A3PAT, an Approach for Certified Automated Termination Proofs *

Évelyne Contejean
Andrei Paskevich
Xavier Urbain †‡
† LRI, Univ. Paris-Sud 11, CNRS;
Orsay, F-91405, France
INRIA Saclay - Île-de-France, ProVal;
Orsay, F-91893, France
{contejea,andrei,urbain}@lri.fr

Pierre Courtieu
Olivier Pons
Cédric, CNAM;
Paris, F-75141, France
{olivier.pons,pierre.courtieu}@cnam.fr

Julien Forest
Cédric, ‡ ENSIIE;
Évry, F-91025, France
julien.forest@ensiie.fr

Abstract

Software engineering, automated reasoning, rule-based programming or specifications often use rewriting systems for which termination, among other properties, may have to be ensured. This paper presents the approach developed in Project A3PAT to discover and moreover certify, with full automation, termination proofs for term rewriting systems.

It consists of two developments: the COCCINELLE library formalises numerous rewriting techniques and termination criteria for the COQ proof assistant; the C*i*ME3 rewriting tool translates termination proofs (discovered by itself or other tools) into traces that are certified by COQ assisted by COCCINELLE.

The abstraction level of our formalisation allowed us to weaken premises of some theorems known in the literature, thus yielding new termination criteria, such as an extension of the powerful subterm criterion (for which we propose the first full COQ formalisation). Techniques employed in C*i*ME3 also improve on previous works on formalisation and analysis of dependency graphs.

Categories and Subject Descriptors F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs; I.2.3 [*Deduction and Theorem Proving*]: Deduction

General Terms Theory, Verification

Keywords Automated Reasoning, Formal Proof, Term Rewriting, Termination

1. Introduction

Verification of programs and specifications may involve a significant amount of formal methods to guarantee required properties for complex or critical systems. In the context of proving, users rely on proof assistants with which they interact, step by step, until the proof is totally and mechanically verified. However, formal proof may be costly, and as proof assistants often lack automation, there is an increasing need to use fully automated tools providing powerful (and intricate) decision procedures. Although some proof assistants can check the soundness of a proof, the results of automated provers are often taken *as is*, even if the provers may be subject to bugs. Since application fields include possibly critical sectors such as security, code verification, cryptographic protocols, etc., reliance on verification tools is a crucial issue.

Automated provers and proof assistants do not mix easily. One of the strengths of proof assistants like COQ [37] is a *highly reliable* procedure that checks the soundness of proofs. For instance, COQ or ISABELLE/HOL [36] have a small and highly reliable *kernel*. In COQ, the kernel type-checks a *proof term* to ensure the soundness of a proof. Those assistants need to check mechanically the proof of *each property used*. Certified-programming environments based on these proof assistants find this an additional guarantee. However, this means that the proof assistant has to check a property proven by an external procedure before accepting it. Therefore, such a procedure must return a *proof trace* checkable by the assistant.

The A3PAT project[1] aims to bridge the gap between proof assistants yielding formal guarantees of reliability, and highly automated tools one has to trust. We want to provide both enhanced automation for proof assistants (by proof delegation), and formal certificates for automatically generated proofs (by mechanical checking of proof traces).

In the framework of first-order *term rewriting* techniques, which has proven to be most useful in programming, specifying and proof automation, we focus on proofs of *termination*: the fundamental property of a program any execution of which yields a result.

Termination is crucial when recursion and induction are involved. It is an unavoidable preliminary for proving various properties of a program. Confluence of a rewriting system, for instance, becomes decidable when the system terminates. Regarding program verification, proving termination is a boundary between *total* and *partial* correctness. In the context of proof assistants like COQ, automating termination is of great interest; for example, a COQ function can be defined only if it is proven to be terminating.

This work takes place in the A3PAT project, it presents contributions of different natures. We describe a methodology for the important challenge of automatically generating proof traces for

* Work partially supported by the French ANR (ANR-05-BLAN-0146 and ANR-08-EMER-005).

PEPM'10, January 18–19, 2010, Madrid, Spain.

[1] `http://a3pat.ensiie.fr`

first-order term rewriting techniques, and in particular for termination proofs. Then we propose two improvements regarding termination/certification techniques: a full formalisation and an extension of an efficient termination criterion, and an alternate approach for dealing with graphs.

We claim that demanding computations should not be handled by a proof assistant alone, *if their results can be checked easily*, but rather delegated to satellite tools. The solution we propose in this work is based on a termination engine and trace generator (C*i*ME3), and a very general formal library on rewriting (COCCINELLE). The trace generator can act as a compiler for proofs traces (as XML files for example) from other termination engines. It generates COQ scripts that refer to the formal library COCCINELLE. The COQ proof assistant can then check the termination proofs, using theorems in COCCINELLE. In the spirit of the early prototype of [13], our approach mixes shallow and deep embeddings that is, respectively, direct use of the proof assistant's own structure of terms and complete encoding of term algebras within the proof assistant. We extend this previous work with new schemes, as modular as possible, and techniques.

An interesting benefit of abstract formalisations is that keeping them as general as possible may lead to relaxing premises of theorems. We give in particular the first full formalisation in COQ of the powerful *subterm criterion* [29] for termination of rewriting systems[2], and we propose an extension (Theorem 4). Another example is our extended notion of matrix interpretations [16].

Proving termination can amount to dealing with properties on graphs, which 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). In particular, the enhanced dependency graph theorem in [25] 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 proving.

Of course, this theorem is not applied explicitly in the search of a termination proof; techniques have been proposed that make an efficient use of how vertices share cycles, for example by analysing strongly connected components (see [29]). We follow the same idea to certify graph-based proofs. We propose here a new formalisation for graphs, based on [15] in which we allowed certification of (termination) proofs based on a graph analysis, and could manage efficiently graphs containing thousands of arcs. In contrast to [15], the new formalisation is deeper, and allows proofs to be more local, while keeping an implicit representation of graphs[2].

We will begin by briefly sketching the background with reference to rewriting and termination, proof assistants and automated provers. Then in Section 3 we will give our notations and define precisely the technical notions involved in this work. Section 4 will describe the overall architecture of our solution, our tool C*i*ME3 and our trace language. We will present our formal library COCCINELLE in Section 5 as well as an extension of a termination criterion. Section 6 will be about our latest improvement on graph techniques. We will eventually conclude and outline directions of future work in Section 8.

2. Background

2.1 Term rewriting, termination

Term (or string) Rewriting Systems (TRS) are basically sets of *oriented* equations, that is: used as directed replacement *rules*. This very abstract formalism is Turing-complete and has numerous applications: in logic or functional programming, automated reasoning, specification, algebraic data types, etc. We refer to [5, 18] for more details.

Termination is fundamental in many ways to verification of programs and certified programming environments. It is an undecidable property, in particular all termination techniques are incomplete. From the automation point of view, a proof of termination is always difficult to discover.

Regarding TRS, powerful techniques arise from the *dependency pairs* (DP) approach, introduced in 1997 by Arts & Giesl [3, 4]. The historical Manna and Ness criterion consists in discovering a well-founded, monotonic (closed by context, noted $\mathsf{SM}(>)$) and stable (w.r.t. instantiation) ordering for which *each rule* of the system decreases strictly, thus embedding the relation in the well-founded ordering and ensuring its termination. The dependency pair approach, in contrast, focuses on the possible *inner recursive calls* of rules (the so-called dependency pairs). This leads to a weakening of the constraints on suitable orderings, very well-suited for automation. For details, see [3]. This approach has been made even more powerful by use of multiple refinements: for example it can benefit greatly from the analysis of a *dependency graph*, especially when different orderings can be used [25].

A powerful termination criterion working in conjunction with dependency graphs is the *subterm criterion* [29], efficiently removing from the dependency graph unnecessary vertices. We provide a formalised extension of this criterion in Section 5.2 (Theorem 4). This is the first full formalisation of a subterm criterion in COQ.

2.2 Automated provers, certification

Research in the area of automation for termination proofs has been very fruitful during the past decade [11, 20, 27, 30, 31], in particular for termination of first order rewriting systems (on which we focus here), but also for logic and functional programming. Regarding TRS, most tools use dependency pairs and their numerous refinements. Recent efficiency improvements are related to the use of external state-of-the-art SAT solvers to handle constraints on orderings [2, 22].

Regarding certification of termination proofs, several libraries have been developed, among which COCCINELLE [8], but also CoLoR [7] and IsaFoR [38].

CoLoR is a library for COQ formalising theorems in a pure deep fashion. On the one hand, proof scripts for CoLoR are very short. On the other hand, some pure deep theorems may be difficult (costly) to compile, as their premises may include completeness properties, for instance graph related theorems, see Sections 3.3 and 6. CoLoR can manage orderings based on polynomials and matrices, DP criterion and relative termination. It provides also some results for termination modulo AC.

IsaFoR is a very recent library for ISABELLE/HOL. Its standalone certifier module is obtained by *extraction*, that is an automatically generated Haskell program (using the code-generation feature of ISABELLE/HOL). The extracted certifier is thus very efficient for checking termination proofs. However, its results have to be accepted as axioms (unchecked) by proof assistants. IsaFoR can manage orderings based on polynomials, and most DP refinements like usable-rules [39], subtle graph approximations, etc. It enjoys also some techniques to certify non-termination proof.

2.3 Proof assistants

Skeptical proof assistants like ISABELLE/HOL [36] or COQ [37] are tools to formalise, to prove, and to check mechanically mathematical results. They are said to be skeptical because formally proving every notion one uses with them is mandatory.

[2] Another formalisation by the IsaFoR team was obtained independently and roughly at the same time for ISABELLE/HOL.

These two proof assistants enjoy very powerful specification languages which can express both logical assertions *and* programs, hence properties of programs. They allow to build interactively proofs using LCF-style tactics to transform recursively an initial goal to simpler subgoals. Tactics correspond to the rules of the underlying logic (HOL for ISABELLE/HOL, CIC for COQ) or to powerful heuristics. The list of tactics used to build a proof is called the proof script.

Specification and scripts can be written in files (`.v` files for COQ) which can be compiled in order to obtain certification. Many formal libraries may be found for both these assistants.

What makes ISABELLE/HOL and COQ highly trustworthy is their kernel, checking the proof once it is built.

We focus in this work on the COQ proof assistant. The COQ proof assistant is based on *type theory*. Its *formal language* can express objects, properties and proofs in a unified way; all these are represented as terms of an expressive λ-calculus: the *Calculus of Inductive Constructions* (CIC) [14]. λ-abstraction is denoted `fun x:T => t`, and application is denoted `t u`. A proof development with COQ consists in trying to build, interactively and using tactics, a λ-term the type of which corresponds to the proven theorem (Curry-Howard style).

The kernel of COQ is a *proof checker* which checks the validity of proofs written as CIC-terms. Indeed, in this framework, a term is a *proof* of its type, and checking a proof consists in type-checking a term. Roughly speaking, the small kernel of COQ simply type-checks λ-terms to ensure soundness.

A very powerful feature of COQ is the ability to define *inductive types* to express inductive data types and inductive properties. For example the following inductive types define the data type `nat` of natural numbers, `O` and `S` (successor) being the two constructors[3], and the property `even` of being an even natural number.

```
Inductive nat : Set := O : nat | S : nat → nat.
Inductive even : nat → Prop :=
  | even_O : even O
  | even_S : ∀n : nat, even n → even (S(S n)).
```

Hence the term `even_S (S(S O)) (even_S O (even_O))` is of type `even (S(S(S(S O))))` so it is a proof that 4 is even.

3. Preliminaries and notations

3.1 Terms, relations on terms

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. We write Λ for the root position in a term, $\Lambda(t)$ denotes the symbol at Λ 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. We write $t \trianglerighteq u$ (resp. $t \triangleright u$) whenever u is a subterm (resp. proper subterm) of t. *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* over a signature $\mathcal{F}$ is a set R of *rewrite rules* $l \to r$ with $l, r \in T(\mathcal{F}, X)$. The systems we consider in this work will be all *finite*. A TRS R defines a monotonic relation $\to_R$ closed under substitution (aka a *rewrite relation*) in the following way: $s \xrightarrow[R]{p} t$ (*s reduces to t* at position p) if $s|_p = l\sigma$ and $t = s[r\sigma]_p$ for a rule $l \to r \in R$ and a substitution σ. Systems and positions that are clear from the context may be omitted. We denote the reflexive-transitive closure of a relation $\to$ by $\to^\star$. Symbols occurring at root position on the left-hand side of rules in R are said to be *defined*, the others are said to be *constructors*.

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

[3] This notion of constructors is different from the one in Section 3.

3.2 Dependency pairs

DEFINITION 1 (Dependency pairs, dependency chain [3]). *The set of* unmarked dependency pairs *of a rewriting system R, denoted* $\mathsf{DP}(R)$ *is defined as*

$$\mathsf{DP}(R) = \{\langle u, v\rangle \mid u \to t \in R \text{ and } t|_p = v \text{ with } \Lambda(v) \text{ defined}\}.$$

Given a TRS R, we note $s \twoheadrightarrow_{D,R} t$ iff $s \xrightarrow[R]{\neq\Lambda\ \star} u\sigma \xrightarrow[\langle u,v\rangle\in D]{\Lambda} v\sigma \equiv t$.
A dependency chain *(of D over R) is a sequence in $\twoheadrightarrow_{D,R}$.*

Distinguishing root symbols of dependency pairs (by means of marks, or 'tuple-symbols') enhances this technique significantly. Marking or not the dependency pairs does *not* interfere with our approach, thus for the sake of readability, we will restrict to unmarked pairs. Further note that our prototype handles both marked and unmarked dependency pairs [13].

The main theorem of dependency pairs [3] can be rephrased as:

THEOREM 1 (Dependency Pairs Criterion). *Let R be a TRS. Then $\mathsf{SN}(\twoheadrightarrow_{\mathsf{DP}(R),R})$ is equivalent to $\mathsf{SN}(\to_R)$.*

Termination of $\twoheadrightarrow_{D,R}$ may be proven directly using ordering pairs (see [32] for a very general definition). Due to our definition of $\twoheadrightarrow_{D,R}$, we use a slightly restricted definition: an ordering pair is a pair $(\succeq, >)$ of relations over $T(\mathcal{F}, X)$, stable, and such that: 1) $\succeq$ is a quasi-ordering, that is, reflexive and transitive, 2) $>$ is a strict ordering, that is, irreflexive and transitive, and 3) $\succeq \cdot > \subseteq >$. An ordering pair $(\succeq, >)$ is *well-founded* (notation $\mathsf{WF}(\succeq, >)$) if there is no infinite strictly decreasing sequence $t_1 > t_2 > \cdots$. An ordering pair $(\succeq, >)$ is said to be weakly monotonic if $\succeq$ is (notation $\mathsf{WM}(\succeq, >)$).

An effective corollary of Theorem 1 consists in discovering a well-founded weakly monotonic ordering pair $(\succeq, >)$ for which $\to_R \subseteq \succeq$ and $D \subseteq >$ to prove $\mathsf{SN}(\twoheadrightarrow_{D,R})$.

3.3 Dependency graphs

As noticed in [3], not all DP can follow another in a dependency chain: one may consider the graph of possible sequences of DP. This graph is finite since we restricted to finite TRS, and 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 2 (Dependency graph [3]). *Let R be a TRS. The* dependency graph of a set of dependency pairs D over a system R, *denoted $\mathcal{G}(D, R)$, is defined as (D, A) where $\langle s, t\rangle \mapsto \langle s', t'\rangle \in A$ if and only if there exists a substitution σ such that $t\sigma \xrightarrow[R]{\neq\Lambda\ \star} s'\sigma$.*

REMARK 1. *It is worth noticing that this dependency graph (D, A) is not computable, so one uses a sound approximation, i.e. a graph $(D, A' \supseteq A)$ that contains it. Arts & Giesl proposed a simple yet efficient approximation, namely* connectability *(with REN/CAP) [3]. The approximation we choose to implement in CiME3 corresponds to this simple one, but the formalisation in* COCCINELLE *is* parameterised by the approximation*, thus well-suited for other provers and approximations.*

The fundamental theorem of graph refinement is as follows:

THEOREM 2 (Dependency graph refinement [25]). *A TRS R terminates if and only if for each circuit C in the dependency graph $\mathcal{G}(\mathsf{DP}(R), R)$ there exists no infinite dependency chain of dependency pairs of C.*

This theorem is not applied *as is* in the practice of discovering termination proofs. In order to reduce the analysis, the decomposition into SCC (or any other convenient decomposition) allows to *search* for some *uniform* arguments for some classes of cycles (for instance, see [29]). Once the search succeeded and produced a proof, the certification of this proof *necessarily* eliminates all cycles (implicitly using Theorem 2). However, at this stage, there is no exponential blow-up, since the proof is generic, and does not examine each cycle of a particular graph, but the few possible shapes of a cycle in a generic split graph. This is how we will certify graph criteria.

This theorem is formalised by the GRAPH rule in Figure 1, which states that one can consider any *compatible* decomposition $\{D_1, \ldots, D_n\} \subseteq \mathbb{P}(D)$ of the set of pairs. We denote $\mathsf{Decomp}(D, \{D_1, \ldots, D_n\})$ such a decomposition. There are several notions of a compatible decomposition, and we give ours in Section 6.2, together with our technique for certifying it.

3.4 Proof sketches

A typical termination proof is in fact a recursive transformation of well-foundedness problems into problems equivalent but easier to solve, until one reaches problems that are trivial or that are proven directly using a well-founded ordering (pair). We call *criterion* a correct and complete method allowing to translate a well-foundedness problem into a set of new problems. Following the idea introduced in [13] and [15] we will model a termination proof by an *inference tree* where inference rules are criteria possibly guarded by conditions (such as the existence of an adequate well-founded ordering, inclusion properties over relations, etc.). The general form of a rule is the following:

$$\text{RULE NAME(PARAM)}\ \frac{p_1 \ldots p_n}{p}\ \text{CONDITIONS}$$

where $p, p_1 \ldots p_n$ are properties on relations, and PARAM is an (optional) ordering (pair) parametrising the rule. This ordering satisfies the properties given in the side condition. The termination criteria described above are summarised by rules of Figure 1.

For example, rule RMVERTEX states that one can split a termination proof by considering separately the set of chains passing through a given dependency pair $\langle u, v\rangle$. With the help of an appropriate ordering pair, we can prove that there is no $\twoheadrightarrow_{D,S}$-chain containing an infinite number of instances of $\langle u, v\rangle$. The rules actually correspond to COQ theorems, this is why RMVERTEX removes only a pair. Removing several pairs would require another theorem proved by induction on the number of removed pairs.

4. A satellite tool with traces

4.1 Architecture

The main purpose of proof assistants is to *check* proofs mechanically and not to *discover* them. An interesting way to add automation to proof assistants is to use satellite tools to which the proof assistant may delegate proofs. This way of proceeding does not add to the complexity of critical kernels, and it helps in delegating costly computations to highly specialised and efficient tools. Then the proof assistant is left with what it is best at: checking that the given proof is correct.

The architecture of A3PAT is summarised in Figure 2. It involves two main components: the C*i*ME3 rewriting tool (including a certification engine that generates proof scripts) and the COCCINELLE library which formalises most of the properties needed.

The sequence of events is as follows. There are various scenarios for start: a development in a proof assistant, a verification of specifications for instance, needs a proof of termination for a rewriting system. The proof assistant may delegate this particular proof to an external (satellite) prover, like C*i*ME3 or APROVE. Alternate case: a user of some termination automated prover may want a certificate for a termination proof, etc.

Once a proof is discovered, it is encoded in a well-suited XML format and submitted to the certification engine in C*i*ME3, which generate a script for COQ. Note that, as the XML format is compact, some more computations may be performed by C*i*ME3 to generate specialised instantiations of lemmas and functions.

In order to certify the proof, COQ must compile the (axiom-free) script. This task usually requires formal definitions and theorems gathered in the COCCINELLE library.

REMARK 2. COQ *scripts may be a bit obscure. Regarding termination proofs, the only things that have to be human readable in scripts are: the definition of the system, and a theorem stating that the aforementioned system defines a well-founded relation.*

All tools developed in the A3PAT project are available from `http://a3pat.ensiie.fr`.

4.2 C*i*ME3

Our tool C*i*ME3 can be used as a satellite prover for the COQ proof assistant. Evolving from the C*i*ME family[4], it consists of

- A rewriting tool box, with a termination engine, and
- A proof compiler, generating COQ proof scripts.

The rewriting tool box provides numerous techniques and procedures on term algebras: matching, unification, rewriting, and completion, either for free algebras or modulo equational theories [10] with commutative or associative and commutative operators, with unit elements, etc. It enjoys also a termination engine for first order rewriting systems.

Certified termination criteria include:

- dependency pairs [3] (with Dershowitz improvement), marked or unmarked,
- dependency graphs refinements [25],
- subterm criterion [29].

Certified termination orderings include:

- polynomial interpretations [12, 33],
- various matrix interpretations [16, 21],
- full RPO with status [17] (including comparison with LEX status of symbols with different arities) and AFS refinements [3].

Some other techniques are not trace-producing yet, even if completely formalised (narrowing, innermost refinements [3], lexicographic composition of orderings, etc.). Note that for efficiency reasons, the termination engine may use an external SAT-solver to find orderings, *à la* [2, 22].

When C*i*ME3 finds a proof, it may either call directly the proof compiler to produce a COQ script for certification, or produce a XML trace to be used later by our proof compiler (or another one).

C*i*ME3 accepts several input languages, depending on how one wants to use it: in batch mode or in interactive session. Interactive sessions require the tool's dedicated programming language. The example below shows a basic interactive session with C*i*ME3. The answers of the top-level are given in *slanted* font. We first define a signature, a term algebra for Peano arithmetic:

```
CiME> let F_peano =
                signature "O : constant; s : unary; plus : binary;";
F_peano : signature = signature "s : 1; plus : 2; 0 : 0"
```

[4] `http://cime.lri.fr`

$$\text{MN}(>)\ \frac{}{\mathsf{SN}(\to_S)} \qquad \mathsf{WF}(>) \wedge \mathsf{SM}(>) \wedge (S \subseteq >)$$

$$\text{DP}\ \frac{\mathsf{SN}(\to\!\!\!\!\to_{DP(S),S})}{\mathsf{SN}(\to_S)} \qquad \text{DP}_{\text{AX}}(\succeq, >)\ \frac{}{\mathsf{SN}(\to\!\!\!\!\to_{D,S})} \quad \begin{array}{l}\mathsf{WF}(\succeq, >) \wedge \mathsf{WM}(\succeq, >) \\ \wedge (S \subseteq \succeq) \wedge (D \subseteq >)\end{array}$$

$$\text{GRAPH}\ \frac{\mathsf{SN}(\to\!\!\!\!\to_{D_1,S}) \quad \ldots \quad \mathsf{SN}(\to\!\!\!\!\to_{D_n,S})}{\mathsf{SN}(\to\!\!\!\!\to_{D,S})}\ \mathsf{Decomp}(D, \{D_1, \ldots, D_n\})$$

$$\text{RMVERTEX}\ \frac{\mathsf{SN}(\to\!\!\!\!\to_{D\setminus\langle u,v\rangle,S})}{\mathsf{SN}(\to\!\!\!\!\to_{D,S})} \quad \begin{array}{l}\mathsf{WF}(\succeq, >) \wedge \mathsf{WM}(\succeq, >) \wedge u > v \\ \wedge (S \subseteq \succeq) \wedge (D\setminus\langle u, v\rangle \subseteq \succeq)\end{array}$$

Figure 1. Termination inference system for criteria: Manna-Ness, dependency pairs and graph refinement.

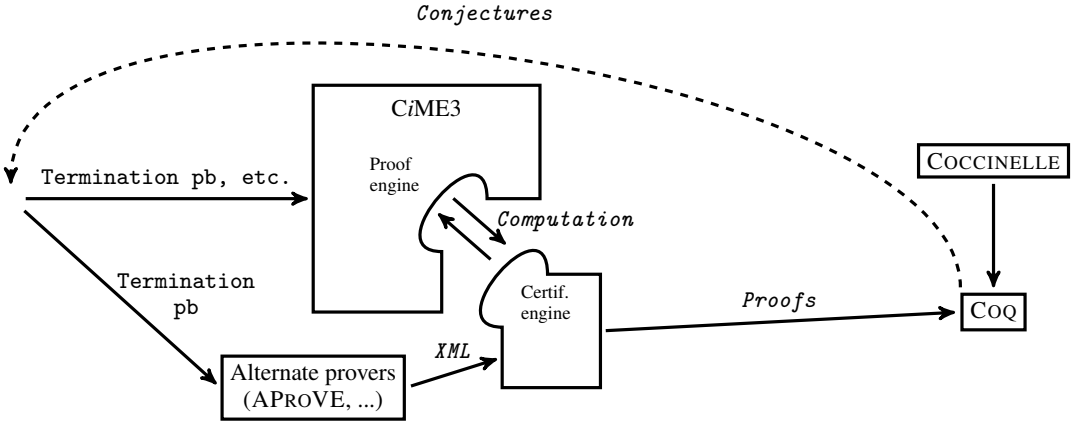

Figure 2. Certification architecture.

```
CiME> let X = variables "x,y,z";
X : variable_set = variables "z,x,y"

CiME> let A_peano = algebra F_peano ;
A_peano : F_peano algebra = algebra F_peano
```

Then we define terms and term rewriting systems on this algebra:

```
CiME> let R_peano = trs A_peano "
      plus(x,O)    -> x ;
      plus(x,s(y)) -> s(plus(x,y)) ;";
R_peano : F_peano trs = trs A_peano ...
```

Now we may check the termination of the system:

```
CiME> termination R_peano;
[...information depending on verbosity...]
- : bool = true
```

and produce an XML trace:

```
CiME> xml_proof_trace R_peano;
<?xml version="1.0" encoding="UTF-8"?>
<PROOF>
  <SIGNATURE>
  ...
</PROOF>
- : unit = ()
```

or directly a COQ certificate:

```
CiME> coq_certify_proof R_peano;
...
 Lemma wf :
  well_founded (algebra.EQT.one_step
    R_peano_deep_rew.R_peano_rules).
....
- : unit = ()
```

Note that the last lemma of the script is the well-foundedness of the given relation.

Termination proof search may be driven with heuristic. Heuristics are described with a small tactic language defined by the grammar below:

```
heuristic ::= Solve [ heuristic_list ]
            | Then [ heuristic_list ]
            | Use criterion
            | Repeat heuristic

heuristic_list ::= heuristic
                 | heuristic ; heuristic_list

criterion ::= DP | DPM | SCC | RMVx | MN  ...
```

The `Use` command applies the criterion given in argument, `Solve` tries to apply a list of heuristics on a goal until one succeeds, `Then` takes a list of heuristics to be applied in sequence and that must succeed on each step, finally `Repeat` repeatedly applies a heuristic until it fails.

The commands of the following example say to use the marked dependency pair criterion (`DPM`) and then to repeat the sequence `SCC ; RMVx`, where `SCC` tries to apply an instance of the GRAPH rule with a decomposition into strongly connected components

which is indeed compatible, and `RMVx` (called on each of the resulting subsets) tries to find a pair $\langle t, u \rangle$ on which rule RMVERTEX applies. If this search is successful RMVERTEX is applied with the relevant pair. Repeating this sequence results in removing pairs recursively until one reaches empty components.

```
let h = heuristic
   "Then [Use DPM; Repeat Then [ Use SCC ; Use RMVx ]]";
set_heuristic "h";
```

It is also possible to drive the ordering constraints solving by selecting which orderings to try. This can be specified using the command `orderparams` with a list of ordering specification (bound or type of polynomials, dimensions of matrices, class of AFS, etc.).

All this can also be done in batch mode. We have to give the input format (among `cime`, `trs`, `srs`, `xml`, `cpf`) and the output format (among `cime`, `trs`, `srs`, `xml`, `cpf`, `coq`). The following example produces an XML certificate `certif.xml` from a given TRS, specified in the `R.trs` file, using the information about orderings, heuristics, etc., provided in file `info.cim`:

```
cime -icime info.cim -itrs R.trs -oxml certif.xml
```

Finally, the next example invokes C*i*ME3 for the compilation of a file `termination_trace.xml` (which may come from the C*i*ME3 termination engine or from another tool) into a COQ source file `coq_certificate.v`)

```
cime -ixml termination_trace.xml -ocoq coq_certificate.v
```

4.3 Trace language

Defining an efficient trace language is a challenging task. Restricting to C*i*ME alone, such a language may be used for objects as different as proof of termination, of confluence, of equality, etc. At the same time, it must keep the verbosity low: those proofs contain so many implicit steps that just emitting their sequence is illusory.

The overall structure of our XML trace is similar to the inference tree mentioned in Section 3.4. It is designed to be as concise as possible, while carrying all information that can be used by C*i*ME to *compute* a proof script. For example, all technical functions depending on the dependency graph will be computed and instantiated by C*i*ME. There will be no trace left of a structure of graph in COQ, just sufficient lemmas, thanks to the techniques of [15] and Section 6.

The XML file lists the system, the applied criteria and gives the relevant orderings to solve the termination problem. An interesting tag is `<PROPERTY>`, which allows the combination of criteria. It has two mandatory attributes, `criterion` which contains the name of the (inference rule) criterion applied and `prop` which expresses the type of problem being solved. For instance `<PROPERTY criterion="dp" prop="sntrs">` means that the DP criterion has to be used to prove that the given system is terminating. The remaining trace contains the system, and the `<PROPERTY>` embedding the proof trace of the generated subproblems.

The full DTD of our XML trace language is available at `http://a3pat.ensiie.fr/pub/a3pat.dtd`. This format is not C*i*ME dependent, it is general enough to allow other provers' proofs to be certified by our approach. In particular, APROVE can output proof traces for COQ certification with C*i*ME3 + COCCINELLE.

However, in order to facilitate interaction between provers and certifiers, the A3PAT and CeTA/IsaFoR [38] teams are involved, with helpful additional comments from the CoLoR team, in the design of an extensible common proof format (CPF). While slightly more verbose, its overall structure is close to C*i*ME's XML format. C*i*ME supports CPF in addition to its own XML trace format, and other provers like APROVE and TTT2 [31] plan to support it in a near future. A detailed description of CPF is available on `http://cl-informatik.uibk.ac.at/software/cpf/`.

5. Formalisation

5.1 COCCINELLE

COCCINELLE is the formalised companion of the C*i*ME3 rewriting tool: its main purpose is to model (in COQ) universal algebras, that is terms built on arbitrary signatures, as well as the usual notions defined in this framework. Among them, rewriting and equational theories generated by a set of rules/equations (seen as pairs of terms) are the key relations over terms, used to model computations and equality.

The links between COCCINELLE and C*i*ME3 are of two distinct natures. Firstly, C*i*ME3 implements data structures, and algorithms on the top of these data, whereas COCCINELLE models (some of) these structures and algorithms, but also states theorems on them. For example, C*i*ME3 uses the Ocaml lists, and has a function to filter a list with respect to a property. In COCCINELLE, there is a modelling of lists and the filtering function, but also some statements that this function actually computes what is intended: an element is in the result of filtering if and only if it belongs to the list given as an argument and fulfils the wanted property. COCCINELLE is hence used to certify the modelling of some crucial algorithms used in C*i*ME3, such as matching modulo associativity-commutativity (AC) [9], unification, RPO orderings, etc.

The second link is looser since it may exist between COCCINELLE and any tool that produces traces in our XML format (Section 4.3). COCCINELLE provides some generic theorems that can be combined with the trace to prove some specific properties, such as the termination of a rewriting system in a given term algebra. COCCINELLE together with the part of C*i*ME3 that turns an XML trace into a COQ file can be seen as a trace compiler, which transforms a trace into a COQ certificate. This second link is at the core of this paper.

COCCINELLE has also been successfully used to prove the accessibility of ordinals less than Γ_0 in the COQ library Cantor, thanks to the RPO modelling (see Section 5.1.2).

5.1.1 COCCINELLE terms and term relations

The modelling of terms is as simple as possible:

```
Inductive term : Set :=
 | Var : variable → term
 | Term : symbol → list term → term.
```

A term is either a variable (`Var`) or a function symbol applied to a list of arguments (`Term`).

Terms in C*i*ME3 enjoy maximal sharing of subterms via a hash-consing mechanism; the modelling in COCCINELLE is a simplified reflection of C*i*ME3 terms. One could argue that this definition is *too* simple since it allows ill-formed terms, and that, thanks to COQ expressive power, a term should contain both its value, and a proof that it is well-formed. We claim, on the contrary, that our definition is well suited to perform proof by induction on terms, and that the ability to deal with ill-formed terms is a nice flexibility which allows us to handle terms transformations (for instance AFS) without building several terms algebras. Moreover, it is possible to define well-formedness in a separate way, and to identify precisely when this property is needed.

The main relation defined over terms is a single step of rewriting (see definition in Section 3). Here comes a reason to introduce the constructor `Var` instead of using the universal quantifiers available in COQ: it allows us to represent a (finite) rewriting system as a finite first-order data, namely a list of pair of terms. For instance, the following rewriting system:

```
plus (x, zero) → x
plus (x, succ y) → succ (plus (x,y))
```

can be represented by the object:

```
[ (Term plus [Var x; Term zero []], Var x);
  (Term plus [Var x; Term succ [Var y]],
      Term succ [Term plus [Var x; Var y]])
```

or by the COQ relation:

```
Inductive Peano (relation term) :=
| Plus_zero : ∀t, Peano t (Term plus [t; Term zero []])
| Plus_succ :
    ∀ t t', Peano (Term succ [Term plus [t; t']])
                  (Term plus [t; Term succ [t']])
```

The first representation is nice since it allows to reason by induction on the number of rules in the system, and define some functions on it, for instance to compute the dependency pairs. The second way allows to use the COQ facilities on inductive definitions (for instance inversion). Both representations are useful, and once the equivalence between them is proven (on a given rewrite system), both are used, according to the wanted property.

The rewriting relations/equational theories are modelled in COQ by our own definition, and not by the COQ native built-in equality. There are several reasons for that:

- the relations we work on are not necessarily symmetric, whereas COQ equality is;
- COQ equality has an underlying matching mechanism which prevents the use of our implicitly quantified variables `Var`;
- COQ matching is not able to handle sophisticated theories such as AC, which would prevent us extending our development to rewriting modulo AC.

The definition of rewriting is done in two separate steps: first, since the variables are implicitly universally quantified, a rule (or an equation) between terms t_1 and t_2 yields all the pairs $(t_1\sigma, t_2\sigma)$ obtained by instantiating the variables by any substitution σ.

```
Inductive axiom (R : relation term): relation term :=
 | instance : ∀t1 t2 sigma, R t1 t2 →
   axiom R (apply_subst sigma t1) (apply_subst sigma t2).
```

Then, it is possible to perform the replacement of $t_1\sigma$ by $t_2\sigma$ under a context $s[\_]_p$, either empty (`at_top`), or deep (`in_context`).

```
Inductive one_step (R : relation term): relation term :=
 | at_top : ∀t1 t2, axiom R t1 t2 → one_step R t1 t2
 | in_context :
     ∀ f l1 l2, one_step_list (one_step R) l1 l2 →
            one_step R (Term f l1) (Term f l2).
```

Then, the usual rewriting relation is simply the transitive closure of the `one_step` relation, and the equational theory is its reflexive, symmetric and transitive closure. Thanks to this formalisation, we have been able to prove all the usual trivial facts on rewriting and equational theories gathered in the COCCINELLE specification file `term_algebra/equational_theory_spec.v`.

5.1.2 Generic termination theorems

A quite important part of COCCINELLE (19 out of 52 kloc) consists of the modelling of term orderings (RPO) and termination criteria together with their formal constructive proofs.

It should be noticed, that due to the COQ definition of well-foundedness for relations, based on the accessibility notion, expressing the termination of a rewriting system amounts to stating that its inverse relation is well-founded. That is in particular why the `Peano` relation defined above is reverted with respect to the original rewriting system.

Before focusing upon the last novelty, namely the subterm refinement of the dependency pair criterion, here is a small catalogue of the available developments:

- the RPO relation with status is defined, proven to be an ordering, compatible with instantiation and addition of context. Moreover, when the underlying precedence over function symbols is well-founded, so is the RPO. The RPO relation can be decided thanks to a COQ function that effectively computes a boolean result[5];
- the dependency pair criterion, both marked and unmarked versions, are proven;
- the following criteria are proven: modularity [40], graphs [25], usable rules [29, 39], subterm [29] and its extension;
- rewriting under some strategies (innermost [26], context sensitive [1]) is modelled, and the corresponding versions of the DP criterion are proven;
- Gramlich's criterion [28] showing plain termination when innermost termination is proven.

5.2 Generalised subterm criterion

All the general theorems that guarantee the soundness of the approach in Section 3.4 are proven via minimal counterexamples: in order to prove that if A is strongly normalising, so is B, one assumes that B is not. Hence there is a *minimal* infinite chain in B, and from this chain, one builds an infinite chain in A, hence a contradiction.

In a proof assistant like COQ, based on intuitionistic logic, a direct translation of these statements and proofs is not possible. The minimality used in the classical proof by counterexample actually appears in the statement of the theorem itself. Given a rewriting system, for each relation over terms, we introduce its restriction over terms such that all their subterms are R-SN.

DEFINITION 3 (Relation restriction). *$sS|_{\mathsf{min}(R)}t$ if and only if sSt and all direct subterms of s and t are R-SN.*

The (stronger) statement of the main DP theorem is as follows:

THEOREM 3. *Let R be a TRS, $\mathsf{SN}(\twoheadrightarrow_{\mathsf{DP}(R),R}|_{\mathsf{min}(R)})$ is equivalent to $\mathsf{SN}(\rightarrow_R)$.*

Concerning the subterm criterion itself, we assume that we have to prove $\mathsf{SN}(\twoheadrightarrow_{D,R}|_{\mathsf{min}(R)})$ for some subset D of $\mathsf{DP}(R)$. Along the line of [29], we define a so-called "projection" over terms, based on a mapping π from function symbols to natural numbers. The projection is also denoted by π. In contrast to [29] we do not require that $\pi(f)$ should be less than the arity of f. Our projection is defined as follows:

$$\pi(x) = x$$
$$\pi(f(t_1, \ldots, t_n)) = \begin{cases} t_{\pi(f)} & \text{if } 1 \leq \pi(f) \leq n \\ f(t_1, \ldots, t_n) & \text{otherwise} \end{cases}$$

We do not require D to be a strongly connected component containing $\langle u_0, v_0\rangle$ since we do not make any assumption on the strategies of trace producing tools: a trace may first remove a pair and then compute a decomposition. The resulting decomposition may significantly differ from what is obtained by first decomposing and then removing $\langle u_0, v_0\rangle$.

THEOREM 4. *Let R be a TRS, and D be a subset of $\mathsf{DP}(R)$. If there exists a projection based on π such that*

- *D can be split into $D' \cup \langle u_0, v_0\rangle$;*
- *$\pi(v_0)$ is a proper subterm of v_0 ;*
- *$\pi(u_0)(\rightarrow_R \cup \rhd)^+\pi(v_0)$;*
- *for every pair $\langle u, v\rangle$ of D' that is strongly connected to $\langle u_0, v_0\rangle$, $\pi(u)(\rightarrow_R \cup \rhd)^\star\pi(v)$;*

then, $\mathsf{SN}(\twoheadrightarrow_{D',R}|_{\mathsf{min}(R)})$ implies $\mathsf{SN}(\twoheadrightarrow_{D,R}|_{\mathsf{min}(R)})$.

[5] This is to date the only RPO used by certifiers.

Our statement is more general than the one given in [29], since we allow a more general projection and the relation $\rightarrow_R \cup \rhd$ instead of $\rhd$, but our proof follows the same line. We remark that if $s \twoheadrightarrow_{\langle u,v\rangle,R} t$ for a pair $\langle u, v\rangle$ of D' which is strongly connected to $\langle u_0, v_0\rangle$, then $\pi(s)(\rightarrow_R \cup \rhd)^\star \pi(t)$, and that if $s \twoheadrightarrow_{\langle u_0,v_0\rangle,R} t$, then $\pi(s)(\rightarrow_R \cup \rhd)^+ \pi(t)$.

Let D_0 denote the strongly connected component of $\langle u_0, v_0\rangle$ in the graph of D, and let D_1 be $D \setminus D_0$, then $\mathsf{SN}(\twoheadrightarrow_{D,R}|_{\mathsf{min}(R)})$ holds if $\mathsf{SN}(\twoheadrightarrow_{D_0,R}|_{\mathsf{min}(R)})$ and $\mathsf{SN}(\twoheadrightarrow_{D_1,R}|_{\mathsf{min}(R)})$ are true.

From $\mathsf{SN}(\twoheadrightarrow_{D',R}|_{\mathsf{min}(R)})$ and since $D_1 \subset D'$, we obtain that $\mathsf{SN}(\twoheadrightarrow_{D_1,R}|_{\mathsf{min}(R)})$ holds. Any chain in $\twoheadrightarrow_{D_0,R}|_{\mathsf{min}(R)}$ is either a chain in $\twoheadrightarrow_{D_0\setminus\langle u_0,v_0\rangle,R}|_{\mathsf{min}(R)}$ or a chain in the composition $(\twoheadrightarrow_{D_0\setminus\langle u_0,v_0\rangle,R}|_{\mathsf{min}(R)})^\star \cdot (\twoheadrightarrow_{\langle u_0,v_0\rangle,R}|_{\mathsf{min}(R)})$. In the first case, $\mathsf{SN}(\twoheadrightarrow_{D_0\setminus\langle u_0,v_0\rangle,R}|_{\mathsf{min}(R)})$ is obtained by inclusion, because of $\mathsf{SN}(\twoheadrightarrow_{D',R}|_{\mathsf{min}(R)})$. In the second case, any chain[6] can be projected by π into a chain in $(\rightarrow_R \cup \rhd)^\star \cdot (\rightarrow_R \cup \rhd)$, containing $\pi(v_0)$, a direct subterm of v_0. By minimality hypothesis, $\pi(v_0)$ is R-SN, hence strongly normalisable by $\rightarrow_R \cup \rhd$. The projected chain is finite, and so is the original one.

Note that our generalised subterm criterion may apply on systems where some redundant rules are added in order to speed up computations. For instance, it proves useful on the following system where the original subterm criterion fails. This system is a slight modification of the one used in [29]. The last rule does not change the underlying equational theory of the rest of the TRS; it is, in fact, expressible as a composition of two other rules. However, such "derived" rules naturally appear in the practice of program optimisation, like deforestation.

$$
\begin{array}{lcl}
x + 0 & \rightarrow & x \\
x + s(y) & \rightarrow & s(x + y) \\
incrlist(nil) & \rightarrow & nil \\
incrlist(x :: l) & \rightarrow & s(x) :: (incrlist(l)) \\
btw(0, 0) & \rightarrow & 0 :: nil \\
btw(0, s(y)) & \rightarrow & 0 :: (btw(s(0), s(y))) \\
btw(s(x), 0) & \rightarrow & nil \\
btw(s(x), s(y)) & \rightarrow & incrlist(btw(x, y)) \\
btw(x + s(z), y + s(z')) & \rightarrow & incrlist(btw((x + z), (y + z'))
\end{array}
$$

The pair $\langle btw(x + s(z), y + s(z')), btw((x + z), (y + z')\rangle$ can be discarded if one uses a projection of btw on its second argument, since one can rewrite $y + s(z')$ into $s(y + z')$ which admits $y + z'$ as a proper subterm. This criterion alone is actually sufficient for a termination proof of this example; the formalisation in Coccinelle allows its certification in Coq.

6. Dealing with graphs

6.1 Implicit representation

As we already mentioned, dealing with explicit representation of graphs in deep embedding certification amounts to showing completeness of some premises: namely, proving that *all* strongly connected parts are considered. Doing this explicitly is a huge resource consuming task *for the proof assistant*, which makes virtually impossible the handling of big graphs, as we may encounter in practice (thousands of arcs, hundreds of vertices, etc.)

We proposed a first solution to this problem in [15], a key point of our approach being to consider implicit graphs only. This section presents a deep version of this technique.

An implicit presentation of relevant connections in the graph is given as a *function* with easy-to-check good properties. This function is computed by the satellite tool *while* it is compiling the proof trace.

Then, in contrast to [15], we may use deep versions of relation-splitting rules which take this function as premise. This way, the proof assistant is freed from heavy computing, while we can use fast versions of deep lemmas.

6.2 Gathering pairs

The main idea is to compute, for a relation $\twoheadrightarrow_{D,S}$ a function *comp* that associates a number to any pair and fulfils the following constraints:

1. every pair in D has a non-zero number,
2. $\forall \langle u, v\rangle, \langle u', v'\rangle \in D$, $comp(\langle u, v\rangle) \geq comp(\langle u', v'\rangle)$ whenever there is a substitution σ such that $v\sigma \xrightarrow[S]{\neq\Lambda\ \star} u'\sigma$.

The decomposition of D, associated with *comp*, that is $\{D_i \mid D_i = \{p \in D \mid comp(p) = i\}\}$ is said to be *compatible*. Note that constraint 2 implies that all pairs in a strongly connected part of the graph share the same number. Further note that in a finite graph, there is a maximum value N reached by *comp* on the pairs in this graph.

The chosen approximation technique is not a limitation here: it is only relevant for proving that *comp* fulfils its constraints. Hence the next key lemma is indeed a generic theorem.

Our deep lemma to deal with graphs is the following:

THEOREM 5. *Let S be a TRS and D be a set of dependency pairs. Let comp be a function fulfilling constraints 1 and 2, let N be its maximum value on D, and $\{D_1, \ldots, D_N\}$ be the finite associated decomposition of D. Then $\mathsf{SN}(\twoheadrightarrow_{D,S})$ if and only if $\forall 0 < n \leq N, \mathsf{SN}(\twoheadrightarrow_{D_n,S})$.*

The proof of this theorem is fully formalised in COCCINELLE.

We illustrate Theorem 5 on the system R_1 below, due to Arts and Giesl [4], which computes the sum of all 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(+(x,y),l))
sum(app(l,cons(x,cons(y,k)))) →
                        sum(app(l,sum(cons(x,cons(y,k)))))
+(0,y) → y
+(s(x),y) → s(+(x,y))
```

The set D of dependency pairs of R_1 consists of:

```
p1 : ⟨+(s(x),y), +(x,y)⟩
p2 : ⟨sum(app(l,cons(x,cons(y,k)))),
        sum(cons(x,cons(y,k)))⟩
p3 : ⟨sum(app(l,cons(x,cons(y,k)))),
        app(l,sum(cons(x,cons(y,k))))⟩
p4 : ⟨sum(app(l,cons(x,cons(y,k)))),
        sum(app(l,sum(cons(x,cons(y,k)))))⟩
p5 : ⟨sum(cons(x,cons(y,l))), +(x,y)⟩
p6 : ⟨sum(cons(x,cons(y,l))), sum(cons(+(x,y),l))⟩
p7 : ⟨app(cons(x,l),k), app(l,k)⟩
```

The (approximated) dependency graph may be found in Figure 3.

We can apply Theorem 5 with the following function *comp*:

$$
\begin{array}{lcllcl}
comp(p_1) & = & 1 & comp(p_2) & = & 5 \\
comp(p_3) & = & 6 & comp(p_4) & = & 7 \\
comp(p_5) & = & 3 & comp(p_6) & = & 4 \\
comp(p_7) & = & 2 & & &
\end{array}
$$

and the rest of the proof will consist in proving $\mathsf{SN}(\twoheadrightarrow_{D_i,R_1})$ for all $0 < i \leq 7$.

Pairs in a connected component must share the same number, and proving termination on such a D_i amounts to removing some of

[6] After the first step, to ensure that the pairs in $D_0 \setminus \langle u_0, v_0\rangle$ are strongly connected to $\langle u_0, v_0\rangle$.

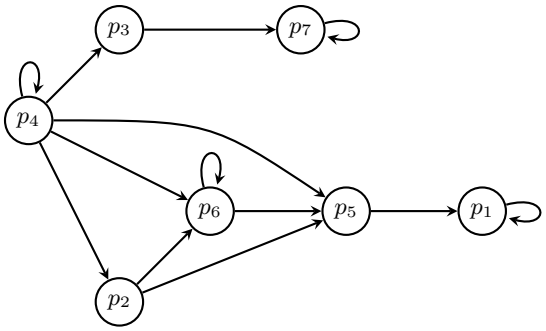

Figure 3. Dependency graph of R_1.

them. To use fully the graph refinement, when a strongly connected component is reached, ordering rules or subterm rules may be used to prune some pairs. New functions *comp* can be computed to apply recursively Theorem 5 on the subgraphs induced by the remaining pairs, and so on.

The main interest of this technique is that the conditions over the function *comp* can be checked *locally*, for every pair of pairs, whereas the general standard graph criterion uses the notion of strongly connected component which is *global*.

Our criterion is very similar to the proposal of [38]. Concerning the splitting of the graph, both statements are roughly equivalent. A minor difference is that in our setting, the splitting is given as a total function from pairs to integers (having in mind a component rank in the graph), whereas in the other, it is an explicit partition given as a list of sets of pairs, sets that have to be checked as disjoint, which is not necessary in our case (we have even removed the necessity for *comp* to be functional, it can be defined as a relation; in this case, all numbers of a pair have to fulfil the inequality).

Our theorem is stated as usually for graph criterion, and in [38] the contraposed statement is emphasised. Obviously this is better in practice, and indeed, when applying our own theorem on actual rewriting systems, we do the same.

Concerning the singleton vertices that are not a SCC, there is no special treatment since we only require a graph splitting according to the condition given by Theorem 5, and not a full decomposition into SCC. Then if the decomposition contains a part reduced to a singleton, either its termination is trivial (no loop) or an extra argument has to be provided by the trace.

7. Experiments

The subterm criterion alone is cheap in proof search and very efficient: we tried our approach on the 1391 TRS of the Termination Problems Data Base [34], the reference benchmark for termination provers.

We used the termination engine of C*i*ME3 to run these tests. Note that, since C*i*ME3's certification engine handles more termination criteria and orderings than its termination engine, higher scores could be reached with, say, the combination of AProVE (for termination) and C*i*ME (for certification), thanks to the common proof format.

The following table summarises our experiments on a 3GHz, 16GB computer running Debian Linux. The first row records the number of problems solved and certified by C*i*ME using marked DP, dependency graphs, and various orderings: RPO (status LEX only) with AFS, polynomials over integers and over matrices, but *without* the subterm criterion. The second row records the results when one adds subterm to the criteria.

	Solved	ST used	SN $\leq 10s$	`.vo` $\leq 60s$
No ST	581	—	547	490
With ST	620	415	583	523

The first column contains the number of problems proven (and certified) to be terminating. The use of the efficient subterm criterion improves this number significantly. Moreover, this criterion contributes to the resolution of 415 problems. The third column states that more than 94% of the 620 problems are proven to be terminating in less than 10 seconds each. The average time for termination proof is $5.47s$ without subterm, whereas is it $5.26s$ using subterm.

Regarding the 620 generated Coq files, about 319 (51.5%) are compiled (into `.vo` files) in less than 10 seconds each (average time $5.24s$), and 523 (84.3%) in less than one minute each (average time $13.5s$). If we omit the subterm criterion, 288 files are compiled in less than 10 seconds (49.5% of the 581) with the average time of $5.52s$. The use of this criterion induces a slight speed increase in compilation.

Note that the times of proof search and certification are reasonably small for C*i*ME3 to be used as a practical tool in everyday development involving TRS and proof assistants.

8. Conclusion and future work

We propose a solution for improving automation in proof assistants. This solution is based on satellite tools that generate traces, and formal libraries to certify those traces. We focus on termination proofs which are of great interest for program verification and automated reasoning, and we target the Coq proof assistant.

There are several works to be mentioned with reference to the communication between automated provers and Coq. Among them, the tableaux-based theorem prover Zénon [19] produces Coq proof terms as certificates. Elan provides means to produce Coq certificates for rewriting [35]. Bezem describes an approach regarding resolution [6]. However, these systems do not tackle the problem of termination proofs.

Regarding termination problems, we mentioned the termination-specialised CoLoR and IsaFoR and discussed pure deep embedding and extraction issues in Section 2.2. It should be noticed that the A3PAT approach neither subsumes nor is subsumed by the techniques of CoLoR or IsaFoR. However, our proof structure, based on inference rules, is highly modular, as well as our graph management (Section 6). In a near future, it should be possible to plug into a proof a certificate coming from another certifier that uses techniques we do not handle yet, e.g. for termination proofs of some of the D_i.

As remarked in Section 4.1, if the proof is not delegated from a proof assistant (but rather directly searched for with an automated prover) and checked for certification only, it is important to verify that the relation defined in the proof assistant is indeed the one the user is interested in! That is why we try to keep the definitions of rewriting systems as human-readable as possible.

An interesting side effect of formal proof of criteria is a possible weakening of their premises. We illustrated this fact with the first full formalisation (and certificate generation) of the subterm criterion [29] for which we could weaken constraints on projections and relations between projected pairs members (Theorem 4).

There exist intricate termination problems coming from practical applications. The TPDB includes, for example, a μ-CRL specification of communicating processes (377 rewriting rules) or transformations of context-sensitive programs (several problems each containing more than 50 rules and leading to dependency graphs of about 1000 arcs and 80 vertices). These problems are handled well by our approach [15].

C*i*ME3 and COCCINELLE can manage other kinds of proofs, completion, etc. We plan on adding automation to proof assistants on these topics in the near future.

Both the C*i*ME3 rewriting tool/certifier and the COCCINELLE library are distributed under the terms of the CeCill-C licence. They may be downloaded from the project's web page: `http://a3pat.ensiie.fr`

References

[1] B. Alarcón, R. Gutiérrez, and S. Lucas. Context-Sensitive Dependency Pairs. In S. Arun-Kumar and N. Garg, editors, *FST&TCS'06*, volume 4337 of *LNCS*, pages 297–308, Kolkata, India, 2006. Springer-Verlag.

[2] P. Schneider-Kamp, R. Thiemann, E. Annov, M. Codish, and J. Giesl. Proving Termination using Recursive Path Orders and SAT Solving. In B. Konev and F. Wolter, editors, *FroCos'07*, volume 4720 of *LNAI*, pages 267–282, Liverpool, UK, Sept. 2007. Springer-Verlag.

[3] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *TCS*, 236:133–178, 2000.

[4] T. Arts and J. Giesl. Automatically Proving Termination Where Simplification Orderings Fail. In M. Bidoit and M. Dauchet, editors, *Theory and Practice of Software Development*, volume 1214 of *LNCS*, Lille, France, Apr. 1997. Springer-Verlag.

[5] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[6] M. Bezem, D. Hendriks, and H. de Nivelle. Automated proof construction in type theory using resolution. *JAR*, 29(3-4):253–275, 2002.

[7] F. Blanqui, S. Coupet-Grimal, W. Delobel, S. Hinderer, and A. Koprowski. Color, a Coq library on rewriting and termination. In Geser and Sondergaard [24].

[8] É. Contejean. The Coccinelle library for rewriting. URL `http://www.lri.fr/~contejea/Coccinelle/coccinelle.html`.

[9] É. Contejean. A certified AC matching algorithm. In V. van Oostrom, editor, *RTA'04*, volume 3091 of *LNCS*, pages 70–84, Aachen, Germany, June 2004. Springer-Verlag.

[10] E. Contejean and C. Marché. CiME: Completion Modulo E. In Ganzinger [23], pages 416–419. URL `http://cime.lri.fr/`.

[11] É. Contejean, C. Marché, B. Monate, and X. Urbain. Proving termination of rewriting with C*i*ME. In A. Rubio, editor, *WST'03*, pages 71–73, June 2003. URL `http://cime.lri.fr`. Technical Report DSIC II/15/03, Univ. Politécnica de Valencia, Spain.

[12] É. Contejean, C. Marché, A. P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *JAR*, 34(4): 325–363, 2005.

[13] É. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. In B. Konev and F. Wolter, editors, *FroCos'07*, volume 4720 of *LNAI*, pages 148–162, Liverpool, UK, Sept. 2007. Springer-Verlag.

[14] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *LNCS*, Tallinn, USSR, 1988. Springer-Verlag.

[15] P. Courtieu, J. Forest, and X. Urbain. Certifying a Termination Criterion Based on Graphs, Without Graphs. In C. Muñoz and O. Ait Mohamed, editors, *TPHOLs'08*, volume 5170 of *LNCS*, pages 183–198, Montréal, Canada, Aug. 2008. Springer-Verlag.

[16] P. Courtieu, G. Gbedo, and O. Pons. Improved matrix interpretations. In J. van Leeuwen et al., editor, *SOFSEM'10*, *LNCS*, Špindleruv Mlýn, Czech Republic, Jan. 2010. Springer-Verlag. To appear.

[17] N. Dershowitz. Orderings for term rewriting systems. *TCS*, 17(3): 279–301, Mar. 1982.

[18] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. North-Holland, 1990.

[19] D. Doligez. Zenon. URL `http://focal.inria.fr/zenon/`.

[20] J. Endrullis. Jambox. URL `http://joerg.endrullis.de/index.html`.

[21] J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *JAR*, 40(2-3):195–220, 2008.

[22] C. Fuhs, A. Middeldorp, P. Schneider-Kamp, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *SAT'07*, volume 4501 of *LNCS*, pages 340–354, Lisbon, Portugal, May 2007. Springer-Verlag.

[23] H. Ganzinger, editor. *RTA'96*, volume 1103 of *LNCS*, New Brunswick, USA, July 1996. Springer-Verlag.

[24] A. Geser and H. Sondergaard, editors. *WST'06*, Aug. 2006.

[25] J. Giesl, T. Arts, and E. Ohlebusch. Modular Termination Proofs for Rewriting Using Dependency Pairs. *JSC*, 34:21–58, 2002. doi:10.1006/jsco.2002.0541.

[26] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Improving Dependency Pairs. In *LPAR'03*, volume 2850 of *LNAI*, pages 165–179, Almaty, Kazakhstan, Sep. 2003. Springer-Verlag.

[27] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In U. Furbach and N. Shankar, editors, *IJCAR'06*, volume 4130 of *LNCS*, Seattle, USA, Aug. 2006. Springer-Verlag.

[28] B. Gramlich. On proving termination by innermost termination. In Ganzinger [23], pages 93–107.

[29] N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: Techniques and Features. *IC*, 205(4):474 – 511, 2007.

[30] A. Koprowski. TPA: Termination Proved Automatically, In F. Pfenning, editor, *RTA'06*, volume 4098 of *LNCS*, pages 257 – 266, Seattle, USA, Aug. 2006. Springer-Verlag.

[31] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In R. Treinen, editor, *RTA'09*, volume 5595 of *LNCS*, pages 295–304, Brasília, Brazil, July 2009. Springer-Verlag.

[32] K. Kusakari, M. Nakamura, and Y. Toyama. Argument filtering transformation. In G. Nadathur, editor, *PPDP'99*, volume 1702 of *LNCS*, pages 47–61, Paris, France, 1999. Springer-Verlag.

[33] D. S. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Mathematics Department, Louisiana Tech. Univ., 1979. URL `http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html`.

[34] C. Marché and H. Zantema. The termination competition 2006. In Geser and Sondergaard [24]. URL `http://www.lri.fr/~marche/termination-competition/`.

[35] Q. H. Nguyen, C. Kirchner, and H. Kirchner. External rewriting for skeptical proof assistants. *JAR*, 29(3-4):309–336, 2002.

[36] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Volume 2283 of *LNCS*, 2002. Springer-Verlag.

[37] The Coq Development Team. *The Coq Proof Assistant Documentation – Version V8.2*, June 2008. URL `http://coq.inria.fr/refman/`.

[38] R. Thiemann and C. Sternagel. Certification of Termination Proofs using CeTA. In T. Nipkow and C. Urban, editors, *TPHOLs'09*, volume 5674 of *LNCS*, pages 452–468, Munich, Germany, Aug. 2009. Springer-Verlag.

[39] R. Thiemann, J. Giesl, and P. Schneider-Kamp. Improved modular termination proofs using dependency pairs. In *IJCAR'04*, volume 3097 of *LNAI*, pages 75–90, Cork, Ireland, 2004. Springer-Verlag.

[40] X. Urbain. Modular and incremental automated termination proofs. *JAR*, 32(4):315–355, 2004.