Tyrolean Termination Tool 2*

Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp

Institute of Computer Science
University of Innsbruck
6020 Innsbruck, Austria

`ttt2@informatik.uibk.ac.at`

Abstract. This paper describes the second edition of the *Tyrolean Termination Tool*—a fully automatic termination analyzer for first-order term rewrite systems. The main features of this tool are its (non-)termination proving power, its speed, its flexibility due to a strategy language, and the fact that the source code of the whole project is freely available. The clean design together with a stand-alone OCaml library for term rewriting, make it a perfect starting point for other tools concerned with rewriting as well as experimental implementations of new termination methods.

Key words: term rewriting, termination, automation

1 Introduction

Termination of term rewrite systems (TRSs) is an undecidable property. Nevertheless a vast number of methods have been developed to determine termination, many of which are suitable for implementation. This paper summarizes the main design issues, features, and successes of the Tyrolean Termination Tool 2 (TTT2 for short), the completely redesigned successor of the award winning[1] Tyrolean Termination Tool (TTT) [10]. TTT2 is a tool for automatically proving termination of TRSs, based on the dependency pair framework [7, 8, 10, 22]. It incorporates several novel methods like increasing interpretations, a modular match-bound technique, uncurrying, and outermost loops, which are not (yet) available in other termination provers. It produces readable output and has a simple web interface. Precompiled binaries, sources and documentation of TTT2 are available at

`http://cl-informatik.uibk.ac.at/software/ttt2/`

In contrast to its predecessor, TTT2 is open source; published under terms of the GNU Lesser General Public License. This work refers to version 1.0 of the tool.

The remainder of the paper is organized as follows. In the next section we describe how TTT2 can be used from the command line and sketch its web interface.

* This research is supported by FWF (Austrian Science Fund) project P18763.

[1] Best paper award, RTA 2003.

Section 3 explains the strategy language of $\mathsf{T_TT_2}$, which gives the user full control over the implemented termination methods. Some of the available termination techniques are listed in Section 4. In Section 5 we discuss the performance of our tool in light of the latest issue of the termination competition before addressing ongoing and future work in Section 6. We conclude in Section 7.

2 Design

The tool is written in OCaml[2] and consists of about 30,000 lines of code. Approximately 13% are dedicated to provide some general useful functions and data structures. Another 24% are used to implement the rewriting library which deals with terms and rules. The biggest fragment—about 49%—is used to implement termination methods and the strategy language. The rest (about 14%) is concerned with input and output. Since our tool provides several techniques that modify a termination problem by transforming it into different problem domains, $\mathsf{T_TT_2}$ interfaces the SAT solver MiniSat [2] and the SMT solver Yices [1]. For interfacing C code the third party contribution CamlIDL[3] is needed. The use of monads to implement the strategy language and several other parts of the tool, allow a clean and abstract treatment of the internal prover state in a purely functional way. Additionally, monads facilitate changes (like the integration of a new termination method).

Besides the actual termination prover, we provide the following libraries:

- `util` extends the functionality of several modules from the standard OCaml library. Furthermore modules for graph manipulation, advanced process and timer handling, as well as monads are included.
- `parsec` is an OCaml port of the Haskell parsec[4] library, i.e., the implementation of a functional combinator parser library.
- `rewriting` provides types and functions dealing with terms, substitutions, contexts, TRSs, etc. The functionality is not only aimed at termination, e.g., the computation of overlaps and normal forms is also supported.
- `logic` provides an OCaml interface that abstracts over the two constraint solvers MiniSat and Yices. To this end arithmetical formulas are encoded in an intermediate datatype. When solving the constraints the user specifies the back-end. In the case of MiniSat, additional information (how many bits are used to represent numbers and intermediate results) can be provided. Afterwards the propositional formula is transformed into conjunctive normal form by a satisfiability-preserving transformation [19]. Yices, on the other hand, does neither require the number of bits as a parameter nor the transformation due to built-in support for *linear* arithmetic and formulas not in conjunctive normal form.
- `processors` collects the numerous (non-)termination methods.
- `ttt2` contains the strategy language and connects the preceding libraries.

[2] `http://caml.inria.fr/`
[3] `http://caml.inria.fr/pub/old_caml_site/camlidl/`
[4] `http://legacy.cs.uu.nl/daan/parsec.html`

2.1 Command Line Interface

In order to run TTT2 from the command line, the user can either download the source code from the TTT2 web page and install it following the installation guidelines or alternatively download the binary of the latest version of TTT2. After a successful installation, TTT2 can be started via the command

```
./ttt2 [options] <file> [timeout]
```

where `[options]` denotes a list of command line options, `<file>` specifies the name of the file containing the TRS of which termination should be proved, and `[timeout]`—a floating point number—defines the time limit for proving termination of the given TRS. The TRS must adhere to the termination problem database format.[5] The timeout is optional. To get a complete list of the command line options of TTT2 either read the documentation provided on the tool's homepage or execute the command `./ttt2 --help`.

2.2 Web Interface

The web interface of TTT2 allows the user to play around with some termination methods and an automatic strategy. The design is intentionally simple to abstract from the challenging task to provide a fast and powerful automatic strategy.

3 The Strategy Language

As mentioned in the introduction, TTT2 is designed according to the dependency pair framework which ensures that all methods are implemented in a modular way. In order to combine these methods in a flexible manner, TTT2 provides a *strategy language*. In the following the most important constructs of this language are explained. For further information please consult the online documentation.

3.1 Syntax

The operators provided by the strategy language can be divided into three classes: *combinators*, *iterators*, and *specifiers*. Combinators are used to combine two strategies whereas iterators are used to repeat a given strategy a designated number of times. In contrast, specifiers are used to control the behavior of strategies. The most common combinators are the infixes '`;`', '`|`', and '`||`'. The most common iterators are the postfixes '`?`', '`+`', and '`*`'. The most common specifier is '$\texttt{[}f\texttt{]}$' (also written postfix), where f denotes some floating point number. In order to obtain a well-formed strategy s, these operators have to be combined according to the grammar

$$s ::= m \mid \texttt{(}s\texttt{)} \mid s\texttt{;}s \mid s\texttt{|}s \mid s\texttt{||}s \mid s\texttt{?} \mid s\texttt{+} \mid s\texttt{*} \mid s\texttt{[}f\texttt{]}$$

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

where m denotes any available method of $\mathsf{T_TT_2}$ (possibly followed by some flags). In order to avoid unnecessary parentheses, the following precedence is used: `?`, `+`, `*`, `[`f`]` $>$ `;` $>$ `|`, `||`.

3.2 Semantics

In the remainder of this section we use the notion *termination problem* to denote a TRS, a dependency pair problem (DP problem), or a relative termination problem. We call a termination problem *terminating* if the underlying TRS (DP problem, relative termination problem) is terminating (finite, relative terminating). A strategy works on a termination problem. Whenever $\mathsf{T_TT_2}$ executes a strategy, internally, a so called proof object is constructed which represents the actual termination proof. Depending on the shape of the resulting proof object after applying a strategy s, we say that *s succeeded* or *s failed*.

This should not be confused with the possible answers of the prover: `YES`, `NO`, and `MAYBE`. Here `YES` means that termination could be proved, `NO` indicates a successful non-termination proof, and `MAYBE` refers to the case when termination could neither be proved nor disproved. On success of a strategy s it depends on the internal proof object whether the final answer is `YES` or `NO`. On failure, the answer always is `MAYBE`. Based on the two possibilities success or failure, the semantics of the strategy operators is as follows.

The combinator '`;`' denotes sequential composition. Given two strategies s and s' together with a termination problem P, s`;`s' first tries to apply s to P. If this fails, then also s`;`s' fails, otherwise s' is applied to the resulting termination problem, i.e., the strategy s`;`s' fails, whenever one of s and s' fails. The combinator '`|`' denotes choice. Different from sequential composition, the choice s`|`s' succeeds whenever at least one of s or s' succeeds. More precisely, given the strategy s`|`s', $\mathsf{T_TT_2}$ first tries to apply s to P. If this succeeds, its result is the result of s`|`s', otherwise s' is applied to P. The combinator '`||`' is quite similar to the choice combinator and denotes parallel execution. That means given the strategy s`||`s', $\mathsf{T_TT_2}$ runs s and s' in parallel on the termination problem P. As soon as at least one of s and s' succeeds, the resulting termination problem is returned. This can be seen as a kind of non-deterministic choice, since on simultaneous success of both s and s', it is more or less arbitrary whose result is taken.

Example 1. Consider the following strategy:

```
dp;edg;sccs;(bounds -dp || (matrix -wm | kbo -af))
```

In order to prove termination of a TRS $\mathcal{R}$ using this strategy, $\mathsf{T_TT_2}$ first computes the dependency pairs $\mathcal{P}$ of $\mathcal{R}$ using the `dp` processor (thereby transforming the initially supplied TRS into a DP problem). After that the estimated dependency graph and the strongly connected components of the DP problem $(\mathcal{P}, \mathcal{R})$ are computed, resulting in a set of DP problems $\{(\mathcal{P}_1, \mathcal{R}), \ldots, (\mathcal{P}_n, \mathcal{R})\}$. Finally, to conclude that the DP problem $(\mathcal{P}, \mathcal{R})$ is finite and hence that the TRS $\mathcal{R}$ is terminating, $\mathsf{T_TT_2}$ tries to prove finiteness of each DP problem $(\mathcal{P}_i, \mathcal{R})$ with

$1 \leqslant i \leqslant n$ by running the match-bound technique and a combination of the matrix method and the Knuth-Bendix order in parallel. Note that the substrategy `matrix -wm | kbo -af` first applies the matrix method to a given termination problem and on failure, applies the Knuth-Bendix order. Here the flag `-wm` indicates that weakly monotone interpretations are used and the flag `-af` specifies that argument filterings should be considered when computing the ordering.

Next we describe the iterators '`?`', '`+`', and '`*`'. The strategy s`?` tries to apply the strategy s to a termination problem P. On success its result is returned, otherwise P is returned unmodified, i.e., s`?` applies s once or not at all to P and always succeeds. The iterators '`+`' and '`*`' are used to apply s recursively to P until P cannot be modified any more. The difference between '`+`' and '`*`' is that s`*` always succeeds whereas s`+` only succeeds if it can prove or disprove termination of P. In other words, s`*` is used to *simplify* problems, since it applies s until no further progress can be achieved and then returns the latest problem. In contrast '`+`' requires the proof attempt to be completed.

Example 2. We extend the strategy of the previous example by adding the iterator '`+`' and two new methods:

```
uncurry?;poly -ib 2 -ob 4*;
 dp;edg;(sccs;(bounds -dp || (matrix -wm | kbo -af)))+
```

To prove termination of a TRS $\mathcal{R}$, $\mathsf{T_TT_2}$ performs the following steps. At first uncurrying is applied. Since this method works only for applicative TRSs, the iterator '`?`' is added in order to avoid that the whole strategy fails if $\mathcal{R}$ is not an applicative system. After that polynomial interpretations with two input bits (coefficients) and four output bits (intermediate results) are used to simplify the given TRS. (Restricting the values for intermediate computations results in efficiency gains.) The iterator '`*`' ensures that a maximal number of rewrite rules is removed by applying the method as often as possible. Finally, after the computation of the dependency pairs and the estimated dependency graph, $\mathsf{T_TT_2}$ tries to prove finiteness of the given DP problems, by applying the strategy `sccs;(bounds -dp || (matrix -wm | kbo -af))` recursively.

At last we explain the specifier '`[`f`]`' which denotes timed execution. Given a strategy s and a timeout f, s`[`f`]` tries to modify a given termination problem P for at most f seconds. If s does not succeed or fail within f seconds (wall clock time), s`[`f`]` fails. Otherwise s`[`f`]` succeeds and returns the termination problem that remains after applying s to P.

Example 3. To ensure that the strategy of the previous example is executed for at most 5 seconds we add the specifier '`[5]`'. In addition we limit the time spend by the match-bound technique to 1 second.

```
(uncurry?;poly -ib 2 -ob 4*;
 dp;edg;(sccs;(bounds -dp[1] || (matrix -wm | kbo -af)))+)[5]
```

Using this strategy, $\mathsf{T_TT_2}$ has at most 5 seconds to prove termination of a given TRS $\mathcal{R}$ and in each iteration 1 second is available to simplify termination problems using the match-bound technique. If the 5 seconds expire, the execution is aborted immediately.

3.3 Specification and Configuration

In order to call $\mathsf{T_TT_2}$ with a certain strategy, the flag `--strategy` (or alternatively the short form `-s`) has to be set. For convenience it is possible to call $\mathsf{T_TT_2}$ without specifying any strategy. In this case a predefined strategy is used (for details execute `./ttt2 --help`). Note that the user is responsible for ensuring soundness of the strategy, e.g., applying the processors in correct order.

Example 4. To call $\mathsf{T_TT_2}$ with the strategy of Example 3, the following command is used: `./ttt2 -s '(uncurry?;poly -ib 2 -ob 4*; ...)[5]' <file>`. Alternatively, one could also remove the outermost time limit of the strategy and pass it as an argument to $\mathsf{T_TT_2}$. In that case the command looks as follows: `./ttt2 -s '(uncurry?;poly -ib 2 -ob 4*; ...)' <file> 5`.

Since strategies can get quite complex (e.g., the strategy used in the November 2008 termination competition consists of about 100 lines), $\mathsf{T_TT_2}$ provides the opportunity to specify a configuration file. This allows to abbreviate and connect different strategies. By convention strategy abbreviations are written in capital letters. To tell $\mathsf{T_TT_2}$ which configuration file should be used, the flag `--conf` (or the short form `-c`) followed by the file name has to be set.

Example 5. Consider the strategy of Example 3. In order to call $\mathsf{T_TT_2}$ with this strategy we write a configuration file `ttt2.conf` containing the following lines:

```
[Abbreviations]
PRE = uncurry?;poly -ib 2 -ob 4*
PARALLEL = (bounds -dp[1] || (matrix -wm | kbo -af))
AUTO = (PRE;dp;edg;(sccs;PARALLEL)+)[5]
```

It is important to note that abbreviations are not implicitly surrounded by parentheses since this allows more freedom in abbreviating expressions. To tell $\mathsf{T_TT_2}$ that the strategy `AUTO` of the configuration file `ttt2.conf` should be used the following flags have to be specified: `./ttt2 -c ttt2.conf -s AUTO <file>`.

4 A Selection of Implemented Techniques

In this section some characteristic methods of $\mathsf{T_TT_2}$ are presented.

Bounds. $\mathsf{T_TT_2}$ provides the match-bound technique [5] which uses tree automata techniques to prove termination of a TRS on a particular language (in general the set of all ground terms). To increase the applicability of the match-bound technique, $\mathsf{T_TT_2}$ was the first tool that incorporated it—in a fully modular way—into the dependency pair framework [14]. Moreover, match-bounds can be used to prove complexity results. It is well-known that match-bounds imply linear derivational complexity for non-duplicating systems [5].

KBO. TTT2 employs the most sophisticated implementations of the Knuth-Bendix ordering. The proof obligations are formulated as a propositional formula (set of pseudo boolean constraints, linear arithmetic constraints) [24] and then solved by MiniSat (MiniSat+,Yices).

Loops. Besides techniques from [18], for string rewrite systems (SRSs) loops are searched with the help of SAT solving. After fixing parameters such as the maximal length of words and the maximal length of the non-terminating sequence, loops are encoded in propositional logic [25]. Additionally, based on the approach from [18], a novel idea [23] allows to check whether a given loop also is an outermost loop, i.e., a loop under the outermost reduction strategy.

Matrices. TTT2 implements matrix [3, 12] and arctic [13] interpretations. Our tool uses higher dimensions than competitors which sometimes results in very short and elegant termination proofs. A direct termination proof by arctic matrices yields linear derivational complexity and direct matrix interpretations (of triangular shape) [15] give polynomial upper bounds.

Polynomials. Apart from polynomial interpretations over different carriers (natural numbers, integers, rationals), additional power is achieved by allowing approximations of minimum and maximum operations [4]. Furthermore, techniques from [26] allow an increase in the constant part of the interpretation for some rules.

Root-Labeling. TTT2 was the first tool that incorporated root-labeling within the dependency pair framework [21]. As a result, in 2007 it was the first automated tool that could prove termination of an SRS with non-primitive recursive derivation length (`Zantema/z090`). Since root-labeling preserves derivational complexity it is a viable transformation for proofs of complexity.

Uncurrying. TTT2 incorporates uncurrying for non-proper systems [9] similar to its predecessor TTT. Furthermore it integrates the method within the dependency pair framework [11]. This makes it a very strong tool on the subclass of applicative systems. Due to the fact that reductions in the uncurried system are strictly longer compared to the original system, upper bounds for complexity considerations are not affected by this transformation.

5 TTT2 in Action

It goes without saying that TTT2 is not the only tool for proving termination of rewrite systems. Since 2004 some of these automated termination analyzers compete against each other in regular competitions.[6] In the following paragraphs we compare our tool with some of the other systems that participated in the latest editions of the international termination competition.[7] TTT2 participated in three categories with the aim to show its flexibility and speed.

[6] `http://termination-portal.org/wiki/Termination_Competition`
[7] `http://termcomp.uibk.ac.at/`

SRS Standard. TTT2 won this category in front of AProVE [6] and Jambox.[8] The main reason was that we used matrix and arctic interpretations of higher dimensions than AProVE and Jambox. Proofs were found for the systems `Trafo/un02`, `Trafo/un15`, `Trafo/un17`, and the randomly generated `Waldmann07b/size-12-alpha-3-num-469` which could not be handled by any other tool (also not in previous competitions).

TRS Standard. TTT2 finished second behind AProVE but in front of Jambox. The main emphasis was put on speed. TTT2 could (dis)prove termination of 970 TRSs out of 1391 TRSs in less than ten minutes. That means, it could handle 79% of the systems AProVE could answer but in just 10% of the time.

TRS Outermost. TTT2 could solve twice as much systems as each of the three competitors. While all other tools employed transformations that allowed to use methods designed for full termination, TTT2 integrated a direct approach for finding loops under a specific strategy [23].

TTT2 is not only successful on its own. Two derivatives of TTT2 were involved in other categories of the competition, namely CaT[9] (which was developed by the authors) and TCT [16] (an independent tool, built on top of the basic components of TTT2). TCT was the first tool dedicated to proving complexity certificates. The aim of CaT was just to show how helpful it is to start from the basis of a well-designed termination prover; the additional implementation effort took a single day. CaT won both categories (*Derivational Complexity – Full Rewriting* and *Derivational Complexity – Innermost Rewriting*) in which it participated. Another tool that builds on TTT2 is MKBTT [20], which implements multi-completion using external termination provers. Experimental results revealed that due to thousands of calls to the external prover, a fast one is preferable over a powerful one. This observation inspired our configuration of TTT2 for the *TRS Standard* category in the November 2008 competition where we used less than 10% of the allowed time, to show how many problems could be solved in a strongly limited amount of time.

6 Future Work

Two main goals for the near future are: the improvement of the output produced by TTT2 and the formalization and certification of (non-)termination methods. Concerning the output we plan to transform the internal proof objects into XML. Afterwards it should be possible to convert this XML format into either human readable output or a proof format suitable for automatic certification. For the second goal a parallel project addresses the formalization of rewriting (IsaFoR, Isabelle Formalization of Rewriting) in the theorem prover Isabelle/HOL [17].

[8] `http://joerg.endrullis.de/`

[9] `http://cl-informatik.uibk.ac.at/software/cat/`

This formalization deals with rewriting in general and (non-)termination based on the dependency pair framework in particular. In order to be usable for automated certification of proofs generated by a termination tool, we employ Isabelle's code-generation facilities to export verified Haskell code. This results in the program CeTA[10] (Certification of Termination Analysis), capable of certifying (non-)termination proofs.

7 Conclusion

In this paper we described the termination prover $\mathsf{T_TT_2}$, the successor of the well-known Tyrolean Termination Tool. We presented its strategy language, some of its characteristic methods, and we compared $\mathsf{T_TT_2}$ with other termination provers to show its flexibility and versatility. We conclude the paper by listing what we believe to be the main attractions of $\mathsf{T_TT_2}$:

- it is open source,
- it provides a strategy language which allows to configure it for all possible applications,
- it benefits from multi-core architecture due to support for parallelism,
- it is one of the fastest and most powerful termination provers, and
- it provides stand-alone libraries for parsing, rewriting, and logic.

Acknowledgments. We thank Nao Hirokawa for providing the sources of $\mathsf{T_TT}$ and Sarah Winkler for writing a first interface for MiniSat.

References

1. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
2. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2004. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
3. Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of rewrite systems. Journal of Automated Reasoning 40(2-3), 195–220 (2008)
4. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: Maximal termination. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 110–125. Springer, Heidelberg (2008)
5. Geser, A., Hofbauer, D., Waldmann, J., Zantema, H.: On tree automata that certify termination of left-linear term rewriting systems. Information and Computation 205(4), 512–534 (2007)
6. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)

[10] `http://cl-informatik.uibk.ac.at/software/ceta/`

7. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework: Combining techniques for automated termination proofs. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 301–331. Springer, Heidelberg (2004)
8. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. Journal of Automated Reasoning 37(3), 155–203 (2006)
9. Hirokawa, N., Middeldorp, A.: Uncurrying for termination. In: Kesner, D., van Raamsdonk, F., Stehr, M.O. (eds.) HOR 2006. pp. 19–24 (2006)
10. Hirokawa, N., Middeldorp, A.: Tyrolean Termination Tool: Techniques and features. Information and Computation 205(4), 474–511 (2007)
11. Hirokawa, N., Middeldorp, A., Zankl, H.: Uncurrying for termination. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 667–681. Springer, Heidelberg (2008)
12. Hofbauer, D., Waldmann, J.: Termination of string rewriting with matrix interpretations. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 328–342. Springer, Heidelberg (2006)
13. Koprowski, A., Waldmann, J.: Arctic termination . . . below zero. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 202–216. Springer, Heidelberg (2008)
14. Korp, M., Middeldorp, A.: Match-bounds revisited. Information and Computation (2009). `doi: 10.1016/j.ic.2009.02.010`
15. Moser, G., Schnabl, A., Waldmann, J.: Complexity analysis of term rewriting based on matrix and context dependent interpretations. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2008. DROPS, vol. 1762, pp. 304–315. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Dagstuhl (2008)
16. Moser, G., Schnabl, A.: Proving quadratic derivational complexities using context dependent interpretations. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 276–290. Springer, Heidelberg (2008)
17. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. vol. 2283 of LNCS. Springer, Heidelberg (2002)
18. Payet, É.: Loop detection in term rewriting using the eliminating unfoldings. Theoretical Computer Science 403(2-3), 307–327 (2008)
19. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. Journal of Symbolic Computation 2(3), 293–304 (1986)
20. Sato, H., Winkler, S., Kurihara, M., Middeldorp, A.: Multi-completion with termination tools (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 306–312. Springer, Heidelberg (2008)
21. Sternagel, C., Middeldorp, A.: Root-labeling. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 336–350. Springer, Heidelberg (2008)
22. Thiemann, R.: The DP Framework for Proving Termination of Term Rewriting. PhD thesis, RWTH Aachen (2007). Available as technical report AIB-2007-17.
23. Thiemann, R., Sternagel, C.: Loops under strategies. In: Treinen, R. (ed.) RTA 2009. LNCS. (2009). This volume.
24. Zankl, H., Hirokawa, N., Middeldorp, A.: KBO orientability. Journal of Automated Reasoning (2009). `doi: 10.1007/s10817-009-9131-z`
25. Zankl, H., Middeldorp, A.: Nontermination of string rewriting using SAT. In: Hofbauer, D., Serebrenik, A. (eds.) WST 2007. pp. 56–59 (2007)
26. Zankl, H., Middeldorp, A.: Increasing interpretations. Annals of Mathematics and Artificial Intelligence (2009). To appear.