The Termination Prover AProVE

Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke

LuFG Informatik II, RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany
{giesl—thiemann}@informatik.rwth-aachen.de
{nowonder—spf}@i2.informatik.rwth-aachen.de

Abstract. We describe the system AProVE, an automated prover to verify (innermost) termination of term rewrite systems (TRSs), functional programs, and logic programs. For this system, we have developed and implemented efficient algorithms based on classical simplification orders (recursive path orders with status, Knuth-Bendix orders, polynomial orders), dependency pairs, and the size-change principle. In AProVE, termination proofs can be performed with a user-friendly graphical interface and the system is currently among the most powerful termination provers available.

1 Introduction

The system AProVE (Automated Program Verification Environment) can be used for automated termination and innermost termination proofs of (conditional) term rewrite systems, as well as for termination proofs of functional and logic programs. AProVE offers a variety of techniques for automated termination proofs: First of all, it provides efficient implementations of classical simplification orders to prove termination "directly" (such as *recursive path orders* possibly with status [7, 19], *Knuth-Bendix orders* [20], and *polynomial orders* [22]), cf. Sect. 2. To increase the power of automated termination proofs, we implemented the *dependency pair* technique [2, 15] in AProVE which allows the application of classical simplification orders to many examples where automated termination analysis would fail otherwise (Sect. 3). In contrast to most other implementations of dependency pairs, we integrated refinements such as *narrowing*, *rewriting*, and *instantiation* of dependency pairs [14] and we improved the dependency pair technique further (e.g., by integrating the generation of *argument filterings* with the computation of *usable rules*) [16] to increase both the efficiency and the power of the approach. Due to these extensions and improvements, AProVE succeeds on many examples where all other automated termination provers fail. Thus, the principles used in the implementation of AProVE are also very helpful for other tools based on dependency pairs ([1], CiME [6], TTT [18]) and we conjecture that they can also be used in other recent approaches for termination of TRSs [5, 11] which have several aspects in common with dependency pairs. Apart from direct termination proofs and dependency pairs, as a third termination technique, AProVE offers the *size-change principle* [9, 23] and it is also possible to combine this principle with dependency pairs [27] (Sect. 4). The tool is written in Java and proofs can be performed both in a fully automated or in an interactive mode via a graphical user interface (Sect. 6), as shown in Fig. 1.

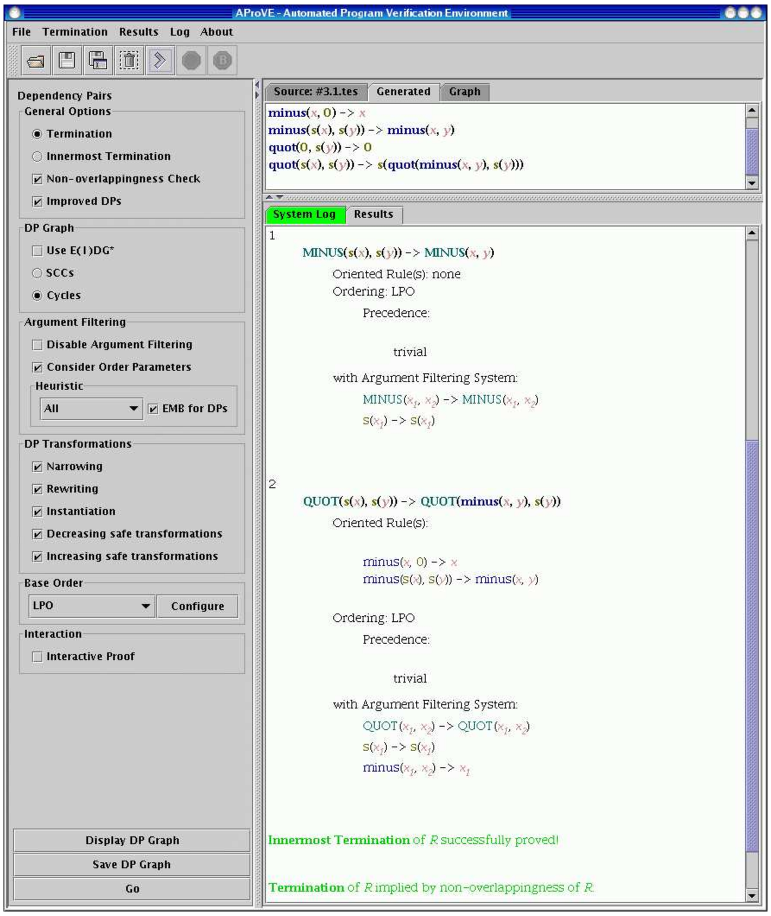

Fig. 1. Screenshot of the AProVE system

2 Direct Termination Proofs

In this section we describe the base orders available in AProVE which can be used for direct termination proofs, but also for proofs with preprocessing techniques like dependency pairs or the size-change principle.

In a direct termination proof of a TRS, the system tries to find a reduction order such that all rules of the TRS are decreasing. Currently, the following *path orders* are implemented in AProVE: the embedding order (EMB), the lexicographic path order (LPO, [19]), the LPO with status which compares subterms lexicographically according to arbitrary permutations (`LPOS`), the recursive path order which compares subterms as multisets (`RPO`, [7]), and the `RPO` with status which is a combination of `LPOS` and `RPO` (`RPOS`).

Path orders may be parameterized by a precedence on function symbols and a status which determines how the arguments of function symbols are compared. To explore the search space for these parameters, the system leaves the precedence and the status as unspecified (or "*minimal*") as possible. The user can decide whether to perform a depth-first or a breadth-first search (where in the latter case, *all* possibilities for a minimal precedence and status are computed which satisfy the current constraints). Moreover, the user can configure the path orders by deciding whether different function symbols may be equivalent according to the precedence used in the path order ("`non-strict precedence`"). It is also possible to restrict potential equivalences to certain pairs of function symbols. When attempting termination proofs with path orders in AProVE, the precedence found by the system is displayed as a graph (in case of success) and in case of failure in the breadth-first search, the system indicates the problematic constraint.

In addition to the above path orders, AProVE offers *Knuth-Bendix orders* (`KBO`, [20]) using the polynomial-time algorithm of [21]. In this algorithm, one has to compute the degenerate subsystem of a system of homogeneous linear inequalities. This is done using the technique of [10].

The last class of orders available in AProVE are *polynomial orders* [22] where every function symbol is associated with a polynomial with natural coefficients. Here, the user can specify three parameters: the degree of the polynomials, the range of the coefficients, and the search algorithm that is used to find suitable coefficients in the given range. Apart from these global options, the user can also provide individual polynomials for some function symbols manually. To prove termination afterwards, AProVE generates a set of polynomial inequalities which state that left-hand sides of rules should be greater than the corresponding right-hand sides. Using the method of partial derivation [13, 22], these inequalities are transformed into a set of inequalities only containing coefficients, but no variables anymore. At that point, a search algorithm has to determine suitable coefficients that satisfy the resulting inequalities. The user can choose between four different search algorithms: we offer brute force search, greedy search, a genetic algorithm, and a constraint-based method based on interval arithmetic.

3 Termination Proofs With Dependency Pairs

The *dependency pair* approach [2, 14–16] increases the power of automated termination analysis significantly, since it permits the application of simplification orders for non-simply terminating TRSs. Instead of comparing left- and right-hand sides of rules, in this approach one compares the left-hand sides with those subterms of right-hand sides that correspond to recursive calls. More precisely, the root symbols of the left-hand sides are called *defined symbols* and for each defined symbol f we introduce a fresh *tuple symbol* F. For each rule

$f(s_1, \ldots, s_n) \to r$ and each subterm $g(t_1, \ldots, t_m)$ of r with defined root symbol g, we build a *dependency pair* $F(s_1, \ldots, s_n) \to G(t_1, \ldots, t_m)$. In order to prove termination one now has to find a weakly monotonic order $\succ$ such that $s \succ t$ for all dependency pairs $s \to t$ and $l \succsim r$ for all rules $l \to r$. When proving innermost termination, $l \succsim r$ is only required for the usable rules of the defined symbols occurring in the dependency pairs. Here, the *usable rules* for a symbol f are the f-rules together with the usable rules for all defined symbols occurring in right-hand sides of f-rules. In AProVE, the user can select whether to use the dependency pair approach for termination or for innermost termination proofs. The system can also check whether a TRS is non-overlapping (then innermost termination already implies termination).

To search for a suitable order $\succ$, the user can select any of the base orders from Sect. 2. However, while most of these orders are *strongly* monotonic, the dependency pair approach only requires *weak* monotonicity. (For polynomial orders, a weakly monotonic variant can be obtained immediately by permitting the coefficient 0 in polynomials. But lexicographic or recursive path orders as well as Knuth-Bendix orders are always strongly monotonic.) For that reason, before searching for a suitable order, some of the arguments of the function symbols in the constraints can be eliminated using an *argument filtering* π [2]. For example, a binary function symbol f can be turned into a unary symbol by eliminating the first argument of f. Then π replaces all terms $f(t_1, t_2)$ in the constraints by $f(t_2)$. Thus, we can obtain a weakly monotonic order $\succ_\pi$ out of a strongly monotonic order $\succ$ and an argument filtering π by defining $s \succ_\pi t$ iff $\pi(s) \succ \pi(t)$. For innermost termination proofs, we developed an improvement such that the argument filtering is also used for reducing the set of constraints [16, Thm. 11].

Since there are exponentially many argument filterings, a crucial problem for any implementation of dependency pairs is to explore this search space efficiently. In AProVE, we use a depth-first algorithm [16] to determine suitable argument filterings by treating the constraints one after another. We start with the set of argument filterings possibly satisfying the first constraint. Here we use the idea of [17] to keep argument filterings as "undefined" as possible. Then this set is reduced further to those filterings which can possibly satisfy the second constraint as well. This procedure is repeated until all constraints have been investigated. By inspecting the constraints in a suitable order, already after the first constraint the set of possible argument filterings is rather small and in this way, one only inspects a small subset of all potential argument filterings. Moreover, one can also combine the search for the argument filtering with the search for the base order by choosing the option "`consider order parameters`". If the user selects this option, then the system additionally stores for each possible argument filtering a minimal set of precedences and stati as described in Sect. 2. This option is only available for path orders.

(Innermost) termination proofs with dependency pairs can be performed in a modular way by constructing an estimated (innermost) *dependency graph* and by regarding its cycles separately [2, 15]. For each cycle, only one dependency pair must be strictly decreasing, whereas all others only have to be weakly decreasing.

As shown in [17], one should not compute all cycles, but only maximal cycles (*strongly connected components (SCCs)*). The reason is that the the chosen argument filtering and base order may make several dependency pairs in an SCC strictly decreasing. In that case, subcycles of the SCC containing such a strictly decreasing dependency pair do not have to be considered anymore. So after solving the constraints for the initial SCCs, all strictly decreasing dependency pairs are removed and one now builds SCCs from the remaining dependency pairs, etc. To inspect estimated (innermost) dependency graphs, they can be displayed in a special "`Graph`"-window. In order to benefit from all refinements on modularity of dependency pairs, we developed and implemented a technique which permits the combination of recent results on modularity of C_ε-terminating TRSs [28] with arbitrary estimations of dependency graphs, cf. [16].

To increase the power of the dependency pair technique, in [2, 14, 16] three different transformation techniques were suggested which transform a dependency pair into several new pairs: *narrowing*, *rewriting*, and *instantiation*. These transformations are often crucial for the success of the proof and in general, their application can never "harm": if the termination proof succeeds without transformations, then it also succeeds when performing transformations, but not vice versa. However, the problem is when to use these transformations, since in general, they may be applicable infinitely often. AProVE automatically performs these transformations in "safe" cases where their application is guaranteed to terminate. There are two kinds of "safe" cases: despite the fact that applying transformations can never prevent a termination proof that would have been possible without transformations, these transformations may increase runtime, since they can produce a large number of similar constraints. However, those transformations which delete dependency pairs or cycles do not have a negative impact on the efficiency and are called *decreasing*. The remaining transformations are called *increasing*. The system offers two switches where the user can enable or disable both kinds of transformation. If turned on, the decreasing transformations are applied before trying to solve the constraints for a cycle. Increasing transformations are only used a limited number of times whenever a proof attempt fails, and then the proof is re-attempted again.

In addition to the fully automated mode, (innermost) termination proofs with dependency pairs can also be performed in an interactive mode. Here, the user can specify which narrowing, rewriting, and instantiation steps should be performed and for any cycle or SCC, the user can determine (parts of) the argument filtering, the base order, and the dependency pair which should be strictly decreasing. Moreover, one can immediately see the constraints resulting from such selections, such that interactive termination proofs are supported in a very comfortable way. This mode is intended for the development of new heuristics as well as for the machine-assisted proof of particularly challenging examples.

4 Termination Proofs with the Size-Change Principle

A new *size-change principle* for termination of functional programs was presented in [23] and this principle was extended to TRSs in [27]. A similar principle is also known for termination proofs of logic programs [9]. The main idea is to build a corresponding *size-change graph* from each dependency pair $F(s_1, ..., s_n) \to G(t_1, ..., t_m)$. This graph is bipartite where the nodes $1_F, \ldots, n_F$ on the left-hand side correspond to the arguments of the F-term and the nodes $1_G, \ldots, m_G$ on the right-hand side correspond to the arguments of the G-term. We draw an edge $i_F \stackrel{\succ}{\to} j_G$ iff $s_i \succ t_j$. Otherwise, there is an edge $i \stackrel{\succsim}{\to} j$ if at least $s_i \succsim t_j$ holds. Furthermore, we add a label $F \to G$ to the whole size-change graph.

We can concatenate two or more size-change graphs to a multigraph if the labels of each two consecutive size-change graphs are compatible (where the label $F \to G$ is *compatible* with $G \to H$ for all tuple symbols F and H). Each path from a left node of the first size-change graph to a right node of the last size-change graph leads to an edge in the multigraph. If there is at least one $\stackrel{\succ}{\to}$- edge on the path, then the resulting edge in the multigraph is labelled with $\succ$, otherwise it is labelled with $\succsim$. We call a multigraph G *maximal* iff the concatenation of G with G results in G again. The main theorem of the size-change principle states that termination can be concluded if each maximal multigraph contains an edge $i \stackrel{\succ}{\to} i$.

In AProVE, the technique of [27, Thm. 11] for size-change termination of TRSs is implemented, where we use the embedding order as underlying base order.[1] AProVE displays all size-change graphs as well as all maximal multigraphs (in case of success) or one critical maximal multigraph without a decreasing edge $i \stackrel{\succ}{\to} i$ (in case of failure).

AProVE also contains the new approach of [27] which combines the size-change principle with dependency pairs in order to prove (innermost) termination. This combined approach has the advantage that it often succeeds with much simpler argument filterings and base orders than the pure dependency pair approach. For each SCC $\mathcal{P}$ of the estimated (innermost) dependency graph, let $\mathcal{C}_\mathcal{P}$ be the constructors in $\mathcal{P}$ and let $\mathcal{D}_\mathcal{P}$ be a subset of the defined symbols in $\mathcal{P}$. Then the system builds the size-change graphs and the maximal multigraphs resulting from $\mathcal{P}$ using an argument filtering and the embedding order on $\mathcal{C}_\mathcal{P} \cup \mathcal{D}_\mathcal{P}$. Again, all these multigraphs must have an edge $i \stackrel{\succ}{\to} i$ and in case of success, the system displays them all. Next, the argument filtering must be extended such that all rules are weakly decreasing w.r.t. the selected base order. When proving innermost termination, instead it suffices if just the usable rules for the symbols $\mathcal{D}_\mathcal{P}$ are weakly decreasing. For reasons of efficiency, the user can impose a limit on the maximal size of $\mathcal{D}_\mathcal{P}$ and one can restrict the number of symbols in dependency pairs which may be argument-filtered.

In case of failure for some SCC, the dependency pairs are transformed by narrowing, rewriting, or instantiation and the proof attempt is re-started. If the user has selected the "`hybrid`" algorithm, then the pure dependency pair method

[1] As shown in [27], only very restricted base orders are sound in connection with the size-change principle. In addition to the results in [27], the full embedding order may be used, where $f(\ldots, x_i, \ldots) \succ x_i$ also holds for defined function symbols f.

is tried as soon as the limits for the transformations are reached. In this way, the combined dependency pair and size-change method can be used as a very fast technique which is checked first for every SCC. Only if this method fails, the ordinary dependency pair approach is used on this SCC.

5 Design of AProVE

The techniques of the previous two sections share one common property: they can be seen as *SCC processors* which transform one SCC into a set of new SCCs. The dependency pair technique generates a set of constraints for each SCC. If the constraints can be solved, then the SCC can be disregarded, while some new SCCs of subgraphs may have to be examined. The transformations "narrowing", "rewriting", and "instantiation" can also produce a set of new SCCs out of a given one. Finally, the combination of dependency pairs with the size-change principle processes the SCCs of the estimated (innermost) dependency graph one by one, too. Hence, all these termination proving algorithms work according to the following structure.

1. Compute the initial SCCs of the (estimated) innermost dependency graph.
2. While there are SCCs left and there is no failure:
   (a) Remove one SCC $\mathcal{P}$ from the set of SCCs.
   (b) Compute a new set of SCCs by processing $\mathcal{P}$ with an SCC processor.
   (c) Add the new set of SCCs to the remaining SCCs.

Thus, the termination proving techniques above are implemented in AProVE as modules which process one SCC and return a set of SCCs. Due to this modular structure, procedures for termination proofs which combine different termination techniques can easily be implemented within AProVE. One just has to configure the system by determining which SCC processors with which parameters should be used in Step 2(b). To obtain an efficient and powerful proof procedure, one should first try to use fast SCC processors which benefit from successful heuristics. In this way, SCCs that are easy to handle can be treated efficiently. Only for SCCs where these fast SCC processors fail, one should use slower but more powerful SCC processors afterwards. Examples for such termination procedures are the hybrid algorithm described in the last section or the "`meta combination`" algorithm of [16] that combines five different SCC processors. This algorithm is particularly useful if one does not want to get involved with the details of termination proving, but one wants to use AProVE in a "black box"-mode. Similar to the modular design of SCC processing, we have implemented the base orders and suitable heuristics in a modular way such that they can be combined freely. Up to now, to realize arbitrary combinations of several different SCC processors, base orders, and heuristics, one has to modify the code of the system, but we are working on a configuration language such that the user will be able to configure new termination proof procedures.

6 Running AProVE

The system AProVE accepts four different input languages: logic and (first-order) functional programs, conditional and unconditional TRSs. Functional programs are translated into conditional TRSs regarding the special semantics of the pre-defined conditional "if". Logic programs are translated into conditional TRSs, too, using the method of [4, 12]. Conditional TRSs are transformed further into unconditional TRSs according to the techniques of [14, 24] to prove their (innermost) quasi-decreasingness. For logic programs, these transformations correspond to the approach of the termination prover TALP [25].

When performing termination proofs, a "`system log`" can be inspected to examine all (possibly failed) proof attempts. The results of the termination proof are displayed in `html`-format and can be stored in `html`- or LaTeX-format. Any termination proof attempt may of course be interrupted by a stop-button. Instead of running the system on only one term rewrite system or program, it is also possible to run it on collections and directories of examples in a `batch mode`. In this case, apart from the information on the termination proofs of the separate examples, the "`result`" also contains statistics on the success and the runtime for the examples in the collection.

While the user can select between several different heuristics for performing termination proofs, we also provided the meta combination algorithm from the previous section which applies selected heuristics in a suitable way [16]. For example, when running the meta combination algorithm on the example collections of [3, 8, 26] (108 TRSs for termination, 151 TRSs for innermost termination), AProVE succeeded on 96.6 % of the innermost termination examples (including all of [3]) and on 93.5 % of the examples for termination. The automated proof for the whole collection took 80 seconds for innermost termination and 27 seconds for termination. These results also illustrate the power and efficiency of AProVE. For more details and to download the system, the reader is referred to the AProVE web-site `http://www-i2.informatik.rwth-aachen.de/AProVE`.

References

1. T. Arts. System description: The dependency pair method. In *Proc. 11th RTA*, LNCS 1833, pages 261–264, 2000.
2. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
3. T. Arts and J. Giesl. A collection of examples for termination of term rewriting using dependency pairs. Technical Report AIB-2001-09, RWTH Aachen, Germany, 2001. Available from `http://aib.informatik.rwth-aachen.de`.
4. T. Arts and H. Zantema. Termination of logic programs using semantic unification. In *Proc. 5th LOPSTR*, LNCS 1048, pages 219–233, 1996.
5. C. Borralleras, M. Ferreira, and A. Rubio. Complete monotonic semantic path orderings. In *Proc. 17th CADE*, LNAI 1831, pages 346–364, 2000.
6. E. Contejean, C. Marché, B. Monate, and X. Urbain. CiME version 2, 2000. Available from `http://cime.lri.fr`.

7. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.
8. N. Dershowitz. 33 examples of termination. In *Proc. French Spring School of Theoretical Computer Science*, LNCS 909, pages 16–26, 1995.
9. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1,2):117–156, 2001.
10. J. Dick, J. Kalmus, and U. Martin. Automating the Knuth-Bendix ordering. *Acta Informatica*, 28:95–119, 1990.
11. O. Fissore, I. Gnaedig, and H. Kirchner. Cariboo: An induction based proof tool for termination with strategies. In *Proc. 4th PPDP*, pages 62–73. ACM, 2002.
12. H. Ganzinger and U. Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. In *Proc. 3rd CTRS*, LNCS 656, pages 113–127, 1993.
13. J. Giesl. Generating polynomial orderings for termination proofs. In *Proc. 6th RTA*, LNCS 914, pages 426–431, 1995.
14. J. Giesl and T. Arts. Verification of Erlang processes by dependency pairs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1,2):39–72, 2001.
15. J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(1):21–58, 2002.
16. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Improving dependency pairs. In *Proc. 10th LPAR*, LNAI 2850, pages 165–179, 2003.
17. N. Hirokawa and A. Middeldorp. Automating the dependency pair method. In *Proc. 19th CADE*, LNAI 2741, 2003.
18. N. Hirokawa and A. Middeldorp. Tsukuba termination tool. In *Proc. 14th RTA*, LNCS 2706, pages 311–320, 2003.
19. S. Kamin and J. J. Lévy. Two generalizations of the recursive path ordering. Unpublished Manuscript, University of Illinois, IL, USA, 1980.
20. D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon, 1970.
21. K. Korovin and A. Voronkov. Verifying orientability of rewrite rules using the Knuth-Bendix order. In *Proc. 10th RTA*, LNCS 2051, pages 137–153, 2001.
22. D. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.
23. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proc. POPL '01*, pages 81–92, 2001.
24. E. Ohlebusch. Termination of logic programs: Transformational approaches revisited. *Applicable Algebra in Engineering, Communication and Computing*, 12(1,2):73–116, 2001.
25. E. Ohlebusch, C. Claves, and C. Marché. TALP: A tool for the termination analysis of logic programs. In *Proc. 11th RTA*, LNCS 1833, pages 270–273, 2000.
26. J. Steinbach. Automatic termination proofs with transformation orderings. In *Proc. 6th RTA*, LNCS 914, pages 11–25, 1995. Full version appeared as Technical Report SR-92-23, Universität Kaiserslautern, Germany.
27. R. Thiemann and J. Giesl. Size-change termination for term rewriting. In *Proc. 14th RTA*, volume LNCS 2706, pages 264–278, 2003.
28. X. Urbain. Automated incremental termination proofs for hierarchically defined term rewriting systems. In *Proc. IJCAR 2001*, LNAI 2083, pages 485–498, 2001.