An Experimental Comparison of Theorem Provers for CTL

Rajeev Goré and Jimmy Thomson
School of Computer Science
Australian National University, Canberra
{*rajeev.gore,jimmy.thomson*}*@anu.edu.au*

Florian Widmann*
Department of Computing
Imperial College, London
f.widmann@imperial.ac.uk

***Abstract*—We compare implementations of five theorem provers for Computation Tree Logic (CTL) based on tree-tableaux, graph-tableaux, binary decision diagrams, resolution and games using formula-classes from the literature. In the process, we gather and analyse a set of test formulae which could form the basis of a suite of benchmark formulae for CTL.**

***Keywords*-computation tree logic; automated reasoning; experimental comparison**

I. Introduction and Motivation

Computation Tree Logic (CTL) is a logic of fundamental importance in Computer Science. A CTL Kripke model can be viewed as a rooted tree where each branch of the tree consists of nodes or states which form an infinite discrete sequence isomorphic to the natural numbers. Such a model can be viewed as all possible runs of a system that moves from one state to the next state, thereby allowing us to reason about properties of the system. Thus, given such a CTL-model M and a CTL-formula φ, we can ask whether all runs of the system M eventually satisfy φ by asking $\mathbf{AF}\ \varphi$, or whether φ is true of every state of some run by asking $\mathbf{EG}\ \varphi$. This model-checking problem has received much attention over the past thirty years and has led to model-checking tools which are now routinely used in the verification of digital circuits for example [5].

Contrast this with the satisfiability problem for CTL, which is to answer whether φ has any CTL-models. Satisfiability testing is important for synthesizing systems from specifications [2] and for generating guarantees about specifications, but it is only in the past five or six years that researchers have attempted to build efficient satisfiability testers for CTL. Currently, there are five main methods for deciding (fixpoint logics like) CTL: Fischer-Ladner-Pratt-like methods, (incremental) semantic tableaux, automata, temporal resolution, and parity games, but there is no clear picture of the state of the art.

Here we present an experimental comparison of the following provers: a Binary Decision Diagram (BDD) based implementation of a Fischer-Ladner-Pratt-like method [12]; implementations of two tableau-based methods [4], [1]; an implementation of a refined resolution-based method [17]; and an implementation of a parity-game method [6]. We are not aware of any other theorem provers for CTL. In particular, we know of no theorem prover for CTL based upon automata, although some [15] consider the BDD method to be related to automata methods, and the parity game method [6] encodes automata states into the game. CTL-satisfiability cannot be decided by CTL model-checking the "free model", as in LTL, because CTL model-checking and CTL-satisfiability have inherently different computational complexity.

* Partially supported by EPSRC grant EP/H016317/1.

II. Computation Tree Logic

Computation Tree Logic (CTL) is a temporal logic of discrete branching time. CTL-formulae are built from atomic formulae using the classical connectives, and the following temporal unary and binary connectives: $\mathbf{AX}\ \cdot, \mathbf{EX}\ \cdot, \mathbf{AG}\ \cdot, \mathbf{EG}\ \cdot, \mathbf{AF}\ \cdot, \mathbf{EF}\ \cdot, \mathbf{A}\,(\cdot\ \mathbf{U}\ \cdot)\,, \mathbf{E}\,(\cdot\ \mathbf{U}\ \cdot)$. We assume the reader is familiar with the semantics of CTL [5]. All unary operators have higher precedence than the binary ones, and conjunction binds tighter than disjunction which binds tighter than implication. Our test harness passes formulae to the individual provers enforcing this precedence, but without enforcing any particular associativity on conjunction and disjunction.

III. Provers and their Underlying Methodology

A. CTL-RP.

CTL-RP is a resolution-based [17] prover for CTL with worst-case EXPTIME complexity. The CTL formula is put into a clausal normal form SNF^{g}_{CTL} using new atomic formulae (renaming). All clauses are translated to first order clauses, except the eventualities (sometime clauses) which are handled separately.

CTL-RP has a saturation phase, where all possible resolvents are found, and an eventuality phase which generates extra clauses that identify unfulfilled eventualities. These two phases are iterated until the empty clause is generated, or no further resolutions are possible. *CTL-RP* is implemented in C and uses SPASS, a highly optimised first-order resolution prover, to perform both phases.

B. BDDCTL.

BDDCTL [12] uses binary decision diagrams (BDDs) to represent all possible worlds of all putative models for the given formula φ, and then applies μ-calculus model checking techniques using BDDs to determine whether φ is satisfiable in any of these worlds. The algorithm has a worst-case complexity of EXPTIME.

This approach constructs a set W of all possible worlds given the formula φ, and then prunes inconsistent and unsatisfiable worlds. To define W, *BDDCTL* takes all the atomic propositions of φ as well as all subformulae starting with a temporal connective, and converts each to a BDD variable. Each truth assignment to these variables represents one world, so the True BDD represents all possible worlds. The pruning process "deletes" worlds which fail CTL modal constraints from (the BDD representing) W. The most expensive part is calculating a least fixpoint for each BDD variable representing any temporal path operator. When no worlds can be pruned, the BDD representing them is checked against a BDD representing φ. If the intersection of these two BDDs is nonempty, then φ is satisfiable.

C. MLSolver.

MLSolver [6] is a framework which provides satisfiability checkers for many modal fixed point logics, including CTL. It converts a formula into a parity game and solves that game using PGSolver (written by the same authors). The actual implementation checks validity. Its worst-case complexity for CTL is EXPTIME.

More precisely, *MLSolver* builds a finite representation of a conceptually infinite tableau using straightforward tableau rules (depending on the logic). To check eventualities are fulfilled *MLSolver* annotates the tableau nodes with states of a deterministic word parity automaton (also depending on the logic) which recognises bad tableau branches. The annotated tableau is then collapsed into a finite graph which is the input of the game solver.

MLSolver offers runtime options. We used `-opt comp -opt litpro -pgs recursive` which seemed to deliver the best results on average. The results we present use the version last updated on September 30 2010.

D. GMUL.

GMUL is our implementation of the graph multipass tableau method based on Ben Ari's algorithm for $\mathcal{UB}$ [4], but with some changes to allow for CTL's Until operator. The algorithm's worst-case complexity is EXPTIME.

Starting with a root node containing the original formula, tableau rules are applied incrementally to create a graph where each node is a unique set of formulae. Nodes containing $\{p, \neg p\}$ are discarded immediately. For each eventuality $\mathbf{A}\,(\varphi\ \mathbf{U}\ \psi)$ and $\mathbf{E}\,(\varphi\ \mathbf{U}\ \psi)$, nodes containing ψ are marked as satisfying the appropriate eventuality. This information is back-propagated through the graph until no further nodes are marked, at which point all unmarked nodes containing the eventuality are pruned. This process is repeated for all eventualities until a fixpoint is reached. If the root node has not been pruned, then the formula is satisfiable. Our Ocaml implementation of *GMUL* uses semantic branching.

E. TreeTab.

TreeTab implements a one-pass tree tableaux method [1] which constructs a tableau from the initial formula, passing information up from the leaves to handle eventualities. The algorithm's worst-case complexity is 2EXPTIME but it has the potential to terminate early when satisfiability can be passed up from the leaves to the root.

Tableau rules are applied to construct a tree with back edges. Some branches close with a contradiction, while others loop back to ancestors. These open branches propagate information on the unfulfilled eventualities up from the leaves, combining the information from other branches as they join up. At the highest node reachable from a subtree, if eventualities remain unfulfilled then the subtree is closed. Our OCaml implementation uses depth-first search and several optimisations including semantic branching, caching of results within AND-branches, and local backjumping.

F. Pre-processing.

We found that *BDDCTL*, *MLSolver* and *CTL-RP* are sensitive to a simple pre-processing step which compacts $\mathbf{AG}\ \varphi_1 \wedge \ldots \wedge \mathbf{AG}\ \varphi_n$ to its logical equivalent $\mathbf{AG}\ (\varphi_1 \wedge \ldots \wedge \varphi_n)$. This reduces the number of BDD variables and fixpoint computations for *BDDCTL* and gives fewer clauses and introduced propositions for *CTL-RP*. The time for compaction was ignored, since it is insignificant.

The *BDDCTL* implementation constrains the size of temporal formulae, so $\bigwedge_i \mathbf{AG}\ \varphi_i$ is "3-compacted" to $\bigwedge_j \mathbf{AG}\ (\varphi_{3j} \wedge \varphi_{3j+1} \wedge \varphi_{3j+2})$ as a trade-off between potential improvement and satisfying this constraint. The resulting incarnation is called *BDDCTL3c*. To allow comparison between *BDDCTL3c* and *CTL-RP* while also showing the full extent of this change, *CTL-RP* is presented using both the limited compaction as *CTL-RP3c* and full compaction as *CTL-RPc*. We also investigate the effect of full compaction, as described above, on *MLSolver*, resulting in *MLSolverc*.

This optimisation has no effect on the tableaux methods because they treat $\mathbf{AG}\ \varphi_1 \wedge \ldots \wedge \mathbf{AG}\ \varphi_n$ the same as $\mathbf{AG}\ (\varphi_1 \wedge \ldots \wedge \varphi_n)$.

G. Hybrids.

We hybridised the potential early-termination of *TreeTab* with *BDDCTL3c* or *CTL-RPc*, giving *TBDDh* and *TRESh*. We give *TreeTab* 5% of the allocated time, and the rest to the other method if required. Hence, graphs for these hybrids can be re-constructed from the given graphs.

H. Discussion.

Conceptually, there is a dichotomy between "top down" and "bottom up" methods. That is, each of *GMUL*, *ML-Solver*, *TreeTab* build a graph/tree based upon the structure of the given formula. On the other hand, both *BDDCTL*, *CTL-RP* do not since *BDDCTL* considers only the atomic formulae extracted from the given formula, while *CTL-RP* transforms it to SNF and looks for clashes.

IV. The Benchmarks

Different criteria for benchmarks have been proposed [3], [13], [16], however these criteria can be difficult to quantify. An oft-used criterion is that the benchmarks should be appropriately difficult, neither too easy nor too hard. But what defines difficulty? Different theorem provers using different algorithms may have vastly different performance on the same problem. Measuring the difficulty of a formula by the size of its model is biased towards methods which attempt to create a model, and we may not discover that an "easy" formula causes difficulties in a different method.

Redundancy is considered undesirable [16], but there is no guarantee that problems from an application domain will not have redundancies. It may be the case that some theorem provers are more adept at discovering and making use of these redundancies, which is something useful to know.

Problems which can be improved by the use of "simple tricks" [3] are also considered undesirable, but what is a simple trick? Here we notice that compacting the original formula can have significant benefits for several theorem provers on most problem classes. If this invalidates the benchmark, then so would any optimisation that could apply to any prover. The example given in [3] is more extreme, but it may also be useful to know how specialist classical methods compare to the more general modal approaches.

The appropriate way to compare different theorem provers has also been considered [13], however we would argue that picking one formula as a baseline is ill-advised. If one method can solve the baseline in 1s and the next hardest in 2s, while another method takes 10s and 11s to do the same, there is little reason to believe that the second method outperforms the first. Instead of reducing the results to single numbers, where possible we present graphs showing the trend of how the different methods perform. This allows comparisons of the time and scaling of each method.

There are currently no widely used benchmarks for CTL, and the various methodologies for constructing such benchmarks are questionable, particularly since we are not interested in declaring a winner but more in analysing the behaviour of the various provers. We have therefore taken formulae from the literature and analysed how the different provers behaved on these formulae. Following Hustadt and Schmidt [11], we have also constructed some formulae ourselves to test hypothesis about the likely behaviour of specific provers, based upon an understanding of their underlying method. Only the benchmarks we created ourselves are fully detailed below. All benchmarks are available at http://users.cecs.anu.edu.au/~rpg/CTLComparisonBenchmarks/

A. Exponential Formulae.

CTL satisfiability is EXPTIME-complete, and these formulae force all their models to have exponentially long paths. The satisfiable exponential formula specifies a binary counter with n bits, while the unsatisfiable formula adds that all n bits must never be true at once [10]. The latter has no models, but a tableau procedure builds an exponential branch to discover this.

B. Pattern Formulae.

We designed a pattern formula (AE) to test a hypothesis that *BDDCTL* would perform badly on a simple formula containing many conflicting EG temporal formulae while other methods would not:

$$(\bigwedge_{i=1}^{n} \mathbf{AG}\,\mathbf{EG}\; p_i) \vee (\bigwedge_{i=1}^{n} \mathbf{AG}\,\mathbf{EG}\; \neg p_i) \qquad (AE)$$

Intuitively, this is a formula with very little choice for tableaux methods, and one with few resolvents and no eventualities. *BDDCTL* finds AE hard because initially *BDDCTL* ignores the specific formula and only considers the individual atomic and temporal components. The two disjuncts of the top-level disjunction are individually simple, but the interactions between them and their negations, which *BDDCTL* considers, result in complex interactions.

C. Reskill.

This formula tests a hypothesis that CTL-RP would perform badly when there were many potential resolutions in a satisfiable formula. It is intended to be difficult for *CTL-RP*, but not for the other methods.

$$\begin{aligned}\Gamma = (\mathbf{AG}\; \textstyle\bigvee_i^n \mathbf{AF}\; q_i) \wedge \bigwedge_i^n \mathbf{AG}\; (q_i \Rightarrow \bigwedge_{j\neq i} \mathbf{EX}\; q_j) \\ \wedge \bigwedge_i^n \mathbf{AG}\; (q_i \Rightarrow \bigwedge_{j\neq i} \neg q_j)\end{aligned}$$

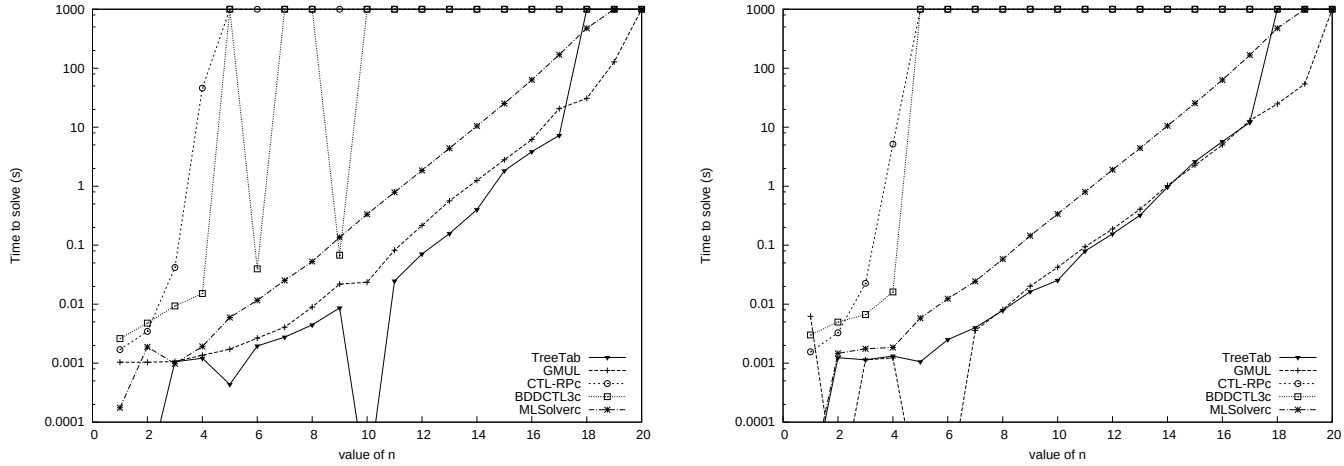
$$reskill = \neg p \wedge (p \Rightarrow (p \wedge \Gamma))$$

The intuition is that a satisfiable formula will cause *CTL-RP* to perform the most work, and that having a large number of nontrivial resolutions which interact with some eventualities will cause the most resolutions, and cause the eventuality resolution rules to trigger the most often.

To further increase the difference between *CTL-RP* and the "top down" methods, the potentially difficult Γ is placed where these methods cannot encounter it since their root node has only two children $\{\neg p\}$ and $\{p, \neg p, \Gamma\}$.

D. Alternating Bit Protocol.

These 3 formulae encode the alternating bit protocol used in a comparison of *CTL-RP* [17], where i represents the initial state, and a_0 and a_1 represent states where a



(a) Satisfiable (b) Unsatisfiable

Figure 1. Exponential formulae

0 or 1 bit is transmitted. The formulae respectively encode whether the following properties hold of the protocol: (i) $\mathbf{A}\,(i\ \mathbf{U}\ a_0)$ (ii) $\mathbf{AG}\ \ (a_0 \Rightarrow \mathbf{A}\,(a_0\ \mathbf{U}\ a_1))$ (iii) $\mathbf{AG}\ \ (a_1 \Rightarrow \mathbf{A}\,(a_1\ \mathbf{U}\ a_0))$. The formulae take the form $\neg\,(\varphi \Rightarrow \psi)$ where φ encodes the protocol and ψ the property being checked. All formulae are unsatisfiable.

E. Step Formulae.

The explicit form of these formulae can be found in a description of *BDDCTL* [12].

F. Montali's Formulae.

These formulae are adapted from formulae used in comparisons of LTL provers [8] and based on formulae originally from the specification of business processes [14]. They are parametrized by n and m:

$$\varphi^i_1 := \mathbf{AF}\ p_i \qquad \varphi^i_m := \mathbf{AF}\,(p_i \wedge \mathbf{AX}\,\varphi^i_{m-1})$$

$$\psi_n := (\mathbf{AG}\ \textstyle\bigwedge_{i=0}^{n-1}(p_i \Rightarrow \mathbf{AX}\,\mathbf{A}\,(\neg p_i\ \mathbf{U}\ p_{i+1})))$$

sat class is $\varphi^0_m \wedge \psi_n$ unsat class is $\varphi^0_m \wedge \psi_n \wedge \neg\varphi^n_m$

G. Business Processes.

These formulae are based on the business process example from [2]. There are actions $allow, deny, never$, and $a_1, b_1, \ldots, a_n, b_n$. Each a_i and b_i action must be followed eventually by its successor, and each action occurs at most once. Each a_i has related mutually-exclusive results au_i, al_i and ah_i, and similarly for the b_i. The $allow$ action requires all al_i true, and $never$ requires at least one bh_i true. We also require $never \Rightarrow \mathbf{AG}\ \neg allow$ and $allow \Rightarrow \mathbf{AG}\ \neg never$, and $deny \Rightarrow (\neg \bigwedge_{i=1}^{n} al_i \vee \bigvee_{i=1}^{n} bh_i)$.

V. Experimental results

We used an Intel Core2 Duo 3.00GHz CPU with 3GB RAM, and a stack limit of 512MB. The solvers were given 1000 seconds for each problem instance (i.e. one value of n and m). Our raw results are at http://users.cecs.anu.edu.au/~rpg/CTLComparisonBenchmarks

Due to the range of numbers and nature of the data being shown, the plots are given using a log scale. Random noise on log plots can result in spikes which appear significant at smaller values but are not. To reduce clutter, the graphs only show maximally compacted versions of the solvers, and show no hybrids. These compacted versions were the best in all cases but two, which are noted in the discussion.

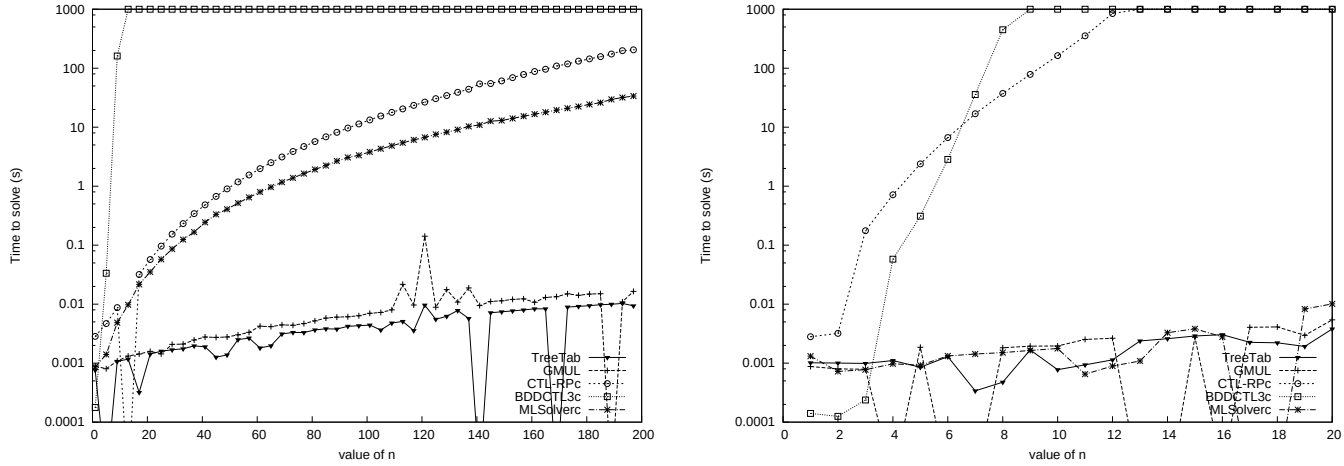
A. Exponential.

Fig. 1 shows the results for the exponential problem set. Note that in Fig. 1a *BDDCTL* fails for size 5, 7 and 8, as well as all values higher than 9. This is due to a flaw in the implementation which restricts the text size of BDD atoms, and temporal formulae are converted into BDD atoms. This flaw affects several other benchmarks, though usually more consistently than here.

As noted earlier, the seemingly large dip at n=10 in Fig. 1a for *TreeTab* is not significant, since the time falls from 10^{-2} to 10^{-5} for only one measurement.

In both figures *TreeTab* and *GMUL* outperform *CTL-RP*, with *GMUL* reaching a slightly higher value of n. *MLSolverc* has similar behaviour to *GMUL*, except it is roughly an order of magnitude slower.

GMUL runs out of memory before it runs out of time at size 20, and *TreeTab* runs out of memory at size 18. *TreeTab* runs out of memory earlier because we increased the amount of memory requested when the managed memory pool is exhausted, causing it to fail when the next step cannot fit in memory, as here. Each path in a potential model corresponds to a branch in *TreeTab*, so exponential paths lead to exponential branches, which cannot be reclaimed until the whole branch is complete.



(a) AE (b) Reskill

Figure 2. Satisfiable Pattern formula

C Type of Goal	Time in Seconds								
	TreeTab	*GMUL*	*BDDCTL*	*BDDCTL3c*	*RES*	*RESc*	*RES3c*	*MLSolver*	*MLSolverc*
(i)	< 0.01	< 0.01	-	1.79	0.58	0.07	0.13	26.71	9.34
(ii)	-	0.12	-	3.37	30.64	0.75	7.53	313.36	192.44
(iii)	-	0.06	-	2.30	48.35	0.80	6.93	265.78	180.77

Figure 3. Results for Alternating Bit Protocol with *RES* abbreviating *CTL-RP*

B. Pattern Formulae AE and Reskill.

Fig. 2a shows that the AE example does indeed cause *BDDCTL* to perform badly, but also affects *CTL-RP*. The AE formulae lead to $5n$ BDD variables, and $2n$ of those require *BDDCTL* to calculate the least fixpoint for an **AU** construct [12, p 233], which is the most expensive calculation. The tableaux methods pick one disjunct of the top-level disjunction and satisfy it easily, which is why they perform so well on this constructed example even though *GMUL* eventually tries both branches. *CTL-RP* suffers in this case because the formula is satisfiable and thus resolution must compute all resolvents before declaring it satisfiable. The two halves of the formula interact to make them exclusive, and this results in a nontrivial number of resolvents. This is the only benchmark where limited compaction (*CTL-RP3c*) performs better than full compaction (*CTL-RPc*) and we are unsure why.

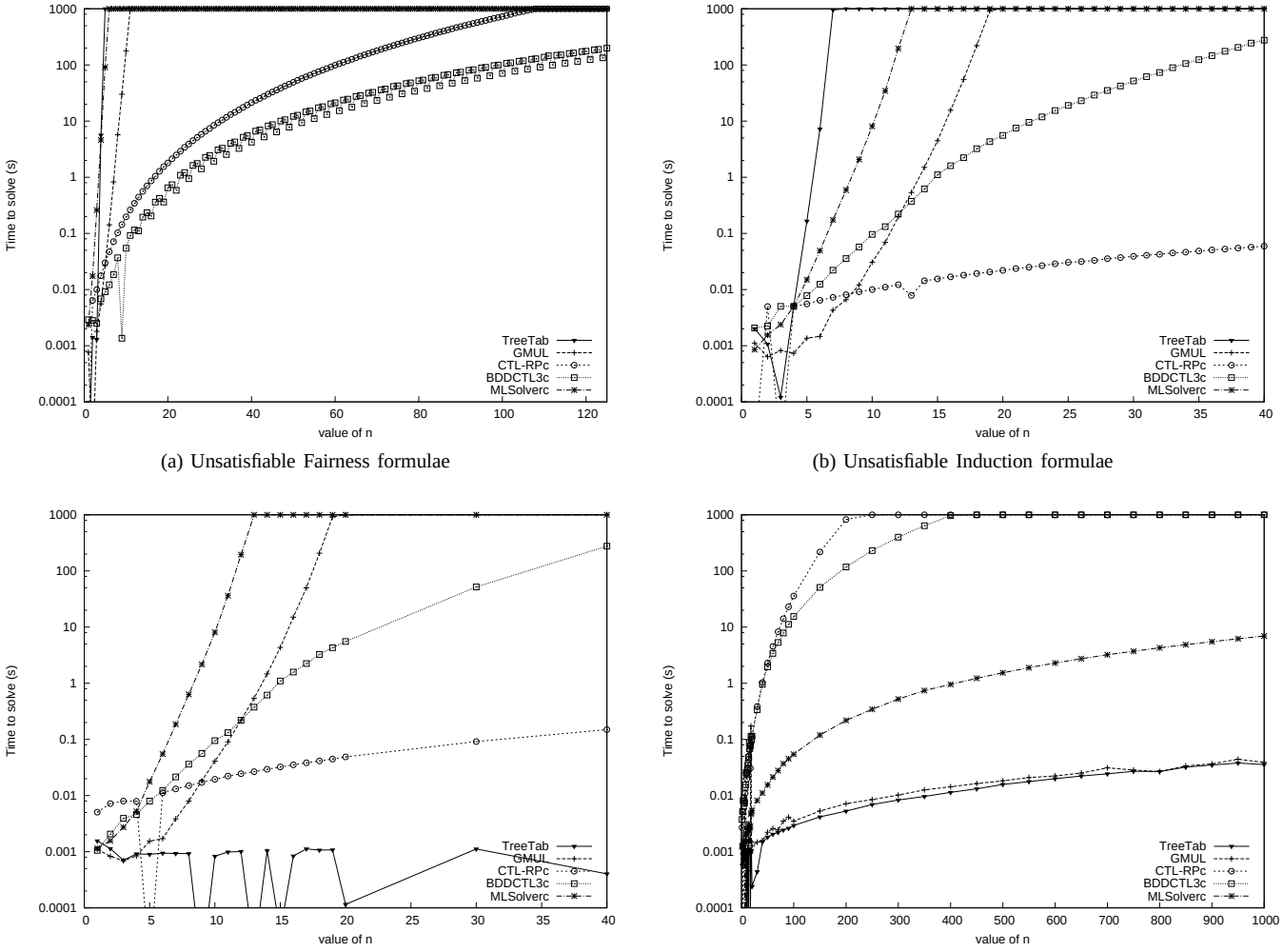
Fig. 2b shows a weakness of *CTL-RP*, as expected, taking a long time on relatively small input. *CTL-RPc* outperforms *BDDCTL3c*, however *CTL-RP3c* still performs worse than *BDDCTL3c*. *BDDCTL* performs badly here because its "bottom up" nature does not allow it to identify Γ as "irrelevant". The "top down" approaches all produce a contradictory node $\{p, \neg p, \Gamma\}$, and never explore Γ.

C. Alternating Bit Protocol.

Fig. 3 shows *GMUL* outperform all other methods, with *CTL-RP* (abbreviated to RES to save space) coming close. We let *BDDCTL* run for longer and it can handle the original formulae after upward of 2000 seconds, so the improvement from compaction here is dramatic, especially when compared with the resolution method given the same preprocessed formula. The reason is that the formula is stated in a modular way with several $\mathbf{AG}\ \varphi$ properties, each of which results in a BDD variable. These components interact in nontrivial ways, so calculating the fixpoints (of their negation) is nontrivial as well, requiring multiple iterations to reach a fixpoint. Compaction reduces the number of these variables and also the number of iterations it takes to propagate changes.

Note that despite (un-compacted) *CTL-RP* (*i.e. RES*) performing better than (un-compacted) *BDDCTL*, *CTL-RP3c* (*i.e. RES3c*) performs worse than *BDDCTL3c* in (ii) and (iii). We also manually compacted the formula further to still fit the size limit of *BDDCTL*, and this improved the time to below half a second in each case, outperforming *CTL-RPc* (*i.e. RESc*) in at least (ii) and (iii) and performing comparably in (i).

GMUL performs well in these cases because the formula restricts the search-space considerably. Even in the hardest of these three instances, *GMUL* constructs only around



(a) Unsatisfiable Fairness formulae

(b) Unsatisfiable Induction formulae

(c) Satisfiable Nobase formulae

(d) Unsatisfiable Precede formulae

Figure 4. Step Formulae

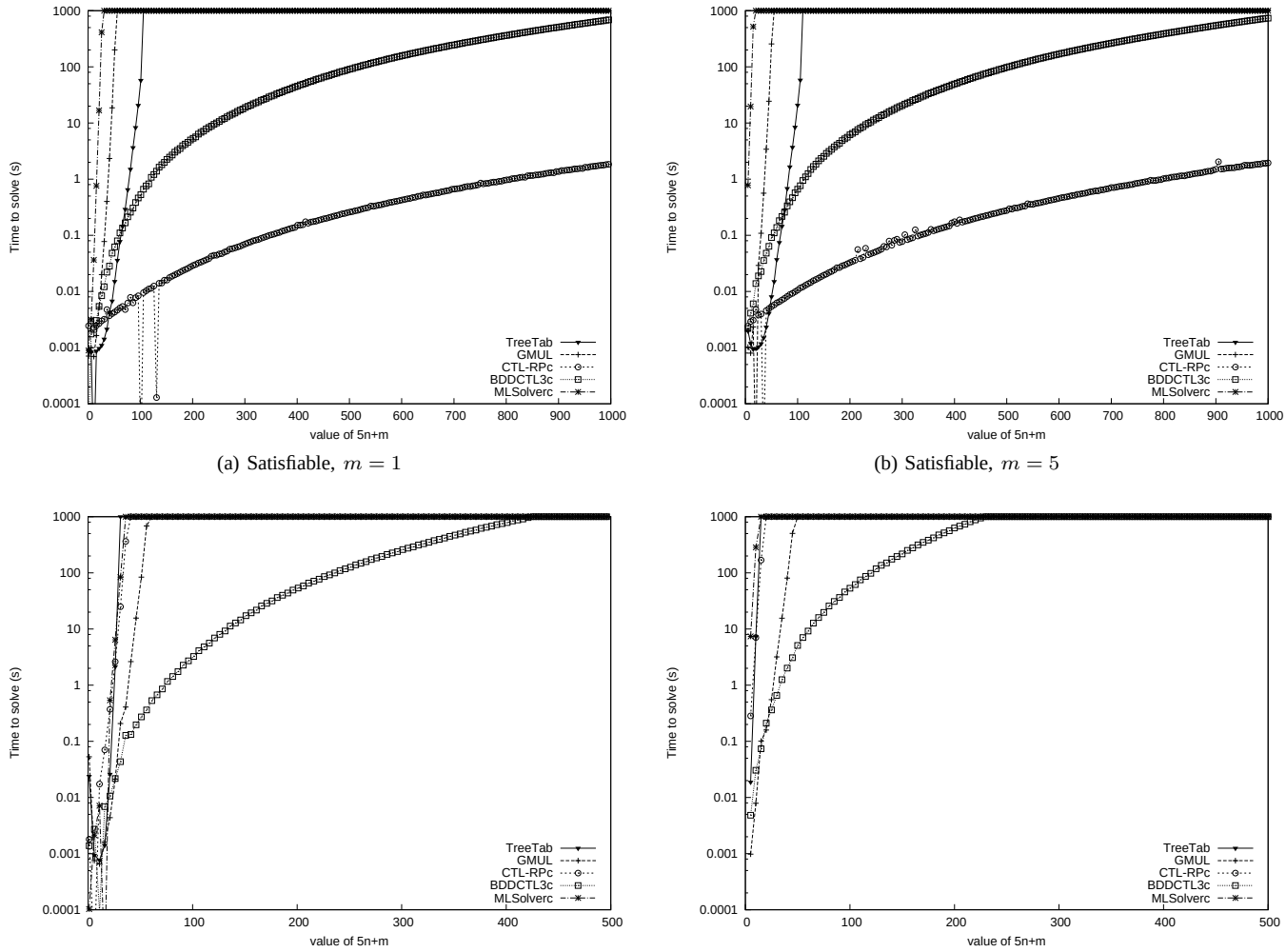
12,000 internal nodes. The properties (i)-(iii) are never true, so only one pass is required to prune the entire graph.

The bad performance of *TreeTab* (especially compared to *GMUL*) is because unsatisfiability is often not a direct contradiction $\{p, \neg p\}$, but instead is a failed eventuality. It is difficult to pinpoint where the unsatisfiability was inevitable for failed eventualities, so the tree method cannot immediately backjump to that point. Thus *TreeTab* must exhaustively try all children of or-nodes in the same search space that *GMUL* computes. Even though this is only some 12,000 distinct nodes, *TreeTab* processes the same node on multiple branches resulting in repeated work to no benefit. The first instance is easier as the condition is rooted at the start, and strongly constrains the states to make a_0 false unless it has previously made i false. The other instances are both global properties, so their negations become eventualities. So each time *TreeTab* fails to satisfy the property, it can try to procrastinate and satisfy it again later which means that it may end up looping, and the search space is much larger.

D. Step formulae.

The results for *BDDCTL* largely agree with previous results [12] after accounting for a more powerful test rig. However, in our experiments, (un-compacted) *BDDCTL* could not complete size 16 for precede in 1000 seconds while [12] gives a time of 1.9s for this size on a less powerful computer. The author of the BDD prover has not responded to our questions.

Fig. 4a shows a zig-zag pattern for *BDDCTL3c*, where it is faster for one in every three cases. This is an artifact of the restricted compaction, where $\mathbf{AG}$ formulae are grouped in threes. *CTL-RP3c* does not zig-zag, but it does have notable steps where a boundary of three is crossed. Interestingly, it appears that these jumps are not on the same formulae.



(a) Satisfiable, $m = 1$

(b) Satisfiable, $m = 5$

(c) Unsatisfiable, $m = 1$

(d) Unsatisfiable, $m = 5$

Figure 5. Montali's formulae

BDDCTL3c times improve when the compacting can fill all **AG**-groups with 3 conjuncts, while the jumps for resolution occur between having an **AG**-group with 1 conjunct and having an **AG**-group with 2 conjuncts.

Fig. 4b shows *CTL-RPc* performing the best by far, a considerable improvement over *CTL-RP* which was slower than *BDDCTL*. Compaction results in fewer input clauses, and many fewer derived clauses. Contradictions are also more direct, using fewer introduced propositions.

The induction and nobase formulae from Figures 4b and 4c are almost identical: the only difference is that the induction formula have an extra p_0 for the base case. Most methods behave similarly for the two benchmarks and perform most of the same work, except for *TreeTab* which easily finds a model for nobase and terminates quickly while timing out early in the unsatisfiable induction case.

The precede formulae in Fig. 4d are the only step formulae where *GMUL* does well. The formula is very restricted with exactly one initial assignment to the p_i permitted, so the search-space is small. The other benchmarks do not restrict the initial assignments, causing *GMUL* to consider exponentially many possible initial states, corresponding to each true/false combination of the p_i.

E. Montali's formulae.

Fig. 5a and 5b show the results for the satisfiable formula with depth 1 and 5. The differences are minor, and when all results from depths 1 to 5 are drawn on the same graph, using $x = 5n + m$, the curve appears smooth. The exception is *TreeTab*, which has large steps as n increases and smaller steps as m increases. In these satisfiable cases *BDDCTL3c* takes 1.5 times the time taken by (uncompacted) *BDDCTL*.

Fig. 5c and Fig. 5d show the results for the unsatisfiable formulae. Here, the depth plays a role in the difficulty of the formula. In this case compaction also makes *BDDCTL*

slower, but not to as great an extent as the satisfiable case. The only explanation we can suggest is that in both the satisfiable and unsatisfiable cases, compaction coincidentally reorders the formulae into a harder to construct BDD. *CTL-RP* fares very badly in these cases, despite doing well in the satisfiable case. This is because in the satisfiable case the only resolutions possible are those caused by the conversion to SNF^g_{CTL}. The unsatisfiable case introduces more clauses which cause many resolutions.

F. Business processes.

Size	Time in Seconds for Business Process Benchmarks			
	TreeTab	*GMUL*	*MLSolver*	*MLSolverc*
1	< 0.01	0.05	303.15	55.47
2	0.44	38.04	-	-
3	216.60	-	-	-

All methods not listed in the table failed on all problem instances. These formulae have many different models, especially as the depth increases. They arise from a synthesis problem [2] where any single model suffices, which is why we believe that the goal-directed methods do better.

VI. Conclusion

No single prover is the best for all problems, and each method has problems where it performs particularly badly.

MLSolver is the slowest prover in general, and tends to be dominated by *GMUL*. However, it caters for many different (fixpoint) logics in one setting, including logics more expressive than CTL like the μ-calculus and CTL*.

TreeTab generally performs better on satisfiable rather than unsatisfiable formulae, but not always. On unsatisfiable formulae in particular, it sometimes terminates instantly, but for other similar size formulae it runs "indefinitely", thus either succeeding or failing spectacularly.

GMUL performs badly on formulae with many distinct satisfying models since it constructs the pseudo-model for all of them. Despite using the same tableaux expansion rules as *TreeTab*, *GMUL* performs better when branches share nodes. An on-the-fly graph tableaux method [9] which interleaves the graph building and pruning phases may inherit the benefits of both. The tableaux implementations may benefit from refinement which the resolution and BDD implementations enjoy from over 20 years of development.

CTL-RP and *BDDCTL* appear more robust than the tableaux methods since they tend to succeed eventually rather than fail spectacularly or succeed spectacularly. Compaction appears to be generally useful for both.

The implementation flaws in *BDDCTL* require attention. A potential optimisation for *BDDCTL* is to intermittently intersect the original formula with the intermediate BDD for consistent worlds to detect unsatisfiability earlier.

Hybrids mixing the potential early success of *TreeTab* and the robustness of *BDDCTL* or *CTL-RP* are promising since *TBBDh* solved the most problems: 2235 out of 2946.

References

[1] P. Abate, R. Goré, and F. Widmann. One-pass tableaux for computation tree logic. In LPAR 2007 LNCS 4790:32–46.

[2] A. Awad, R. Goré, J. Thomson, and M. Weidlich. An iterative approach for business process template synthesis from compliance rules. In *Proc. CAiSE 2011 (to appear)*, 2011.

[3] P. Balsiger, A. Heuerding, and S. Schwendimann. A benchmark method for the propositional modal logics K, KT, S4. *J. Autom. Reasoning*, 24(3):297–317, 2000.

[4] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *POPL '81*, pages 164–176, ACM, 1981.

[5] O. G. Edmund M. Clarke and D. A. Peled. *Model Checking*. Cambridge, Mass. : MIT Press, 2000.

[6] O. Friedmann and M. Lange. A solver for modal fixpoint logics. *ENTCS*, 2009.

[7] J. Geldenhuys and H. Hansen. Larger automata and less work for LTL model checking. In LNCS 3925:53–70, 2006.

[8] V. Goranko, A. Kyrilov, and D. Shkatov. Tableau tool for testing satisfiability in LTL: Implementation and experimental analysis. *ENTCS*, 262:113–125, May 2010.

[9] R. Goré and F. Widmann. An optimal on-the-fly tableau-based decision procedure for PDL-satisfiability. In CADE-22, LNCS 5663:437–452, Springer-Verlag, 2009.

[10] J. Y. Halpern and Y. Moses. A guide to completeness and complexity for modal logics of knowledge and belief. *Artif. Intell.*, 54:319–379, April 1992.

[11] U. Hustadt and R. A. Schmidt. Scientific benchmarking with temporal logic decision procedures. In *KR*, pages 533–546. Morgan Kaufmann, 2002.

[12] W. Marrero. Using BDDs to decide CTL. *Lecture Notes in Computer Science*, 3440:222–236, 2005.

[13] F. Massacci and F. M. Donini. Design and results of tancs-2000 non-classical (modal) systems comparison. In *TABLEAUX 2000*, LNCS 1847: 52–56. Springer, 2000.

[14] M. Montali, P. Torroni, M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, and P. Mello. Verification from declarative specifications using logic programming. In LNCS 5366:440–454.

[15] G. Pan, U. Sattler, and M. Y. Vardi. Bdd-based decision procedures for K. In CADE-18, pages 16–30, LNCS, 2002.

[16] P. F. Patel-Schneider and R. Sebastiani. A new general method to generate random modal formulae for testing decision procedures. *JAIR*, 18:351–389, 2003.

[17] L. Zhang, U. Hustadt, and C. Dixon. A refined resolution calculus for CTL. In *CADE*, pages 245–260, 2009.