Certifying Safety and Termination Proofs for Integer Transition Systems*

Marc Brockschmidt[1], Sebastiaan J.C. Joosten[2], René Thiemann[2], and Akihisa Yamada[2]

[1] Microsoft Research Cambridge, UK
[2] University of Innsbruck, Austria

Abstract. Modern program analyzers translate imperative programs to an intermediate formal language like integer transition systems (ITSs), and then analyze properties of ITSs. Because of the high complexity of the task, a number of incorrect proofs are revealed annually in the Software Verification Competitions.
In this paper, we establish the trustworthiness of termination and safety proofs for ITSs. To this end we extend our Isabelle/HOL formalization IsaFoR by formalizing several verification techniques for ITSs, such as invariant checking, ranking functions, etc. Consequently the extracted certifier CeTA can now (in)validate safety and termination proofs for ITSs. We also adapted the program analyzers T2 and AProVE to produce machine-readable proof certificates, and as a result, most termination proofs generated by these tools on a standard benchmark set are now certified.

1 Introduction

A number of recently introduced techniques for proving safety or termination of imperative programs, such as Java [1,29,32] and C [6,16,34], rely on a two-step process: the input program is abstracted into an intermediate formal language, and then properties of the intermediate program are analyzed. These intermediate languages are usually variations of integer transition systems (ITSs), reflecting the pervasive use of built-in integer data types in programming languages, as well as common abstractions like modeling algebraic datatypes by their size. For example, the C program in Fig. 1 can be abstracted to the ITS in Fig. 2.

To establish the *trustworthiness* of such program analyzers, two problems need to be tackled. First, the soundness of the translation from the source programming language to ITSs needs to be proven, requiring elaborate models that capture the semantics of advanced programming languages [21,24,39]. Then, the soundness of safety and termination proofs on ITSs needs to be validated.

* This work was partially supported by FWF project Y757. The authors are listed in alphabetical order regardless of individual contributions or seniority. We thank the anonymous reviewers for their helpful comments.

```
int x, y, z;
z = -1;
while (x >= -5) {
  x = x + z;
  y = 0;
  z = z - 1;
  while (y < x)
    y = y + 1;
}
```

Fig. 1: Input C program

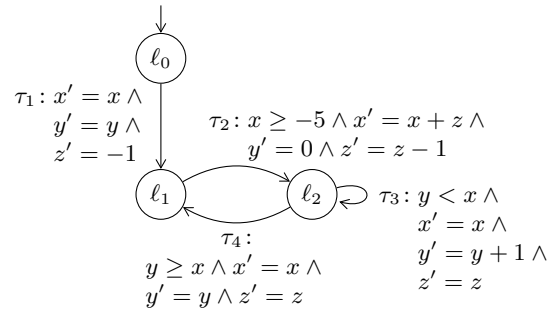

Fig. 2: ITS $\mathcal{P}$ corresponding to Fig. 1

In this work, we tackle the second problem by extending IsaFoR [35], the Isabelle Formalization (originally) of Rewriting, by termination and safety proving techniques for ITSs. We then export verified code for the *certifier* CeTA, which validates proof *certificates* generated by untrusted program analyzers. In order for CeTA to read proofs for ITSs, we extend an XML certificate format [33] with syntax for ITS inputs and various proof techniques. Moreover, we adapt the program analyzers AProVE [18] and T2 [9] to produce certificates following the XML format.

The rest of the paper is organized as follows. In Sect. 2, we formalize *logic transition systems (LTSs)*, a generalization of ITSs. The termination and safety proofs are developed on LTSs, so that we can easily extend our results to bit vectors, arrays, etc. A number of approaches reduce the termination analysis problem to a sequence of program safety problems that derive suitable invariants [6,8,14,37]. Thus in Sect. 3, we formalize program invariant proofs as generated by the Impact algorithm [26], yielding a certifier for safety proofs. In Sect. 4 we consider certifying termination proofs. We recapitulate and formalize the concept of cooperation programs [8] and then present how to certify termination proofs. To instantiate the general results to ITSs, in Sect. 5 we discuss how to reason about linear integer arithmetic. In Sect. 6 we report on an experimental evaluation, showing that a large number of termination proofs can now be certified.

This paper describes what program analyzers need to provide in a proof certificate, and what CeTA has to check to certify them. As all proofs are checked by Isabelle [27], we have omitted them from this paper. The full formalization, consisting of around 10 000 lines of Isabelle code and an overview that links theorems as stated in this paper to the actual formalization is available at `http://cl-informatik.uibk.ac.at/ceta/experiments/lts`. The website further contains certificates for the two termination proofs of the ITS $\mathcal{P}$ in Fig. 2 that are developed in this paper.

Related Work: A range of methods has been explored to improve the trustworthiness of program analyzers. Most related to this work is the *certification* of termination and complexity proofs of term rewrite systems [5,12,35]. Here

certification means to validate output of untrusted tools using a trustable certifier whose soundness is formally verified. Although our work is built upon one of them [35], the techniques for ITSs required a substantial addition to the library. SparrowBerry [11] follows a similar approach to validate numerical program invariants obtained by abstract interpretation [15] of C programs.

A less formal approach, taken in the context of complexity [2] and safety [3] proofs, is to cross-check a tool output using another (unverified) tool. A weakness of this approach is that, even if a "cross-checker" accepts a proof, it does not mean the proof is fully trustable. The cross-checker may have a bug, and both tools may be based on wrong (interpretations of) paper proofs. In contrast, we aim at termination, and we have formally proven in Isabelle that if our certifier accepts a proof, then the proof is indeed correct.

Another approach is to develop fully verified program analyzers, in which all intermediate steps in the proof search are formalized and verified, and not only the final proof as in our case. Examples of this approach have been used to develop a static analyzer for numerical domains [20] and to validate a Java Bytecode-like intermediate language in JINJA [21]. Compared to this approach, our certification approach demands much less work on the tool developers: they only have to output proofs that comply with the certificate format.

2 Logic Transition Systems

While our goal is specific to linear integer arithmetic, the used techniques apply to other logics as well. We separate the generic parts from the logic-specific parts. This clarifies the explanation of the generic parts, and makes it easier to extend our development to other logics in the future. We assume a sound validity checker for clauses (disjunctions of atoms) of the underlying logic. Linear integer arithmetic (i.e., Presburger arithmetic) can be considered as the canonical instance, but one may consider bit vectors, arrays, etc.

A logic describes how to interpret formulas of a certain shape. We first formalize the notion of *many-sorted algebras* [10,38] and formulas over them. We base our development on the (untyped) *term* datatype in IsaFoR.

Definition 1. *A* many-sorted signature Σ *consists of a set* $\mathcal{S}$ *of* sorts *and a disjoint family that assigns a set* $\Sigma_{\sigma_1 \cdots \sigma_n \sigma}$ *of function symbols to each list* $\sigma_1, \ldots, \sigma_n, \sigma \in \mathcal{S}$ *of sorts. A* sorted variable *is a pair of a variable symbol* v *and a sort* σ *(written* $v : \sigma$*, or just* v *when the sort is clear from the context). Given a set* $\mathcal{V}$ *of sorted variables, the set* $\mathcal{T}_\sigma(\mathcal{V})$ *of* expressions *of sort* σ *is defined inductively as follows:* $v : \sigma \in \mathcal{T}_\sigma(\mathcal{V})$*, and* $f(e_1, \ldots, e_n) \in \mathcal{T}_\sigma(\mathcal{V})$ *if* $f \in \Sigma_{\sigma_1 \cdots \sigma_n \sigma}$ *and* $e_i \in \mathcal{T}_{\sigma_i}(\mathcal{V})$ *for each* $i = 1, \ldots, n$*.*

Definition 2. *A* many-sorted Σ-algebra $\mathcal{A}$ *specifies the domain* A_σ *of each sort* $\sigma \in \mathcal{S}$ *and an interpretation* $[\![f]\!] : A_{\sigma_1} \times \cdots \times A_{\sigma_n} \to A_\sigma$ *of each* $f \in \Sigma_{\sigma_1 \cdots \sigma_n \sigma}$*.*

An assignment α *on a set* $\mathcal{V}$ *of sorted variables assigns each variable* $v : \sigma$ *a value* $\alpha(v) \in A_\sigma$*. We define the interpretation* $[\![e]\!]_\alpha$ *of an expression* e *under* α *as usual:* $[\![v]\!]_\alpha = \alpha(v)$ *and* $[\![f(e_1, \ldots, e_n)]\!]_\alpha = [\![f]\!]([\![e_1]\!]_\alpha, \ldots, [\![e_n]\!]_\alpha)$*.*

Definition 3. *We define a* many-sorted logic Λ *as a tuple consisting of a set of sorts* $\mathcal{S}$*, a many-sorted signature* Σ *on* $\mathcal{S}$*, and a* Σ*-algebra* $\mathcal{A}$ *such that* $\mathsf{bool} \in \mathcal{S}$ *and* $\mathsf{true}, \mathsf{false} \in A_{\mathsf{bool}}$. Formulas $\Lambda(\mathcal{V})$ *over typed variables from* $\mathcal{V}$ *are defined by the grammar* $\phi ::= a \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi$,[3] *where an* atom $a \in \mathcal{T}_{\mathsf{bool}}(\mathcal{V})$ *is an expression of sort* bool.

We say an assignment α satisfies *a formula* ϕ*, written* $\alpha \models \phi$*, if* ϕ *evaluates to* true *when every atom* a *in the formula is replaced by* $[\![a]\!]_\alpha$*. We write* $\phi \models \psi$ *iff* $\alpha \models \phi$ *implies* $\alpha \models \psi$ *for every assignment* α*.*

We define the notion of *logic transition systems (LTSs)* over a logic Λ. Note that an LTS can be seen as a labeled transition system, which also is commonly abbreviated to LTS.

In the following, we fix a set $\mathcal{L}$ of *locations* in a program and a set $\mathcal{V}$ of *variables* that may occur in the program.

Definition 4. *A* state *is a pair of* $\ell \in \mathcal{L}$ *and an assignment* α *on* $\mathcal{V}$*.*

To define state transitions, we introduce a fresh variable v' for each variable $v \in \mathcal{V}$. We write $\mathcal{V}'$ for the set $\{v' \mid v \in \mathcal{V}\}$, α' for the assignment on $\mathcal{V}'$ defined as $\alpha'(v') = \alpha(v)$, and e' (resp. ϕ') for the expression e (resp. formula ϕ) where all variables v are replaced by v'.

Definition 5. *A* transition rule *is a triple of* $\ell, r \in \mathcal{L}$ *and a* transition formula $\phi \in \Lambda(\mathcal{V} \uplus \mathcal{V}')$,[4] *written* $\ell \xrightarrow{\phi} r$*. A* logic transition system (LTS) $\mathcal{P}$ *is a set of transition rules, coupled with a special location* $\ell_0 \in \mathcal{L}$ *called the* initial location.

In the rest of the paper, we always use ℓ_0 as the initial location. Hence we identify an LTS and the set of its transition rules.

For our formalization, we extend LTSs with *assertions*, i.e., a mapping Φ that assigns a formula describing all valid states to each location. We assume no assertions for an input LTS, i.e., $\Phi(\ell) = \mathsf{true}$ for every $\ell \in \mathcal{L}$.

Definition 6. *The* transition step $\to_\tau$ *w.r.t. a transition rule* $\tau : \ell \xrightarrow{\phi} r$ *and an assertion* Φ *is defined by* $(\ell, \alpha) \to_\tau (r, \beta)$ *iff* $\alpha \uplus \beta' \models \phi$*, where* $\alpha \models \Phi(\ell)$ *and* $\beta \models \Phi(r)$*. For an LTS* $\mathcal{P}$*, we write* $\to_{\mathcal{P}} = \bigcup_{\tau \in \mathcal{P}} \to_\tau$*.*

Throughout the paper we establish methods that reduce a desired property of an LTS to zero or more subproblems of proving properties of refined LTSs. Hence the certificate forms a proof tree, where the root concludes the desired property of the input LTS, and all leafs are concluded by methods that yield no more subproblems.

In the formalization, corresponding theorems assume that LTSs are *well-typed*, i.e., atoms in transition formulas and assertions are of type bool. Well-typedness is checked only when an input LTS is given, and is statically proven for LTSs which are introduced as subproblems.

[3] In the Isabelle formalization and the certificate XML, formulas are represented in negation normal form and conjunction and disjunction are not necessarily binary.

[4] In the Isabelle formalization we admit *auxiliary* variables to appear in the transition formula. To ease readability we omit this ability in the paper.

3 Certifying Invariants and Safety Proofs

The safety of a program means that certain "bad" states cannot be reached, and is usually modeled by a set of *error locations* $\mathcal{L}_\lightning \subseteq \mathcal{L}$ that are reached from such bad states. Safety then reduces to the unreachability of error locations.

Definition 7. *We say a state* (ℓ_n, α_n) *is* reachable *if there is a sequence of transition steps starting from the initial location:* $(\ell_0, \alpha_0) \to_\mathcal{P} \cdots \to_\mathcal{P} (\ell_n, \alpha_n)$*. A location* ℓ_n *is* reachable *if there is a reachable state* (ℓ_n, α_n)*.*

A *program invariant* maps every $\ell \in \mathcal{L}$ to a formula ϕ such that $\alpha \models \phi$ for all reachable states (ℓ, α). Program safety can thus be proven by finding a program invariant that maps every error location to an unsatisfiable formula (e.g., false).

Definition 8. *We say a mapping* $\mathcal{I} : \mathcal{L} \to \Lambda(V)$ *is an* invariant *of an LTS* $\mathcal{P}$ *iff* $\alpha \models \mathcal{I}(\ell)$ *whenever* (ℓ, α) *is reachable in* $\mathcal{P}$*.*

One way to prove that a mapping is an invariant is to find an *unwinding* [26] of a program. We integrate support for invariant checking and safety proofs into CeTA by formalizing unwindings in Isabelle.

Definition 9. *An* unwinding *of LTS* $\mathcal{P}$ *under assertion* Φ *is a graph* $\mathcal{G} = (N_t \cup N_c, \longrightarrow \cup \dashrightarrow)$ *with* $N_t \cup N_c \subseteq \mathcal{L} \times \Lambda(\mathcal{V})$*, where nodes in* N_t *are called* transition nodes*, those in* N_c *are* covered nodes*, edges in* $\longrightarrow$ *are* transition edges*, those in* $\dashrightarrow$ *are* cover edges*, and the following conditions are satisfied:*

1. $(\ell_0, \mathsf{true}) \in N_t \cup N_c$*;*
2. *for every transition node* $(\ell, \phi) \in N_t$*, either* ϕ *is unsatisfiable or for every transition rule* $\ell \xrightarrow{\chi} r \in \mathcal{P}$ *there is a transition edge* $(\ell, \phi) \longrightarrow (r, \psi)$ *such that* $\Phi(\ell) \wedge \phi \wedge \chi \models \psi'$*;*
3. *for every covered node* $(\ell, \phi) \in N_c$*, there exists exactly one outgoing edge, which is a cover edge* $(\ell, \phi) \dashrightarrow (\ell, \psi)$ *with* $(\ell, \psi) \in N_t$ *and* $\phi \models \psi$*.*

Each node (ℓ, ϕ) in an unwinding represents the set of states $\{(\ell, \alpha) \mid \alpha \models \phi\}$. If ϕ is unsatisfiable then the node represents no state, and thus no successor of that node needs to be explored in condition 2. A location ℓ in the original program is represented by multiple transition nodes $(\ell, \phi_1), \ldots, (\ell, \phi_n)$, meaning that $\phi_1 \vee \ldots \vee \phi_n$ is a disjunctive invariant in ℓ.

Example 1. Consider the LTS $\mathcal{P}$ in Fig. 2 again. In order to prove the termination of the outer while loop, the invariant $z < 0$ in location ℓ_1 is essential. To prove this invariant, we use the graph $\mathcal{G}$ in Fig. 3, a simplified version of the unwinding constructed by the Impact algorithm [26].

Our definition of unwindings only roughly follows the original definition [26], in which nodes and transition edges are specified as a *tree* and cover edges are given as a separate set. This turned out to be unwieldy in the formalization; our definition is not restricted to trees, since being a tree or not is irrelevant for

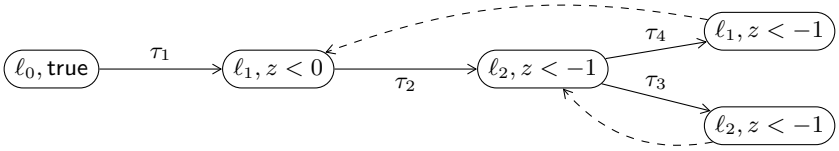

Fig. 3: An unwinding $\mathcal{G}$ of $\mathcal{P}$

soundness. This flexibility gives some benefits. For instance, instead of introducing an additional node in Fig. 3 as the target of τ_4, the corresponding transition edge could just point back to the node $(\ell_1, z < 0)$. More significantly, we are able to certify invariants obtained by other means (e.g., abstract interpretation [15]). For this, an inductive invariant $\mathcal{I} : \mathcal{L} \to \Lambda(\mathcal{V})$ can be cast as an unwinding with transition nodes $\{(\ell, \mathcal{I}(\ell)) \mid \ell \in \mathcal{L}\}$ and transition edges $\{(\ell, \mathcal{I}(\ell)) \longrightarrow (r, \mathcal{I}(r)) \mid \ell \xrightarrow{\phi} r \in \mathcal{P}\}$, with no covered nodes.

Theorem 1 (Invariants via Unwindings). *Let $\mathcal{G}$ be an unwinding for an LTS $\mathcal{P}$. Then a mapping $\mathcal{I} : \mathcal{L} \to \Lambda(V)$ is an invariant of $\mathcal{P}$ if for every $l \in \mathcal{L}$,*

$$\left(\bigvee_{(\ell,\phi)\in N_t} \phi\right) \models \mathcal{I}(\ell) \tag{1}$$

To verify that Thm. 1 is applied correctly, CeTA needs the invariant $\mathcal{I}$ and the unwinding $\mathcal{G}$ to be specified in the certificate. It checks conditions 1–3 of Def. 9, and then the entailment (1) for each location. For efficiency we further assume that each transition edge is annotated by the corresponding transition rule.

To use invariants in proofs for the desired properties (safety or termination), we turn invariants into *assertions*. As we have proven that the invariant formula is satisfied whenever a location is visited, "asserting" the formula merely makes implicit global information available at each location. This approach has the advantage that invariants become a part of input for the later proofs, and thus we do not have to prove that they are invariant repeatedly when we transform programs as required in Sect. 4.

Theorem 2 (Invariant Assertion for Safety Proofs). *Let $\mathcal{P}$ be an LTS, Φ an assertion on $\mathcal{P}$, and Ψ an invariant of $\mathcal{P}$. Then $\mathcal{P}$ under assertion Φ is safe if $\mathcal{P}$ under assertion Ψ is safe.*

Thm. 2 requires nothing to be checked, besides that the invariant is certified via Thm. 1. A typical application of Thm. 2 refines the current assertion Φ by a stronger one Ψ. When sufficiently strong assertions are made, one can conclude safety as follows.

Theorem 3 (Safety by Assertions). *Let $\mathcal{P}$ be an LTS, Φ an assertion on $\mathcal{P}$, and $\mathcal{L}_\lightning$ a set of error locations. If $\Phi(\ell)$ is unsatisfiable for every $\ell \in \mathcal{L}_\lightning$, then $\mathcal{P}$ under assertion Φ is safe.*

4 Certifying Termination Proofs

Now we present our formalization of techniques for proving termination of LTSs.

Definition 10. *An LTS $\mathcal{P}$ is* terminating *iff there exists no infinite transition sequence starting from the initial location:* $(\ell_0, \alpha_0) \to_{\mathcal{P}} (\ell_1, \alpha_1) \to_{\mathcal{P}} \cdots$.

We formalize a collection of transformation techniques for LTSs such that every transformation preserves nontermination, i.e., the termination of all resulting LTSs implies the termination of the original LTS. The *cooperation graph* method [8], the foundation of the termination tool T2, will be modeled by a combination of such transformations. The split into smaller techniques not only simplifies and modularizes the formalization task, but also provides a way to certify termination proofs of tools which use related methods. They can choose a subset of supported termination techniques that is sufficient to model their internally constructed termination proofs. For instance, we also integrate certificate export for LTSs in AProVE, which internally does *not* utilize cooperation graphs.

4.1 Initial Transformation

The key component of cooperation graphs is the use of two copies of the program: the original part is used to describe reachable program states, and a termination part is progressively modified during the termination proof. This approach makes it possible to apply transformations which would be unsound when performed on the original program; e.g., one can remove transitions from the termination part if they are proven to be usable only a finite number of times. This is unsound if it is performed on the original program; consider, e.g., a non-terminating LTS consisting of only the two transition rules $\ell_0 \xrightarrow{\mathsf{true}} \ell_1$ and $\ell_1 \xrightarrow{\mathsf{true}} \ell_1$. Clearly, the first transition rule can be applied only once. Nevertheless, if it is removed, ℓ_1 becomes unreachable, and the resulting LTS is terminating.

To describe the copies of programs, we introduce a fresh location $\ell^\sharp$ for each location $\ell \in \mathcal{L}$. We write $\mathcal{L}^\sharp$ for the set $\{\ell^\sharp \mid \ell \in \mathcal{L}\}$.

Definition 11. *A* cooperation program *$\mathcal{Q}$ is an LTS on locations $\mathcal{L} \cup \mathcal{L}^\sharp$ which is split into three parts: $\mathcal{Q} = \mathcal{Q}^\natural \cup \mathcal{Q}^\sharp \cup \mathcal{Q}^{\sharp\sharp}$, where $\mathcal{Q}^\natural$, $\mathcal{Q}^\sharp$, $\mathcal{Q}^{\sharp\sharp}$ consist of transitions of form $\ell \xrightarrow{\phi} r$, $\ell \xrightarrow{\phi} r^\sharp$, $\ell^\sharp \xrightarrow{\phi} r^\sharp$, respectively, where $\ell, r \in \mathcal{L}$.*

We say $\mathcal{Q}$ is CP-terminating *if there exists no infinite sequence of form*

$$(\ell_0, \alpha_0) \to_{\mathcal{Q}^\natural} \cdots \to_{\mathcal{Q}^\natural} (\ell_n, \alpha_n) \to_{\mathcal{Q}^\sharp} (\ell_n^\sharp, \alpha_n) \to_{\mathcal{Q}^{\sharp\sharp}} (\ell_{n+1}^\sharp, \alpha_{n+1}) \to_{\mathcal{Q}^{\sharp\sharp}} \cdots$$

where each transition rule used after the n-th step must be used infinitely often.

We call $\mathcal{Q}^\natural$ the *original part* and $\mathcal{Q}^{\sharp\sharp}$ the *termination part*. The termination of an LTS can be reduced to the CP-termination of a cooperation program which has the input LTS as the original part and its copy (where every location ℓ is renamed to $\ell^\sharp$) as the termination part, and additionally includes ϵ-*transitions* that allow to jump from locations ℓ to $\ell^\sharp$.

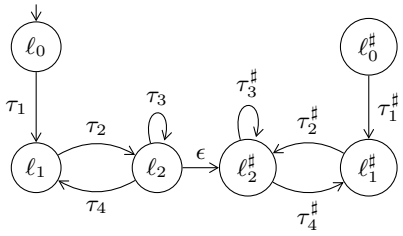

Fig. 4: Cooperation program $\mathcal{Q}$ constructed from $\mathcal{P}$

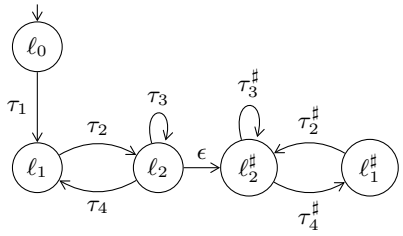

Fig. 5: Cooperation program $\mathcal{Q}_1$ resulting from SCC decomposition of $\mathcal{Q}$

Definition 12. *We say a transition rule $\tau : \ell \xrightarrow{\phi} r$ is an* ϵ-transition *iff $(\ell, \alpha) \to_\tau (r, \alpha)$ for any assignment α, i.e., $\alpha \uplus \alpha' \models \phi$. We write $\ell \xrightarrow{\epsilon} r$ to denote an ϵ-transition.*

Canonically, one can consider $\ell \xrightarrow{\epsilon} \ell^\sharp$ for *every location* ℓ, but we can also do a little better by employing the notion of *cutpoints*. For this, we view an LTS $\mathcal{P}$ as a *program graph* with nodes $\mathcal{L}$ and edges $\{(\ell, r) \mid \ell \xrightarrow{\phi} r \in \mathcal{P}\}$.

Definition 13. *A set $\mathcal{C} \subseteq \mathcal{L}$ of locations is a* cutpoint set *of an LTS $\mathcal{P}$ if the program graph of $\mathcal{P} \setminus \mathcal{C}$ is acyclic.*

Intuitively, if $\mathcal{C}$ is a cutpoint set of $\mathcal{P}$, then any infinite execution of $\mathcal{P}$ must visit some *cutpoint* $\ell \in \mathcal{C}$ infinitely often.

Theorem 4 (Initial Cooperation Program). *Let $\mathcal{P}$ be a finite LTS over $\mathcal{L}$, $\mathcal{Q}$ a cooperation program, and $\mathcal{C} \subseteq \mathcal{L}$ such that*

1. *for each $\ell \xrightarrow{\phi} r \in \mathcal{P}$, there exist $\ell \xrightarrow{\phi} r \in \mathcal{Q}$ and $\ell^\sharp \xrightarrow{\phi} r^\sharp \in \mathcal{Q}$;*
2. *for each $\ell \in \mathcal{C}$, there exists $\ell \xrightarrow{\epsilon} \ell^\sharp \in \mathcal{Q}$; and*
3. *$\mathcal{C}$ is a cutpoint set for $\mathcal{P}$.*

Then $\mathcal{P}$ is terminating if $\mathcal{Q}$ is CP-terminating.

Example 2. In order to construct an initial cooperation program for $\mathcal{P}$ in Fig. 2, termination provers need to choose a cutpoint set. Let us consider a minimal one: $\mathcal{C} = \{\ell_2\}$. We obtain the cooperation program $\mathcal{Q}$ in Fig. 4, where each transition $\tau_i^\sharp$ has the same transition formula as τ_i for $i = 1, \ldots, 4$ and ϵ has transition formula $x' = x \wedge y' = y \wedge z' = z$.

To check that Thm. 4 is applied correctly, we only require the added ϵ-transitions to be specified in the certificate. The other parts, e.g., the cutpoint set $\mathcal{C}$, are automatically inferred by CeTA.

Condition 1 of Thm. 4 is always fulfilled, since these transitions are automatically generated by CeTA, and statically proven correct.

For condition 2, CeTA checks if the transition formula is of form $\bigwedge_{v \in \mathcal{W}} v' = v$ for some set of variables $\mathcal{W} \subseteq \mathcal{V}$. Allowing $\mathcal{W} \subset \mathcal{V}$ can be useful: Consider a C

fragment `x = x + 1; x = 2 * x`. This might be encoded into a single transition formula using an auxiliary variable, e.g., as $aux = x + 1 \wedge x' = 2 * aux$. It would make sense not to mention the auxiliary variables in epsilon transitions.

For condition 3, i.e., to check that $\mathcal{C}$ is indeed a cutpoint set, we must check acyclicity of graphs as required in Def. 13. Luckily we could reuse the certified implementation of Gabow's *strongly connected component (SCC)* decomposition algorithm [23] and check that after removing $\mathcal{C}$ from $\mathcal{P}$, it has only trivial SCCs.

To reason about the termination of LTSs, we often require program invariants to allow us to reason about reachable program states. Thus, analogous to Thm. 2 for safety proofs, we provide a way to introduce program invariants in termination proofs. The following result is formalized both for normal LTSs w.r.t. Def. 10 as well for cooperation programs w.r.t. Def. 11.

Theorem 5 (Invariant Assertion for Termination). *Let $\mathcal{P}$ be an LTS, Φ an assertion on $\mathcal{P}$, and Ψ an invariant of $\mathcal{P}$. Then $\mathcal{P}$ under assertion Φ is (CP-)terminating if $\mathcal{P}$ under assertion Ψ is (CP-)terminating.*

4.2 SCC and Cutpoint Decompositions

In the setting of cooperation programs, it is sound to decompose the termination part into SCCs.

Theorem 6 (SCC Decomposition). *Given a cooperation program $\mathcal{Q}$, if the cooperation program $\mathcal{Q}^\natural \cup \mathcal{Q}^\sharp \cup \{\ell^\sharp \xrightarrow{\phi} r^\sharp \in \mathcal{Q}^{\sharp\sharp} \mid \ell^\sharp, r^\sharp \in \mathcal{S}\}$ is CP-terminating for every non-trivial SCC $\mathcal{S}$ of the program graph of $\mathcal{Q}^{\sharp\sharp}$, then $\mathcal{Q}$ is CP-terminating.*

To certify an application of Thm. 6, the certificate has to list the subproofs for each SCC. CeTA invokes the same certified SCC algorithm as in the cutpoint validation to check applications of the SCC decomposition.

Example 3. Using SCC decomposition, $\mathcal{Q}$ in Fig. 4 can be transformed into the new problem $\mathcal{Q}_1$ in Fig. 5, where location $\ell_0^\sharp$ and transition $\tau_1^\sharp$ are removed.

We can also decompose a cooperation program by case distinction depending on which ϵ-transition for a cutpoint is taken. This can also be used to delete ϵ-transitions leading to locations whose outgoing transitions have already been removed by other means.

Theorem 7 (Cutpoint Decomposition). *Let $\mathcal{P}$ be a cooperation program with $\mathcal{P}^\sharp = \mathcal{Q}_0^\sharp \cup \mathcal{Q}_1^\sharp \cup \ldots \cup \mathcal{Q}_n^\sharp$, where for every $\ell \xrightarrow{\psi} \ell^\sharp \in \mathcal{Q}_0^\sharp$ there is no transition rule of form $\ell^\sharp \xrightarrow{\phi} r^\sharp$ in $\mathcal{P}^{\sharp\sharp}$. Then $\mathcal{P}$ is CP-terminating if $\mathcal{P}^\natural \cup \mathcal{Q}_i^\sharp \cup \mathcal{P}^{\sharp\sharp}$ is CP-terminating for every $i = 1, \ldots, n$.*

A certificate for Thm. 7 needs to provide the considered partition $\mathcal{Q}_1^\sharp \cup \ldots \cup \mathcal{Q}_n^\sharp$ and a corresponding subproof for each of the newly created cooperation programs. CeTA determines $\mathcal{Q}_0^\sharp$ and checks that it has no succeeding transitions in $\mathcal{P}^{\sharp\sharp}$.

4.3 Transition Removal

A cooperation program is trivially CP-terminating if its termination part is empty. Hence we now formalize a way to remove transitions via rank functions, the core termination proving procedure for cooperation programs, and also for other termination methods as implemented by, e.g., AProVE or VeryMax [6].

Roughly speaking, a rank function is a mapping from program states to a mathematical domain on which a well-founded order exist (e.g., the natural numbers). We formalize such domains reusing a notion from term rewriting.

Definition 14. *We call a pair* $(\geq, >)$ *of relations a* (quasi-)order pair *if* $\geq$ *is reflexive, both are transitive, and they are "compatible", i.e.,* $(\geq \circ > \circ \geq) \subseteq >$. *We say that the order pair is* well-founded *if* $>$ *is well-founded.*

We model a rank function as a mapping that assigns an expression $f(\ell^\sharp) \in \mathcal{T}_\sigma(\mathcal{V})$ of sort σ to each location $\ell^\sharp$. Here we assume that the domain A_σ of σ has a well-founded order pair $(\geq, >)$. If some transitions in $\mathcal{Q}^{\sharp\sharp}$ strictly decrease a rank function and all other transitions "do not increase" this rank function, then the decreasing transitions can be used only finitely often, and thus can be removed from the termination part.

Theorem 8 (Transition Removal). *Let* $\mathcal{Q}$ *be a cooperation program with assertion* Φ, $(\geq, >)$ *a well-founded order pair*[5] *on* A_σ, $f : \mathcal{L}^\sharp \to \mathcal{T}_\sigma(\mathcal{V})$, *and* $\mathcal{D}^{\sharp\sharp} \subseteq \mathcal{Q}^{\sharp\sharp}$ *a set of transition rules such that for every* $\ell^\sharp \xrightarrow{\phi} r^\sharp \in \mathcal{Q}^{\sharp\sharp}$,

- $\Phi(\ell^\sharp) \wedge \Phi(r^\sharp)' \wedge \phi \models f(\ell^\sharp) > f(r^\sharp)'$ *if* $\ell^\sharp \xrightarrow{\phi} r^\sharp \in \mathcal{D}^{\sharp\sharp}$; *and*
- $\Phi(\ell^\sharp) \wedge \Phi(r^\sharp)' \wedge \phi \models f(\ell^\sharp) \geq f(r^\sharp)'$ *otherwise.*

Then $\mathcal{Q}$ *is CP-terminating if* $\mathcal{Q} \setminus \mathcal{D}^{\sharp\sharp}$ *is CP-terminating.*

To certify the correct application of Thm. 8, naturally the rank function and deleted transitions have to be specified in the certificate. For integer arithmetic σ is fixed to int, but one also needs to choose the well-founded order. Note that $>$ on integers is not well-founded, but its bounded variant $>_b$ is, where $s >_b t$ iff $s > t$ and $s \geq b$. Note also that $(\geq, >_b)$ forms an order pair.

Example 4. The program $\mathcal{P}$ from Fig. 2 can be shown terminating by repeatedly applying Thm. 8. Assume that we have applied Thm. 5 on $\mathcal{P}$ and established the assertion $z < -1$ on ℓ_2, based on the unwinding from Ex. 1, before transforming $\mathcal{P}$ into $\mathcal{Q}_1$ of Fig. 5. We then apply Thm. 8 with rank function x and bound -5 for all locations in $\mathcal{Q}_1^{\sharp\sharp}$. With the assertion $z < -1$, this allows us to remove $\tau_2^\sharp$. Then, using the constant rank functions 1 for $\ell_1^\sharp$ and 0 for $\ell_2^\sharp$, we can remove the transition $\tau_4^\sharp$ (alternatively, we could use SCC decomposition here). Finally, the rank function $x - y$ and bound 0 can be used to remove the last remaining transition $\tau_3^\sharp$.

[5] In the paper we use symbols $\geq$ and $>$ also for *formulas*. In the formalization we encode, e.g., by a formula $e_1 \geq_\mathsf{f} e_2$ such that $\alpha \models e_1 \geq_\mathsf{f} e_2$ iff $[\![e_1]\!]_\alpha \geq [\![e_2]\!]_\alpha$.

Simple rank functions on integers are sometimes too weak, so we also integrate *lexicographic* orderings.

Definition 15. *Given order pairs* $(\succsim_1, \succ_1), \ldots, (\succsim_n, \succ_n)$ *on* A*, their* lexicographic composition *is the order pair* $(\succsim^{\mathsf{lex}}_{1,\ldots,n}, \succ^{\mathsf{lex}}_{1,\ldots,n})$ *on length-*n *lists of* A *defined as follows:* $\langle x_1, \ldots, x_n\rangle \succ^{\mathsf{lex}}_{1,\ldots,n} \langle y_1, \ldots, y_n\rangle$ *iff*

$$\exists i \leq n.\ x_1 \succsim_1 y_1 \wedge \cdots \wedge x_{i-1} \succsim_{i-1} y_{i-1} \wedge x_i \succ_i y_i \tag{2}$$

and $\langle x_1, \ldots, x_n\rangle \succsim^{\mathsf{lex}}_{1,\ldots,n} \langle y_1, \ldots, y_n\rangle$ *iff* (2) *holds or* $x_1 \succsim_1 y_1 \wedge \cdots \wedge x_n \succsim_n y_n$*.*

The lexicographic composition of well-founded order pairs forms again a well-founded order pair. Hence, to conclude the correct application of Thm. 8, CeTA demands a list of bounds $b_1, \ldots, b_n$ to be given in the certificate, and then uses the lexicographic composition induced by bounded order pairs $(\geq, >_{b_1}), \ldots, (\geq, >_{b_n})$. An application is illustrated at the end of the next subsection in Ex. 6.

4.4 Variable and Location Additions

Transition removal is an efficient termination proving method, but relies on local syntactic structure of the program. Most significantly, it cannot find termination arguments that depend on interactions between succeeding transitions on a cycle. Safety-based termination proofs thus instead consider *evaluations* that represent a full cycle through a program SCC, from a cutpoint back to itself, and show that every such evaluation decreases some well-founded measure. In order to do this, a *snapshot variable* v_{s} is introduced for each program variable v and the program is extended to set v_{s} to the value of v on every transition leaving a cutpoint. Then, a rank function for the SCC satisfies $f(v_{1\mathsf{s}}, \ldots, v_{n\mathsf{s}}) > f(v_1, \ldots, v_n)$ whenever an evaluation reaches the cutpoint again. In our modified version of T2, we implement the setting of snapshot variables and checking of rank functions by adding dedicated fresh locations after and before a cutpoint.

Theorem 9 (Location Addition). *Let* $\mathcal{P}$ *be a cooperation program and* $\mathcal{Q}^{\sharp\sharp}$ *a set of transitions such that for every transition* $\ell^\sharp \xrightarrow{\phi} r^\sharp \in \mathcal{P}^{\sharp\sharp} \setminus \mathcal{Q}^{\sharp\sharp}$ *there exists a location* f *such that* $\ell^\sharp \xrightarrow{\phi} f, f \xrightarrow{\epsilon} r^\sharp \in \mathcal{Q}^{\sharp\sharp}$ *or* $\ell^\sharp \xrightarrow{\epsilon} f, f \xrightarrow{\phi} r^\sharp \in \mathcal{Q}^{\sharp\sharp}$*. Then* $\mathcal{P}$ *is CP-terminating if* $\mathcal{P}^\natural \cup \mathcal{P}^\sharp \cup \mathcal{Q}^{\sharp\sharp}$ *is CP-terminating.*

In certificates the new component $\mathcal{Q}^{\sharp\sharp}$ does not have to be provided. Instead it suffices to provide the new ϵ-transition $f \xrightarrow{\epsilon} r^\sharp$ (resp. $\ell^\sharp \xrightarrow{\epsilon} f$) with fresh location f. Then $\mathcal{Q}^{\sharp\sharp}$ is computed from $\mathcal{P}^{\sharp\sharp}$ by redirecting every transition with target $r^\sharp$ (resp. source $\ell^\sharp$) towards f.

Example 5. Here and in Ex. 6, we provide an alternative termination proof for the cooperation program $\mathcal{Q}_1$ of Fig. 5. We use the global reasoning that every cycle from $\ell_2^\sharp$ back to itself decreases the measure $\langle x, x - y\rangle$, bounded by -5 and 0 respectively. Note that x decreases in every iteration of the outer loop, and $x - y$ decreases in every iteration of the inner loop.

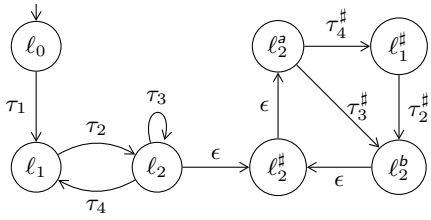

Fig. 6: Cooperation program $\mathcal{Q}_2$ resulting from $\mathcal{Q}_1$ by adding a location ℓ_2^a after $\ell_2^\sharp$ and a location ℓ_2^b before $\ell_2^\sharp$

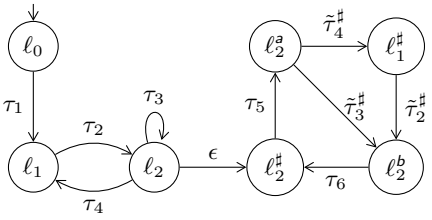

Fig. 7: Cooperation program $\mathcal{Q}_3$ resulting from $\mathcal{Q}_2$ by adding snapshot variables

As a first step, we transform $\mathcal{Q}_1$ into $\mathcal{Q}_2$ of Fig. 6 by applying Thm. 9 twice, providing the transitions $\ell_2^b \xrightarrow{\epsilon} \ell_2^\sharp$ and $\ell_2^\sharp \xrightarrow{\epsilon} \ell_2^a$ to introduce fresh locations before and after the cutpoint $\ell_2^\sharp$.

The addition of snapshot variables is not trivially sound, as the operation involves *strengthening* transition formulas, e.g., from ϕ to $\phi \wedge x'_{\mathsf{s}} = x$. Thus to ensure soundness, CeTA demands the new variable x_{s} and the formula added to each transition, and checks that no existing transition formula mentions x'_{s}, and the added formulas do nothing more than giving a value to x'_{s}. The latter condition is more precisely formulated as follows.

Definition 16. *We say a variable x of sort σ is* definable *in a formula ψ iff for any assignment α, there exists $v \in A_\sigma$ such that $\alpha[x \mapsto v] \models \psi$, where $\alpha[x \mapsto v]$ maps x to v and $y \neq x$ to $\alpha(y)$.*

Theorem 10 (Variable Addition). *Let $\mathcal{P}$ and $\mathcal{Q}$ be cooperation programs, x a variable, and Ψ a mapping from transitions to formulas, such that for every transition $\tau : \ell \xrightarrow{\phi} r \in \mathcal{P}$, x does not occur in ϕ and there exists $\ell \xrightarrow{\phi \wedge \Psi(\tau)} r \in \mathcal{Q}$ where x is definable in $\Psi(\tau)$. Then $\mathcal{P}$ is CP-terminating if $\mathcal{Q}$ is CP-terminating.*

Example 6. We can transform $\mathcal{Q}_2$ to $\mathcal{Q}_3$ in Fig. 7. Here, each $\tilde{\tau}_i^\sharp$ extends $\tau_i^\sharp$ to keep the values of $x_{\mathsf{s}}, y_{\mathsf{s}}, z_{\mathsf{s}}$ unchanged; i.e., $x'_{\mathsf{s}} = x_{\mathsf{s}} \wedge y'_{\mathsf{s}} = y_{\mathsf{s}} \wedge z'_{\mathsf{s}} = z_{\mathsf{s}}$ is added to the transition formulas. The transition τ_6 keeps all variables unchanged, and τ_5 initializes the snapshot variables: $\ldots \wedge x'_{\mathsf{s}} = x \wedge y'_{\mathsf{s}} = y \wedge z'_{\mathsf{s}} = z$. This transformation is achieved by repeatedly adding snapshot variables x_{s}, y_{s}, and z_{s}. To add x_{s}, for instance, we apply Thm. 10 on $\mathcal{Q}_2$ with $\Psi(\ell_2^\sharp \xrightarrow{\epsilon} \ell_2^a) = \big(x'_{\mathsf{s}} = x\big)$ and $\Psi(\tau) = \big(x'_{\mathsf{s}} = x_{\mathsf{s}}\big)$ for all other transitions $\tau \in \mathcal{Q}_2^{\sharp\sharp}$.

Every cycle from $\ell_2^\sharp$ back to itself decreases the bounded measure $\langle x, x - y\rangle$, so we are able to remove the transition τ_6 by Thm. 8, using the rank function f with $f(\ell_2^\sharp) = \langle x, x - y\rangle$ and $f(\ell^\sharp) = \langle x_{\mathsf{s}}, x_{\mathsf{s}} - y_{\mathsf{s}}\rangle$ for all other locations $\ell^\sharp \neq \ell_2^\sharp$. To this end we need to be able to show $\langle x_{\mathsf{s}}, x_{\mathsf{s}} - y_{\mathsf{s}}\rangle >^{\mathsf{lex}}_{-5,0} \langle x, x - y\rangle$ for τ_6. Weak decreases required for other transitions are immediate from the transition formulas. So we need an invariant on ℓ_2^b that is strong enough to prove

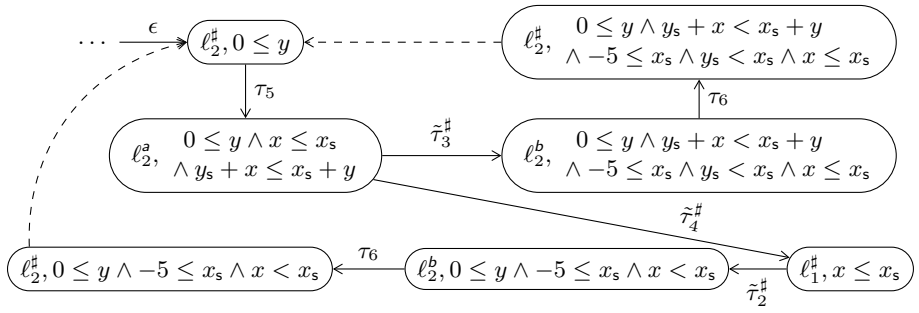

Fig. 8: Partial unwinding of $\mathcal{Q}_2$ from Fig. 6

$\langle x_\mathsf{s}, x_\mathsf{s} - y_\mathsf{s}\rangle >^{\mathsf{lex}}_{-5,0} \langle x, x - y\rangle$. To this end the following invariant works, and can be proven using the unwinding (partially) shown in Fig. 8.

$$(0 \le y \wedge y_\mathsf{s} + x < x_\mathsf{s} + y \wedge -5 \le x_\mathsf{s} \wedge y_\mathsf{s} < x_\mathsf{s} \wedge x \le x_\mathsf{s}) \\ \vee\, (0 \le y \wedge -5 \le x_\mathsf{s} \wedge x < x_\mathsf{s})$$

Having τ_6 removed, $\mathcal{Q}_3^{\sharp\sharp}$ contains no SCC anymore, and thus SCC decomposition (requiring no further subproofs) finishes the proof.

5 Linear Integer Arithmetic

In the preceding sections we have assumed that we can certify entailments $\psi \models \chi$, i.e., the validity of formulas $\neg\psi \vee \chi$. In this section, we provide such a validity checker when the underlying logic is linear integer arithmetic. Note that although Isabelle has already builtin support for reasoning about linear arithmetic, we cannot use these results: Isabelle *tactics* like `linarith` and `presburger` are not accessible to CeTA, since CeTA is a stand-alone Haskell program that has been constructed via code generation from Isabelle, and it has no access to Isabelle tactics at all.

5.1 Reduction to Linear Programming

As the initial step, CeTA converts the input formula (whose validity has to be verified) into conjunctive normal form (CNF). Note that here we cannot use the Tseitin transformation [36] since we are interested in checking validity, not satisfiability. By default, CeTA completely distributes disjunctions to obtain CNFs, but we also provide a "hint" format to indicate that some part of the formula should be erased or to explicitly apply distributivity rules at some position.

Next, we ensure the validity of a CNF by equivalently checking the validity of every clause. Hence, the underlying logic should provide at least a validity

checker for disjunctions of literals, or equivalently an unsatisfiability checker for conjunctions of literals.

For linear integer arithmetic, all literals can be translated into inequalities of the form $e \geq 0$ by using straightforward rules such as $\neg(e_1 \leq e_2) \hookrightarrow e_1 - e_2 - 1 \geq 0$. Thus, we only need to prove unsatisfiability of a conjunction of linear inequalities, a question in the domain of integer linear programming (ILP).

Since the unsatisfiability of ILP instances is a coNP-complete problem [30, Chapter 18], there is little hope in getting small certificates which are easy to check. We provide two alternatives. Both interpret ILPs as linear programming problems (LPs) over $\mathbb{Q}$, not $\mathbb{Z}$, and thus are incomplete but sound, in the sense that the resulting LP might be satisfiable although the input ILP is unsatisfiable, but not vise versa. In our experiments the incompleteness was never encountered when certifying proofs generated by AProVE and T2.

Simplex Algorithm: The first alternative employs the existing Isabelle formalization of the simplex algorithm by Spasić and Marić [31]. We only had to manually rebase the formalization from Isabelle 2012 to Isabelle 2016-1, and then establish a connection between the linear rational inequalities as formalized by Spasić and Marić and our linear integer inequalities.

Farkas' Lemma: The second alternative demands the certificate to provide the coefficients as used in Farkas' Lemma [17]: Given an LP constraint $e_1 \geq 0 \wedge \ldots \wedge e_n \geq 0$ and a list of non-negative coefficients $\lambda_1, \ldots, \lambda_n$, we conclude $\sum_{i=1}^{n} \lambda_i e_i \geq 0$ and then check that this inequality is trivially unsatisfiable, i.e., that $\sum_{i=1}^{n} \lambda_i e_i$ is a negative constant. It is well known that this criterion is of the same power as the first alternative. The advantage of this alternative is that it is faster to validate—at the cost of more demanding certificates.

5.2 Executable Certifier for ITSs

To summarize, we developed a validity checker for formulas in linear integer arithmetic, whose correctness is formally proven. Hence we derive an executable checker for the correct application of Thms. 1–10 on linear ITSs. Thus CeTA is now able to certify safety and termination proofs for linear ITSs.

Corollary 1 (Safety and Termination Checker). *Let $\mathcal{P}$ be a linear ITS. If CeTA accepts a safety proof certificate (resp. termination proof certificate) for $\mathcal{P}$, then $\mathcal{P}$ is safe (resp. terminating).*

Our validity checker has exponential worst-case complexity and is incomplete, but the experimental results show that the current implementation of CeTA is good enough to validate all the proofs generated by AProVE and T2. A reason for this is that the transition formulas in the example ITSs are all conjunctions of atoms, and thus disjunctions are only due to invariants from the Impact algorithm and encoded lexicographic orderings. As a consequence, the CNF of formulas that have to be validated is at most quadratically larger than the original formula.

Table 1: Experimental results with AProVE, T2 and CeTA

Tool	# Yes	# No	# Certified	# Rejected	average time tool	average time CeTA
certifiable T2	562	–	560	2	7.98s	1.24s
certifiable T2 (w. hints)	540	–	539	1	8.35s	0.54s
full T2	615	420	–	–	8.60s	–
certifiable AProVE	543	–	535	8	14.52s	1.39s
full AProVE	512	369	–	–	21.19s	–

6 Experiments

For our experiments, we used *full* (unmodified) versions of AProVE and T2 as well as *certifiable* versions, where the latter have to produce termination certificates in XML using only the techniques described in this work. Additionally, we also consider a version of T2 that provides "hints" to prove entailments of linear arithmetic formulas. These certificates will then be checked by CeTA version 2.30.

The modification to obtain the certifiable version of T2 consists of about 1 500 lines of additional code, mostly to produce the machine-readable certificates and to keep the required information about all proof steps. The certifiable version uses precisely the techniques presented without formalization in [8], and all of these techniques can be modeled by the formalized theorems of this paper. The difference between the certifiable and the full version of T2 is that the latter uses Spacer [22] instead of Impact, supports additional termination techniques [13], and searches for nontermination proofs, but does not produce certifiable output.

Although AProVE does not explicitly work on cooperation programs, its certifiable version inserts an application of Thm. 4 at the beginning of each certificate. Afterwards, SCC decompositions and ranking functions that AProVE internally computes are reformatted into the applications of Thm. 6 and Thm. 8, respectively. Ranking functions over rational numbers are converted into ones over the integers by multiplication with the common denominator. The difference between the certifiable and the full version of AProVE is that the latter tries more termination techniques like non-linear ranking functions and searches for nontermination proofs, but does not produce certifiable output.

We performed experiments of our implementation on an Intel Xeon E5-1620 (clocked at 3.6GHz) with 16GB RAM on the 1222 examples from the "Integer Transition System" category from the Termination Competition 2016. The source code of CeTA is exported in Haskell using Isabelle's code export function, and compiled by ghc. All tools were run with a timeout of 60 seconds.

Table 1 summarizes our experiments. The table contains five rows, one for each configuration. The column "# Yes" indicates the number of successful termination proofs, "# No" the number of successful nontermination proofs, "# Certified" the number of proofs that were validated by CeTA, and "# Rejected" the number of certificates that were not validated by CeTA. We note that for termination, the certifiable version of T2 already has 91 % of the power of the

full version, even though many advanced techniques (e.g., polyranking rank functions [7]) are disabled. The certifiable version of AProVE was even more powerful than the full version w.r.t. termination proving, most likely since the full version also spends a significant amount of time to detect nontermination. Nearly all of the certificates were successfully validated by CeTA, except for two from AProVE where non-linear arithmetic reasoning is essential, and six which could not be certified in the given time. Currently, CeTA ignores all non-linear constraints when invoking the simplex algorithm. For T2, three certificates lead to CeTA parsing errors, caused by bugs in the certificate export.

Certification for T2 took in average about a sixth of the time T2 required to find a termination proof—the average time for successful runs of T2 (certifiable) is 7.98s. Generating and exporting hints for entailments in T2 more than halves the time CeTA needs to check certificates.

All experimental details including links to AProVE, T2 and CeTA can be found on `http://cl-informatik.uibk.ac.at/ceta/experiments/lts`.

7 Conclusion and Future Work

We have presented a formalization of safety and termination proofs using the unwinding and cooperation graph techniques. Furthermore, we have implemented the certification of proof certificates in CeTA, and have extended T2 to produce such certificates. While we have focused on two specific techniques in this paper, our formalization is general enough to accommodate proofs produced by other safety and termination provers, witnessed by AProVE. It remains as future work to extend other tools to export proof certificates and support additional techniques they require.

Our experiments show that extending our formalization to also support *non-termination* proof certificates would be valuable. We are also interested in supporting other related program analyzes, such as inferring runtime complexity bounds or proving properties in temporal logics.

As the most part of our formalization is independent of the chosen logic, formalized decision procedures for other logics than linear integer arithmetic, such as non-linear arithmetic, bit-vectors, arrays, etc. will immediately extend our results to systems which cannot be expressed as linear ITSs. For example, the two rejected certificates from AProVE can be certified if non-linear arithmetic reasoning is supported. Incorporating the certified quantifier elimination algorithms by Nipkow [28] would not only lead to another alternative validity checker but also allow for quantified formulas appear in transition formulas and invariants.

Finally, we note that CeTA is usually an order of magnitude faster than the termination tools on term rewriting, a statement that is not yet true for ITSs. Here, profiling reveals that the validity checker for formulas over linear integer arithmetic is the bottleneck. Consequently, it seems to be fruitful to develop a formalized SMT solver by extending work on SAT solving [4,19,25].

References

1. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination analysis of Java Bytecode. In: FMOODS'08. pp. 2–18
2. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Puebla, G., Román-Díez, G.: A formal verification framework for static analysis. Software & Systems Modeling 15(4), 987–1012 (2016)
3. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: exchanging verification results between verifiers. In: FSE'16. pp. 326–337. ACM (2016)
4. Blanchette, J.C., Fleury, M., Weidenbach, C.: A verified SAT solver framework with learn, forget, restart, and incrementality. In: IJCAR'16. pp. 25–44. Springer (2016)
5. Blanqui, F., Koprowski, A.: CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. Mathematical Structures in Computer Science 21(4), 827–859 (2011)
6. Borralleras, C., Brockschmidt, M., Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving termination through conditional termination. In: TACAS'17. To appear.
7. Bradley, A.R., Manna, Z., Sipma, H.B.: The polyranking principle. In: ICALP'05. pp. 1349–1361
8. Brockschmidt, M., Cook, B., Fuhs, C.: Better termination proving through cooperation. In: CAV'13. pp. 413–429
9. Brockschmidt, M., Cook, B., Ishtiaq, S., Khlaaf, H., Piterman, N.: T2: Temporal property verification. In: TACAS'16. pp. 387–393
10. Caleiro, C., Gonçalves, R.: On the algebraization of many-sorted logics. In: WADT'06. pp. 21–36
11. Cho, S., Kang, J., Choi, J., Yi, K.: SparrowBerry: A verified validator for an industrial-strength static analyzer, `http://ropas.snu.ac.kr/sparrowberry/`
12. Contejean, E., Paskevich, A., Urbain, X., Courtieu, P., Pons, O., Forest, J.: A3PAT, an approach for certified automated termination proofs. In: PEPM'10. pp. 63–72
13. Cook, B., See, A., Zuleger, F.: Ramsey vs. lexicographic termination proving. In: TACAS'13. pp. 47–61
14. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI'06. pp. 415–426
15. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL'77. pp. 238–252 (1977)
16. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: RTA'11. pp. 41–50
17. Farkas, J.: Theorie der einfachen Ungleichungen. Journal für die reine und angewandte Mathematik 124, 1–27 (1902)
18. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing program termination and complexity automatically with AProVE. J. Autom. Reasoning 58, 3–31 (2017)
19. Heule, M.J., Hunt, W.A., Wetzler, N.: Trimming while checking clausal proofs. In: FMCAD 2013. pp. 181–188. IEEE
20. Jourdan, J., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: POPL'15. pp. 247–259
21. Klein, G., Nipkow, T.: A Machine-Checked Model for a Java-like Language, Virtual Machine and Compiler. ACM Trans. Progr. Lang. Syst. 28(4), 619–695 (2006)

22. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: CAV'14. pp. 17–34
23. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: Klein, G., Gamboa, R. (eds.) ITP'14. pp. 325–340
24. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
25. Marić, F., Janičić, P.: Formal correctness proof for DPLL procedure. Informatica 21(1), 57–78 (2010)
26. McMillan, K.: Lazy abstraction with interpolants. In: CAV'06. pp. 123–136
27. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
28. Nipkow, T.: Linear quantifier elimination. J. Autom. Reasoning 45(2), 189–212 (2010)
29. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Automated termination analysis of Java Bytecode by term rewriting. In: RTA'10. pp. 259–276
30. Schrijver, A.: Theory of linear and integer programming. Wiley (1999)
31. Spasić, M., Marić, F.: Formalization of incremental simplex algorithm by stepwise refinement. In: Giannakopoulou, D., Méry, D. (eds.) FM'12. pp. 434–449
32. Spoto, F., Mesnard, F., Payet, É.: A termination analyser for Java Bytecode based on path-length. ACM Trans. Progr. Lang. Syst. 32(3), 8:1–8:70 (2010)
33. Sternagel, C., Thiemann, R.: The certification problem format. In: UITP 2014. EPTCS, vol. 167, pp. 61–72 (2014)
34. Ströder, T., Giesl, J., Brockschmidt, M., Frohn, F., Fuhs, C., Hensel, J., Schneider-Kamp, P., Aschermann, C.: Automatically proving termination and memory safety for programs with pointer arithmetic. J. Autom. Reasoning 58, 33–65 (2017)
35. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: TPHOLs'09. pp. 452–468
36. Tseitin, G.S.: On the complexity of proof in prepositional calculus. Studies in constructive mathematics and mathematical logic. Part II 8, 234–259 (1968)
37. Urban, C., Gurfinkel, A., Kahsai, T.: Synthesizing ranking functions from bits and pieces. In: TACAS'16. pp. 54–70
38. Wang, H.: Logic of many-sorted theories. J. Symbolic Logic 17(2), 105–116 (1952)
39. Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations. In: POPL'12. pp. 427–440