Automated Certified Proofs with CiME3*

É. Contejean[1,2] P. Courtieu[3] J. Forest[4,3] O. Pons[2]
X. Urbain[4,2]

[1] CNRS, LRI, UMR 8623, Orsay, F-91405
[2] INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893
Univ Paris-Sud 11, Orsay, F-91405
[3] Lab. Cédric, CNAM, Paris, F-75141
[4] ENSIIE, Évry, F-91025

evelyne.contejean@lri.fr, pierre.courtieu@cnam.fr,
julien.forest@ensiie.fr, olivier.pons@cnam.fr,
xavier.urbain@ensiie.fr

Abstract

We present the rewriting toolkit C*i*ME3. Amongst other original features, this version enjoys two kinds of engines: to handle and discover proofs of various properties of rewriting systems, and to generate Coq scripts from proofs traces input in *certification proof format* (CPF) (an XML format widely accepted by the certified rewriting community) in order to certify them with a skeptical proof assistant like Coq. C*i*ME3 may thus be used to add automation to proofs of termination or confluence in a formal development in the Coq proof assistant.

1 Introduction

Automated tools based on complex mathematical arguments have become widely used in various domains of computer science. Cryptographic systems, proof assistants or systems analysing programs involve highly intricate deduction procedures that are indeed beyond human capabilities. However, this situation raises a real issue as the counterpart of such a useful deduction power is the difficulty to trust a result that no human can check.

Regarding rewriting, and in particular automated termination proof, there has been many new tools since the introduction of the dependency pair approach [2] at the end of the 90's: C*i*ME 2, AProVE [13], TTT 2 [16], Jambox[1], etc., to cite a few of them.

However, all these tools exhibited incorrect behaviour at some point, in particular during the Termination Competition [17] or its preparatory rounds. Several approaches to certify the results of automated provers with skeptical proof assistants have been developed: A3PAT [6, 5] and CoLoR/Rainbow [3] targeted to Coq, CeTa [18] targeted to Isabelle/HOL.

*This work was partially supported by the french ANR projec A3PAT (ANR-05-BLAN-0146).

[1] http://joerg.endrullis.de/jambox.html

New member of the C*i*ME family[2], C*i*ME3 is a toolbox dedicated to the handling and analysis of rewriting programs. It allows one to define term algebras, rewriting systems, etc., and to perform a range of treatments over these terms and systems: namely computation, normalisation, matching and unification (modulo equational theories), completion (Knuth-Bendix) and proofs of equality (both modulo equational theories), etc. An important part of C*i*ME3 is dedicated to the proofs of termination (including solving of ordering constraints), local confluence and convergence. C*i*ME3 enjoys a top level mode for interactive development of rewrite programs; it can also be used in batch mode with full automation.

C*i*ME3 has been developed in the context of the A3PAT project[3] regarding certification of automated proofs and automation for proof assistants. In comparison to its ancestors, C*i*ME3 features, in addition to its various proof engines, a certification mechanism which issues proof traces and *certificates*, and which allows one to check that the result of a (semi-) decision procedure is correct. To be certified, a proof trace can be translated into a script checked by a trusted tool: a skeptical proof assistant like COQ[4], or it can be given directly to a certified dedicated tool like CETA. A unique feature of C*i*ME3 is in particular the (discovery and) certification of proofs of convergence. C*i*ME3 was in 2010 the only certifier tool targeted for COQ in the Category "certifying" of the Termination Competition.

The present article is organised as follows.We present in Section 2 some of the proof engines at work in C*i*ME3, for termination in Sections 2.1.2 and 2.1.3, and local confluence, KB completion and convergence in Section 2.2. Some criteria allow for parallel solving, this is presented in Section 2.1.4. The certification engine is then described in Section 3, along with its input, output, and the general structure of emitted COQ scripts. Section 3.3 sketches the use of C*i*ME3 in batch mode. We conclude with a list of resources and companion tools, and with future work regarding the current implementation.

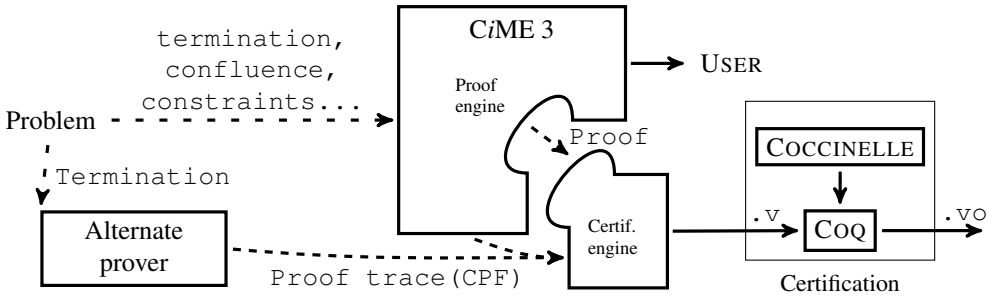

Figure 1: Global C*i*ME 3 architecture.

[2] http://cime.lri.fr
[3] http://a3pat.ensiie.fr
[4] http://coq.inria.fr

2 Proof Engines

2.1 Core Language, Termination

C*i*ME3 can be used in interactive or batch mode. We hereafter describe some of the basic commands for the interactive mode. Note that the batch mode can accept additional files in various formats (like `TRS` or `SRS` formats from the Termination Competition, or in CPF[5], developed by the CETA group and our team), however the commands described here can be used in both modes. See Section 3.3 for batch specific command line options.

The core top level language of C*i*ME3 is a simple language which allows the user to type in expressions terminated by a semicolon. Simple expressions consist of booleans, integers, strings, and basic operations on them. As in previous versions of C*i*ME the nominal language is powerful enough to define functions (polymorphic higher order, allowing partial application, functions as arguments) with a **`let`** `fun` construct. Application is denoted by a juxtaposition of the function and its arguments, as in LISP or other functional languages based on λ-calculus. A detailed manual is available online on the A3PAT web page. We focus here on the new syntax and features of C*i*ME3 for rewrite systems and constraints.

2.1.1 Declarations

The user can define and name objects, algebras, and systems in the C*i*ME input language akin to the definitions found in the literature: sets of variables, signatures (symbols with arities), term algebras, rewrite systems, term ordering constraints...

A **`let`** construct introduces *global* declarations. Listing 1 declares successively a set `X` of variables, a signature `F` for Peano numbers and addition, and the corresponding term algebra `T`. One can then easily declare terms, rewrite systems, and even ordering constraints (built from conjunctions `/\` and disjunctions `\/` of atomic constraints).

Listing 1: A few simple declarations.

```
let X = variables "x,y";
let F = signature "plus : binary; 0:constant; S:unary;";
let T = algebra F;
let t1 = term T "S(0)";
let R = trs T " plus(0,x) -> x; plus(S x, y) -> S(plus(x,y)); ";
let c = order_constraints T "0 < S(0) /\ S(plus(x,y)) < plus(S(x),y)";
```

2.1.2 Ordering Parameters

C*i*ME3 features a proof engine dedicated to the discovery of (well-founded) term orderings that fulfil a set of (ordering) constraints. These constraints may come from a direct declaration by the user (see Listing 1), or may appear as the result of operations like the application of termination criteria on a termination problem, i.e. from termination constraints. As soon as the ordering constraints are provided, one can try to discover a relevant well-suited ordering pair $(\preceq, <)$ using the command `ordering_solve` or `ordering_solve_strict` depending on the expected monotonicity of the ordering, that is depending on whether only $\preceq$, or both $\preceq$ and $<$, respectively, are supposed to be monotonic.

[5]The Certification Proof Format, `http://cl-informatik.uibk.ac.at/software/cpf/index.php`

Listing 2: Ordering constraints solving commands.

```
ordering_solve c; (* Search for weakly monotonic well-founded ordering fulfilling c *)
ordering_solve_strict c; (* Search for strictly monotonic well-founded ordering *)
```

C*i*ME3 can search for different kinds of orderings, all of which being parameterised by the user. The presently implemented orderings range from full RPO (with or without argument filtering systems) to various polynomial and matrix interpretations. Note that the *certification* engine can accept traces from other provers, and, thus, can handle more orderings than those that are currently searched for by the termination proof engine.

The parameters for the search are provided using the grammar in Figure 2. Ordering specifications (*Ordering_param*) are of the generic form n_1 *ord_kind* n_2 n_3 $n_4 \ldots$ where *ord_kind* specifies the kind of ordering for which to search, n_1 is the (optional) timeout of the solver in seconds, n_2 (only for polynomial and matrix interpretations) specifies the upper bound of the (nonnegative integral) coefficients in polynomials or matrices, n_3 (matrices only) specifies the size of the matrix coefficients, and n_4 and following parameters are the list of allowed sizes of strict sub-matrices as defined in [10]. For example, `matrix 1 3 1 2` specifies a search for matrix interpretations, with matrix coefficients less or equal to 1, 3×3 matrices, and strict sub-matrices of size either 1 *or* 2. As the optional timeout is not provided, the default will be used (no timeout). Similarly, `30 linear 3` specifies a search for linear polynomials with coefficients less or equal to 3, and a search timeout after 30s.

The user can declare several ordering parameters for different situations. This is illustrated on Listing 3 where two different ordering parameters sequences `op1` and `op2` are declared, in order to be used later in heuristics (see Listing 4).

Listing 3: Ordering parameters.

```
let op1 = params "30 linear 3; 30 simple 2; 30 rpo; 30 matrix 2 2 1; 30 matrix 1 3 1 2";
let op2 = params "linear 3; simple 2; rpo; matrix 2 2 1; matrix 1 3 1";
```

Ordering parameters are interpreted sequentially, unless a parallel search is enabled. In the latter case, several orderings are searched for at the same time, depending on the computer architecture and the specific parameters given by the user, see Section 2.1.4.

2.1.3 Criteria and Heuristics

Termination criteria are commonly seen as transformations of termination problems into sets of other termination problems (possibly empty); they can be represented as inference rules [6, 9] or processors [14]. The strategy of application of termination criteria is specified in C*i*ME3 with the help of *heuristics*. Intuitively, a successful heuristic must describe a cover of a closed termination proof tree.

C*i*ME comes with a simple heuristic description language which contains built-in criteria and criteria combinators. Heuristics are specified following the grammar described in Figure 2 where id is a previously declared heuristic, `manna_ness` stands for the well-known Manna and Ness criterion, `lex_manna_ness` for rule removal using lexicographic combination of orderings, `DP` and `DPM` stand respectively for unmarked and marked dependency pair criteria [2], `DPG` for the graph refinement (estimation EDG) of dependency pairs [12] (DPG returns the set of strongly connected components), `ST` denotes the extended sub-term criterion [5], and `RMVx` denotes vertex removal in dependency graphs components [15] and its variant `RMC` which removes all vertices of the component.

Ordering_param	::=	`linear` n | `simple` n | `quadratic` n | `rpo` | `matrix` n n $n+$
Solver_params	::=	n *Ordering_param* (`;` *Solver_params*)?
Criterion	::=	`manna_ness` | `lex_manna_ness` | `DP` | `DPM` | `DPG` | `ST` | `RMVx` | `RMC`
Heuristic	::=	*Criterion* {*Solver_params*}? | id {*Solver_params*}?
		| `Then` [*Heuristic_l*] | `Do` n *Heuristic* | `Repeat` *Heuristic* | `Solve` [*Heuristic_l*]
Heuristic_l	::=	*Heuristic* (`;` *Heuristic_l*)?

Figure 2: Grammar for heuristics and ordering parameters.

A heuristic x applies to a problem and returns the set of generated sub-problems. The following constructs allow one to combine heuristics. `Then[` x`;`$y\ldots$ `]` applies heuristic x on a problem and then, if it succeeds, heuristic y on each generated sub-problem. It fails if either x or y fails. `Repeat` x calls x on a problem and, if x succeeds, applies `Repeat` x again recursively on sub-problems. It never fails. `Do` n x is similar to `Repeat` but uses a limit n for recursive depth. Finally `Solve [` x`;`$y\ldots$ `]` tries to apply x, and in the case x does not succeed tries to apply y, etc., it fails if all heuristics failed. When all recursive calls have ended, the proof is successful if all sub-problems are empty.

Listing 4 provides two examples. Heuristic `h2` first tries to apply repeatedly rule removal by lexicographic combination (using strictly monotonic orderings) until it fails to do so. It transforms then the remaining TRS termination problem into a marked dependency pairs termination problem and repeats recursively the previously declared `h1`. Heuristic `h1` first splits the given (DP) problem into sub-problems corresponding to the strongly connected components of the dependency graph, and then tries on each component: firstly the extended sub-term criterion, and then, if it failed, vertex removal with a weakly monotonic ordering.

Some criteria may be parameterised specifically, for instance the number of rewrite steps allowed in the extended sub-term criterion may be set to n using command `subtermparams` n.

Listing 4: Heuristics declarations and commands.

```
let h1 = heuristic " Repeat Then [ DPG ; Solve [ ST ; RMVx {op2} ] ] ";
let h2 = heuristic " Then [ Repeat lex_manna_ness {op1} ; DPM ; h1] ";
set_heuristic "h2";                          (* Sets the termination heuristic. *)
```

When relevant parameters are set, proof search may be launched using commands `termination`, `confluence`..., which print answers and possibly store traces.

Listing 5: Proof search.

```
termination R;
local_confluence R;
convergence R;
complete o R;                              (* Completion of R using term ordering o *)
prove_goal o R t1 t2;(* Ordered completion of R, stops when t1 and t2 are found equal*)
unify (term T "plus(0,x)") (term T "plus(y,0)");
```

2.1.4 External Solvers, Parallel Search for Orderings

External Solvers Ordering constraints are nowadays usually translated into SAT problems. Thus, the main solving parts are delegated to a SAT solver [11, 1] or an SMT

solver [16].This scheme is implemented in C*i*ME3, and the discovery of relevant interpretations, RPO and AFS, and application of the extended sub-term criterion amount to solving SAT problems.

The directive `#Set_sat_solver` *invoc_name* allows one to specify the invocation name of the external SAT solver, which must fulfil the requirements of the SAT format for input and output, and the invocation of which must be of the form: *invoc_name in_file out_file*.

Note that polynomial interpretations and LPO+AFS may still be found without the help of any external solver, using the internal Diophantine constraint solver of C*i*ME [8].

Parallel Search Ordering solvers may be used in parallel. When ordering constraints are generated during termination analysis, C*i*ME3 forks one solver process by ordering constraints set and ordering parameter (Listing 3), thus searching for different orderings in parallel. Note that solving of different ordering constraints is thus also parallelised. The maximum number of processes that can be launched in parallel may be set using the `#Set_nb_proc` directive. If this limit is reached, new forks are put in a queue and wait for previous ones to stop before they can be activated. Default behaviour is sequential computation.

2.2 Local Confluence, Completion and Convergence

A noticeable feature of C*i*ME3 is its ability to check and moreover certify local confluence. Combining this with its possibility to prove and certify termination, one can easily obtain *proof and certification* of convergence. To date and to our knowledge C*i*ME3 is the only tool that can prove and certify convergence of rewrite systems. The proof search of local confluence (and hence of convergence) is obtained by checking joinability of critical pairs. When proving local confluence, C*i*ME3 stores a trace in order to be able to certify it later.

Since critical pairs computation is the core of the standard Knuth-Bendix completion, our implementation of KB completion (commands `complete` or `prove_goal`) also benefits from the trace production: given two terms which are found equal thanks to ordered completion, C*i*ME3 yields a trace of equality between these terms, and can also produce a COQ certificate [4]. Completion modulo AC is also a part of C*i*ME3, but is not instrumented yet.

Certification of convergence proof is obtained by application of Newman's lemma, either using a known proof of termination, or assuming termination of the considered system. Convergence is then proved by showing that in particular each critical pair is joinable: we try to normalise each member of the pair and to show that both reduce to the same term.

3 Certification Engine

As proof engines perform proof search, various verbosity levels allow one to control the process. Once a proof is discovered, its description is printed on the standard output. Generation of a CPF trace[6], or directly of the corresponding COQ script (for

[6]Presently for termination proofs only as CPF is not yet extended to handle confluence proofs.

termination, local confluence and convergence), is done by command or through batch mode (Section 3.3).

The purpose of the certification engine is to take a proof trace as an input, and to output a COQ script for this trace's certification. Traces may come from proofs discovered with C*i*ME3, or with other provers (APROVE, TTT ...) provided they come as CPF files[6].

3.1 Input, Output

The normal input trace format for termination proofs is CPF. C*i*ME3 is not yet able to certify all criteria supported by the CPF format, but may generate scripts for all criteria it uses in proof discovery. For properties other than termination (including termination for convenience), one can take as an input a *problem* instead of a proof trace. In this case all the problem formats described in Section 2.1 are supported, C*i*ME will search the proof by itself and will generate the script, without any intermediate trace. See Listing 7.

The techniques for the generation by C*i*ME of COQ scripts for certification have been described in previous works [5, 7, 9]. The compilation of those scripts relies on the COCCINELLE library, which allows for deep and shallow embeddings of the theory of rewriting. Relying on a deep embedding to reuse generic theorems instead of reproving them for each instances is usually fast. However it is interesting to notice that C*i*ME does use both embeddings depending on the proofs performed. We claim in particular that dealing with a shallow representation of the dependency graph is very efficient from a computational point of view, as presented in [9]. Listing 6 illustrates the fact that the formalisation is partly shallow: on the one hand, symbols and rewriting rules are defined by new inductive types (`symb` and `R_rules`), on the other hand, term structure and rewriting relations rely on generic deep definitions (`Term`, `Var` and `one_step`). However, depending on the content of the proof, a deep version of `R_rules`, automatically proved equivalent, is defined when needed.

3.2 Structure of a Proof

Listing 6 presents an excerpt of the proof of convergence for the example in Listing 1. It displays in particular the general structure of a proof script for certification using the A3PAT approach. Symbols (line 4) and signature (line 7) are defined at the beginning of the file, followed by the definition of the system itself (line 10)[7]. The main lemma for termination is at the end of the last module. The proofs related to the instances of the different criteria mentioned in the input trace are formalised and proved in a (nested) sequence of modules (from line 16), thus mirroring roughly the structure of proof tree described in [5].

Eventually, the confluence proof may be generated with or without an actual proof of termination (in the latter case, the confluence is proved *with the assumption* that the system terminates). As termination is shown in our example, there is no assumption in Listing 6.

To compile the script, and thus to ensure its validity, one must have the COCCINELLE library installed, and a shell variable `COCCINELLE` set to its root directory. The last lines of the generated COQ file contains the `coqc` command line to run.

[7] Notice that `R_rules t u` means that `u` rewrites to `t`.

Listing 6: Coq script structure (notations are simplified).

```
1 Require Import equational_theory...      (* Preamble: require coccinelle files. *)
2 Module algebra.
3   Module F <:term_spec.Signature.                    (* Signature definition *)
4     Inductive symb:Set := plus: symb | S: symb | O: symb.        (* Symbols *)
5     ...
6   End F.
7   Module Alg := term.Make(F) (IntVars)      (* Algebra by functor instantiation *)
8 End Algebra.
9
10 Inductive R_rules: term → term → Prop :=          (* Definition of the TRS *)
11 |R_rule0: R_rules (Var 1) (Term plus [Term O [];Var 1])      (* plus(O,x)→x *)
12 |R_rule1: R_rules (Term S [Term plus [Var 1;Var 2]])
   (* plus(S(x),y)→S(plus(x,y)) *)
13                    (Term plus [Term S [Var 1];Var 2]).
14
15 Module WF_R.
16   ...                                        (* Criteria application proofs *)
17   Lemma wf: well_founded (one_step R_rules).      (* Main termination lemma *)
18     ...
19   Qed.
20 End WF_R.
21 Module Confluence := Newman.Confluence(...).
22 ...
23 Lemma confluence: ∀ x, Newman.confluence _ (one_step R_rules) x.
24   ...                                       (* Joinability of critical pairs.  *)
25 Qed.
```

3.3 Batch Mode Command Line Options

In batch mode C*i*ME takes an input file in a specified format, and returns a result file in a specified output format. The whole process may include proof searches and/or generation of COQ scripts. Options `-term`/`-noterm` and `-confl`/`-noconf` specify which properties must be considered. Input options include: `-itrs` (`-isrs`) for `TRS` (`SRS`) (non XML) formats, `-icime` for C*i*ME language, and `-icpf` for CPF format. Similarly, output options include: `-ocpf`, `-ocoq`, and `-ocime` (outputs the global environment in C*i*ME format).

Listing 7: Sample command lines.

```
1 cime -itrs foo.trs -term -ocpf foo.cpf
2 cime -icpf foo.cpf -ocoq foo.v
3 cime -itrs foo.trs -term -ocoq foo.v
4 cime -itrs foo.trs -term -confl -ocoq foo.v
5 cime -icime preamble.cim3 -itrs foo.trs -term -confl -ocoq foo.v
6 cime -icime preamble.cim3 -noterm -confl  -itrs foo.trs -ocoq foo.v
```

Line 1 asks for a termination proof for the system in file `foo.trs` and then generates its CPF trace into `foo.cpf`. Line 2 generates the COQ script `foo.v` to certify CPF trace `foo.cpf`. Line 3 is equivalent to line 1 followed by line 2 (no CPF file is generated). Line 4 additionally generates the COQ proof of confluence. Line 5 is similar to 4 but loads a preamble in C*i*ME format before proof search, this is a usual way to set parameters in batch mode. Line 6 produces a COQ confluence proof parameterised

by a proof of termination (not discovered). All these options may finally be chained in one command line:

```
cime -noconfl -itrs foo1.trs -ocpf foo1Term.cpf -confl -ocoq foo1TermConf.v\
     -icpf foo2.cpf -ocoq foo2Term.v -noterm -ocoq foo2Conf.v
```

This asks for a termination proof for the system in `foo1.trs`, generates the corresponding CPF file `foo1Term.cpf`, then produces (without any new search) a termination and confluence COQ proof `foo1TermConf.v`. It then generates the COQ script `foo2Term.v` to certify the CPF trace `foo2.cpf`, and build script `foo2Conf.v` to certify the confluence of the TRS in `foo2.cpf`. The last proof is parameterised by a proof of termination (not computed).

3.4 Experiments

The following tables present some experiments run on a 24GB machine equipped with 12 cores and running linux. The base for termination is the category TRS (taken *raw* of the TPDB 5.0 consisting of 2506 problems[8]; 11 simultaneous processes are allowed. Local confluence and convergence tests are run on the problems that are proved to be terminating. The version of COQ is the 8.3 release. A global timeout is set to 180s.

		$< 0.1s$	$< 1s$	$< 10s$	$< 30s$
Termination	1155	354 (30.7%)	730 (63.2%)	1012 (87.6%)	1080 (93.51%)

The vast majority of termination proofs are discovered in less than 10s, and 30% in less than a tenth of a second. Compared to certification by extracted tools like CETA, the compilation of COQ script is slow. However, 730 termination proofs are certified under the time limit. Actually, the only failures observed (out of this run) are due to time or memory limitations.

Regarding local confluence and convergence, 310 proofs of convergence are discovered, 306 of which are certified under 180s (4 timeouts). The average compile time for convergence is less than 17s. More than 53% of the convergence proofs are certified in under 10s.

		$< 5s$	$< 10s$	$< 30s$	$< 60s$
Certif. Term.	730	200 (27.5%)	293 (40.3%)	481 (66.2%)	573 (78.8%)
Certif. Confl.	306	157 (51.3%)	234 (76.5%)	287 (93.8%)	299 (97.7%)
Certif. Conv.	306	53 (17.3%)	164 (53.6%)	271 (88.6%)	293 (95.8%)

4 Resources and Perspectives

C*i*ME3 is an open source piece of software; it is available under the CeCiLL-C licence on the resources page of the A3PAT project: `http://a3pat.ensiie.fr/pub`. The web site provides ELF executables for 64 bit architectures, a tarball of the sources and installation instructions, as well as a short user-manual (`http://a3pat.ensiie.fr/pub/manual-cime3.html`), and a script tool to ease benches and tests over databases of problems. Note that it is also possible to give C*i*ME3 a try online through our dedicated web interface `http://a3pat.ensiie.fr/`

[8] In particular strategies are *not* taken into account.

`online`, with a limited choice of options. The companion library COCCINELLE is also available from this page, and requires version `8.3` of the COQ proof assistant.

Perspectives regards proof and certification engines. Regarding proofs, some techniques implemented in C*i*ME 2 have not been transferred yet, notably termination modulo equational theories and modular techniques, including usable rules refinements.

Regarding certification techniques for termination a short term perspective is the handling of arctic matrices, min/max polynomials, useable rules, and proofs under strategies, as all the formal material is ready in COCCINELLE.

References

[1] Elena Annov, Michael Codish, Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. A SAT-based Implementation for RPO Termination. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (Short Paper)*, November 2006.

[2] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[3] 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.

[4] Évelyne Contejean and Pierre Corbineau. Reflecting proofs in first-order logic with equality. In *20th International Conference on Automated Deduction (CADE-20)*, number 3632 in Lecture Notes in Artificial Intelligence, pages 7–22, Tallinn, Estonia, July 2005. Springer-Verlag.

[5] Évelyne Contejean, Pierre Courtieu, Julien Forest, Andrei Paskevich, Olivier Pons, and Xavier Urbain. A3PAT, an Approach for Certified Automated Termination Proofs. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 10)*, pages 63–72. ACM, 2010.

[6] 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.

[7] Évelyne Contejean, Julien Forest, and Xavier Urbain. Deep-Embedded Unification. Technical Report 1547, Cédric, 2008.

[8] Évelyne Contejean, Claude Marché, Ana Paula Tomás, and Xavier Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325–363, 2005.

[9] Pierre Courtieu, Julien Forest, and Xavier Urbain. Certifying a Termination Criterion Based on Graphs, Without Graphs. In César Muñoz and Otmane Ait Mohamed, editors, *21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*, volume 5170 of *Lecture Notes in Computer Science*, pages 183–198, Montréal, Canada, August 2008. Springer-Verlag.

[10] Pierre Courtieu, Gladys Gbedo, and Olivier Pons. Improved Matrix Interpretation. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *36th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2010)*, volume 5901 of *Lecture Notes in Computer Science*, pages 283–295, Špindlerův Mlýn, Czech Republic, January 2010. Springer-Verlag.

[11] Carsten Fuhs, Aart Middeldorp, Peter Schneider-Kamp, and Harald Zankl. SAT solving for termination analysis with polynomial interpretations. In *SAT 07*, volume 4501 of *Lecture Notes in Computer Science*, pages 340–354. Springer-Verlag, 2007.

[12] 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.

[13] 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.

[14] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

[15] 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.

[16] Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean Termination Tool 2. In Ralf Treinen, editor, *20th International Conference on Rewriting Techniques and Applications (RTA 09)*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304, Brasília, Brazil, July 2009. Springer-Verlag.

[17] Claude Marché and Hans Zantema. The Termination Competition. In Franz Baader, editor, *18th International Conference on Rewriting Techniques and Applications (RTA 07)*, volume 4533 of *Lecture Notes in Computer Science*, pages 303–313, Paris, France, June 2007. Springer-Verlag.

[18] René Thiemann and Christian Sternagel. Certification of Termination Proofs using CeTa. In Tobias Nipkow and Christian Urban, editors, *22st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468, Munich, Germany, August 2009. Springer-Verlag.