# Interactive Termination Proofs using Termination Cores*

Panagiotis Manolios and Daron Vroon

College of Computer and Information Science
Northeastern University
360 Huntington Ave., Boston MA 02115, USA
pete@ccs.neu.edu, daron.vroon@gmail.com

**Abstract.** Recent advances in termination analysis have yielded new methods and tools that are highly automatic. However, when they fail, even experts have difficulty understanding why and determining how to proceed. In this paper, we address the issue of building termination analysis engines that are both highly automatic and easy to use in an interactive setting. We consider the problem in the context of ACL2, which has a first-order, functional programming language. We introduce the notion of a *termination core*, a simplification of the program under consideration which consists of a single loop that the termination engine cannot handle. We show how to extend the Size Change Termination (SCT) algorithm so that it generates termination cores when it fails to prove termination, with no increase to its complexity. We show how to integrate this into the Calling Context Graph (CCG) termination analysis, a powerful SCT-based automatic termination analysis that is part of the ACL2 Sedan. We also present several new, convenient ways of allowing users to interface with the CCG analysis, in order to guide it to a termination proof.

## 1 Introduction

Recent years have seen great advances in the field of automated proofs of program termination (*e.g.* [1, 3, 8, 11]). In this paper, we will explore one such termination analysis, the Calling Context Graph (CCG) algorithm, in detail [9, 13]. The motivation for developing the CCG algorithm came from our desire to integrate mechanized program verification into the undergraduate curriculum. We used ACL2 [6, 5, 7], in part because it is based on a simple, applicative programming language and has a relatively simple logic. One of the first issues students confront is that functions must be shown to terminate. We wanted to avoid discussing ordinals and measure functions, so we developed and implemented the CCG termination analysis, which is able to automatically prove termination for the kinds of functions arising in undergraduate classes.

* This research was funded in part by NASA Cooperative Agreement NNX08AE37A and NSF grants CCF-0429924, IIS-0417413, and CCF-0438871.

Unfortunately, any termination analysis (and CCG is no exception) is bound to fail, either because the program under consideration is non-terminating or because the analysis is just not powerful enough to prove termination. In this paper, we address the issue of what to do when that happens. The idea is to leverage all of the information that was discovered during the termination effort. For example, the analysis may have discovered that certain loops in the program are terminating. Rather than throwing out all of that analysis and asking the user to prove termination from scratch, we want to present the user with an explanation of why the termination analysis failed. We propose to do that by generating a *termination core*, a new program that the termination analysis cannot prove terminating and which corresponds to a single simple cycle of the original program. Termination cores reveal the true reason that the termination analysis failed. Termination cores are a general notion that we believe can be fruitfully applied to any number of termination analyses, but in this paper we show how to compute termination cores for algorithms based on Size Change Termination (SCT) [8].

We start by reviewing CCG analysis in Section 2. We then introduce the notion of termination cores in Section 3. We prove that the complexity of the termination core generation problem for SCT is PSPACE-complete and we provide a practical algorithm. Just reporting termination cores is not enough. We need a mechanism that enables the user to interact with the termination analysis. We discuss this in Section 4, where we present several new, convenient ways of allowing users to interface with the CCG analysis, in order to guide it to a termination proof.

All the techniques described here are implemented in the current version of ACL2s, the ACL2 Sedan [4, 2], a freely available, open-source, well-supported theorem prover that is based on ACL2, but was designed with greater usability and automation as primary design considerations. ACL2s provides a modern integrated development environment and includes fully automatic bug-finding methods based on a synergistic combination of theorem proving and random testing. CCG analysis is an integral part of ACL2s. In extensive experimental trials, it was able to prove over 98% of the more than 10,000 functions in the ACL2 regression suite terminating with no user input. ACL2s has been used at Northeastern University, UT Austin, and Georgia Tech to teach hundreds of undergraduate students how to reason about programs.

## 2 Termination Using Calling Context Graphs

We give a brief, simplified overview of the CCG analysis. For a more complete and detailed treatment, see [9, 13]. The domain for the CCG analysis is a universe of programs, PROG, written in an applicative first-order functional programming language. For the sake of simplicity, we limit our discussion here to a very simple language, part of whose semantics is sketched in Figure 1: $\mathcal{F}$ denotes the universe of function names; $\mathcal{X}$ the universe of variable names; $\mathcal{V}$ the universe of values; $\mathcal{E}$ the universe of expressions; HIST the universe of *histories*, which map previously

defined functions to their signature and definitions; and ENV the universe of environments, which map variables to values.

$$f, g \in \mathcal{F} \qquad v, u \in \mathcal{V} \qquad x, y, z \in \mathcal{X} \qquad e, m \in \mathcal{E}$$
$$h \in \text{HIST} = \mathcal{F} \to \mathcal{X}^* \times \mathcal{E} \qquad \epsilon \in \text{ENV} = \mathcal{X} \to \mathcal{V}$$

$$[\![\texttt{if } e_{test} \texttt{ then } e_{then} \texttt{ else } e_{else}]\!]^h \epsilon = \begin{cases} [\![e_{then}]\!]^h \epsilon \text{ if } [\![e_{test}]\!]^h \epsilon \neq \texttt{nil} \\ [\![e_{else}]\!]^h \epsilon \text{ otherwise} \end{cases}$$

$$[\![f\ e_1\ e_2\ \ldots\ e_n]\!]^h \epsilon = [\![e]\!]^h [x_i \mapsto v_i]_{i=1}^n \quad \text{where } v_i = [\![e_i]\!]^h \epsilon \text{ and } h(f) = \langle \langle x_i \rangle_{i=1}^n, e \rangle$$

**Fig. 1.** A rough sketch of a simple language and its semantics.

Consider the following function definition:

```
f x y = if (x < y) then 1 + (f (x+1) y)
        else if (x > y) then 1 + (f x (y+1))
        else 0
```

First, we create an abstraction of the program that captures its recursive behaviors while ignoring everything else. In `f`, the value returned in the base case and the fact that we add `1` to the value returned by each recursive call is irrelevant to the termination proof. We therefore reduce the program to its *calling contexts*. Intuitively, these are the recursive calls of the program along with the conditions under which each call is made. More formally, given a program $F \in \text{PROG}$, a calling context is a triple, $\langle f, G, e \rangle \in \text{CONTEXTS} = \mathcal{F} \times 2^{\mathcal{E}} \times \mathcal{E}$, such that $f$ is a function defined in $F$ ($F$ can contain several function definitions), $e$ is a call to a function defined in $F$, and $G$ is the set of *governors* of $e$, *i.e.*, the *exact* set of conditions under which $e$ is executed. The calling contexts for `f` are as follows.

1. $\langle$`f`, {`x < y`}, `(f (+ x 1) y)`$\rangle$
2. $\langle$`f`, {`not (x < y)`, `x > y`}, `(f x (+ y 1))`$\rangle$

Note that, in the governor for the second context, the second condition implies the first. We could therefore simplify this governor to be {`x > y`}.

The calling contexts are used to approximate the behavior of the program via the construction of a *Calling Context Graph (CCG)*, whose nodes are the calling contexts of the program and whose edges represent possible paths of execution from one context to the next. The minimal CCG for $F$ is as follows.


Notice that if `x` $<$ `y`, then in the next iteration, `x` $\leq$ `y`, since `x` is incremented by `1`. Likewise, if `x` $>$ `y`, then in the next iteration, `x` $\geq$ `y` since `y` is incremented by `1`. Therefore, it is not possible for execution of `f` to move from one context to

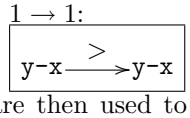
the other. This is a critical observation for proving termination, since if the flow of the program could alternate between the contexts, it could enter an infinite loop where `x` was increased, then `y`, and so on, without `x` and `y` ever being equal.

In order to formalize our notion of a CCG, we need the notion of a *call substitution*. Given a function call, $e = f\ \ e_1\ \ e_2\ \ \ldots\ \ e_n$, to a function with parameters $x_1, x_2, \ldots, x_n$, the call substitution for $e$, denoted $\sigma_e$, substitutes each $e_i$ for the corresponding $x_i$.

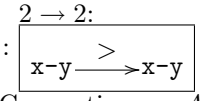
A CCG is a graph, $\mathcal{G} = \langle C, E \rangle$, whose nodes, $C \subseteq$ Contexts, and whose edges, $E \subseteq$ Contexts$\times$Contexts and for any pair of contexts, $c_1 = \langle f_1, G_1, e_1 \rangle$, $c_2 = \langle f_2, G_2, e_2 \rangle \in C$, if $e_1$ is a call to $f_2$ and $\left[\!\left[ \bigwedge_{g_1 \in G_1} g_1 \wedge \bigwedge_{g_2 \in G_2} g_2 \sigma_{e_1} \right]\!\right]^h \epsilon \neq$ `nil` for some $\epsilon \in$ Env, then $\langle c_1, c_2 \rangle \in E$. Such an environment is called a *witness* for $\langle c_1, c_2 \rangle$. Notice that it is in general undecidable to determine if an edge must be in a CCG. This is why the condition for including an edge is an if rather than an iff. The goal is to create a safe approximation of the minimal CCG. Also, note that the *trivial CCG*, defined as the CCG in which there is an edge from $c_1$ to $c_2$ iff $c_1$ represents a call to the function containing $c_2$, gives us the exact same information as a standard call graph (in which the nodes are function names and there is an edge between $f$ and $g$ if $f$ contains a call to $g$). We use theorem prover queries to generate CCGs that are smaller than the trivial one [9]. For example, using the ACL2 theorem prover, we are able to generate the minimal CCG for `f` as given above.

The next step in the CCG analysis is to annotate each edge of the CCG with a *generalized size change graph (GSCG)*. These tell us which values are decreasing or non-increasing from one context to the next in our CCG. A valid set of GSCGs for `f` is as follows.

$G_1$: $1 \to 1$:




$G_2$ : $2 \to 2$:




GSCGs are then used to annotate the CCG, creating an *Annotated CCG (ACCG)*. More formally, we define ACCGs and GSCGs as follows.

$$
\begin{aligned}
p, q, r \in \textsc{Lab} &= \{>, \geq\} \\
\mathcal{G}, \mathcal{H} \in \text{ACCG} &= 2^{\textsc{Contexts}} \times 2^{\textsc{Contexts}\times\text{GSCG}\times\textsc{Contexts}} \\
G, H \in \text{GSCG} &= 2^{\mathcal{E}} \times 2^{\mathcal{E}} \times 2^{\mathcal{E}\times\textsc{Lab}\times\mathcal{E}} \times \textsc{Contexts} \times \textsc{Contexts}
\end{aligned}
$$

Each edge in the ACCG is annotated with a GSCG. We write $c_1 \xrightarrow{G} c_2$ to denote that $\langle c_1, G, c_2 \rangle$ is an edge $\in \mathcal{G}$. A GSCG is a bipartite graph with a set of expressions corresponding to the left nodes, a set of expressions corresponding to the right nodes, a set of labeled edges, and the pair of contexts the GSCG annotates. The tuple $\langle M_1, M_2, E, c_1, c_2 \rangle$ is a GSCG if for every $\langle m_1, r, m_2 \rangle \in E$ we have that $[\![m_1]\!]^h\, \epsilon \quad r \quad [\![m_2]\!]^h\, \epsilon$ for each $\epsilon$ that is a witness for $\langle c_1, c_2 \rangle$. We write $m_1 \xrightarrow{r} m_2$ to denote that $\langle m_1, r, m_2 \rangle$ is an edge $\in G$.

GSCGs and ACCGs are similar in concept to *size change graphs (SCGs)* and *annotated call graphs (ACGs)* that form the basis of the Size Change Termination analysis of Lee, Jones, and Ben-Amram for use in their size-change analysis [8].

The differences are that GSCGs have arbitrary expressions rather than just variables for nodes, and ACCGs mirror the recursive flow from recursive call to recursive call rather than from function to function. The result is a more detailed analysis of program behavior. However, structurally, these concepts are the same, which allows us to apply the size change analysis to ACCGs as follows.

**Definition 1.** *A **multipath** $\pi$ through an ACCG $\mathcal{G}$ is a (potentially infinite) path in $\mathcal{G}$: $\pi = f_0 \xrightarrow{G_1} f_1 \xrightarrow{G_2} f_2 \xrightarrow{G_3} \cdots$.*

We write $\mathcal{G}^\omega$ for the set of infinite multipaths over $\mathcal{G}$ and $\mathcal{G}^+$ for the set of finite, nonempty ones. We sometimes write $G_1, G_2, \ldots$ or $\langle G_i \rangle$ to describe a multipath when the function names are irrelevant. Paths in ACCGs are called *multipaths* because their elements are graph structures and may contain many *threads*.

**Definition 2.** *A **thread** in a multipath $\pi = \langle G_i \rangle$ is a sequence of size-change edges $\langle x_{i-1} \xrightarrow{r_i} x_i \rangle$ such that $x_{i-1} \xrightarrow{r_i} x_i \in G_i$ for all $i > 0$.*

For example, consider the multipath `f` $\xrightarrow{G_1}$ `f` $\xrightarrow{G_1}$ `f`. Its only thread is `y-x` $\xrightarrow{>}$ `y-x` $\xrightarrow{>}$ `y-x`. A thread tells us that the values of certain expressions do not increase during a sequence of calls, and can be used to prove termination as follows.

**Definition 3.** *The **Size Change Termination (SCT) problem** takes an ACCG as input and returns* `true` *if every infinite multipath through the ACCG has a suffix with a thread $\langle m_i \xrightarrow{r_i} m_{i+1} \rangle$ such that infinitely many $r_i = >$, or* `false` *otherwise.*

By well-foundedness, no infinite path through the ACCG that has such a thread can be an actual computation.

**Theorem 1 (Lee, et al. [8]).** *SCT is PSPACE complete.*

The SCT problem can be solved by *composing* GSCGs to create new GSCGs representing multiple transitions in the ACCG.

**Definition 4.** ***Composition** of GSCG labels and GSCGs is defined as follows.*

1. $p \cdot q = \begin{cases} \geq \text{ if } p = \geq \text{ and } q = \geq \\ > \text{ otherwise} \end{cases}$
2. $G_1 \cdot G_2 = \langle M_1, M_3, E, c_1, c_3 \rangle$, *where* $G_1 = \langle M_1, M_2, E_1, c_1, c_2 \rangle$, *and* $G_2 = \langle M_2, M_3, E_2, c_2, c_3 \rangle$, *and* $E = \{m_1 \xrightarrow{p \cdot q} m_3 \mid m_1 \xrightarrow{p} m_2 \in G_1 \ \wedge \ m_2 \xrightarrow{q} m_3 \in G_2\}$

**Definition 5.** *The **evaluation** of $\pi = \langle G_1, G_2, \ldots, G_n \rangle \in \mathcal{G}^+$ is $[\![\pi]\!] = G_1 \cdot G_2 \cdots G_n$.*

**Proposition 1.** *$m \xrightarrow{r} m' \in [\![\pi]\!]$ iff there exists a thread $m \xrightarrow{r_1} m_1 \xrightarrow{r_2} \cdots \xrightarrow{r_{n-1}} m_{n-1} \xrightarrow{r_n} m'$ in $\pi$, with $r = r_1 \cdot \ \cdots \ \cdot r_n$.*

Composition occurs until a fixed point is reached, at which point certain GSCGs, called *idempotents* are examined. An idempotent is a GSCG, $G$, of the form $\langle M, M, E, c, c\rangle$ such that $G \cdot G = G$. If all idempotents have an edge, $e \xrightarrow{>} e \in G$, then the algorithm returns `true`. Otherwise, it returns `false`.

**Theorem 2 (Lee et al. [8]).** *The algorithm above solves SCT.*

**Theorem 3 (Manolios and Vroon [9]).** *If a program, $D \in$ PROG has a corresponding ACCG, $\mathcal{G}$, such that SCT($\mathcal{G}$) = `true`, $D$ is terminating on all inputs.*

## 3 Termination Cores

The idea behind termination cores is to present the user with a single simple cycle that embodies the reason for a failure to prove termination. We want this to be a general notion that applies to any termination prover, so we begin by defining a general notion of a termination analysis.

**Definition 6.** *A **termination analysis**, $\mathcal{T}$, is a function that takes in a set of function definitions, $F$, and returns* `true` *or* `false`*, such that if $\mathcal{T}(F) =$* `true`*, it is the case that the definitions of $F$ will terminate for all inputs according to the semantics of the language.*

When the termination analysis fails, we want to create a new program that is simpler than the original, but still reflects the reason for the failure. We therefore must link the recursive behaviors of two programs, which we express as a relationship between their CCGs.

**Definition 7.** *Two calling contexts, $c, c'$ of the form $\langle f, G, (g\ e_1\ \ldots\ e_n)\rangle$ and $\langle f', G, (g'\ e_1\ \ldots\ e_n)\rangle$, respectively, are said to be **similar**, denoted $c \sim c'$. We can denote $c'$ as $[c]^{f'}_{g'}$, or $c$ as $[c']^{f}_{g}$.*

Using this notion of context similarity, we develop the notion of a *similarity-preserving path homomorphism* between ACCGs that will form the basis of our definition of termination cores. We begin by defining path homomorphism.

**Definition 8.** *Given two directed graphs, $G = (C, E), G' = (C', E')$, a **path homomorphism** is a function $\phi : C \to C'$ such that $\langle c_1, c_2\rangle \in E \Rightarrow \langle \phi(c_1), \phi(c_2)\rangle \in E'$. If such a $\phi$ exists, we say that $G$ is homomorphic to $G'$. If $C$ and $C'$ are sets of contexts, we say $\phi$ is **similarity-preserving** if $c \sim \phi(c)$ for all $c \in C$.*

In other words, a similarity-preserving homomorphism from $\mathcal{G}$ to $\mathcal{G}'$ demonstrates that the original program associated with $\mathcal{G}'$ contains a superset of the recursive behaviors of the new program associated with $\mathcal{G}$.

**Definition 9.** *Let $\mathcal{T}$ be a termination analysis, $F$ be a set of function definitions such that $\mathcal{T}(F) =$* `false`*, and $\mathcal{G} = (C, E)$ be a CCG for $F$. Then a **termination core** for $F$ modulo $\mathcal{T}$, is a set of function definitions, $F'$ that satisfy all of the following:*

– *The trivial CCG, $\mathcal{G}' = (C', E')$, of $F'$ is a simple cycle,*
– *There exists a similarity-preserving path homomorphism, $\phi : \mathcal{G}' \to \mathcal{G}$,*
– $\mathcal{T}(F') = \texttt{false}$.

Thus, the termination core is a single loop through the original program for which the termination analysis fails. The *Termination Core modulo $\mathcal{T}$ ($TC_{\mathcal{T}}$)* is the problem of finding such cores.

**Definition 10.** *Given a termination analysis $\mathcal{T}$, the* ***termination core modulo $\mathcal{T}$ ($TC_{\mathcal{T}}$) problem*** *takes a program $P$ as input and generates* `null` *if $\mathcal{T}(P) =$* `true` *and a termination core for $P$ modulo $\mathcal{T}$ otherwise.*

### 3.1 Termination Cores in CCG via Size Change Cores

In order to create termination cores for CCG, we use the notion of *size change cores (SCC).*

**Definition 11.** *A* ***size change core*** *is a finite multipath of the form $\pi = c \xrightarrow{G_1} c_1 \xrightarrow{G_2} \cdots \xrightarrow{G_n} c$ such that, $\pi^\omega$ has no suffix with a corresponding thread of infinite descent.*

**Proposition 2.** *SCT($\mathcal{G}$) =* `false` *iff there exists a size change core for $\mathcal{G}$.*

*Proof.* By the definition of SCT, it is clear that if such an SCC exists, $SCT(\mathcal{G}) =$ `false`. For the other direction, suppose $SCT(\mathcal{G}) =$ `false`. Then by Theorem 2, there exists $\pi = c \xrightarrow{G_1} c_1 \xrightarrow{G_2} \cdots \xrightarrow{G_n} c$ such that $[\![\pi]\!]$ is idempotent and has no edge of the form $m \xrightarrow{>} m$. Notice that $\pi$ is an SCC: by Proposition 1, there is no thread for $\pi$, $m \xrightarrow{r_1} m_1 \xrightarrow{r_2} \cdots \xrightarrow{r_n} m$ such that one of $r_1 \cdots r_n$ is ">". Now, suppose that $\pi^\omega$ has a suffix with an infinitely decreasing thread. By the pigeon hole principle, this means that there exists some $k$ and $m$ such that $\pi^k$ has a thread $m \xrightarrow{r_1} m_1 \xrightarrow{r_2} \cdots \xrightarrow{r_{nk}} m$ such that some $r_i$ is ">". By Proposition 1, this means that $[\![\pi^k]\!]$ has an edge $m \xrightarrow{>} m$ in it. But $[\![\pi^k]\!] = [\![\pi]\!]$ since $[\![\pi]\!]$ is idempotent. Therefore, no such edge can exist. □

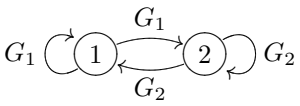
Consider the following function:

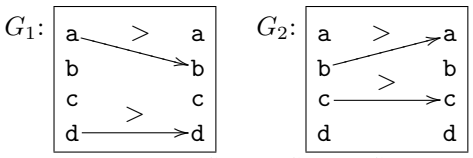
```
f a b c d = if (a > 0 and b > 0 and c > 0 and d > 0)
            then 1 + (f (b+1) (a-1) (c+1) (d-1))
                   + (f (b-1) (a+1) (c-1) (d+1))
            else 0
```

The contexts for this function are as follows.

1. ⟨`f`, {`a > 0`, `b > 0`, `c > 0`, `d > 0`}, `(f (b+1) (a-1) (c+1) (d-1))`⟩
2. ⟨`f`, {`a > 0`, `b > 0`, `c > 0`, `d > 0`}, `(f (b-1) (a+1) (c-1) (d+1))`⟩

Suppose we use the measures `a`, `b`, `c`, and `d` for both contexts. Then we get the following ACCG.




Notice, then, that an SCC for this program is $1 \xrightarrow{G_1} 1 \xrightarrow{G_1} 2 \xrightarrow{G_2} 1$, for which there is no thread at all. The problem is that this is not a simple cycle. In order to create a termination core, then, we must derive a simple cycle that corresponds to an SCC. We do this by renaming the functions in the contexts in order to distinguish between occurrences of a single context in the loop.

**Definition 12.** *Given a cycle $\boldsymbol{c} = c_1, c_2, \ldots, c_n, c_1$ in an ACCG, and a sequence of fresh, distinct function symbols, $\boldsymbol{f} = \langle f_i \rangle_{i=1}^{n} \in \mathcal{F}^n$, the **similar simple cycle with respect to $\boldsymbol{f}$**, denoted $[\boldsymbol{c}]_{\boldsymbol{f}}$, is the sequence $\boldsymbol{c'} = \langle [c_i]_{f_{i+1}}^{f_i} \rangle$ where $f_{n+1} = f_1$. Given a cyclical multipath, $\pi = c_1 \xrightarrow{G_1} c_2 \xrightarrow{G_2} \cdots \xrightarrow{G_n} c_1$, the **set of similar simple cyclical multipaths with respect to $\boldsymbol{f}$**, denoted $[\pi]_{\boldsymbol{f}}$ is the set containing all $\pi' = [c_1]_{f_2}^{f_1} \xrightarrow{G'_1} [c_2]_{f_3}^{f_2} \xrightarrow{G'_2} \cdots \xrightarrow{G'_n} [c_1]_{f_2}^{f_1}$ such that each $G'_i$ is a subgraph of the corresponding $G_i$.*

The two key features of similar simple cyclical multipaths are that they are simple cycles and that they preserve the non-terminating behavior of an SCC. Thus, we get the following result.

**Lemma 1.** *Let $\pi = \langle G_i \rangle_{i=1}^{n}$ be an SCC and $\pi' = \langle G'_i \rangle_{i=1}^{n} \in [\pi]_{\boldsymbol{f}}$. Then $\pi'$ is an SCC of the ACCG consisting of the contexts, edges, and GSCGs of $\pi'$.*

*Proof.* By definition, there is no infinitely decreasing thread corresponding to $\pi^\omega$. But every thread of $\pi^\omega$ is a thread of $\pi'^\omega$ by construction. Therefore, there is no infinitely decreasing thread corresponding to $\pi'^\omega$. By definition, this makes $\pi'$ an SCC.

Returning to our example, we can construct a similar simple cycle to our SCC as follows:

1. $\langle$`f0`, {`a > 0`, `b > 0`, `c > 0`, `d > 0`}, `(f1 (b+1) (a-1) (c+1) (d-1))`$\rangle$
2. $\langle$`f1`, {`a > 0`, `b > 0`, `c > 0`, `d > 0`}, `(f2 (b+1) (a-1) (c+1) (d-1))`$\rangle$
3. $\langle$`f2`, {`a > 0`, `b > 0`, `c > 0`, `d > 0`}, `(f0 (b-1) (a+1) (c-1) (d+1))`$\rangle$

We use a similar simple cycle to an SCC to create a corresponding termination core. We do this by creating a trivial function for each context in the similar simple cycle as follows.

**Definition 13.** *Given a calling context, $c = \langle f, \{e_1, e_2, \ldots, e_n\}, e \rangle$, the **minimal function definition for** $c$ is the following function, where $\langle x_i \rangle_{i=1}^{m}$ are the parameters for function $f$.*[1]

[1] Notice that we can return anything at all in the `else` case below. In our implementation, we return the list of parameters for technical, ACL2-related reasons.

$f\ x_1\ x_2\ \ldots\ x_m$ =
  if ($e_1$ and $e_2$ and ... and $e_n$) then $e$ else
  [$x_1$; $x_2$; ...; $x_m$]

The point is to use the similar simple cycles to construct the minimal definitions. In the case of our example, we have the following.

```
f0 a b c d = if (a > 0 and b > 0 and c > 0 and d > 0)
             then (f1 (b+1) (a-1) (c+1) (d-1))
             else [a; b; c; d]
f1 a b c d = if (a > 0 and b > 0 and c > 0 and d > 0)
             then (f2 (b+1) (a-1) (c+1) (d-1))
             else [a; b; c; d]
f2 a b c d = if (a > 0 and b > 0 and c > 0 and d > 0)
             then (f0 (b-1) (a+1) (c-1) (d+1))
             else [a; b; c; d]
```

What remains is to prove that this corresponds to a termination core in general. Our work thus far allows us to prove that the resulting functions satisfy the first two conditions of termination cores.

**Lemma 2.** *Let* $\pi = c_1 \xrightarrow{G_1} c_2 \xrightarrow{G_2} \cdots \xrightarrow{G_n} c_1$ *be a size change core for ACCG* $\mathcal{G}$, $\boldsymbol{c} = c_1, c_2, \ldots, c_n, c_1$, $\boldsymbol{f}$ *be a sequence of fresh, distinct function names, and* $\boldsymbol{c'} = [\boldsymbol{c}]_{\boldsymbol{f}} = c'_1, c'_2, \ldots, c'_n, c'_1$. *Then the trivial CCG,* $\mathcal{G}'$ *of the minimal function definitions of* $\boldsymbol{c'}$, *is a simple cycle such that there exists a similarity-preserving path homomorphism from* $\mathcal{G}'$ *to* $\mathcal{G}$.

*Proof.* This follows from the freshness and distinctness of the function names in $\boldsymbol{f}$. □

All that remains, then, is to show that our construction results in functions that cannot be proved terminating by our analysis. In order to do this, we need to make some assumptions about the construction of ACCGs. Intuitively, we need to assume that we are consistent in our choice of measures and our ability to prove the necessary queries. We require that our ACCG generator is not "smarter" when analyzing the termination core than it is when analyzing the the original function.

**Definition 14.** *Let* `build-accg` : PROG → *ACCG be a function that computes an ACCG corresponding to the input program. Then we say that* `build-accg` *is* ***monotonic*** *if when given* $P, P' \in$ PROG *such that there exists a similarity-preserving homomorphism,* $\phi$ *from the trivial CCG of* $P$ *to the trivial CCG of* $P'$, *the following conditions hold, where* $\mathcal{G}_P =$ `build-accg`$(P)$ *and* $\mathcal{G}_{P'} =$ `build-accg`$(P')$*:*

- $c_1 \xrightarrow{G} c_2 \in \mathcal{G}_P$ *if the call of* $c_1$ *is a call to the function containing* $c_2$ *and* $\phi(c_1) \xrightarrow{G'} \phi(c_2) \in \mathcal{G}_{P'}$, *and*

– *For all* $c_1 \xrightarrow{G} c_2 \in \mathcal{G}_P$ *and* $\phi(c_1) \xrightarrow{G'} \phi(c_2)$, $G$ *is a subgraph of* $G'$.

From this point forward, we assume a fixed, monotonic `build-accg`. The interesting about this monotonicity property for us is that it means that the ACCG we construct for our termination core contains a similar simple multipath to the SCC we used to construct it. More formally, we have the following.

**Lemma 3.** *Let* $P$ *be a program and* $\mathcal{G}_P = \texttt{build-accg}(P)$ *such that* $\pi$ *is an SCC for* $\mathcal{G}_P$ *that traverses contexts* $\boldsymbol{c}$*. Then if* $\boldsymbol{c'} = [\boldsymbol{c}]_{\boldsymbol{f}}$ *and* $P'$ *is the set of minimal function definitions for* $\boldsymbol{c'}$*, then the multipath* $\pi'$ *that visits in order the contexts of* $\boldsymbol{c'}$ *in* $\mathcal{G}_{P'} = \texttt{build-accg}(P')$ *is a similar simple multipath to* $\pi$*.*

*Proof.* Follows from the definition of monotonicity. □

Now we fix our definition of `ccg` and `ccg-tc`, our termination analysis and termination core solver, respectively; but first, we introduce the size change core problem.

**Definition 15.** *The* ***Size Change Core (SCC) problem*** *takes an ACCG,* $\mathcal{G}$*, as input and returns a size change core if SCT(*$\mathcal{G}$*) =* `false` *and* `null` *otherwise.*

**Definition 16.** *We define* `ccg` *as SCT(*`build-accg`$(P)$*). We define* `ccg-tc` : Prog → Prog *as follows: Given program,* $P$*,let* $\pi = SCC(\texttt{build-accg}(P))$*. If* $\pi =$ `null`*, return* `null`*. Otherwise, return the minimal function definitions for* $[\pi]_{\boldsymbol{f}}$ *for some fresh function names,* $\boldsymbol{f}$*.*

Based on the work we've done so far, it is fairly straightforward to prove the following.

**Theorem 4.** `ccg-tc` *solves* $TC_{\texttt{ccg}}$*.*

*Proof.* By the definition of SCC, `ccg-tc`$(P)$ returns `null` iff `ccg`$(P) =$ `true`, so we only need to concern ourselves with the case in which `ccg`$(P) =$ `false`.

If `ccg-tc`$(P)$ produces a program' $P'$, then $P'$ satisfies the first two properties of a termination core by Lemma 2. It satisfies the final property by a combination of Lemma 3 with Lemma 1. □

### 3.2 Constructing Size Change Cores in CCG

We prove that $SCC \in PSPACE$. To do this, we use a characterization of the problem using Büchi automata. Given an ACCG, $\mathcal{G}$, consider the two sets of infinite multipaths:

$$FLOW^\omega = \mathcal{G}^\omega$$
$$DESC^\omega = \{\pi \in \mathcal{G}^\omega \mid \pi \text{ has a suffix with an infinitely decreasing thread}\}$$

By results in [8], both $FLOW^\omega$ and $DESC^\omega$ are $\omega$-regular subsets of $\text{GSCG}^\omega$ for which there are Büchi automata that solve each in space polynomial with

respect to the size of the original program. Note that SCT is equivalent to determining if $FLOW^\omega \subseteq DESC^\omega$. This, in turn can be expressed as the problem of determining that $FLOW^\omega \cap \overline{DESC^\omega}$ is empty.

What we want is to find $\pi \in \mathcal{A} = FLOW^\omega \cap \overline{DESC^\omega}$ when such a $\pi$ exists. One idea is to construct the Büchi automaton corresponding to $\mathcal{A}$ and to search for such a path. Unfortunately, this does not work because complementing a Büchi automaton can lead to an exponential blowup. Fortunately, there are methods that allow us to traverse $\mathcal{A}$ in polynomial space without actually constructing it [10]. The PSPACE completeness of SCC can then be proved as follows.

**Theorem 5.** *SCC is PSPACE complete.*

*Proof.* Showing PSPACE hardness is trivial. By Proposition 2, SCT is reducible to SCC, and by Theorem 1, SCT is PSPACE-complete.

To show that SCC is in PSPACE, we non-deterministically find a multipath $\pi = \pi_1\pi_2 \in \mathcal{A}$ such that $s \xrightarrow{\pi_1} a \xrightarrow{\pi_2} a$ for some initial state $s$ and accepting state $a$, using the following algorithm, where $S^0$ is the set of initial states, $S$ is the set of states, and $F$ is the set of final states of $\mathcal{A}$. Also, the alphabet of $\mathcal{A}$ is the set of GSCG's of the ACCG $\mathcal{G}$.

```
1:  s ← s_0 ← choose(S^0); a ← choose(F)
2:  for i = 1 to |S| do
3:      s ← choose({t ∈ S | s -G-> t ∈ A})
4:      if s = a then
5:          for i = 1 to |S| do
6:              G ← choose(G)
7:              s ← choose({t ∈ S | s -G-> t ∈ A})
8:              Output G
9:              if s = a then
10:                 return found
11:         return null
12: return null
```

Here, `choose` denotes a non-deterministic choice of an element in the given set if such an element exists. If not, it causes the entire algorithm to halt and return `null`. Note that $\mathcal{A}$ is non-empty iff such a path exists [12]. However, the algorithm only outputs $\pi_2$. By the definition of $\mathcal{A}$, we see that $\pi_1\pi_2^\omega$ is an infinite multipath such that no suffix of the multipath has an infinitely decreasing thread. Thus, $\pi_2^\omega$ is also such a multipath, since any suffix of $\pi_2^\omega$ is a suffix of $\pi_1\pi_2^\omega$. By definition, this makes $\pi_2^\omega$ a size change core. Therefore, this algorithm solves SCC.

At any given point, all we are storing is four states ($s_0$, $a$, $s$, and $t$), two counters between 1 and $|S|$, and a single GSCG, $G$. All of this plus determining if $s \xrightarrow{G} t \in \mathcal{A}$ can be done in polynomial space. Therefore, $SCC \in NPSPACE = PSPACE$. □

**Definition 17.** *An **enhanced size change graph (ESCG)** is a triple, $\langle G, p, l\rangle$ where $G$ is a GSCG and one of the two conditions hold:*

- $p = G$ *is an GSCG and* $l = 1$*, or*
- $p$ *is a pair of ESCGs,* $\langle H', H'' \rangle$ *such that* $G = G' \cdot G''$*, and* $l = l' + l''$*, where* $H' = \langle G', p', l' \rangle$ *and* $H'' = \langle G'', p'', l'' \rangle$*.*

*The* ***enhancement*** *of an existing GSCG,* $G$*, denoted* $\overline{G}$*, is the ESCG,* $\langle G, G, 1 \rangle$*.*

**Definition 18.** *The* ***corresponding multipath*** *of an ESCG,* $H = \langle G, p, l \rangle$*, denoted* $\texttt{path}(H)$*, is* $G$ *if* $p = G$*, and* $\texttt{path}(H_1)\texttt{path}(H_2)$ *if* $p = \langle H_1, H_2 \rangle$*.*

**Definition 19.** *The* ***composition*** *of two ESCGs,* $H_1 = \langle G_1, p_1, l_1 \rangle$ *and* $H_2 = \langle G_2, p_2, l_2 \rangle$*, denoted* $H_1 \cdot H_2$ *is the ESCG,* $\langle G_1 \cdot G_2, \langle H_1, H_2 \rangle, l_1 + l_2 \rangle$*.*

**Lemma 4.** *Given an ESCG,* $H = \langle G, p, l \rangle$*,* $G = [\![\texttt{path}(H)]\!]$*.*

Let $S' \leftarrow$ the enhancements of all the GSCGs of the ACCG.
Let $\prec$ s.t. $\langle G, p, l \rangle \prec \langle G', p', l' \rangle$ iff $l < l'$.
**repeat**
  $S \leftarrow S'$
  **for all** $H = \langle G, p, l \rangle, H' = \langle G', p', l' \rangle \in S$ **do**
    **if** $\exists H'' = \langle G'', p'', l'' \rangle \in S'$ s.t. $G'' = G \cdot G'$ **then**
      $S' \leftarrow S' - \{H''\} \cup \min_{\prec}\{H \cdot H', H''\}$
    **else**
      $S' \leftarrow S' \cup \{H; H'\}$
**until** $S = S'$
**if** $\exists H = \langle G, p, l \rangle$ s.t. $G$ is idempotent with no edge of the form $m \xrightarrow{>} m$ **then**
  **return** $\texttt{path}(H)$
**else**
  **return** `true`

**Fig. 2.** An algorithm for solving SCC.

**Theorem 6.** *The algorithm given in Figure 2 solves SCC.*

*Proof.* The algorithm behaves exactly as the one in Theorem 2, except that we keep track of which ESCGs were composed to create each new ESCG and the length of the path corresponding to the ESCG. Thus there exists an idempotent ESCG without an edge $m \xrightarrow{>} m$ iff SCT also discovers such an edge. Therefore, if $SCT(\mathcal{G}) = \texttt{true}$, $SCC(\mathcal{G}) = \texttt{null}$, and if $SCT(\mathcal{G}) = \texttt{false}$, $SCC(\mathcal{G})$ returns a circular multipath, $\pi$ such that $[\![\pi]\!]$ is idempotent and contains no edge of the form $m \xrightarrow{>} m$. Suppose there was some decreasing thread corresponding to $\pi^\omega$. Then by the pigeon hole principle, there would be some $k$ such that $\pi^k$ such that there existed a decreasing thread from some $m$ to itself. By Proposition 1, this means that $[\![\pi^k]\!]$ would have an edge $m \xrightarrow{>} m$. But by the definition of idempotence, $[\![\pi^k]\!] = [\![\pi]\!]$, and we already stated that no $[\![\pi]\!]$ has no edge of the form $m \xrightarrow{>} m$. Therefore, there is no infinite decreasing thread for $\pi^\omega$. By definition, then, $\pi$ is an SCC. □

**Theorem 7.** *The algorithms in Theorem 2 and Figure 2 have the same complexity.*

*Proof.* To the original data structures, we add pointers to two ESCGs and an integer representing the length of a path whose evaluation leads to the corresponding ESCG. Since there can be exponential ESCGs, these added fields require linear length in the size of the original problem. In the loop itself, we perform one addition and one comparison of the lengths, which takes linear time, and create two pointers to the ESCGs being composed, which takes constant time. This is eclipsed by the composition of the GSCGs, which has complexity that is polynomial and greater than linear. Thus, there is no overall change in the complexity from the SCT algorithm to the SCC algorithm. □

## 4 Interactive CCG

Termination core analysis, as described in this paper is implemented in ACL2s, the ACL2 Sedan. When reporting termination cores to ACL2s users, our goal is to give as specific and concrete a reason for the failure to prove termination as possible, taking into account everything the termination prover discovered in its proof attempt. The hope is that termination cores will be effective tools for helping users efficiently debug failed termination proofs.

Once a user figures out why the termination proof attempt failed, she must then be able to interact with the termination engine, in order to guide it towards a proof. We have developed a clean and intuitive interface that gives the user a way to interact with the theorem prover without getting bogged down in the details of CCG analysis. The interface is based on the three possible reasons CCG can fail to prove termination.

The first and most obvious source of failure is a non-terminating program. In this case, our analysis will reach a point at which it finds an SCC that represents an actual infinite run of the program. This is particularly useful in programs with multiple recursive behaviors, as it enables the user to find the relevant code and make revisions. Consider, for example, the following program. The reader is encouraged to figure out what is going on before reading further.

```
f1 w r z s x y a b zs =
  if (a > 0) then
     f2 w r z 0 r w 0 0 zs
  else w = r^zs

f2 w r z s x y a b zs =
  if z > 0 then
     f3 w r z s x y y s zs
  else f1 s r (z-1) 0 0 0 0 0 zs

f3 w r z s x y a b zs =
   if a > 0 then
      f3 w r z s x y (a-1) (b+1) zs
   else f2 w r z b (x-1) y 0 0 zs
```

ACL2s produces the following core:

```
f3_0 w r z s x y a b zs =
  if a <= 0 then
    f2_0 w r z b (x-1) y 0 0 zs
  else
    [w; r; z; s; x; y; a; b; zs]
```

```
f2_0 w r z s x y a b zs
  if z > 0 then
    f3_0 w r z s x y y s zs
  else
    [w; r; z; s; x; y; a; b; zs]
```

From the termination core, we see that the only value consistently decreasing in this loop is `x`, which decreases by 1 each time through the loop. The problem is that there is no test to see that `x` is positive. Instead, we check that `z` is positive. This is easily remedied by changing `z > 0` to `x > 0` in the definition of `f2`. Does the program terminate now? Readers are encouraged to construct a measure and to mechanically verify it. (It took us about 20 minutes.) If we submit the updated program, CCG analysis proves termination in under 2 seconds, fully automatically with no user guidance.

The above program was generated by applying weakest precondition analysis to a triply-nested loop. An expert with over a decade of theorem proving experience spent 4–6 hours attempting to construct a measure function that could be used to prove termination, before giving up. So, this example highlights how useful termination analysis can be and how termination core analysis can help users discover and correct termination bugs.

A second reason that CCG analysis may fail to prove termination is that it may fail to guess the necessary Calling Context Measures (CCMs) [9, 13]. Calling context measures can be thought of as building blocks for conventional measures. We use simple heuristics to guess CCMs, and while they are effective for many programs, they are certainly not complete. Consider, the following program.

```
dec x = if x <= 0 then 255 else x-1
f x = if x = 1 then 0 else 1 + (f (dec x))
```

Our heuristics choose |`x`| as the CCM for the sole recursive call in `f`. However, this measure does not always decrease across the recursive call. For example, if `x` is `0`, `(dec x)` is `255`. The reader is encouraged to prove termination using the standard measure-based approach.

The termination core produced by our algorithm is as follows.

```
f_0 x = if x = 1 then [x] else (f_0 (dec x))
```

This termination core is not terribly helpful, since it has the exact same looping behavior as the original. However, our core generator also lists the CCMs chosen for each context, as well as the edges of the relevant GSCGs. For our example, our analysis will inform the user that the sole CCM chosen was |`x`|, which cannot be shown to be non-increasing or decreasing from one iteration of the loop to the next. A quick look at the definition of `dec` confirms that this is not an appropriate CCM. However, `(dec x)` is a useful CCM. That is, it is easy to prove that if `x` is not `1`, `(dec (dec x)) < (dec x)`.

We provide an interface that allows users to override the heuristics for guessing CCMs by providing one of two hints to the CCG algorithm. The first is

the `:CONSIDER` hint, which takes a list of expressions over the parameters of the function to which the user wishes to apply the hint. This tells the CCG analysis to add the given CCMs to those heuristically generated for all the contexts in the function to which it is applied. In our example, a `:CONSIDER [(dec x)]` hint will result in measures {|x|, `(dec x)`} for the sole context.

In some cases, users may want even more control. For example, |x| is irrelevant for the termination proof. Such CCMs lead to needless theorem prover queries. Therefore, we also provide a `:CONSIDER-ONLY` hint. This is identical in usage to the `:CONSIDER` hint, but tells the CCG analysis to use *only* those measures provided by the user for the given function. Thus, the hint `:CONSIDER-ONLY [(dec x)]` in our example will result in `(dec x)` being the only measure for the sole context. Giving this hint leads to a simpler termination proof.

The final reason that CCG may fail to prove termination is that it was unable to prove a necessary theorem about either the exclusion of an edge from the CCG or about the relationship between two measures across a recursive call. Consider, for example the following definition of merge sort.

```
mergesort x =
   if x = [] or tl x = []
   then x
   else mergelists (mergesort (evens x)) (mergesort (odds x))
```

Here, `(evens [`$e_0$`; `$e_1$`; ...; `$e_n$`])` returns `[`$e_0$`; `$e_2$`; ...]` and `odds` applied to the same list returns `[`$e_1$`; `$e_3$`; ...]`. Our analysis produces the following core:

```
mergesort_0 x = if not (x = []) and (not (tl x = []))
                then  mergesort_0 (evens x) else [x]
```

It also tells us that the sole measure |x| (*i.e.*, the length of `x`) could not be shown to be decreasing. The problem is that the theorem prover was unable to prove that `evens` always returns a list that is smaller than the input if that input list has 2 or more elements in it. It turns out that ACL2d needs some guidance to get this proof to go through. If we prove this lemma, termination still fails, but we get a new core.

```
mergesort_0 x = if not (x = []) and (not (tl x = []))
                then mergesort_0 (odds x) else [x]
```

We have the same problem with `odds` that we had with `evens`. A similar lemma leads to a successful termination proof. Note the interactive nature of this process. There were two different reasons for CCGs inability to prove termination. Rather than simply giving up, the CCG analysis shows the user each reason for non-termination, one at a time, thereby enabling the user to successfully address these issues and prove termination.

## 5 Conclusions

We examined the issue of building termination analysis engines that are both highly automatic and easy to use in an interactive setting. The challenge is in un-

derstanding and then dealing with failure. To this end, we introduced the notion of a *termination core*, a simplification of the program under consideration which consists of a single loop that the termination engine cannot prove terminating. Termination cores are used to help users understand why a termination analysis engine failed to prove termination. We showed how to extend Size Change Termination so that it generates a termination core when it fails to prove termination. We showed that this is a PSPACE-complete problem and presented a practical algorithm that adds termination core generation to the Calling Context Graph termination analysis, a recent, powerful termination analysis that is part of the ACL2 Sedan. We also presented several convenient ways of allowing users to interact with CCG analysis, so that they can guide it to a termination proof after analyzing the termination core that was generated. These techniques are implemented in the ACL2 Sedan, a freely available, open-source, well-supported theorem prover that is based on ACL2, but was designed with greater usability and automation as primary design considerations.

## References


1. James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *POPL '08*, pages 101–112. ACM, 2008.
2. Harsh Raju Chamarthi, Peter C. Dillinger, Panagiotis Manolios, and Daron Vroon. ACL2 Sedan homepage. See URL `http://acl2s.ccs.neu.edu/`.
3. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI '06*, pages 415–426. ACM, 2006.
4. Peter C. Dillinger, Panagiotis Manolios, Daron Vroon, and J. Strother Moore. ACL2s: "The ACL2 Sedan". *Electr. Notes Theor. Comput. Sci.*, 174(2):3–18, 2007.
5. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.
6. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.
7. Matt Kaufmann and J Strother Moore. ACL2 homepage. See URL `http://-www.cs.utexas.edu/users/moore/acl2` (08/2007).
8. Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL '01*, volume 28, pages 81–92. ACM, 2001.
9. Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In *CAV '06*, pages 401–414. Springer, 2006.
10. A. Prasad Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for büchi automata with applications to temporal logic. *Theor. Comput. Sci.*, 49(2-3):217–237, 1987.
11. Reneé Thiemann and Jürgen Giesl. Size-change termination for term rewriting. Technical Report AIB-2003-02, RWTH Aachen, January 2003.
12. Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher order workshop conference on Logics for concurrency*, pages 238–266. Springer, 1996.
13. Daron Vroon. *Automatically Proving the Termination of Functional Programs*. PhD thesis, Georgia Intitute of Technology, 2007.