedge i < i');\}$$

Our termination analysis algorithm is now able to infer the transition invariant $i < 10 \wedge i < i'$ for the simple loop. Since the transition invariant $i < 10 \wedge i < i'$ is well-founded, we conclude that the outer loop is terminating as well. □

Contributions.

- We design and implement an algorithmic-learning-based termination analyzer. As far as we know, our work is the first to apply the algorithmic learning to termination analysis problem.
- We empirically show that the prototype implementation of our technique outperforms the previous tools [8, 22, 27] both in efficiency and effectiveness.

Organization. Section 2 reviews termination analysis via transition invariants and presents our formalism of transition invariants in intentional representation. Section 3 explains algorithmic-learning-based inference approach and how to apply it to the problem of inferring disjunctively well-founded transition invariants. Section 4 presents our experiment results. Section 5 discusses related work and Section 6 concludes.

2 Termination Analysis via Transition Invariants

This section explains the termination analysis technique based on transition invariants. The technique was first introduced by Podelski and Rybalchenko [25] and later implemented on top of the SLAM model checker [13]. We first review the original theory of transition invariants in an extensional view [18]. We then present our formalism of transition invariants in an intensional view which we compute via algorithmic-learning-based approach.

2.1 Program Termination and Transition Invariant

A program $P = \langle W_P, I_P, R_P \rangle$ consists of a set W_P of states, a set $I_P \subseteq W_P$ of initial states, and a transition relation $R_P \subseteq W_P \times W_P$.

A program P terminates if there is no infinite sequence $s_1, s_2, \cdots$ of states such that $s_1 \in I_P$ and $(s_i, s_{i+1}) \in R_P$. This condition is equivalent to the well-foundedness of $R_P \cap \mathit{Reach}(P) \times \mathit{Reach}(P)$. Here, the set $\mathit{Reach}(P)$ denotes the set of reachable states.

Instead of showing $R_P \cap \mathit{Reach}(P) \times \mathit{Reach}(P)$ is well-founded, we prove the termination by finding its disjunctively well-founded transition invariant [25]. A transition invariant T of P is a relation that contains a reachable portion of the transitive closure of R_P:

$$R_P^+ \cap \mathit{Reach}(P) \times \mathit{Reach}(P) \subseteq T.$$

Furthermore, we say the transition invariant T is disjunctively well-founded when it is a union of a finite number of well-founded relations $T_1, \cdots, T_N$.

Theorem 1 ([25]). *A program P terminates if and only if there exists a disjunctively well-founded transition invariant T of P.*

Thanks to Theorem 1, the problem of program termination now becomes finding a disjunctively well-founded transition invariant for a given program P.

Cook et al. [13] showed that transition invariants can be reduced to reachability analysis. The authors named the relation $R_P^+ \cap \mathit{Reach}(P) \times \mathit{Reach}(P)$ *binary reachability relation*, which is the least fixpoint of the following functional F_P starting from the relation $\bot_P$.

$$\begin{aligned} F_P(X) &\triangleq (X \cup id_{(2)}(X)) \circ R_P \\ id_{(2)}(X) &\triangleq \{(\nu_2, \nu_2) \in W_P \times W_P : \exists \nu_1.(\nu_1, \nu_2) \in X\} \\ X \circ Y &\triangleq \{(\nu_1, \nu_3) \in W_P \times W_P : \exists \nu_2.(\nu_1, \nu_2) \in X \text{ and } (\nu_2, \nu_3) \in Y\} \\ \bot_P &\triangleq \{(\nu_1, \nu_2) \in W_P \times W_P : \nu_1 \in I_P \text{ and } (\nu_1, \nu_2) \in R_P\} \end{aligned}$$

In the next subsection, we show how to compute an over-approximation of this binary reachability relation via intensional representations of transition invariants.

2.2 Intensional Transition Invariants

Simple Loop Programs. For presentation, we consider a simple loop program P with the following abstract syntax.

$$\begin{aligned} P &::= \{l\}\ \texttt{while}\ l\ \texttt{do}\ S \\ S &::= v \leftarrow e \mid v \leftarrow \texttt{nondet} \mid \texttt{assume}\ l \mid S\ []\ S \mid S;\ S \\ l &::= e \leq n \mid l \wedge l \mid l \vee l \\ e &::= n \mid v \mid n \times e \mid e + e \mid e - e \qquad (v \in V, n \in \mathbb{Z}) \end{aligned}$$

where V and $\mathbb{Z}$ is a set of variables and integers respectively, and l represents quantifier-free formulae over integer affine predicates. A loop with a loop guard

is annotated with a formula specifying a precondition. We write κ_P for the loop guard and δ_P for the precondition. In the syntax, we have non-deterministic assignments ($v \leftarrow \texttt{nondet}$) to emulate the behaviors of unsupported features such as arrays or function calls. For brevity, we use choice ($S \ [] \ S$) and assume statements ($\texttt{assume}\ l$) instead of traditional if statements.

A program state $\nu \in W_P$ of the program P is a map from V to $\mathbb{Z}$. Given a formula l, we write $\nu \models_{\text{sat}} l$ when ν *satisfies* l. We write $\models_{\text{sat}} l$ if there exists a state that satisfies l. When the formula is satisfied by all states, we write $\models l$. We define the set $\mathcal{W}(l)$ to be $\{\nu \in W_P : \nu \models_{\text{sat}} l\}$. For a simple loop program P, the set I_P of initial states is the same as $\mathcal{W}(\delta_P)$.

To describe the transition relation R_P for a simple loop program, we define transition semantics $[\![P]\!]$ of P. The transition semantics is a quantifier-free formula over sets V and V' describing the current state and the state after the transition, respectively. The transition semantics is defined as follows:

$$
\begin{aligned}
[\![\{\delta_P\}\ \texttt{while}\ \kappa_P\ \texttt{do}\ S]\!] &\triangleq \kappa_P \wedge [\![S]\!] \\
[\![v \leftarrow e]\!] &\triangleq v' = e \wedge \bigwedge_{w \in V \setminus \{v\}} w' = w \\
[\![v \leftarrow \texttt{nondet}]\!] &\triangleq v' = v'' \wedge \bigwedge_{w \in V \setminus \{v\}} w' = w \qquad (v'' : \text{fresh}) \\
[\![\texttt{assume}\ l]\!] &\triangleq l \wedge \bigwedge_{w \in V \setminus \mathit{Vars}(l)} w' = w \\
[\![S_0\ []\ S_1]\!] &\triangleq [\![S_0]\!] \vee [\![S_1]\!] \\
[\![S_0; S_1]\!] &\triangleq [\![S_0]\!][V' \mapsto V''] \wedge [\![S_1]\!][V \mapsto V''] \qquad (V'' : \text{fresh})
\end{aligned}
$$

where $\mathit{Vars}(l)$ is the set of variables appeared in l and $f[v_1 \mapsto v_2]$ is the formula obtained by substituting the variable v_2 for v_1 in f. Given a formula f over V and V', we write $\nu, \nu' \models_{\text{sat}} f$ when the formula obtained by replacing $v \in V$ in f with $\nu(v)$ and $v' \in V'$ in f with $\nu'(v')$ is satisfiable. The notations $\models_{\text{sat}} f$ and $\models f$ are defined accordingly. The notation $\mathcal{R}(f)$ denotes the relation $\{(\nu, \nu') \in W_P \times W_P : \nu, \nu' \models_{\text{sat}} f\}$. Thus the transition relation R_P of a simple loop program P is $\mathcal{R}([\![P]\!])$.

In summary, a simple loop program P defines the program $\langle W_P, \mathcal{W}(\delta_P), \mathcal{R}([\![P]\!])\rangle$.

Intensional Transition Invariants. For a simple loop program P, we define the intensional representations of the functional F_P and the relation $\perp_P$ (written $F_P^\sharp$ and $\perp_P^\sharp$ respectively) as follows:

$$
\begin{aligned}
F_P^\sharp(f) &\triangleq (f \vee id_{(2)}^\sharp(f)) \circ^\sharp [\![P]\!] \\
id_{(2)}^\sharp(f) &\triangleq f[V \mapsto V''] \wedge V = V' \qquad (V'' : \text{fresh}) \\
f \circ^\sharp g &\triangleq f[V' \mapsto V''] \wedge g[V \mapsto V''] \qquad (V'' : \text{fresh}) \\
\perp_P^\sharp &\triangleq \delta_P \wedge [\![P]\!]
\end{aligned}
$$

where $f[\{v_1, \cdots, v_n\} \mapsto \{v_1', \cdots, v_n'\}] \triangleq f[v_1 \mapsto v_1'] \cdots [v_n \mapsto v_n']$. The following lemmas show that $F_P^\sharp$ and $\perp_P^\sharp$ correspond to F_P and $\perp_P$ respectively.

Lemma 1. *For any simple loop program P, $\mathcal{W}(\bot_P^\sharp) = \bot_P$.*

Lemma 2. *Let f be a quantifier-free formula over V and V'. For any simple loop program P, $\mathcal{R}(F_P^\sharp(f)) = F_P(\mathcal{R}(f))$.*

From the properties of $F_P^\sharp$ and $\bot_P^\sharp$, we compute a transition invariant of a simple loop program P by finding a formula $\mathcal{T}$ that satisfies the following conditions:

1. $\models \bot_P^\sharp \implies \mathcal{T}$;
2. $\models \mathcal{T} \implies \kappa_P$;
3. $\models F_P^\sharp(\mathcal{T}) \implies \mathcal{T}$.

The first condition is to guarantee that $\mathcal{T}$ subsumes the first iteration starting from the initial state. The second condition is to guarantee that $\mathcal{T}$ expresses only the iterations within the loop. The last condition is to guarantee that $\mathcal{T}$ is a fixpoint. Note that this fixpoint is not necessarily a *least* fixpoint. We call the formula $\mathcal{T}$ *intensional* transition invariant which is an intensional representation of a transition invariant.

Lemma 3. *Let $\mathcal{T}$ be an intensional transition invariant of a simple loop program P. Then $\mathcal{R}(\mathcal{T})$ is a transition invariant; i.e. $\mathcal{R}(\mathcal{T}) \supseteq R_P^+ \cap Reach(P) \times Reach(P)$.*

We say $\mathcal{T}$ is disjunctively well-founded when $\mathcal{R}(\mathcal{T})$ is disjunctively well-founded. Disjunctively well-founded intensional transition invariants are proofs of program termination.

Theorem 2. *A simple loop program P terminates if there exists a disjunctively well-founded intensional transition invariant $\mathcal{T}$ of P.*

In the rest of the paper, transition invariants mean intensional transition invariants unless stated otherwise.

3 Algorithmic-Learning-Based Inference of Transition Invariants

The key idea of the algorithmic-learning-based framework [19–21] is to apply CDNF algorithm [7] to infer a formula with a mechanical teacher. CDNF algorithm is an exact learning algorithm for Boolean formulae. It infers an arbitrary Boolean formula over fixed variables by interacting with a teacher. In our case, we are particularly interested in finding transition invariants over the given set of atomic transition predicates. In order to apply CDNF algorithm, we will design a mechanical teacher to guide the learning algorithm to infer a transition invariant for a simple loop program.

In this section, we explain our design of the mechanical teacher in details. We first introduce CDNF algorithm for Boolean formulae. Through transition predicate abstraction, the correspondence between Boolean formulae and quantifier-free formulae over atomic transition predicates is explained. Lastly, we present our design of the mechanical teacher.

3.1 CDNF Learning Algorithm

CDNF algorithm is an exact learning algorithm for Boolean formulae. It infers an unknown target formula by posing queries to a teacher. The teacher is responsible for answering two types of queries. The learning algorithm may ask if a valuation satisfies the target formula by a membership query. Or it may ask if a conjectured formula is equivalent to the target in an equivalence query. According to the answers to queries, CDNF algorithm will infer a Boolean formula equivalent to the unknown target within a polynomial number of queries in the formula size of the target.

In order to apply CDNF algorithm, a mechanical teacher that answers queries from the learning algorithm is needed. The mechanical teacher consists of two algorithms. The membership query resolution algorithm (MEM) answers membership queries; the equivalence query resolution algorithm (EQ) resolves equivalence queries. The algorithm MEM returns YES if the given valuation satisfies the unknown target and NO otherwise. The algorithm EQ returns YES if the given conjecture is equivalent to the target and a counterexample otherwise. Let $\mathbf{x}$ be a set of Boolean variables, and $BF[\mathbf{x}]$ and $Val_{\mathbf{x}}$ be the set of Boolean formulae and valuations over $\mathbf{x}$, respectively. The signatures of these query resolution algorithms are as follows:

$$\begin{aligned} MEM &: Val_{\mathbf{x}} \to \{YES, NO\} \\ EQ &: BF[\mathbf{x}] \to \{YES\} + Val_{\mathbf{x}} \end{aligned}$$

3.2 Learning Algorithm as an Inference Engine

We establish a connection between Boolean formulae and quantifier-free formulae. The connection enables CDNF algorithm to infer transition invariants. We consider transition predicate abstraction [26] over a set $\mathcal{P}$ of atomic predicates defined over V and V'. A quantifier-free formula f over $\mathcal{P}$ is generated by the following syntax.

$$f ::= p \mid \neg f \mid f \vee f \mid f \wedge f$$

where $p \in \mathcal{P}$. We write $QF[\mathcal{P}]$ for the set of quantifier-free formulae over $\mathcal{P}$. The set $QF[\mathcal{P}]$ and the set $BF[\mathbf{x}]$ of Boolean formulae, where $\mathbf{x} = \{x_{p_i} \mid p_i \in \mathcal{P}\}$, establishes the following Galois connection.

$$QF[\mathcal{P}] \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} BF[\mathbf{x}]$$

From the connection, we know that once the learning algorithm finds a Boolean formula, then it has a corresponding quantifier-free formula that we want to find.

We now show how to make a mechanical teacher under the transition predicate abstraction. We define the following two functions $\overline{\alpha}$ and $\overline{\gamma}$ that translate valuations over V and $\mathbf{x}$, respectively.

$$\begin{aligned} \overline{\alpha}(\nu, \nu') &\triangleq \mu \text{ such that } \mu \models \bigwedge_{\nu,\nu' \models_{\text{sat}} p} x_p \wedge \bigwedge_{\nu,\nu' \not\models_{\text{sat}} p} \neg x_p \\ \overline{\gamma}(\mu) &\triangleq \bigwedge_{\mu(x_p)=\top} p \wedge \bigwedge_{\mu(x_p)=\bot} \neg p. \end{aligned}$$

The design of the query resolution algorithms MEM and EQ amounts to that of two concrete algorithms $MEM^\sharp$ and $EQ^\sharp$ with the following signatures:

$$\begin{aligned} MEM^\sharp &: QF[\mathcal{P}] \to \{\mathit{YES}, \mathit{NO}\} \\ EQ^\sharp &: QF[\mathcal{P}] \to \{\mathit{YES}\} + \mathit{Val}_V \times \mathit{Val}_V \end{aligned}$$

With the two concrete algorithms and the translation functions $\overline{\alpha}$ and $\overline{\gamma}$, we derive the query resolution algorithms MEM and EQ as follows:

$$\begin{aligned} MEM(\mu) &= MEM^\sharp(\overline{\gamma}(\mu)) \\ EQ(b) &= \begin{cases} \overline{\alpha}(\nu, \nu') & \text{when } EQ^\sharp(\gamma(b)) = (\nu, \nu') \\ \mathit{YES} & \text{otherwise} \end{cases} \end{aligned}$$

3.3 Algorithms for Mechanical Teacher

In the rest of this section, we explain how to design the algorithms $MEM^\sharp$ and $EQ^\sharp$ for transition invariants. One technical question is how we can make $MEM^\sharp$ and $EQ^\sharp$ answer questions on the formula that we do not know yet. We solve this problem simply by giving random answers when we cannot answer conclusively. Interesting observation is that as far as those answers are consistent and algorithm $EQ^\sharp$ returns YES when it really finds the one, CDNF algorithm can still infer the target formula. We exploit the fact that there can exist multiple formulae that are equivalent to the target.

Membership Query Resolution. In a membership query $MEM(\mu)$ with $\mu \in \mathit{Val}_{\mathbf{x}}$, we would like to know if μ is included in an unknown target Boolean formula that represents a disjunctively well-founded transition invariant. Since we do not know any disjunctively well-founded transition invariant yet, we can not answer every membership query conclusively.

To see what amount of answers we can give conclusively, we first consider the conditions that μ should respect. Suppose $\mathcal{T}$ is a transition invariant. If μ satisfies the target Boolean formula, we have $\overline{\gamma}(\mu) \implies \mathcal{T}$. Moreover, we have $\bot_P^\sharp \implies \mathcal{T} \implies \kappa_P$ for $\mathcal{T}$ is a transition invariant. Therefore, we have the following relationship:

1. If $\not\models \overline{\gamma}(\mu) \implies \kappa_P$, then $\not\models \overline{\gamma}(\mu) \implies \mathcal{T}$;
2. If $\models \overline{\gamma}(\mu) \implies \bot_P^\sharp$, then $\models \overline{\gamma}(\mu) \implies \mathcal{T}$.

In the first case, we can conclusively answer NO. Similarly, we answer YES for the second case conclusively.

For the others cases, we can give random answers. Since we are looking for disjunctively well-founded transition invariants, we heuristically answer NO when $\mathcal{R}(\overline{\gamma}(\mu))$ is not well-founded.

Algorithm 1 summarizes the membership query resolution. The $MEM^\sharp(f)$ algorithm first checks if $\models f \implies \kappa_P$. If not, it returns NO. The algorithm then checks if $\mathcal{R}(f)$ is well-founded. If not, it heuristically returns NO. Finally, the algorithm checks if $\models f \implies \bot_P^\sharp$. If so, it returns YES since we know for sure

Algorithm 1. $MEM^\sharp(f)$

```
   Input: f ∈ QF[P]
   Output: YES or NO
 1 if ⊨_sat f ∧ ¬κ_P then
 2 |  return NO
 3 else
 4 |  if R(f) is well-founded then
 5 |  |  if ⊨_sat f ∧ ¬⊥^♯_P then
 6 |  |  |  return YES or NO randomly
 7 |  |  else
 8 |  |  |  return YES
 9 |  else
10 |  |  return NO
```

Algorithm 2. $EQ^\sharp(f)$

```
   Input: f : a CDNF formula such that f = ⋀_{i=1}^{n} f_i
   Output: YES or a counterexample (ν, ν') ∈ Val_V × Val_V
 1 if isTransitionInvariant(f) is (ν, ν') ∈ Val_V × Val_V then
 2 |  return (ν, ν')
   // f is a transition invariant
 3 if hasDWFConjunct(f) is YES then
 4 |  return YES
   // f_i is not disjunctively well-founded for every i
 5 if findCounterexample(f) is (ν, ν') ∈ Val_V × Val_V then
 6 |  return (ν, ν')
 7 restart CDNF algorithm
```

that $\overline{\gamma}^{-1}(f)$ is the member of the target formula. Otherwise, it gives a random answer to the learning algorithm.

Equivalence Query Resolution. In an equivalence query $EQ(b)$, we are given a CDNF formula b over $\mathbf{x}$ as the conjecture. The algorithm should check whether $\gamma(b)$ is a disjunctively well-founded transition invariant for the simple loop program P. If not, it returns a valuation over $\mathbf{x}$ as a counterexample.

Algorithm 2 presents the equivalence query resolution algorithm $EQ^\sharp(f)$. The algorithm first checks if f is a transition invariant. If not, it returns a counterexample. Next, it checks if the formula has a disjunctively well-founded conjunct f_i. If so, we have found a disjunctively well-founded transition invariant. Otherwise, the algorithm tries to find a counterexample that possibly makes the formula not disjunctively well-founded. If it cannot find a counterexample, the algorithm simply restarts to find another transition invariant.

Invariance Check. Algorithm 3 shows the procedure to check if f satisfies the three conditions of transition invariants. If the conjecture f does not satisfy one of them, the algorithm returns a counterexample.

Algorithm 3. *isTransitionInvariant*(f)

Input: f : a CDNF formula
Output: *YES* or a counterexample $(\nu, \nu') \in Val_V \times Val_V$

```
if ⊨ ⊥_P^♯ ⟹ f and ⊨ f ⟹ κ_P and ⊨ F_P^♯(f) ⟹ f then
    return YES
if ν, ν' ⊨_sat ⊥_P^♯ ∧ ¬f then
    return (ν, ν')
if ν, ν' ⊨_sat f ∧ ¬κ_P then
    return (ν, ν')
if ν, ν' ⊨_sat F_P^♯(f) ∧ ¬f then
    return (ν, ν')
```

Algorithm 4. *hasDWFConjunct*(f)

Input: f : a CDNF formula such that $f = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} f_{ij}$
Output: *YES* if f_i is disjunctively well-founded for some i; *NO* otherwise

```
foreach i = 1, ··· , n do
    isWellFounded ← ⊤
    foreach j = 1, ··· , m_i do
        if R(f_ij) is not well-founded then
            isWellFounded ← ⊥
            break
    if isWellFounded then return YES
return NO;
```

Disjunctively Well-foundedness Check. Algorithm 4 checks if f_i is disjunctively well-founded for some i. Recall that f is a CDNF formula such that $f = \bigwedge_{i=1}^{n} f_i$ and each f_i is also a transition invariant since f implies f_i. If the algorithm has found one disjunctively well-founded f_i, we have found a disjunctively well-founded transition invariant. The following lemma states the correctness of the algorithm.

Lemma 4. *Let $f = \bigwedge_{i=1}^{n} f_i$ be a CDNF formula. If f is a transition invariant and f_i is disjunctively well-founded for some i, f_i is a disjunctively well-founded transition invariant.*

For each DNF formula f_i, we check if all of its disjuncts are well-founded. Each disjunct f_{ij} is a conjunction of atomic transition predicates and we can check the well-foundedness using existing well-foundness checkers [4, 5, 11, 24].

Counterexample Generation. Conjectures from learning algorithms are sometimes not disjunctively well-founded even if they are a transition invariant. Those are either containing an idle transition, which does nothing during an iteration, or the ones that become disjunctively well-founded once proper bound conditions are added. For example, transition invariant $x' \leq x$ contains an idle transition and transition invariant $x' < x$ becomes well-founded if additional bound condition $x > 0$ is added. We implemented an algorithm that generates a

Algorithm 5. *findCounterexample*(f)

Input: f : a CDNF formula
Output: a counterexample $(\nu, \nu') \in Val_V \times Val_V$ or *FAIL*
1 **if** $(\nu, \nu') \models_{sat} f \wedge V' = V$ **then return** (ν, ν') *as a counterexample*
2 **return** *FAIL*

Algorithm 6. Pseudo-code of the main loop

Input: set $\mathcal{P}$ of transition predicates
1 **while** *there exists a simple loop* P *in the program* **do**
2 repeat N times to infer d.wf transition invariant $\mathcal{T} \in QF[\mathcal{P}]$ using CDNF algorithm
3 **if** $\mathcal{T}$ *is found* **then**
4 replace P with assume$(\kappa_P \wedge \mathcal{T} \vee \neg\kappa_P \wedge V = V')$;
5 **else**
6 replace P with assume$(\kappa_P \wedge true \vee \neg\kappa_P \wedge V = V')$;

counterexample for both cases, but for space reason, we present in Algorithm 5 a simplified procedure that handles only the first case. If the algorithm finds an idle transition ($\models_{\text{sat}}\ f \wedge V' = V$), it returns a counterexample. Otherwise it returns *FAIL*, hoping that the learning algorithm finds another formula next time.

4 Experiments

To evaluate our approach, we implemented our algorithm and compared it with existing tools. In the implementation, we use Z3 SMT solver [16] for satisfiability check and our own implementation of RankFinder algorithm [24] for well-foundedness check.

Algorithm 6 shows the pseudo-code of the main loop of our analyzer. The algorithm essentially handles the nested loop in the manner similar to that of [27]; it finds a non-nested simple loop and tries to find a disjunctively well-founded transition invariant; when it finds one, we can use it as a summary of the loop and make the outer-loop also non-nested; if the inference fails within the given limit N, it simply assumes that the loop can change the variable arbitrarily and uses *true* as its summary.

In Algorithm 6, we make CDNF algorithm repeat only a certain number of times because the learning algorithm loops indefinitely if a given loop does not terminate or it does but there is no disjunctively well-founded transition invariant expressible with the given set of predicates. In practice, CDNF algorithm could find a disjunctively well-founded transition invariant within several trials.

We implement a simple heuristic that generates atomic transition predicates using loop guards and branch conditions. First, all loop guards and branch conditions are used as atomic transition predicates. Second, for each loop guard, say $E_1 \geq E_2$, we generate predicates $E_1' - E_2' < E_1 - E_2$. The intuition is that the gap

between values of E_1 and E_2 should decrease so that the loop guard would be eventually violated. According to our experience, even with this simple heuristic we could verify almost all terminating examples (only four predicates are required to add manually in the whole experiments).

For comparison, we use the following four tools.

LTA) Our prototype algorithmic-Learning-based Termination Analyzer (LTA) with a simple heuristic for transition predicate generation.

LF) LoopFrog [27], a summary-based termination analyzer. LoopFrog can be configured with five different templates of transition invariants and we use only the template $i' \diamond i$ where $\diamond = \{<, >\}$, which showed the best performance according to [27].

LR) LinearRankTerm [8], an abstract interpretation-based termination analyzer.

CTA) Compositional Termination Analyzer (CTA) [22].

All experiments are done on Intel Core i7 3.07 Ghz CPU with 24GB memory running Linux 2.6.35. The timeout for CTA is set to one hour and LTA is configured with the retrial limit (N in Algorithm 6) 100.

In all experiments, we report only the elapsed time for cases that tools could prove the termination. If there are multiple loops, we report the elapsed time aggregated only on terminating cases (denoted by '+' after numbers). The reason is that our technique is semi-algorithm; it is not meaningful to report the elapsed time to eventually fail since it simply depends on the parameter N. We run each case 100 times and take the average of them.

We use four sets of examples[1] from the literature, which are examples from Octagon library [23], PolyRank distribution [5, 6], Windows device drivers [2, 8], and SNU real-time benchmark suite [27]. Since our prototype supports a fragment of full ANSI-C, some examples are manually translated when they use unsupported features. The experiment results are given in Figure 1.

Figure 1(a) and (b) shows the results on examples from Octagon Library and PolyRank distribution[2], respectively. All examples are known to terminate. Our tool is the only one that proves all examples from Octagon library (note that we got a different result from the one in [8]; we tried our best but we could not make LinearRankTerm prove example 3). In terms of efficiency, LTA outperforms the others except LoopFrog; since LoopFrog considers only one iteration of loops with the pre-defined transition invariant template, it is very efficient for simple programs. For the examples from PolyRank distribution, only LTA and LinearRankTerm can prove the first two.

Figure 1(c) shows the result on examples from Windows device drivers. Example 2, 3, and 9 are known to have termination bugs and the others terminate. Only LTA and LinearRankTerm can prove all the terminating cases and LTA

[1] We made them available at `http://ropas.snu.ac.kr/cav12/`. Windows device driver examples cannot be made available due to the license issue.

[2] As already noted in [2, 8], there was no example 5 in the original distribution. We used the same numbering to avoid confusion.

	1	2	3	4	5	6
LTA	0.01	0.01	0.59	0.12	0.01	0.03
LF	0.01	0.01	0.02	0.03	⊘	0.01
LR	0.20	0.16	⊘	0.32	0.21	0.79
CTA	0.58	0.26	9.48	⊘	⊘	0.48

(a) Results on examples from the Octagon Library

	1	2	3	4	6	7	8	9	10	11	12
LTA	0.03	0.45	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘
LF	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘
LR	0.71	0.34	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘
CTA	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘	⊘

(b) Results on examples from the PolyRank distribution

	1	2	3	4	5	6	7	8	9	10
LTA	0.43	⊘	⊘	0.02	0.01	0.60	0.30	0.20	⊘	0.03
LF	⊘	⊘	⊘	0.00	0.00	⊘	⊘	⊘	⊘	⊘
LR	0.66	⊘	⊘	0.32	0.16	0.34	0.54	0.29	⊘	0.28
CTA	T/O	⊘	⊘	0.41	0.44	2.04	8.86	8.87	⊘	T/O

(c) Results on small arithmetic examples taken from Windows device drivers

Example	**Tool**	✓	⊘	**Time**
bs	**LTA**	1	0	0.01
1 loop	**LF**	0	1	N/A
1 terminates	**CTA**	0	1	N/A
fft1k	**LTA**	3	0	0.04
3 loops	**LF**	2	1	0.03+
3 terminate	**CTA**	2	1	0.54+
fft1	**LTA**	2	3	0.03+
5 loops	**LF**	2	3	0.18+
2 terminate	**CTA**	2	3	0.66+
insertsort	**LTA**	1	1	0.01+
2 loops	**LF**	1	1	0.01+
1 terminates	**CTA**	1	1	0.29+

Example	**Tool**	✓	⊘	**Time**
ludcmp	**LTA**	11	0	0.13
11 loops	**LF**	5	6	0.07+
11 terminate	**CTA**	4	7	1.50+
minver	**LTA**	15	2	0.23+
17 loops	**LF**	16	1	0.22+
15 terminate	**CTA**	15	2	5.21+
qsort-exam	**LTA**	2	4	0.66+
6 loops	**LF**	0	6	N/A
2 terminate	**CTA**	0	6	N/A
select	**LTA**	0	4	N/A
4 loops	**LF**	0	4	N/A
0 terminates	**CTA**	0	4	N/A

(d) Results on modified examples from SNU real-time benchmark

Fig. 1. Experiment Results. **LTA** is used to represent algorithmic-learning-based termination analyzer. **LF** is used to represent LoopFrog, summary-based termination analyzer. **LR** is used to represent LinearRankTerm, abstract interpretation-based termination analyzer. **CTA** is used to denote compositional termination analyzer. Symbol '+' means that the time is aggregated only when the tool proved the termination. ✓="termination proven". ⊘="termination not proven". N/A="not comparable". T/O="time out". **Tool** and **Time** show the name and the runtime of tools.

shows better performance than LinearRankTerm for all examples except example 6.

Figure 1(d) is the result on SNU real-time benchmark suite[3]. The original examples in the suite contain many trivial, non-nested loops of form `for(i=0; i<n; ++i){...}` (52 out of 107 loops). We leave them out and make suite contain only non-trivial, nested loops. We show in the figure the number of terminating loops in each example, which was found manually.

Figure 1(d) shows that LTA outperforms LoopFrog and CTA, both in efficiency and effectiveness. Note that there is no comparison between LinearRankTerm and LTA; we could not compare LTA with LinearRankTerm on the examples that have non-terminating loops since LinearRankTerm stops the analysis as soon as it finds any single termination bug. We report here the results on the examples with terminating loops only; for three such examples (bs, fft1k, and ludcmp), LinearRankTerm tool can prove only one example (bs) and it takes 0.59 seconds.

Our approach shows a promising result; even by a prototype implementation with a simple heuristic for atomic transition predicate generation, our tool outperforms other tools both in efficiency and effectiveness.

5 Related Work

Our work is inspired by the recent success of the algorithmic-learning-based approach to loop invariant inference [19–21]. In those papers, the problem of loop invariant generation is formulated as a problem of inferring an unknown quantifier-free formula. With a simple randomized mechanical teacher, a learning algorithm is adopted to infer an invariant for the given annotated loop. Instead of the costly fixpoint iteration, the learning algorithm revises its purported invariants by counterexamples from the teacher. The randomized teacher can guide the learning algorithm to find a loop invariant very efficiently since there are usually sufficiently many loop invariants.

Terminator [13] is the most prominent termination analyzer which is successfully applied to an industrial practice [1]. Using transition invariants [25], Terminator decomposes a termination problem of complex loops into easier ones. However, as reported in [13], the initial approach reveals that most of the analysis time is spent in reachability analysis that is to check if the current transition invariant reached a fixpoint.

Our work shares the same goal as several techniques [2, 8, 22, 27] which aims to improve the performance of the initial approach. To compute fixpoints efficiently, Berdine et al. [2] and Chawdhary et al. [8] use abstract interpretation [15]. We use in experiments LinearRankTerm [8] which adopts a new abstract domain tailored for termination proof. The new abstract domain is effective to prove the termination in most of the practical examples, but it simply gives up when a transition invariant of a loop is beyond its expressivity. Kroening et al. [22] and

[3] The original benchmark suite can be also found at `http://archi.snu.ac.kr/realtime/benchmark/`.

Tsitovich et al. [27] use compositional transition invariants. Compositional transition invariants are the ones that are closed under composition with themselves. If a transition invariant covers several iterations of a loop and is compositional, it covers the entire iterations. Since compositional transition invariants can be found by considering only several iterations, they are sometimes discovered earlier than the one that covers the entire iterations. However, not all terminating programs have a compositional transition invariant.

Our technique can be easily extended with more sophisticated ranking function synthesis algorithms, such as lexicographic linear ranking functions [4] or bit-vector relations [11]. In this paper we use the ranking function synthesis algorithm for simple linear loops [24], which has been proven to be effective on realistic programs.

6 Conclusion

In this paper, we present an algorithmic-learning-based termination analysis technique. By combining transition predicate abstraction, algorithmic learning, and decision procedures, the technique can efficiently compute transition invariants as proofs of program termination. Compared to the previous approaches, our technique does not commit to any particular one, thus can prove the termination of the examples that previous techniques simply give up and report possible non-termination. We compare our technique with others on several benchmarks from literature. The result shows that the new technique outperforms the others both in efficiency and effectiveness.

Although our heuristic for selecting initial atomic transition predicates is effective, a complete predicate synthesis technique will be useful. Extending our learning-based framework to support more features such as function calls and pointers is certainly desirable. Several optimizations under the learning-based framework are to be explored. A more powerful well-foundedness checker should make the framework even more effective. An incremental learning algorithm for Boolean functions [9] should improve the efficiency of our technique as well.

Acknowledgement. We would like to thank anonymous referees for their comments and appreciations. We are grateful to Aziem Chawdhary, Peter O'Hearn, and Hongseok Yang for letting us use Windows device drivers examples. Especially, Aziem helps us a lot on the comparison with LinearRankTerm analyzer. We also thank to Hakjoo Oh, Daejun Park, Yungbum Jung, Deokhwan Kim, and Sungkeun Cho for their comments.

References

1. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys (2006)

2. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.: Variance analyses from invariance analyses. In: POPL (2007)
3. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.W.: Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006)
4. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear Ranking with Reachability. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 491–504. Springer, Heidelberg (2005)
5. Bradley, A.R., Manna, Z., Sipma, H.B.: The Polyranking Principle. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1349–1361. Springer, Heidelberg (2005)
6. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination of Polynomial Programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 113–129. Springer, Heidelberg (2005)
7. Bshouty, N.H.: Exact learning Boolean function via the monotone theory. Information and Computation 123(1), 146–153 (1995)
8. Chawdhary, A., Cook, B., Gulwani, S., Sagiv, M., Yang, H.: Ranking Abstractions. In: Gairing, M. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 148–162. Springer, Heidelberg (2008)
9. Chen, Y.F., Wang, B.Y.: Learning Boolean Functions Incrementally. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 55–70. Springer, Heidelberg (2012)
10. Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: Proving Conditional Termination. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 328–340. Springer, Heidelberg (2008)
11. Cook, B., Kroening, D., Rümmer, P., Wintersteiger, C.M.: Ranking Function Synthesis for Bit-Vector Relations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 236–250. Springer, Heidelberg (2010)
12. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction Refinement for Termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)
13. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI (2006)
14. Cook, B., Podelski, A., Rybalchenko, A.: Proving thread termination. In: PLDI (2007)
15. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
16. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
17. Harris, W.R., Lal, A., Nori, A.V., Rajamani, S.K.: Alternation for Termination. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 304–319. Springer, Heidelberg (2010)
18. Heizmann, M., Jones, N.D., Podelski, A.: Size-Change Termination and Transition Invariants. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 22–50. Springer, Heidelberg (2010)
19. Jung, Y., Kong, S., Wang, B.-Y., Yi, K.: Deriving Invariants by Algorithmic Learning, Decision Procedures, and Predicate Abstraction. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 180–196. Springer, Heidelberg (2010)

20. Jung, Y., Lee, W., Wang, B.-Y., Yi, K.: Predicate Generation for Learning-Based Quantifier-Free Loop Invariant Inference. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 205–219. Springer, Heidelberg (2011)
21. Kong, S., Jung, Y., David, C., Wang, B.-Y., Yi, K.: Automatically Inferring Quantified Loop Invariants by Algorithmic Learning from Simple Templates. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 328–343. Springer, Heidelberg (2010)
22. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination Analysis with Compositional Transition Invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
23. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
24. Podelski, A., Rybalchenko, A.: A Complete Method for the Synthesis of Linear Ranking Functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
25. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS (2004)
26. Podelski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. In: POPL (2005)
27. Tsitovich, A., Sharygina, N., Wintersteiger, C.M., Kroening, D.: Loop Summarization and Termination Analysis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 81–95. Springer, Heidelberg (2011)