Representing Strategies for the Connection Calculus in Rewriting Logic

Bjarne Holen[1], Einar Broch Johnsen[1], and Arild Waaler[2,1]

[1] Department of Informatics, University of Oslo, Norway
[2] Finnmark College, Norway
{bjarneh,einarj,arild}@ifi.uio.no

Abstract. Rewriting logic can be used to prototype systems for automated deduction. In this paper, we illustrate how this approach allows experiments with deduction strategies in a flexible and conceptually satisfying way. This is achieved by exploiting the reflective property of rewriting logic. By specifying a theorem prover in this way one quickly obtains a readable, reliable and reasonably efficient system which can be used both as a platform for tactic experiments and as a basis for an optimized implementation. The approach is illustrated by specifying a calculus for the connection method in rewriting logic which clearly separates rules from tactics.

Keywords: Rewriting logic, reflection, meta-programming, Maude, connection method, propositional logic.

1 Introduction

The aim of this paper is to show how rewriting logic [10] may be used to design a rapid prototype of a logical system and use this as a framework for experimenting with search strategies. We shall, more precisely, provide a simple rewriting logic specification of the connection calculus [1, 3] and invoke the Maude interpreter [7, 9, 13] for the purpose of executing the specification. Although we have implemented a system for a full first-order language without equality [8], it is sufficient for the purposes of this paper to focus on a propositional language.

Rewriting logic is ideal for the specification of rule-based logical systems. If efficiency is not an issue, specifying deduction rules as rewrite rules is in fact sufficient for obtaining a search engine; the breadth-first rewrite strategy of Maude will systematically traverse the search space spanned by the rules. If, on the other hand, efficiency is at stake, more sophisticated tactics than breadth-first should be adopted. In rewriting logic this can be achieved through specifications at the meta-level.

The theoretical basis for representing tactics at the meta-level resides in the reflective property of rewriting logic, by means of which we can construct meta-programs that control the execution of other rewrite theories. The rewrite theory

simulating execution of other rewrite theories is often referred to as the Universal Rewrite Theory. More precisely, there exists in rewriting logic a finitely presented rewrite theory $\mathcal{U}$ (the universal theory) which allows any finitely presented rewrite theory $\mathcal{R}$ to be specified as a term $\overline{\mathcal{R}}$ such that $\mathcal{R} \vdash t \longrightarrow t'$ if and only if $\mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle$. Moreover, the reflective property gives us the ability to simulate execution using the Universal Rewrite Theory. If a rewrite theory specifies a logical calculus this is very helpful as it in principle provides full freedom at each step in the deduction process. To see how the Universal Rewrite Theory can be implemented in Maude, see [5, 6].

We shall compare our executable specifications with the connection-based Prolog implementation leanCoP of Otten and Bibel [4], an efficient implementation in the family of lean provers (like leanTAP [2], ileanTAP [11], and linTAP [12]). Since leanCoP's search strategy is intimately tied to the internal control mechanism of Prolog, it illustrates both the strength and the shortcomings of Prolog as an implementation language. Exploiting the internal language constructs of Prolog gives a remarkably compact code; it is, on the other hand, hard to experiment with tactics without fundamentally changing the underlying structure of the program. The main reason for this is that the formulation of logical calculus and strategic choices are intermixed.

A novel feature of the rewriting logic specification is that the specification itself is executable, we also obtain readable and highly reliable code. Even though the Maude platform is reasonably efficient, it is not competitive to, say C; however, if we want to optimize speed we may still benefit considerably from tactic experiments in a Maude prototype prior to an implementation in an imperative language.

2 Constructing the Calculus

A matrix representation of a formula is obtained by converting the formula into DNF, where each column in the matrix is a DNF clause. A path consists of one literal from each DNF clause (column in the matrix). The formula is valid if it is equivalent to a conjunction of tautologies. This is the case if all paths through the matrix contain connections; i.e., all paths contain a literal and its negation.

A column in the matrix will be specified as a multiset of literals, a matrix will consist of a multiset of clauses. In rewriting logic we construct multisets by constructing commutative associative lists. Atoms will be represented by lowercase letters for readability. The sort `Lit` of literals is defined by constructors:

```
ops a b c d e f ..... x y z : -> Lit [ctor] .

op -_ : Lit -> Lit [ctor] .   *** a negated atom is also a literal
```

Remark that to get a dynamic representation we can use an enumerating scheme instead, using a constructor defined as `op p : Nat -> Lit [ctor]`. A clause is a multiset of literals inside brackets; matrices are represented as multisets of clauses inside brackets:

```
subsort Lit < LitSet .
op none : -> LitSet [ctor] .
op _,_  : LitSet LitSet -> LitSet [ctor assoc comm id: none] .
op [_]  : LitSet -> Clause [ctor] .

subsort Clause < ClauseSet .
op none : -> ClauseSet [ctor] .
op _,_  : ClauseSet ClauseSet -> ClauseSet [ctor assoc comm id: none] .
op [_]  : ClauseSet -> Matrix [ctor] .
```

Below, the matrix to the left will get the representation to the right.

$$\left[\begin{bmatrix}\neg Q\\ P\end{bmatrix}\begin{bmatrix}\neg R\\ S\end{bmatrix}\begin{bmatrix}\neg S\\ Q\end{bmatrix}[\neg P]\right] \qquad \texttt{[[- q, p], [- r, s], [- s, q], [- p]]}$$

Like LeanCoP [4], the calculus operates on a structure which contains three elements: the active path, the active clause, and the remaining matrix. The active path consists of a set of literals, represented by a term of the sort `LitSet`. The active clause is a term of the sort `Clause` and the remaining matrix is a term of the sort `Matrix`. The structures that our calculus operates on will be terms of the sort `SearchState`, which hold terms for active path, active clause, and the remaining matrix:

```
op <_;_;_> : LitSet Clause Matrix -> SearchState [ctor] .
```

Some of our deductive rules will split the `SearchState` element into two other `SearchState` elements, so we need to be able to represent more than one of these structures. We form lists of them, called `SearchStateList`s.

```
subsort SearchState < SearchStateList .

op nil : -> SearchStateList [ctor] .
op __  : SearchStateList SearchStateList -> SearchStateList [ctor assoc id: nil] .
```

If a connection is located, we are on the right track. Should we, on the other hand, encounter a path through the matrix which contains no connections, this path represents a countermodel and we can terminate the search. A `SearchState` element with a connection will have an empty active clause, while a `SearchState` element with an empty remaining matrix does not contain any connections. The sort which represents paths with connections and paths without connections will be a subsort of `SearchState`, and contains two constants.

```
sort TerminationValue .
subsort TerminationValue < SearchState .

ops  valid notvalid : -> TerminationValue [ctor] .
```

There are eight rules in the calculus, the first of these will simply select a clause from the matrix and make this our active clause.

```
rl [init]:
  < none ; emptyClause ; [CLA, CLASET] >
=>--------------------------------------
  < none ; CLA ; [CLASET] > .
```

The `CLA` variable is a variable of the sort `Clause`, `CLASET` is a variable of the sort `ClauseSet`. The constant `emptyClause` represents an empty clause (we have not selected an active clause).

```
rl [negLitInPath]:
  < LITSET1, - LIT ; [LIT, LITSET2] ; M >
=>-------------------------------------
  < LITSET1, - LIT ; [LITSET2] ; M > .
```

This rule locates a connection between an element inside the active clause and the active path. We cut off all paths which contain connections by eliminating the element `LIT` from our active clause. All paths through the matrix containing the active path (`LITSET1, - LIT`) which also contain the element `LIT` inside the active clause, will have a connection. We eliminate these paths by removing the element `LIT` from the active clause. This rule has a dual rule where the negated literal is inside the active clause instead of the active path, but is otherwise the same rule. This is how we prune the exponential search space using the connection method: we eliminate paths already known to have a connection.

```
rl [negLitInMatrix]:
  < PATH ; [LIT, LITSET1] ; [[- LIT, LITSET2], CLASET] >
=>-----------------------------------------------------
  < PATH, LIT ; [LITSET2] ; [CLASET] >
  < PATH      ; [LITSET1] ; [[- LIT, LITSET2], CLASET] > .
```

This rule is a bit more complex than the previous one, but it prunes the search space in the same manner. All paths that enter the active clause will contain the element `LIT` or some element in `LITSET1` (there really is no other option). All paths containing the element `LIT` inside the active clause which also contain the element `- LIT` in the remaining matrix, will have connections. The first `SearchState` element states this. We add `LIT` to our active path, then make `[- LIT, LITSET2]` from our remaining matrix our new active clause, but we

remove the element `- LIT` since these paths will have connections (since the active path contains `LIT`). The second `SearchState` element represents all paths not containing the element `LIT` inside the active clause of our first `SearchState` element. This rule also has its own dual rule where the negated literal is located inside the active clause instead of the remaining matrix, but is otherwise identical.

If we fail to locate connections between elements inside the active clause and the active path (the second deductive rule), and we also fail to locate connections between elements inside the active clause and the remaining matrix, our hope of pruning the search space is lost. It is then necessary to extend the active path, which is done by applying the fourth deductive rule.

```
rl [extendPath]:
  < PATH ; [LIT, LITSET] ; [CL, CLASET] >
=>--------------------------------------
  < PATH, LIT ; CL       ; [CLASET] >
  < PATH      ; [LITSET] ; [CL, CLASET] > .
```

All we need now to complete the calculus is to locate the axioms or paths with and without connections, which is done by the following two structural rules.

```
rl [removeConnectedPaths]:
  < PATH ; [none] ; M >
=>--------------------
        valid  .

rl [counterModel]:
  < PATH ; [LIT, LITSET] ; [none] >
=>--------------------------------
             notvalid   .
```

The specification of the calculus is now complete. However, note that unconstrained use of `extendPath` leads to inconsistency, since the rule can be applied to a `SearchState` element as long as the remaining matrix and active clause are inhabited. If we repeatedly apply this rule to a `SearchState` element we will end up with a `SearchState` element somewhere in the `SearchStateList` on the form `< PATH ; [LITSET] ; [none] >`, which represents a countermodel. If soundness shall not be compromised we have to look for connections (apply the rules `negLitInPath` and `negLitInMatrix`) before we apply the rule `extendPath`. This could have been solved using conditional rules (implementing side conditions on rules). However, that reduces efficiency, since the side conditions would have to be tested at each deductive step. Instead we shall use a meta-program to control the order of rule applications.

3 Controlling the Rules of Deduction at the Meta-level

Maude is specifically designed to construct meta-programs. The most important facility of the Universal Rewrite Theory (which is here used to simulate execution of other rewrite theories) is pre-implemented in a module called `META-LEVEL`. To use this module we have to meta-represent terms and other modules according to the specifications that can be found in the pre-implemented modules `META-TERM` and `META-MODULE`. The module `META-LEVEL` contains several functions that help us simulate execution at the meta-level [7, 9]. The most versatile of these is:

```
op metaXapply : Module Term Qid Substitution Nat Bound Nat -> Result4Tuple .
```

This function will be the core of all strategic choices implemented at the meta-level. The first three arguments of the `metaXapply` function are a meta-represented `Module` (rewrite theory), a meta-represented `Term`, and a meta-represented rule-name, which represents the rewrite rule we would like to apply to our `Term`. Terms of sort `Qid` are used for uninterpreted symbols in Maude and are written as a quoted list of characters.

A call to the `metaXapply` function will produce a `Result4Tuple` if it is possible to apply the desired rule to the `Term`. The `Result4Tuple` contains some extra information besides the resulting term, and we extract the resulting term by means of the `getTerm` function in our examples. If the rule application was not possible a failure term is returned by this function. This function gives us full freedom with respect to which rule of a given rewrite theory we want to apply to a given term of the theory.

A basic strategy. A first meta-level function will try to apply the rules of the calculus in a specific order. We initialize the term by applying the rule `'init` to select an active clause inside our `SearchState` element. We then apply the other rules recursively in the following order: `negLitInPath`, `negLitInMatrix`, and `extendPath`. This will produce a list of `SearchState` elements. The list is passed to another function which determines whether or not the elements represent paths with or without connections, by applying the deductive rules `counterModel` and `removeConnectedPaths` recursively.

The meta-represented `Module` will be a meta-representation of the deductive rules of the calculus and the `Term` will be a meta-represented `SearchState` element which is un-initialized; i.e., it has no active clause. The init function applies the `'init` rule to the `SearchState` element.

```
op init : Module Term -> Term [ctor] .

eq init(M, T) =
getTerm(metaXapply(M, T, 'init,  none, 0, unbounded, 0)) .
```

Below is the result of applying this function to a term, one of the clauses inside the remaining matrix becomes the first active clause.

```
Before: < none ; emptyClause ; [[a, - b] , [b, - a], [c, d], [a, b], [b, a]] >
After:  < none ; [a, - b] ; [[b, - a], [c, d], [a, b], [b, a]] >
```

When the active clause has been selected and removed from the remaining matrix, we are ready to start looking for connections. This will be handled by the next function.

```
op basicStrat : Module Term -> Term [ctor] .

eq basicStrat(M, T) =

if (metaXapply(M, T, 'negLitInPath, none, 0, unbounded, 0) =/= failure)
then
     basicStrat(M, getTerm(metaXapply(M, T, 'negLitInPath, none, 0, unbounded, 0)))
else if
     (metaXapply(M, T, 'negLitInMatrix, none, 0, unbounded, 0) =/= failure)
then
     basicStrat(M, getTerm(metaXapply(M, T, 'negLitInMatrix, none, 0, unbounded, 0)))
else if
     (metaXapply(M, T, 'extendPath, none, 0, unbounded, 0) =/= failure)
then
     basicStrat(M, getTerm(metaXapply(M, T, 'extendPath, none, 0, unbounded, 0)))
else
     T
fi fi fi .
```

We test whether a `'rule` can be applied with the statement:

```
if (metaXapply(M, T, 'rule, none, 0, unbounded, 0)=/= failure)
```

If `metaXapply` returns a failure term this implies that the `'rule` cannot be applied to the term `T`, and the if-test will fail. We must assure that we only apply applicable rules to our term, so the function `basicStrat` has if-tests before each attempted rule application. The function `basicStrat` is called recursively with the resulting term after a rule has been applied to the term (we extract the term from the `Result4Tuple` that `metaXapply` returns with the function `getTerm`). The function `basicStrat` produces a list of `SearchState` elements which is handed over to the function `simplify`.

```
op simplify : Module Term -> Term [ctor] .

eq simplify(M, T) =
if (metaXapply(M , T, 'removeConnectedPaths, none, 0, unbounded, 0)  =/= failure)
then
    simplify(M, getTerm(metaXapply(M , T, 'removeConnectedPaths, none, 0, unbounded, 0)))
```

```
else if
    (metaXapply(M , T, 'counterModel, none, 0, unbounded, 0) =/= failure)
then
    simplify(M, getTerm(metaXapply(M , T, 'counterModel, none, 0, unbounded, 0)))
else T
fi fi .
```

This function determines whether `SearchState` elements represent paths with or without connections. Note that all elements inside the `SearchStateList` produced by the function `basicStrat` will represent paths either with or without connections. The three functions just described are combined to produce a search strategy:

```
op proveBasic : Module Term -> Term [ctor] .

eq proveBasic(M, T) =
simplify(M, basicStrat(M, init(M, T))) .
```

Far from optimized, this implements a sound strategy for the calculus. A propositional connection based theorem prover should terminate the proof search as soon as a path without a connection is located, the first strategy presented does not.

A refined strategy. A more refined strategy is now introduced, which addresses this problem. This refined strategy also solves the efficiency problems that occur when we try to match large terms to a set of rewrite rules. Some of the deductive rules produce a new `SearchState` element. The strategy will place newly generated `SearchState` elements onto a stack. This gives us the ability to terminate the proof search as soon as a path without a connection is located. We avoid matching large terms (i.e., the whole `SearchStateList`) to the rewrite rules, which would reduce the efficiency of the algorithm. The idea behind this strategy is to investigate one `SearchState` element a time, loading new `SearchState` elements onto a stack. If the `SearchState` element we investigate represents paths with connections, we pop an element of the stack and start to investigate this element. If the `SearchState` element we investigate represents a path without a connection, the proof search ends. We try to apply the rule `negLitInPath` first to ensure soundness, since the last rule application can leave us with `SearchState` elements on the form `< PATH , LIT ; [- LIT] ; [none] >`.

```
eq refinedStrat(M, ACTIVE, STACK) =
if (metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, 0) =/= failure)
 then
refinedStrat(M,
     getTerm(metaXapply(M, ACTIVE, 'negLitInPath, none, 0, unbounded, 0)), STACK)
```

```
 else if (simplify(M, ACTIVE) ==  'notvalid.TerminationValue)
 then 'notvalid.TerminationValue  **** proof search terminates ****

 else if (simplify(M, ACTIVE) ==  'valid.TerminationValue)
 then  refinedStrat(M, metaPop(M, STACK), metaPopped(M, STACK))

 else if (metaXapply(M, ACTIVE, 'negLitInMatrix, none, 0, unbounded, 0) =/= failure)
then
refinedStrat(M,  metaFirst(M, getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0))),
metaPush(M, metaRest(M, getTerm(metaXapply(M, ACTIVE,
'negLitInMatrix, none, 0, unbounded, 0))), STACK))

else if

(metaXapply(M, ACTIVE, 'extendPath, none, 0, unbounded, 0) =/= failure)
then
refinedStrat(M,  metaFirst(M, getTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0))),
metaPush(M, metaRest(M, getTerm(metaXapply(M, ACTIVE,
'extendPath, none, 0, unbounded, 0))) , STACK))

else 'valid.TerminationValue

fi fi fi fi fi .
```

In order to create a decision strategy, we initialize the `SearchState` element and call the function `refinedStrat` with the meta-representation of an empty stack.

```
op proveRefined : Module Term Term -> Term [ctor] .

eq proveRefined(M, T) =
refinedStrat(M, init(M, T), 'nil.SearchStateList) .
```

A small example is now given to illustrate how the refined strategy works compared to the basic strategy previously introduced. The difference is best seen with a matrix without any connections: `[[a, - b], [c, d], [- e, f]]`. Both strategies start by loading this matrix into the remaining matrix part of a `SearchState` element, and then initialize the term (select an active clause). This will be the resulting term:

```
< none ; [a, - b] ; [[c, d], [- e, f]] >
```

Then the functions `basicStrat` and `refinedStrat` are called to apply the deductive rules to this term. The only rule we are able to apply is `extendPath` since there are no connections in the matrix. After the first function call to both functions (`basicStrat` and `refinedStrat`), we end up with the following term. The two elements in the `SearchStateList` are enumerated to see the difference between the strategies.

```
1. < a  ; [c, d] ; [[- e, f]] >
2.  < none ; [- b] ; [[c, d], [- e, f]] >
```

The function `basicStrat` will now try to apply the deductive rules to the `SearchStateList`, while `refinedStrat` will place element no. 2 on its stack of un-investigated elements, and continue with no. 1. This has considerable impact when terms become large, since all deductive rules will try to match each element inside the list when the `basicStrat` is employed. Now as both functions call themselves recursively, the `refinedStrat` function has placed the second `SearchState` element onto its stack. After the second recursive call to both functions we obtain the following term or `SearchStateList`, enumerated to highlight the difference between the strategies.

```
1.  < a, c ; [- e, f] ; [none] >
2.  < a  ; [d] ; [[- e, f]] >
3.  < none ; [- b] ; [[c, d], [- e, f]] >
```

Recall that we are still only able to apply the rule `extendPath` since no connections exist, `basicStrat` will use more time to figure out this as well since it will match the rules against a larger term. After the second recursive call to both functions `basicStrat`'s term consists of the three elements above, while the `refinedStrat` has placed `2` and `3` on its stack of un-investigated `SearchState` elements. The next recursive call to the function `refinedStrat` will reveal that `SearchState` element no. `1` represents a path without a connection (or actually two). Then the whole search can be called off, there is no need to investigate the paths lying on the stack since we already have located a path without a connection. The function `basicStrat` still finds applicable rules to its term since it consists of all three elements, and the rule `extendPath` can be applied to element `2` and `3`, this process will continue until all the elements inside the `SearchStateList` are on the form `< PATH ; [LITSET] ; [none] >`. When all the elements are on this form `basicStrat` will be unable to apply any of the three deductive rules and the `SearchStateList` is handed over to the simplify function which reduces the whole list of paths without connections to the constant `notvalid`.

Performance. In Fig. 1 we can see the performance results of the two strategies presented on some test formulas. We have tested the same formulas on the Prolog prover of Otten and Bibel for comparison. As soon as matrices become large the implementation in Maude struggles. However, when we compare the two strategies introduced in the paper we can see that we have been able to optimize the term rewriting (deductive process) a great deal with the refined strategy. Prolog struggles with invalid formulas since this can cause a lot of backtracking (this is only necessary for the first order part, so this causes unnecessary computation on

propositional formulas). When formulas are valid there will be no backtracking with leanCoP, and it will usually be faster.

Test formula	basicStrat	refinedStrat	leanCoP
1 GRA001-1.p	530 ms	40 ms	10 ms
2 NUM285-1.p	-	36.8 min	-
3 PUZ009-1.p	90 ms	20 ms	20 ms
4 PUZ013-1.p	1.5 sec	90 ms	420 ms
5 PUZ016-2.004.p	-	220 ms	-
6 SYN092-1.003.p	-	70 ms	10 ms
7 SYN011-1.p	20 ms	10 ms	0 ms
8 SYN093-1.002.p	2.7 sec	130 ms	0 ms
9 SYN004-1.007.p	340 ms	40 ms	10 ms
10 SYN008-1.p	0 ms	0 ms	0 ms
11 SYN003-1.006.p	110 ms	20 ms	0 ms
12 SYN85-1.010.p	20 ms	10 ms	0 ms
13 SYN86-1.003.p	-	20 ms	20 ms
14 SYN087-1.003.p	-	20 ms	0 ms
15 SYN089-1.002.p	10 ms	0 ms	0 ms
16 SYN091-1.003.p	-	70 ms	20 ms

Fig. 1. Performance results. The formulas are fetched from the TPTP library (version 3.0.1). An open entry (−) means that this problem was not solved within one hour.

4 Conclusion

This paper has considered how deduction strategies may be prototyped and compared for calculi specified in rewriting logic. The reflective property of rewriting logic provides a conceptually clear distinction between the rules of the calculus and the deductive strategies of the proof search. We have illustrated the approach by providing a rewriting logic specification of a calculus for the connection method and defining two different search strategies for this calculus. A similar calculus for the first-order language without equality has also been specified and similar strategies have been defined [8]. Although the explicit representation of variable unification and backtracking makes the specification slightly more demanding, the gain from experimentation with search strategies should increase with the complexity of the language.

The main advantage gained by this approach to prototyping automated deduction systems is that rewriting logic's reflective property gives us full flexibility to experiment with strategies relative to a calculus, as the two levels are not intermixed. Consequently, the approach seems well-suited for prototyping search strategies before porting a deductive system to a low-level efficient implementation, where the two levels are most often intermixed.

Acknowledgment
We are grateful for several helpful comments from Jens Otten.

References

1. P. B. Andrews. Refutations by matings. IEEE Transactions on Computers, C-25:193-214, 1976.
2. B. Beckert and J. Posegga: leanTAP: Lean, Tableau-based Deduction. Journal of Automated Reasoning, Vol. 15, No. 3, 339-358, 1995.
3. W. Bibel. An approach to a systematic theorem proving procedure in first order logic. Computing 12:43-55, 1974.
4. W. Bibel and J. Otten. leanCoP: lean connection-based theorem proving. In Proceedings of the Third International Workshop on First-Order Theorem Proving, pages 152–157. University of Koblenz, 2000.
5. M. Clavel and J. Meseguer. Reflection in Conditional Rewriting Logic. Theoretical Computer Science. 285, 2, 245-288, 2002.
6. M. Clavel, Reflection in Rewriting Logic, Metalogical foundations and Metaprogramming Applications, CLSI publications, 2000.
7. M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer and C. Talcott. Maude Manual, 2004.
8. B. Holen. A Reflective Theorem Prover for the Connection Calculus. MSc thesis, Dep. of Informatics, University of Oslo, 2005.
9. T. McCombs. Maude Primer, 2003.
10. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science. 96, 73–155, 1992.
11. J. Otten. ileanTAP: An Intuitionistic Theorem Prover. International Conference TABLEAUX'97, LNAI 1227, pages 307-312. Springer Verlag, 1997.
12. J. Otten and H. Mantel. linTAP: A Tableau Prover for Linear Logic. International Conference TABLEAUX'99. LNAI 1617, pages 217-231, Springer Verlag, 1999.
13. Maude web-site: http://maude.cs.uiuc.edu/