Certifying a termination criterion based on graphs, without graphs*

Pierre Courtieu[1], Julien Forest[2], and Xavier Urbain[2]

[1] CÉDRIC – CNAM, Paris, France
[2] CÉDRIC – ENSIIE, Évry, France

Abstract. Although graphs are very common in computer science, they are still very difficult to handle for proof assistants as proving properties of graphs may require heavy computations. This is a problem when it comes to issues such as the certification of a proof of well-foundedness, since premises of generic theorems involving graph properties may be at least as difficult to prove as their conclusion. We define a framework and propose an original approach based on both shallow and deep embeddings for the mechanical certification of these kinds of proofs without the help of any graph library. This framework actually avoids concrete models of graphs and handles those implicitly. We illustrate this approach on a powerful refinement of the dependency pairs approach for proving termination. This refinement makes heavy use of graph analysis and our technique is powerful enough to deal efficiently –and with full automation– with graphs containing thousands of arcs, as they may occur in practice.

1 Introduction

The halting problem is a well-known undecidable problem and its related property, *termination*, plays a fundamental role at several levels in many proofs and definitions. For instance the termination of a relation $\rightarrow$, i.e. the *well-foundedness* of its inverse, is crucial for induction proofs with reference to $\leftarrow$; functions which are *total*, i.e. functions whose computation always terminates, are often compulsory in some proofs assistants; some properties like the confluence of a relation become decidable as soon as the relation is proven to be terminating, etc. Termination is also compulsory when proving *total correctness* of programs.

Discovering a termination proof is often very tricky. The past decade has been rich in developments of automated tools dedicated to termination proofs [9, 11, 14, 15, 21], in particular in the context of first order term/string rewriting systems which we address here. However, skeptical proof assistants [17, 18] cannot take for granted the answers of these tools: they need a formal proof of handled properties. Hence, two of the main concerns are: 1) developing powerful techniques to prove termination of more and more relations with full automation, and 2) obtaining formal (mechanical) certificates of well-foundedness for these relations, in order to enable their definition and use in skeptical proof assistants. We will address here point 2).

* Work partially supported by A3PAT project of the French ANR (ANR-05-BLAN-0146-01).

Regarding termination criteria, most tools now use the Dependency Pairs (DP) approach, introduced in 1997 by Arts & Giesl [1,2]. Contrary to the historical Manna and Ness criterion, the main idea of dependency pairs does not consist in discovering a well-founded, monotonic (i.e., closed by context) and stable (w.r.t. instantiation) ordering for which *each rule* of the system decreases strictly. Roughly speaking, it focuses instead on the possible inner recursive calls of rules (the so-called dependency pairs). This leads to constraints on suitable orderings that are much easier to fulfil. For details, see [2]. This approach has been made even more powerful by use of multiple refinements: in particular it can benefit greatly from the analysis of a *dependency graph*, especially when different orderings can be used [10].

Regarding certification of automated proofs, the project A3PAT[1] aims at improving cooperation between automated provers and skeptical proof assistants [20]. The idea is to get some proof trace from an (efficient) automated prover and translate it into a proof script which can be certified by a targeted proof assistant (which often lacks automation), possibly with the help of dedicated libraries. Eventually we obtain *automatically* a proof of the wanted theorem. One original point of our approach is that it mixes shallow and deep embeddings, this may ease the work of the proof assistant significantly. This work takes place in project A3PAT and focuses on proofs involving *graph analysis*.

Regarding termination in particular, a few libraries model the base theory of dependency pairs [7, 19, 20]. However, regarding the use of graphs, some properties may require heavy computations and particularly involved algorithmics, and may be very difficult to overcome for a proof assistant (even with the help of dedicated libraries). For instance the strong version of the enhanced dependency graph theorem [10] states that one has to find a suitable ordering "for *each* cycle of the graph", that is: the property has to be shown for each cycle separately, and moreover one has to prove that *all* cycles have been considered. Applying directly such a theorem is currently out of reach regarding the size of graphs that occur in the practice of termination proof.

We propose here a method which allows certification of (termination) proofs based on a graph analysis. Our technique can manage efficiently graphs containing thousands of arcs; it is implemented in the prototype developed by the A3PAT project [20]. Our prototype instantiates our approach and generates termination proof scripts that just have to be compiled and type-checked by the COQ proof assistant[2], possibly with the help of our library COCCINELLE [19].

It is to date the only one able to use the power of the enhanced graph criterion in its strongest version (yet without the so-called usable rules). We consider enhanced graph criteria [10] only as they are much more powerful than original ones [2] which they subsume; for the sake of readability, we will simply write "graph criteria".

Since our approach uses shallow embedding, we describe an instantiation of it, on termination proofs. Thus, in Preliminaries, we will recall some notions and results about graphs, termination of term rewriting, termination proofs and the DP approach with graphs refinements. As noted in [20] and pictured as *processors* in [12] criteria can be expressed in a uniform setting. We will see in Section 3 that we can even write them as formal inference rules. For each of the considered graph criteria, we will give the corre-

[1] `http://a3pat.ensiie.fr`
[2] `http://coq.inria.fr`

sponding rule. Then, in Section 4, we will focus on how we model dependency graphs so as to certify termination proof using a proof assistant like Coq. We will eventually provide some experiments to illustrate the efficiency of our approach.

2 Preliminaries

We assume the reader to be familiar with basic concepts of graphs [6], and of term rewriting [5, 8] and termination, in particular with the Dependency Pairs approach [2]. We recall the usual notions, and give our notations.

2.1 Graphs

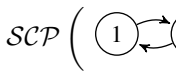
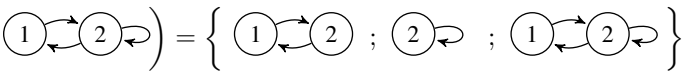
Definition 1 (Graph, Path). *A* graph $\mathcal{G}$ *is a pair* (N, A) *where* N *is a set of* vertices*, and* $A \subset N^2$ *is a set of* arcs*. We say that an arc* $a = (n_1, n_2)$ *goes from vertex* n_1 *to vertex* n_2 *(noted* $n_1 \mapsto_{\mathcal{G}} n_2$*).*
A finite path *from a vertex* u *to a vertex* v *is a sequence of arcs* $(u \mapsto n_0, \ldots, m_i \mapsto n_i, \ldots, m_{k-1} \mapsto v)$ *such that* $\forall i, 0 < i < k, n_{i-1} = m_i$*. In our case a path is completely determined by the sequence of vertices that it encounters.*

Definition 2 (Subgraph). *let* $\mathcal{G} = (\mathscr{D}, A)$ *be a graph, the* subgraph of $\mathcal{G}$ generated by $\mathscr{D}' \subset \mathscr{D}$ *is the graph* $\mathcal{G}_{\mathscr{D}'} = (\mathscr{D}', A')$ *such that* $A \supseteq A' = \{n \mapsto n' \mid n, n' \in \mathscr{D}'\}$*. We also note* $\mathcal{G} \backslash d$ *the subgraph* $\mathcal{G}_{\mathscr{D} \backslash d}$*.*

Definition 3 (Strongly connected parts of a graph). *Let* $\mathcal{G} = (N, A)$ *be a graph. A* strongly connected part *(scp) of* $\mathcal{G}$ *is a subset* C *of* N *such that for all* $a, b \in C$ *there is a path from* a *to* b *in subgraph* $\mathcal{G}_C$*. We denote* $\mathcal{SCP}(\mathcal{G})$ *the set of all scp of* $\mathcal{G}$*.*

A strongly connected component *(scc) of a graph* $\mathcal{G}$ *is a maximal strongly connected part of* $\mathcal{G}$*. We denote* $\mathcal{SCC}(\mathcal{G})$ *the set of all scc of* $\mathcal{G}$*.*

In the example below, we see that a scc contains a set of scp:

$\mathcal{SCP}$  $=$ 

Definition 4 (directed acyclic graph of connected components). *Let* $\mathcal{G}$ *be a graph. The directed acyclic graph (DAG) of strongly connected components of G is the graph:* $\mathcal{G}^{\mathcal{SCC}} = (\mathcal{SCC}(\mathcal{G}), \{c_1 \mapsto c_2 | c_1, c_2 \in \mathcal{SCC}(\mathcal{G})$ *and* $\exists (n_1, n_2) \in c_1 \times c_2, n_1 \mapsto_{\mathcal{G}} n_2\})$.

It is easy to see that $\mathcal{G}^{\mathcal{SCC}}$ is the directed acyclic graph of strongly connected components of $\mathcal{G}$ since components are *maximal* strongly connected parts. All graphs in this work are finite, in particular $\mapsto^+_{\mathcal{G}^{scc}}$ is a finite (well-founded) ordering.

2.2 Rewriting

A *signature* $\mathcal{F}$ is a finite set of *symbols* with arities. Let X be a countable set of *variables*; $T(\mathcal{F}, X)$ denotes the set of finite *terms* on $\mathcal{F}$ and X. $\Lambda(t)$ is the symbol at root position in term t. We write $t|_p$ for the subterm of t at position p and $t[u]_p$ for term t

where $t|_p$ has been replaced by u. *Substitutions* are mappings from variables to terms and $t\sigma$ denotes the application of a substitution σ to a term t.

A *term rewriting system* (TRS for short) over a signature $\mathcal{F}$ is a set R of *rewrite rules* $l \to r$ with $l, r \in T(\mathcal{F}, X)$. In this work we restrict to *finite* systems. A TRS R defines a monotonic relation $\to_R$ closed under substitution (aka a *rewrite relation*) in the following way: $s \to_R t$ (s *reduces to* t) if there is a position p such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$ for a rule $l \to r \in R$ and a substitution σ. We shall omit systems and positions that are clear from the context. We denote the reflexive-transitive closure of a relation $\to$ by $\to^\star$. Symbols occurring at root position in the left-hand sides of rules in R are said to be *defined*, the others are said to be *constructors*.

A term is *R-strongly normalizable* (R-SN) if it cannot reduce infinitely many times for $\to_R$. A rewrite relation $\to_R$ *terminates* if any term is R-SN, which we denote $\mathsf{SN}(\to_R)$. In such case we may say that R terminates. This is equivalent to $\leftarrow_R$ is well-founded that is, every term is *accessible* for $\leftarrow_R$.

Dependency pairs. In this section, we briefly recall main definitions and results about dependency pairs, dependency chains and dependency graphs. We introduce some of our notations.

Definition 5 (Dependency pairs, Dependency chain [2]). *The set of* unmarked *dependency pairs of a TRS R, denoted $\mathsf{DP}(R)$ is defined as $\{\langle u, v\rangle \mid u \to t \in R$ and $t|_p = v$ and $\Lambda(v)$ is defined$\}$. Let $\mathscr{D}$ be a set of dependency pairs, a dependency chain in $\mathscr{D}$ is a sequence of dependency pairs $\langle u_i, v_i\rangle$ with a substitution σ such that*

$$\forall i,\ v_i\sigma \xrightarrow[R]{\neq\Lambda\ \star} u_{i+1}\sigma$$

It is worth noticing that distinguishing root symbols of dependency pairs (by means of marks, or 'tuple-symbols') enhances significantly this technique. Marking or not dependency pairs does *not* interfere with our approach, thus for readability's sake, we will restrict to unmarked pairs. Further note that our approach and prototype handle marks without any problem [20].

Definition 6. *Given a TRS R, we note $s \to_{\mathsf{DPR}(\mathscr{D})} t$ iff $s \xrightarrow[R]{\neq\Lambda\ \star} u\sigma \xrightarrow[\langle u,v\rangle\in\mathscr{D}]{\Lambda} v\sigma \equiv t$.*

The main theorem of dependency pairs of [2] can be rephrased using the DPR relation:

Theorem 1 (Dependency Pairs Criterion). *Let R be a TRS, $\to_{\mathsf{DPR}(\mathsf{DP}(R))}$ terminates if and only if $\to_R$ terminates.*

Termination of relation $\to_{\mathsf{DPR}(\mathscr{D})}$ may be directly proved by mean of an ordering pair, a very general notion of which may be found in [16]. Due to our definition of the DPR relation, we use a slightly restricted definition of those but it does not interfere with the topic of this work.

Definition 7 (Ordering pair). *An ordering pair is a pair $(\succeq, >)$ of relations over $T(\mathcal{F}, X)$ such that: 1) $\succeq$ is a quasi-ordering, i.e. reflexive and transitive, 2) $>$ is a*

strict ordering, i.e. irreflexive and transitive, and 3) $\succeq \cdot > = >$.
An ordering pair $(\succeq, >)$ *is* well-founded *if there is no infinite strictly decreasing sequence* $t_1 > t_2 > \ldots$, *which we denote* $\mathsf{WF}(\succeq, >)$.

An effective corollary of Theorem 1 consists in discovering a well-founded ordering pair $(\succeq, >)$ for which $\to_R \subseteq \succeq$ and $u > v$ for all $\langle u, v\rangle \in \mathscr{D}$ to prove that $\to_{\mathsf{DPR}(\mathscr{D})}$ terminates.

Many efficient termination tools (see for instance [11, 14, 21]) use this criterion as a first step. Then, one is left with proving that there is no infinite dependency chain. In the following we describe the graph criterion which allows to split this proof into easier ones.

Dependency Pairs with graph. Not all DPs can follow another in a dependency chain, one may consider the graph of possible sequences of DPs. Note that since we restricted to finite TRSs, this graph is finite. Thus, each dependency chain corresponds to a *path* in this graph. Therefore if there is no infinite path *corresponding to a dependency chain* in the graph, then there is no infinite dependency chain.

Definition 8 (Dependency graph [2]). *Let* R *be a rewriting system. The* dependency graph of R *is the graph* $\mathcal{G} = (\mathsf{DP}(R), A)$ *where* $\langle s, t\rangle \mapsto \langle s', t'\rangle \in A$ *if and only if there exists a substitution* σ *such that* $t\sigma \xrightarrow{\neq \Lambda\ \star} s'\sigma$.

Remark 1. It is worth noticing that this graph $(\mathscr{D}, A)$ is not computable, so one uses a sound approximation i.e., a graph $(\mathscr{D}, A' \supseteq A)$ that contains it. Arts & Giesl proposed a simple yet efficient approximation, namely *connectability* (with REN/CAP) [2]. The approximation we choose to implement in our prototype corresponds to this simple one (see Section 4.2).

The (enhanced) graph refinement, as stated by Giesl, Arts and Ohlebusch is:

Theorem 2 (Dependency graph refinement [10]). *A TRS* R *terminates if and only if for each circuit* C *in the dependency graph there exists no infinite dependency chain of dependency pairs of* C.

Note that it is *not* sufficient to consider elementary cycles instead of circuits[3] (for a counterexample, see [4]).

Further note that proving in a proof assistant that his theorem can be applied to a particular termination problem amounts to proving that *all* cycles have been considered, which is difficult in practice. We provide hereafter an approach to avoid this problem.

2.3 Modelling rewriting and graphs in Coq

The goal of our methodology and prototype is to be able to derive *with full automation*, from the definition of a rewrite system R, a proof certified (i.e. checked) by a skeptical

[3] This is why we use the word "circuit" instead of the original "cycle", cf. [6].

proof assistant. Regarding termination proofs, our tool generates a lemma of the form `well_founded R` together with its proof.

We will reuse the model we introduced in [20]. We only recall here the notions and notations used in this paper.

We illustrate our approach using the COQ proof assistant which is based on *type theory* and enjoys in particular the ability to define *inductive types* to express inductive data types and inductive properties, and a very expressive tactic language. Tactics in COQ unsafely produce proof terms which are safely validated at saving time by type checking the proof.

If `R` is the relation modelling a TRS $\mathcal{R}$, we should write `R` $u\ t$ (which means $u < t$) when a term t rewrites to a term u. For the sake of readability we will use as much as possible the COQ notation: t `-[R]>` u (and t `-[R]*>` u for $t \to^* u$) instead.

We use in this work a deep embedding for term algebras and shallow embedding for rewriting relations. In COQ scripts below a term $f(x, y, z)$ will be denoted `Term f [x;y;z]`, and $f(x, y, z) \to_R^\star g(a, z)$ will be denoted `Term f [x;y;z] -[R]*> Term g [a;z].`

3 Formalizing graph refinements

Our project aims at making skeptical proof assistants and automated provers cooperate. Hence, our presentation of the graph refinement differs from the original one from Arts and Giesl [2] in order to fit our general scheme for proving properties on graphs. Such a general scheme could form a basis for a general trace language, similar to the processors setting [11]. For example, in our framework, Theorem 1 is expressed formally by the following inference rule:

$$\frac{\mathsf{SN}(\to_{\mathsf{DPR}(\mathsf{DP}(R))})}{\mathsf{SN}(\to_R)}\ \text{DP}$$

The graph criterion consists in proving that there is no infinite dependency chain by proving that *strongly connected parts* of the graph cannot be crossed infinitely many times by a dependency chain.

Definition 9. *Let P be a strongly connected part of a dependency graph, we denote by $\mathsf{SNG}(P)$ the property that there is no reduction in $\to_{\mathsf{DPR}(P)}$ such that each vertex of P is crossed infinitely many times. Given $E = \{P_1, \ldots, P_k\}$ we denote $\mathsf{SNG}(E) \equiv \bigwedge_{1 \le i \le k} \mathsf{SNG}(P_i)$.*

The main theorem of the graph criterion can be rephrased as follows:

Theorem 3 (graph criterion). *Let R be a rewriting system, let $\mathcal{G}$ be its dependency graph. Let $P_1, \ldots, P_k$ be the k scp of $\mathcal{G}$ (the $P_i \in \mathcal{SCP}(\mathcal{G})$ are the subgraphs of $\mathcal{G}$). $\mathsf{SN}(\to_{\mathsf{DPR}(\mathsf{DP}(R))})$ if and only if $\mathsf{SNG}(\{P_1, \ldots, P_k\})$.*

This theorem can in turn be expressed by the following inference rule:

$$\frac{\mathsf{SNG}(\mathcal{SCP}(\mathcal{G}))}{\mathsf{SN}(\to_{\mathsf{DPR}(\mathsf{DP}(R))})}\ \textsc{Graph}$$

Where $\mathcal{G}$ is the (approximated) dependency graph of R. Note that the termination proof of each scp may be done using a different ordering. In practice this is expensive. Instead, we will gather scp into subsets of $\mathcal{SCP}(\mathcal{G})$, which will be recursively proved to be terminating separately. Actually, the graph criterion can be completed by the following rule for recursive splitting:

$$\frac{\mathsf{SNG}(X_1) \dots \mathsf{SNG}(X_k)}{\mathsf{SNG}(X)} \text{ SUBGRAPH}$$

where $\bigcup_{1 \leq i \leq k} X_i = X$.

Since the set of strongly connected *components* covers all scp of $\mathcal{G}$, one way to use the graph criterion is to prove that $\to_{\mathsf{DPR}(X_i)}$ terminates for all $X_i \in \mathcal{SCC}(\mathcal{G})$. A weak version of the graph criterion consists in providing *for each strongly connected component* an ordering pair that decreases strictly for all its vertices, and weakly for all rules of the initial system.

The whole termination proof for a system R by this weak graph criterion may be represented by a proof tree like:

$$\dfrac{\dfrac{\dfrac{\mathsf{WF}(\leq_1, <_1) \quad \to_R \subseteq \leq_1 \quad \to_{\mathsf{DPR}(\mathcal{G}_{C_1})} \subseteq <_1}{\mathsf{SNG}(\mathcal{SCP}(C_1))} \text{ ORD} \quad \dots \quad \dfrac{\mathsf{WF}(\leq_k, <_k) \quad \to_R \subseteq \leq_k \quad \to_{\mathsf{DPR}(\mathcal{G}_{C_k})} \subseteq <_k}{\mathsf{SNG}(\mathcal{SCP}(C_k))} \text{ ORD}}{\mathsf{SN}(\to_{\mathsf{DPR}(\mathsf{DP}(R))})} \text{ GRAPH}}{\mathsf{SN}(\to_R)} \text{ DP}$$

Where $\mathcal{G}$ is the (approximated) dependency graph of R, and $C_1 \dots C_k$ are the strongly connected components.

The graph criterion in its strong version consists in partitioning the set of scp of $\mathcal{G}$ in parts smaller than scc. An efficient technique, due to Middledorp and Hirokawa [13], is to apply recursively the following steps for each scc C of $\mathcal{G}$:

1. choose a node $p = \langle t, u \rangle$ of C;
2. prove that each scp D containing p is such that $\mathsf{SNG}(D)$;
3. prove (recursively) that each scp of the remaining graph is SNG.

This technique can be formalized by the following application of SUBGRAPH:

$$\frac{\mathsf{SNG}(\mathcal{SCP}(\mathcal{G} \backslash \langle t, u \rangle)) \quad \mathsf{SNG}(\{P \in \mathcal{SCP}(\mathcal{G}) | \langle t, u \rangle \in P\})}{\mathsf{SNG}(\mathcal{SCP}(\mathcal{G}))} \text{ SUBGRAPH}$$

where $\langle t, u \rangle \in \mathcal{G}$. Notice that the rule SUBGRAPH is applied correctly since $\mathcal{SCP}(\mathcal{G} \backslash \langle t, u \rangle) \cup \{P \in \mathcal{SCP}(\mathcal{G}) | \langle t, u \rangle \in P\} = \mathcal{SCP}(\mathcal{G})$.

Usually step 3 is done by computing $\mathcal{G}_1 \dots \mathcal{G}_n$ (the scc of $\mathcal{G} \backslash \langle t, u \rangle$), and by applying recursively the SUBGRAPH rule to each $\mathcal{G}_i$ until one of the following happens:

- there is no more scc,
- one finds an ordering pair $(\leq, <)$ such that $\to_R \subseteq \leq$ and $\to_{\mathsf{DPR}(\mathcal{G})} \subseteq <$.

Usually step 2 is done by discovering a well-founded ordering pair $(\leq, <)$ such that $\to_R \subseteq \leq$, $\to_{\mathsf{DPR}(C \backslash \langle t,u \rangle)} \subseteq \leq$ and $t < u$. This is rule VERTEX.

Finally a typical graph criterion application is illustrated by Figure 1.

$$\textsc{SubGraph}\ \dfrac{\dfrac{\vdots}{\mathsf{SNG}(\mathcal{SCP}(\mathcal{G}_1))}\ \ldots\ \dfrac{\vdots}{\mathsf{SNG}(\mathcal{SCP}(\mathcal{G}_k))}}{\mathsf{SNG}(\mathcal{SCP}(\mathcal{G}\backslash\langle t,u\rangle))}\quad \dfrac{\begin{array}{cc}\mathsf{WF}(\leq,<) & t<u\\ \rightarrow_R\subseteq\leq & \rightarrow_{\mathsf{DPR}(\mathcal{G}\backslash\langle t,u\rangle)}\subseteq\leq\end{array}}{\mathsf{SNG}(\{P\in\mathcal{SCP}(\mathcal{G})|\langle t,u\rangle\in P\})}\ \textsc{Vertex}}{\mathsf{SNG}(\mathcal{SCP}(\mathcal{G}))}\ \textsc{SubGraph}$$

where $\mathcal{G}_1 \ldots \mathcal{G}_k$ are scc of $\mathcal{G}\backslash\langle t,u\rangle$.

Fig. 1. Typical application of strong graph criterion

4 Mechanical certification of the graph refinement

The key point of our approach is that the graph will be defined implicitly. We never actually model a graph, we just use a set of vertices and a relation between them to build it implicitly as we prove the relevant property on its parts, in a hierarchical fashion. Regarding termination proofs: vertices will be dependency pairs, a pair p_1 will be in relation with a pair p_2 if p_1p_2 may occur in a dependency chain.

In the formal proof that is generated automatically by our technique, each rule applied corresponds to an independent lemma. We described the proof techniques relevant to some of these rules in a previous work [20]. Some of these lemmas are proved using generic theorems previously formalized in a deep embedding, some others are proved by generating directly a shallow embedded proof. Graph refinement rules are of the latter category. As explained in the introduction, the reason is that the premises of rules GRAPH and SUBGRAPH are computationally hard to deal with. For example to prove an application of rule SUBGRAPH one has not only to prove recursively $\mathsf{SNG}(X_1)\ldots\mathsf{SNG}(X_k)$ *but also* to prove that $\bigcup_{1\leq i\leq k} X_i = X$. Such completeness properties are known to be difficult to prove.

Instead of relying on a generic proof of GRAPH and SUBGRAPH, we generate a direct proof for each application of these rules. This proof is done by induction on the possible dependency chains in the initial graph X. In particular this induction follows a graph that is now *implicit*.

Suppose we have to prove an application of a graph refinement as shown in Figure 1. The goal of our methodology is to build from this tree (output by an automated tool), a formal proof of the property $\mathsf{SNG}(\mathcal{SCP}(\mathcal{G}))$. To that purpose, we[4] will generate two lemmas proved by different techniques:

- $\mathsf{SNG}(\mathcal{SCP}(\mathcal{G}\backslash\langle t,u\rangle)) \Rightarrow \mathsf{SNG}(\mathcal{SCP}(\mathcal{G}))$ proved by induction on the well-founded ordering $<$, and
- $\mathsf{SNG}(\mathcal{SCP}(\mathcal{G}\backslash\langle t,u\rangle))$ proved by a hierarchical decomposition of $\mathcal{G}\backslash\langle t,u\rangle$.

We describe this hierarchical decomposition in more details in the next section.

4.1 Hierarchical decomposition of $\mathcal{SCC}(\mathcal{G})$

In order to prove $\mathsf{SNG}(\mathcal{SCP}(\mathcal{G}\backslash\langle t,u\rangle))$ in a shallow embedded way, we proceed as follows:

[4] More precisely "the tool in which this approach is implemented", since all this is done without any human interaction.

– Compute the DAG $(\mathcal{G}\backslash\langle t,u\rangle)^{\mathcal{SCC}}$.
– Prove *successively*, for each sub-DAG S_i (rooted by $\mathcal{G}_i$) and *in a bottom-up fashion*: $\left(\bigwedge_{S\subset S_i}\mathsf{SNG}(\mathcal{SCP}(S))\right)\Rightarrow\mathsf{SNG}(\mathcal{SCP}(S_i))$ (See Figure 3 for an example). Therefore we can formalize this part of the proof as follows:

$$\textsc{SubGraph}\ \frac{\bigwedge_{S_i}\left(\left(\bigwedge_{S\subset S_i}\mathsf{SNG}(\mathcal{SCP}(S))\right)\Rightarrow\mathsf{SNG}(\mathcal{SCP}(S_i))\right)}{\mathsf{SNG}(\mathcal{SCP}(\mathcal{G}\backslash\langle t,u\rangle))}$$

Each proof of $\left(\bigwedge_{S\subset S_i}\mathsf{SNG}(\mathcal{SCP}(S))\right)\Rightarrow\mathsf{SNG}(\mathcal{SCP}(S_i))$ is done by proceeding the same way recursively with the proofs of $\mathsf{SNG}(\mathcal{SCP}(\mathcal{G}_i))$ from Figure 1.

4.2 Formalization of hierarchical decomposition

Dependency chains. In our methodology dependency graphs and sub-graphs are not concrete. Instead, we will work directly on dependency chains which we model by inductive relations. Using inductive types, reasoning on all possible dependency chains can be done by induction on the definition of the relation.

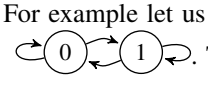
For example suppose we have a scc `SCC` defined by the set of pairs:

$$\texttt{SCC} = \{\langle \texttt{plus(s x,y)},\texttt{plus(x,y)}\rangle, \langle \texttt{plus(x,s y)},\texttt{plus(x,y)}\rangle\}$$

The corresponding dependency chain relation is generated as follows:

```
Inductive SCC : term → term → Prop:=
  SCC0: ∀V₀ V₁, x -[R]*> S(V₀) → y -[R]*> V₁
                       → plus(x,y) -[SCC]> plus(V₀,V₁)
| SCC1: ∀V₀ V₁, x -[R]*> V₀ → y -[R]*> s(V₁)
                       → plus(x,y) -[SCC]> plus(V₀,V₁)
```

Note that `SCC y x` is *exactly* equivalent to `x` $\rightarrow_{\mathsf{DPR}(\texttt{SCC})}$ `y`, in particular notice how head reduction by `R` is disallowed *by construction*. Further note that this relation is not constructively defined: for instance the set of terms `x` such that `x -[R]*> S(V₀)` is not defined explicitly and actually, it cannot be computed in general. There are several possible approximations of this set as noted in Remark 1. In our methodology the approximation lies, *during reasoning*, in the way we discard terms t that cannot reduce to u. Currently our generated proofs implement the simple connectability relation of Arts & Giesl [2].

Sub-DAGs of the dependency chains. A scp $\mathcal{G}_i$ of the graph $\mathcal{G}$ implicitly modelled by DPR is built by restraining the constructors of `SCC` to the set of pairs inside $\mathcal{G}_i$. For example let us consider the following graph corresponding to the relation above: 

. The scp of `SCC` containing only pair $\langle \texttt{plus(s x,y)},\texttt{plus(x,y)}\rangle$ is modelled by the following relation, which corresponds to 

:

```
Inductive SCP₀ : term → term → Prop:=
  SCP₀0: ∀V₀ V₁, x -[R]*> S(V₀) → y -[R]*> V₁
                    → plus(x,y) -[SCP₀]> plus(V₀,V₁)
```

Finally, to prove $\bigwedge_{S_i} \left(\left(\bigwedge_{S \subset S_i} \mathsf{SNG}(\mathcal{SCP}(S)) \right) \Rightarrow \mathsf{SNG}(\mathcal{SCP}(S_i)) \right)$ as explained above, we prove the following equivalent lemmas:

```
Lemma Acc_S0: ∀x y, SCC₀ x y → Acc SCC x.
...
Lemma Acc_Sn: ∀x y, SCCₙ x y → Acc SCC x.
```

The proof of `Acc_Si` may use any `Acc_Sj` for `j<i`. Note that the conclusion is about accessibility in the current graph instead of the well-foundedness of the sub-DAG S_i. Since all dependency chains starting in `SCC_i` can only stay in S_i, those lemmas are equivalent to $\bigwedge_{S_i} \left(\left(\bigwedge_{S \subset S_i} \mathsf{SNG}(\mathcal{SCP}(S)) \right) \Rightarrow \mathsf{SNG}(\mathcal{SCP}(S_i)) \right)$.

5 Examples

5.1 A weak graph criterion example

The example R_1 below is due to Arts and Giesl [1] and computes the sum of the elements of a list:

```
app(nil, k) → k
app(l, nil) → l
app(cons(x,l), k) → cons(x, app(l, k))
sum(cons(x,nil)) → cons(x,nil)
sum(cons(x,cons(y,l))) → sum(cons(plus(x,y),l))
sum(app(l,cons(x,cons(y,k)))) → sum(app(l,sum(cons(x,cons(y,k)))))
plus(0,y) → y
plus(s(x),y) →s(plus(x,y))
```

The dependency pairs of this system are the following:

$1 : \langle \texttt{plus(s(x),y)}, \texttt{plus(x,y)} \rangle$
$2 : \langle \texttt{sum(app(l,cons(x,cons(y,k))))}, \texttt{sum(cons(x, cons(y,k)))} \rangle$
$3 : \langle \texttt{sum(app(l,cons(x,cons(y,k))))}, \texttt{app(l,sum(cons(x,cons(y,k))))} \rangle$
$4 : \langle \texttt{sum(app(l,cons(x,cons(y,k))))}, \texttt{sum(app(l,sum(cons(x,cons(y,k)))))} \rangle$
$5 : \langle \texttt{sum(cons(x,cons(y,l)))}, \texttt{plus(x,y)} \rangle$
$6 : \langle \texttt{sum(cons(x,cons(y,l)))}, \texttt{sum(cons(plus(x,y),l))} \rangle$
$7 : \langle \texttt{app(cons(x,l),k)}, \texttt{app(l,k)} \rangle$

The (approximated) dependency graph, the induced DAG of scc and corresponding sub-DAGs may be found Figure 2 and Figure 3. Each scc is modelled by the corresponding sub-relation of `DPR`:

```
Inductive SCC₀ : term → term → Prop :=
 | SCC₀₀ : ∀x₀ x₁ V₂ V₃,
      x₀ -[R]*> Term s [V₂] → x₁ -[R]*> V₃
    → Term plus [x₀;x₁] -[SCC₀]> Term plus [V₂;V₃].
```

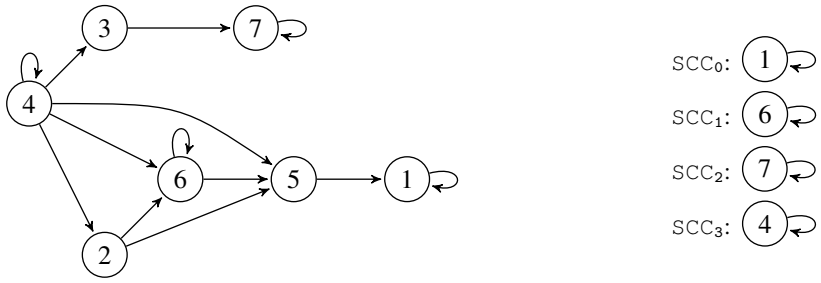


Fig. 2. Dependency graph of R_1 and its scc

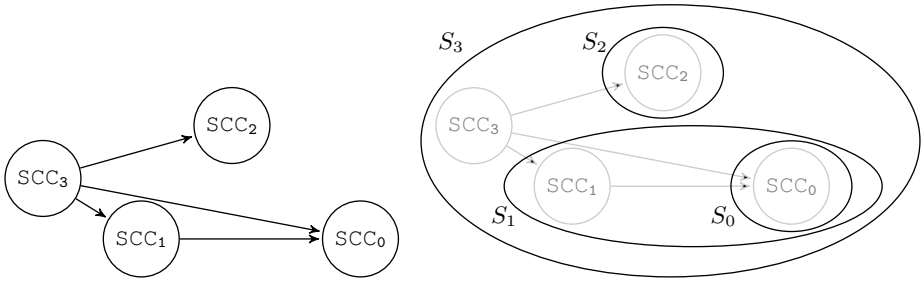


Fig. 3. DAG of scc of R_1 and corresponding sub-DAGs

```
Inductive SCC₁ : term → term → Prop :=
 | SCC₁₀ : ∀x₀ V₁ V₂ V₃,
     x₀ -[R]*> (Term cons [V₂; (Term cons (V₃;V₁))])
   → (Term sum [x₀])
      -[SCC₁]> Term sum [Term cons [Term plus [V₂; V₃];V₁]].

Inductive SCC₂ : term → term → Prop :=
 | SCC₂₀ : ∀x₀ x₁ V₀ V₁ V₂, x₀ -[R]*> (Term cons [V₂;V₁])
    → x₁ -[R]*> V₀
    → Term app [x₀;x₁] -[SCC₂]> Term app [V₁;V₀].

Inductive SCC₃ : term → term → Prop :=
| SCC₃₀ : ∀x₀ V₀ V₁ V₂ V₃,
   x₀ -[R]*> Term app [V₁;Term cons [V₂;Term cons [V₃;V₀])]] →
  Term sum [x₀] -[SCC₃]>
   Term sum [Term app
              [V₁;Term sum [Term cons [V₂;term cons [V₃;V₀]]]]].
```

Each scc is proved to be terminating using a different ordering by classical induction (see [20]). Then the final proof of $\mathsf{SNG}(\mathcal{G})$ (`well_founded DPR` below) must be built. We show below how this is done by composing results on sub-DAGs in a bottom-up fashion as described in Section 4.2.

```
(*Now suppose scc are SNG and prove that then DPR is well-founded.*)
```

```
Hypothesis Well_Founded_SCC₀ : well_founded SCC₀.
Hypothesis Well_Founded_SCC₁ : well_founded SCC₁.
Hypothesis Well_Founded_SCC₂ : well_founded SCC₂.
Hypothesis Well_Founded_SCC₃ : well_founded SCC₃.

Lemma Acc_SCC₀ : ∀x y, SCC₀ x y → Acc DPR x.
Proof. (*well-founded induction on SCC₀.*) Qed.

Lemma Acc_SCC₁ : ∀x y, SCC₁ x y → Acc DPR x.
Proof. (*well-founded induction on SCC₁ + Acc_SCC₀.*) Qed.

Lemma Acc_SCC₂ : ∀x y, SCC₂ x y → Acc DPR x.
Proof. (*well-founded induction on SCC₂. *) Qed.

Lemma Acc_SCC₃ : ∀x y, SCC₃ x y → Acc DPR x.
Proof. (*well-founded induction on SCC₃ + Acc_SCC₁ + Acc_SCC₂.*) Qed.

Lemma Well_Founded_DPR : well_founded DPR.
Proof. (*case analysis on DPR + Acc_SCC₃.*) Qed.
```

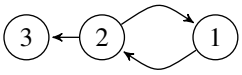
5.2 A strong graph criterion example

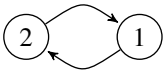
The example below is also due to Arts and Giesl [3] and checks whether the first argument of `evenodd` is even or not.

```
not(true) →false
not(false) →true
evenodd(x,0) →not(evenodd(x,s(0)))
evenodd(0,s(0)) →false
evenodd(s(x),s(0)) →evenodd(x,0)
```

The dependency pairs of this new system is:

$$1 : \langle \texttt{evenodd(x,0)}, \texttt{evenodd(x,s(0))} \rangle$$
$$2 : \langle \texttt{evenodd(s(x),s(0))}, \texttt{evenodd(x,0)} \rangle$$
$$3 : \langle \texttt{evenodd(x,0)}, \texttt{not(evenodd(x,s(0)))} \rangle$$

The corresponding (approximated) dependency graph is : 

and its only scc is $\texttt{SCC}_0$: 

The first step of the graph proof is very similar to the previous example one:

```
Inductive SCC₀ : term →term →Prop :=
 | SCC₀₁ : ∀x₀ x₁ V₀,
     x₀ -[R]*> V₀ →x₁ -[R]*> Term 0 [] →
     Term evenodd [x₀;x₁] -[SCC₀]>
       Term evenodd [V₀;Term s [Term 0 []]]
 | SCC₀₂ : ∀x₀ x₁ V₀,
      x₀ -[R]*> Term s [V₀] →x₁ -[R]*> Term s [Term 0 []] →
      Term evenodd [x₀;x₁] -[SCC₀]> Term evenodd [V₀;Term 0 []]

Variable Well_Founded_SCC₀ : well_founded SCC₀.

Lemma Acc_SCC₀ : ∀x y, SCC₀ x y →Acc DPR x.
Proof. (*well-founded induction on SCC₀.*) Qed.

Lemma Well_Founded_DPR : well_founded DPR.
Proof. (*case analysis on DPR + Well_Founded_SCC₀.*) Qed.
```

Despite the simplicity of this graph, our automated (termination) prover does use the enhanced version of the graph criterion in order to split the single component $\texttt{SCC}_0$. The pair 2 is strictly oriented by the discovered ordering pair while the pair 1 is only weakly oriented.

The proof of $\mathsf{SNG}(\mathcal{SCP}(\texttt{SCC}_0))$ is obtained as follows :

```
 Inductive SCC₀_large : term →term →Prop :=
 | SCC₀_large₁ : ∀x₀ x₁ V₀,
     x₀ -[R]*> V₀ →x₁ -[R]*> Term 0 [] →
     Term evenodd [x₀;x₁] -[SCC₀]>
       Term evenodd [V₀;Term s [Term 0 []]]

Variable Well_Founded_SCC₀_large : well_founded SCC₀_large.

Inductive SCC₀_strict : term →term →Prop :=
```

```
 | SCC₀_strict₁ : ∀x₀ x₁ V₀,
     x₀ -[R]*> Term s [V₀] →x₁ -[R]*> Term s [Term 0 []] →
     Term evenodd [x₀;x₁] -[SCC₀]> Term evenodd [V₀;Term 0 []]

Variable lt le : term →term →Prop.
Hypothesis lt_le_compat : ∀x y z, lt x y →le y z →lt x z.
Hypothesis wf_lt : well_founded lt.
Hypothesis SCC₀_strict_in_lt :
  Relation_Definitions.inclusion _ SCC₀_strict lt.
Hypothesis SCC_large_in_le :
 Relation_Definitions.inclusion _ SCC₀_large le.

Lemma Well_Founded_SCC₀ : well_founded SCC₀.
Proof. (*well-founded induction on lt and SCC₀_large+
       case analysis on SCC₀*) Qed.
```

6 Experiments

Our approach is implemented in a prototype which is an automated prover dedicated to termination, based on a restricted version of the termination engine of C*i*ME 2.04 [21]. We ran experiments using this technique on a 3GHz, 8GB, Debian-linux machine. Up to now it gives termination certificates for more than 550 problems of the Termination Problems DataBase (TPDB) 4.0[5] (i.e. ∼27% of the standard category, some problems of which being non-terminating). It is important to notice that the limiting factor in these results is *not* the certification process itself but the termination techniques used by the prototype! Actually, *all* proofs found by the prototype are certified by Coq.

Some interesting proofs should be highlighted here. We can certify problems with 159 nodes (`TRS/TRCSR/PALINDROME_complete-noand_FR.trs`, certified in 568.77 s), with 1015 edges (`TRS/TRCSR/ExSec11_1_Luc02a_iGM.trs`, certified in 113.49s), with 10 strongly connected components at top level (`TRS/TRCSR/inn/Ex26_Luc03b_C.trs`, certified in 78.80s) or even using 6 times graph splitting (`TRS/TRCSR/Ex9_BLR02_iGM.trs.v`, certified in 64.15s).

Of course, the certification time for those examples is not representative of the average certification time. It emphasizes what we are able to certify in the TPDB's worst cases. Further note that the certification time takes *all* the certification process into account (orderings, etc.) and not only the graph management. Yet, those times are still reasonable for such tricky examples.

The average certification time on all certified problems is 14s (83% of them are certified in less than 15s and 58% in less than 5s). This makes our approach exploitable in practice, for developments where (involved) proofs of well-foundedness are required.

7 Conclusion

We described an approach to deal efficiently with some graph analysis in skeptical proof assistants. This approach is based on proof scripts generation and uses well-founded

[5] `http://www.lri.fr/~marche/tpdb`

induction and dependent inductive types. One of the key points is that induction on the paths in the graph can be simulated by judicious generation of intermediate lemmas that propagate the desired property along the arcs. It benefits from shallow embedding by handling the graph implicitly, without any concrete model. Thus, some premises that would be difficult to prove in a full deep embedding setting (like completeness, for instance) are avoided.

Regarding termination proofs, this technique is fully implemented in a prototype in the context of the A3PAT project; the prototype is available and can be tried online from the web-page of the project: `http://a3pat.ensiie.fr`. Note that our implementation takes benefit of the expressive tactic language of Coq. Experiments with our prototype illustrate the power of this approach, in particular it allows us to certify in COQ, *in a few seconds*, (scripts of) termination proofs that rely on the circuit analysis of graphs consisting of more than a thousand arcs. Another approach for certifying termination proofs is the Rainbow+Color approach [7], which is based on deep embedding only. Restricted to termination problems, the Color library models several orderings, among which powerful orderings induced by matrix interpretations. However it cannot handle graph criteria as involved as Theorem 2.

Although we illustrate our approach through its instantiation for termination proofs in COQ, it is general enough to adapt to other skeptical proof assistant (e.g. Isabelle/HOL [17]), provided that the targeted assistant is powerful enough to handle inductive relations and associated reasoning tools, provides a mechanism to deal with well-founded induction and enjoys an expressive enough tactic language. Among the perspectives, a first one could be to implement this approach in other assistants than COQ. A second one could be to generalize this shallow model + external prover technique to deal with generic graph analysis and other properties that rely on it.

Acknowledgments

The authors would like to thank Christiane Goaziou for her help in improving the readability of this paper, and the anonymous referees for their comments.

References

1. Thomas Arts and Jürgen Giesl. Automatically Proving Termination Where Simplification Orderings Fail. In Michel Bidoit and Max Dauchet, editors, *Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, Lille, France, April 1997. Springer-Verlag.
2. Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
3. Thomas Arts and Jürgen Giesl. A collection of examples for termination of term rewriting using dependency pairs. Technical report, RWTH Aachen, September 2001.
4. Thomas Arts and Jürgen Giesl. Verification of Erlang Processes by Dependency Pairs. *Application Algebra in Engineering, Communication and Computing*, 12(1,2):39–72, 2001.
5. Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
6. Claude Berge. *Graphs*, volume 6 of *North-Holland mathematical library*. North-Holland, third edition, 1991. ISBN : 0-444-87603-0.

7. Frédéric Blanqui, Solange Coupet-Grimal, William Delobel, Sébastien Hinderer, and Adam Koprowski. Color, a coq library on rewriting and termination. In Alfons Geser and Harald Sondergaard, editors, *Extended Abstracts of the 8th International Workshop on Termination, WST'06*, August 2006.
8. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. North-Holland, 1990.
9. Jörg Endrullis. Jambox. `http://joerg.endrullis.de/index.html`.
10. Jürgen Giesl, Thomas Arts, and Enno Ohlebusch. Modular Termination Proofs for Rewriting Using Dependency Pairs. *Journal of Symbolic Computation*, 34:21–58, 2002. doi:10.1006/jsco.2002.0541.
11. Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In Ulrich Furbach and Natarajan Shankar, editors, *Third International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, Seattle, USA, August 2006. Springer-Verlag.
12. Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
13. Nao Hirokawa and Aart Middeldorp. Automating the dependency pair method. In Franz Baader, editor, *19th International Conference on Automated Deduction (CADE-19)*, volume 2741 of *Lecture Notes in Computer Science*, pages 32–46, Miami Beach, FL, USA, July 2003. Springer-Verlag.
14. Nao Hirokawa and Aart Middeldorp. Tyrolean termination tool. In Jürgen Giesl, editor, *16th International Conference on Rewriting Techniques and Applications (RTA'05)*, volume 3467 of *Lecture Notes in Computer Science*, pages 175–184, Nara, Japan, April 2005. Springer-Verlag.
15. Adam Koprowski. TPA. `http://www.win.tue.nl/tpa`.
16. Keiichirou Kusakari, Masaki Nakamura, and Yoshihito Toyama. Argument filtering transformation. In Gopalan Nadathur, editor, *Principles and Practice of Declarative Programming, International Conference PPDP'99*, volume 1702 of *Lecture Notes in Computer Science*, pages 47–61, Paris, 1999. Springer-Verlag.
17. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
18. The Coq Development Team. *The Coq Proof Assistant Documentation – Version V8.1*, February 2007. `http://coq.inria.fr`.
19. Évelyne Contejean. The Coccinelle library for rewriting. `http://www.lri.fr/~contejea/Coccinelle/coccinelle.html`.
20. Évelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Certification of automated termination proofs. In Boris Konev and Frank Wolter, editors, *6th International Symposium on Frontiers of Combining Systems (FroCos 07)*, volume 4720 of *Lecture Notes in Artificial Intelligence*, pages 148–162, Liverpool,UK, September 2007. Springer-Verlag.
21. Évelyne Contejean, Claude Marché, Benjamin Monate, and Xavier Urbain. Proving termination of rewriting with CiME. In Albert Rubio, editor, *Extended Abstracts of the 6th International Workshop on Termination, WST'03*, pages 71–73, June 2003. `http://cime.lri.fr`.