A Solver for Modal Fixpoint Logics

Oliver Friedmann Martin Lange

Dept. of Computer Science
University of Munich
Munich, Germany

Abstract

We present MLSolver, a tool for solving the satisfiability and validity problems for modal fixpoint logics. The underlying technique is based on characterisations of satisfiability through infinite (cyclic) tableaux in which branches have an inner thread structure mirroring the regeneration of least and greatest fixpoint constructs in the infinite. Well-foundedness for unfoldings of least fixpoints is checked using deterministic parity automata. This reduces the satisfiability and validity problems to the problem of solving a parity game. MLSolver then uses a parity game solver in order to decide satisfiability and derives example models from the winning strategies in the parity game. Currently supported logics are the modal and linear-time μ-calculi, CTL*, and PDL (and therefore also CTL and LTL). MLSolver is designed to allow easy extensions in the form of further modal fixpoint logics.

Keywords: tool support, modal logic, satisfiability checking

1 Introduction

Modal logics are important and very successful tools in various areas in computer science, philosophy, mathematics etc. They are being used – in various shapes and forms – in order to specify correct program behaviour (temporal logics, dynamic logics), to model and to reason about knowledge (epistemic logics, description logics), etc.

Any modal logic inherently faces the issue of expressiveness vs. complexity. On the one hand, logics are desirably very expressive, on the other hand, they should come with efficient decision procedures. But naturally, high expressive power entails high complexity.

Standard modal logic is particularly weak because of the locality aspect of the diamond and box operators. Since they transfer properties of worlds in a Kripke structure only to their immediate neighbours, any fixed formula of modal logic can only assert properties of worlds that depend on a neighbourhood of bounded size. Very simple properties like reachability – which are vital for some applications like program specification for instance – therefore cannot be expressed in standard modal logic and, thus, require stronger operators.

A generic mechanism that has proved to be successful in extending the expressive power of modal logics is that of incorporating operators which can be characterised as solutions to fixpoint equations over modal logic formulas. The modal μ-calculus $\mathcal{L}_\mu$ [14] does this in the most explicit form by adding fixpoint quantifiers to standard modal logic. Similar constructs – possibly in restricted form – are also present in other logics, for example as the Kleene-star in propositional dynamic logic PDL [8], as the until operator in temporal logics LTL, CTL, or CTL* [20,6,7], as transitive-closure operators in query languages [27] or in description logics [1], etc.

In any case, the satisfiability and, by duality, validity problems for such logics are of vital interest for specific problems in their domains of applications because many of those are in fact instances of the satisfiability problem for example. The subsumption problem in description logics (decide whether or not a given concept is contained in another given one), or the question whether or not a program specification given in temporal logic is realisable easily reduce to the satisfiability problem for these logics for instance.

Despite the similarities between various modal logics, tools for their satisfiability problems usually target a specific logic only. This makes sense because it is easier and more promising to optimise algorithms for specific rather than general problems. Furthermore, different communities seem to prefer different methodologies, for instance the automata-theoretic inclined temporal logic community [29] vs. the tableaux inclined description logic community [2]. On the other hand, similarities are not exploited and optimisations found for one logic may not be transferred to other logics where they may be applicable as well.

One difficulty that is common to satisfiability problems for all modal logics with fixpoint constructs is the regeneration or unfolding problem for least fixpoints. While it is sound and complete to unwind a least fixpoint operator according to its definition once, twice, and any finite number of times in order to build a tableau or an automaton or some other data structure from the input formula, one must ensure that such an unwinding does not continue ad infinitum. There are various ways to ensure this which all boil down to excluding certain cycles in certain graphs.

Incidentally, the same problem occurs in model checking CTL* [16,3]. Note that for logics like CTL, PDL, or the modal μ-calculus, satisfaction of a formula in a state of a transition system can be reduced to satisfaction of subformulas in states [26]. For CTL* this is not the case because of the mixture between state and path formulas. A CTL* model checker usually has to consider satisfaction of a set of formulas in a state. This introduces the same difficulties that arise with least fixpoint constructs in satisfiability checking procedures.

In this paper we describe a new tool called MLSolver which primarily provides a framework for satisfiability and validity checking for various modal fixpoint logics. It can also be used as a model checker for these logics, albeit not necessarily a competitive one.

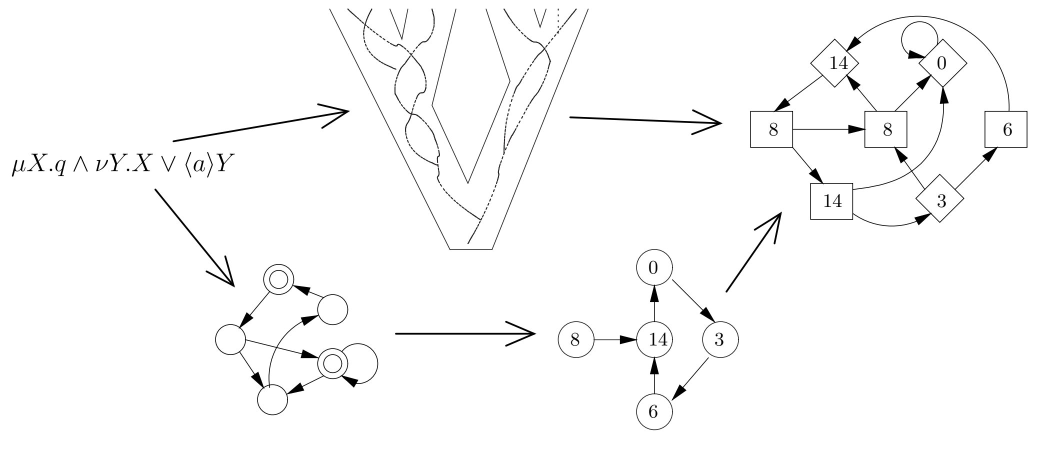

Fig. 1. A method for solving satisfiability for modal fixpoint logics.

2 The Underlying Theory

2.1 The Framework

MLSolver is a satisfiability and validity checker for various modal fixpoint logics. Satisfiability of such logics can be characterised through the existence of possibly infinite tableaux in which nodes are data structures containing formulas. Typically, these will simply be sets of subformulas of the input formula but this need not be the case. Also, these tableaux come with a notion of a good infinite branch which is one that does not contain any least fixpoint construct which regenerates itself along that branch. A tableau is then a finite graph on which every path starting in a designated initial node is either good or bad.

In order to distinguish good and bad branches and in particular detect bad ones we employ automata-theory. Bad branches can be accepted by (a combination of) nondeterministic finite ω-automata which essentially guess the occurrence of a least fixpoint construct in some tableau node and trace its infinite regeneration. Automata-theory provides algorithms for the determinisation and complementation of such automata into automata with a parity condition. The question of the existence of a tableau is then reduced to the problem of determining for a given node in a parity game which of the two players has a winning strategy for the game starting in that node. The nodes of the parity game are nodes which may occur in a tableau annotated with states of a deterministic parity automaton. Fig. 1 depicts this method in a diagramm: starting from a modal fixpoint formula, one creates a nondeterministic automaton and a (finite representation of an) infinite tableau with internal structure on the branches. The automaton is determinised and the product of the resulting automaton with the tableau yields a parity game.

The vast majority of modal fixpoint logics can be handled in this way. This has been shown explicitly for variants of the modal μ-calculus including the graded and the probabilistic μ-calculus [4] or for the linear-time μ-calculus [5]. It is also implictly present in other work, again for example for $\mathcal{L}_\mu$ [18] or for LTL and CTL [15] which simply do not mention explicitly the possibility of using automata-theoretical results for the incurring problem of detecting bad branches. It is also easily seen to be applicable for PDL and therefore also for the description logic $\mathcal{ALC}_{reg}$ [10], etc. Furthermore, a technically more involved but still similar construction yields a satisfiability checker for CTL*.

2.2 *μ-Threads*

A rule-based tableau system comes with a *connection relation* which relates a formula in a tableau node to formulas in a predecessing node. This gives rise to an internal structure of *threads* in an infinite branch which is an infinite sequence of connected formulas. Since fixpoint constructs are typically handled through unfolding rules which replace such a construct with a defining fixpoint expression (which can contain the construct again), an infinite unfolding leads to a thread on a branch. These threads can now be characterised as μ- or ν-threads depending on the outermost / topmost / dominating type of fixpoint construct that is unfolded infinitely often on this thread.

Next, one considers rule applications of this tableau system as an alphabet for a nondeterministic automaton $\mathcal{A}_{thread}$ which accepts all infinite sequences of rule applications (i.e. encoded tableau branches) which contain a μ-thread.

2.3 *Determinisation and Complementation of ω-Automata*

Remember that a tableau is a graph in which no path is bad in the sense that it contains a μ-thread. Thus, it should be obvious that complementation of ω-automata is needed in such a decision procedure because the nondeterministic automata mentioned above accept bad branches. Furthermore, for many genuinely modal logics these automata also need to be deterministic. This is the case iff the tableau system contains rules with more than one premiss. Then the tableau can have two bad branches which share a common prefix such that the two μ-threads on these bad branches split before the two branches split. Thus, a nondeterminisitic automaton may have accepting runs on these branches that differ on the common prefix, and a labelling of the tableau nodes with single automaton states would not be possible.

Note that determinisation and complementation commute, but in general it is easier to complement a deterministic automaton. Thus, $\mathcal{A}_{thread}$ will be determinised first. We make use of two constructions depending on its acceptance type.

(i) If $\mathcal{A}_{thread}$ is a nondeterministic Büchi automaton then we use Piterman's construction [19] in order to obtain a deterministic parity automaton from it. [1] It is a refinement of Safra's famous determinisation procedure [22] which yields deterministic Rabin automata and which are algorithmically not that easy to handle after being complemented into Streett automata.

Note that nondeterministic Büchi automata accept all ω-regular languages, and bad branches in tableaux for all the logics mentioned here form an ω-regular language.

(ii) In cases of logics structurally simpler than the modal μ-calculus, in particular those without nested fixpoint constructs like LTL, CTL, PDL, etc. bad branches are recognisable by nondeterministic co-Büchi automata. Their expressive power is strictly below that of full ω-regularity, but – as opposed to

[1] Alternatively, one could use the newer determinisation procedure given in [13].

Büchi automata – they enjoy determinisability. The Miyano-Hayashi construction [17] that originally transforms alternating into nondeterministic Büchi automata also works for this purpose. Additionally, it is only marginally more complex than the powerset construction for automata on finite words and way less complex than the Piterman construction for instance. Furthermore, deterministic co-Büchi automata can easily be complemented into deterministic Büchi automata which means that in this case satisfiability reduces to the solving of Büchi games, a strict subclass of parity games.

2.4 Solving Parity Games

A parity game is a finite graph whose node set is partitioned into nodes owned by player 0 and nodes owned by player 1. Additionally, each node carries a non-negative natural number, its priority. A play is an infinite sequence of adjacent nodes. It is won by player 0 iff the highest priority seen infinitely often in this sequence is even. Otherwise, player 1 wins this play.

The problem of solving a parity game is to compute for each node v, the player who has a strategy that allows him to win every play starting in v that complies with this strategy. It is well-known that this problem is well-defined, i.e. that for each such node exactly one of the players wins this node [31].

The are various algorithms for solving parity games. The most successful ones are the recursive proof of determinacy [31], the small progress measures algorithm [12], and strategy improvement [30,23]. Even though each of those (or the others) requires exponential time in the worst-case, parity games can be solved efficiently in practice [9].

One way of reducing the complexity of the resulting parity games avoids the mapping of every tableau node, annotated with a state of the deterministic automaton, to a node in the game graph. Instead, only those tableau nodes to which the usual modal rule is applied, are mapped. This rule, in CTL for example written as

$$\frac{\varphi_1, \psi_1, \ldots, \psi_m \qquad \varphi_2, \psi_1, \ldots, \psi_m \qquad \ldots \qquad \varphi_n, \psi_1, \ldots, \psi_m}{\mathtt{EX}\varphi_1, \ldots, \mathtt{EX}\varphi_n, \mathtt{AX}\psi_1, \ldots, \mathtt{AX}\psi_m, \ell_1, \ldots, \ell_k}$$

is applied whenever all boolean constraints about the current state have been resolved and the sequent consists of literals and diamond- and box-formulas only. This directly corresponds to a state in a possible model which is labeled with the present propositions and has successors given by the diamond-formulas.

Typically, this rule is the only one that creates universal branching in a tableau, and existential branching because of binary disjunctions in between can be collapsed to a choice by the existential parity game player after the universal player chooses one of the premisses of this rule. This leads to significantly smaller parity games, and can also speed up the construction of those because less effort is needed for the detection of cycles. However, one has to accummulate the priorities of the automaton states that occur on a path between two applications of the modal rule. Also, this optimisation is not easily possible if formulas are unguarded meaning that the tableau rules do not guarantee that every set of formulas will eventually be transformed into one to which the modal rule only applies. This is possible for

PDL with nested Kleene-stars and arbitrary formulas of the modal μ-calculus.

3 System Description

MLSolver provides a platform for satisfiability and validity checkers for various modal fixpoint logics. In order to allow for domain-specific optimisations and to reuse code for common functionalities, it is built in a modular way, separating the construction of tableaux for example from the automata-theoretic procedures like determinisation.

MLSolver is written in OCaml for the purposes of execution speed and source code readability. It is able to test input formulas of the supported logics for satisfiability or validity, or to check their satisfaction in a transition system given explicitly as a labeled directed graph. The is done as described above: a parity game is generated from the formula as the product of a tableau with a deterministic automaton. The parity game is then solved using PGSolver, a highly efficient and configurable solver for parity games [9]. PGSolver can be linked into MLSolver which allows for direct access to the solving routines in there and avoids costly printing and parsing of large parity games.

MLSolver currently supports the following logics: the modal μ-calculus, the linear-time μ-calculus, PDL, and CTL*. Note that CTL is a simple fragment of CTL* and so is LTL which is also a fragment of the linear-time μ-calculus. Thus, MLSolver is also capable of determining satisfiability and validity of LTL and CTL formulas. However, μ-threads in these two logics are co-Büchi-recognisable whereas Büchi automata are required for their superlogics. The decision procedures for LTL and CTL obtained in this way are therefore not optimal.

Extending MLSolver with another modal fixpoint logic is relatively easy. One has to provide an abstract data type modelling formulas of that logic and to implement the tableaux rules for that logic as well as the nondeterministic automata recognising bad branches of these tableaux. The remaining tasks, i.e. the automata determinisation and construction of parity games, as well as the decoding of the winning strategy into a model / countermodel for the input formula can use available routines.

4 Benchmarks

In this section we describe hand-crafted benchmarks formalised in any of the logics that are currently supported by MLSolver and report on performance tests on these benchmarks.

Note that the series presented in the tables to follow do not start with the smallest instances. We only present instances with non-negligable running times. On the other hand, the solving of larger instances not presented in the tables anymore has experienced time-outs after one hour, marked †, or the constructed games were already to large to be stored in the heap space.

All tests have been carried out on a 64-bit machine with four quad-core Opteron^{TM} CPUs and 128GB RAM space. The algorithm used to solve the resulting parity games is Zielonka's recursive one [31]. It has proved to be generally

				Without Compaction			With Compaction		
n	$\lvert\varphi_n\rvert$	$\lvert$NBA$\rvert$	$\lvert$DPA$\rvert$	$\lvert$Game$\rvert$	t_{generate}	t_{solve}	$\lvert$Game$\rvert$	t_{generate}	t_{solve}
1	34	6	15	29	0.00s	0.00s	8	0.00s	0.00s
2	87	33	892	1,623	0.04s	0.01s	343	0.04s	0.00s
3	140	69	10,077	21,435	1.11s	0.18s	4,999	1.24s	0.06s
4	201	116	231,884	556,552	86.36s	22.54s	133,602	132.10s	7.18s

Fig. 2. Runtime results on Hard Formulas with Fixpoint Alternation

the best among those implemented in PGSolver [9].

Hard Formulas with Fixpoint Alternation

It is well-known that alternation between least and greatest fixpoint quantifiers causes formulas to be difficult to solve. We therefore use for benchmarking a family of formulas – in the linear-time μ-calculus – that features increasing alternation of fixpoint quantifiers. It is built as follows.

For every $n \geq 1$, $\psi_n := \nu X. \bigcirc X \wedge \bigvee_{i=1}^{n} q_i \wedge \bigwedge_{j \neq i} \neg q_j$ expresses that in every state of a model exactly one of the propositions $q_1, \ldots, q_n$ is true. Let

$$\varphi_n \;:=\; \psi_n \;\rightarrow\; \Big((\sigma X_n \ldots \nu X_2 . \mu X_1 . \bigwedge_{i=1}^{n} q_i \rightarrow \bigcirc X_i) \;\leftrightarrow\; \bigvee_{i \text{ even}} (\nu X.(\mu Y. q_i \vee \bigcirc Y) \wedge \bigcirc X) \wedge \bigwedge_{\substack{j>i \\ j \text{ odd}}} \mu X.(\nu Y. \neg q_j \wedge \bigcirc Y) \vee \bigcirc X \Big)$$

where $\sigma = \nu$ if n is even, otherwise $\sigma = \mu$. Note that φ_n has alternation depth $n-1$. It expresses that a deterministic parity condition is expressible as a nondeterministic Büchi condition. The left part of the bi-implication states that the greatest index i s.t. infinitely many states are labeled q_i, is even. The right part states that there is an even index i with q_i occurring infinitely often and no q_j doing so if j is odd and greater than i. Intuitively, these two are equivalent. For technical reasons it is necessary to demand uniqueness of propositions at each state.

The times needed to generate and solve the games resulting from determining validity of φ_n as well as their sizes are presented in Fig. 2. The columns in the left part show the index n of the instance, the size of φ_n, as well as the sizes of the thread-finding automaton before and after determinisation. Note that $|\varphi_n|$ grows quadratically in n and validity checking for the linear-time μ-calculus is PSPACE-complete [24,28].

The middle and right parts contain the size of the resulting game as well as the time it takes to generate and solve it. This is done in two different ways: "without compaction" maps every tableau node annotated with a state of the deterministic automaton to a node in the parity game, "with compaction" does so only for those nodes that precede an application of the modal rule as explained in Sect. 2.4 above. As one can see, this reduces the size of the resulting parity game and makes them easier to solve, but generating the games becomes harder.

					Without Compaction			With Compaction		
Δ	n	$\lvert\Delta\rvert$	$\lvert$NBA$\rvert$	$\lvert$DPA$\rvert$	$\lvert$Game$\rvert$	t_{generate}	t_{solve}	$\lvert$Game$\rvert$	t_{generate}	t_{solve}
φ_n	200	$1,413$	$1,404$	$1,404$	$403,005$	65.09s	11.80s	$1,203$	31.82s	0.20s
	460	$3,233$	$3,224$	$3,224$	$2,122,905$	$3,238.67$s	34.79s	$2,763$	491.52s	1.48s
	470	$3,303$	$3,294$	$3,294$	†	†	†	$2,823$	533.80s	1.40s
	600	$4,213$	$4,204$	$4,204$	†	†	†	$3,603$	$1,182.64$s	2.69s
	840	$5,893$	$5,884$	$5,884$	†	†	†	$5,043$	$3,431.83$s	7.80s
ψ_n	50	363	5	5	$75,472$	20.77s	0.54s	9	11.40s	0.00s
	100	713	5	5	$300,922$	358.87s	5.60s	9	172.05s	0.00s
	160	$1,133$	5	5	$769,462$	$2,944.04$s	41.39s	9	$1,137.01$s	0.00s
	170	$1,203$	5	5	†	†	†	9	$1,458.52$s	0.00s
	210	$1,483$	5	5	†	†	†	9	$3,544.81$s	0.00s

Fig. 3. Runtime results on Nesting Stars in PDL

Nesting Stars in PDL

It is a well-known fact that the nesting-depth of Kleene stars in the programs of a PDL-formula causes formulas to be difficult to be solved. Particularly, the decision procedure has to make sure that certain formulas are not unfolded infinitely often without also seeing infinitely many applications of the modal rule.

We therefore consider two simple families of formulas that feature programs with deep nestings of Kleene stars. Let $\alpha_0 := \mathtt{tt}?^*$ and $\alpha_{n+1} := (a^*\alpha_n b^*)^*$ and

$$\varphi_n \;:=\; \langle (a \cup b)^* \rangle q \vee [\alpha_n]\neg q \qquad \psi_n \;:=\; \langle \alpha_n \rangle q \vee [(a \cup b)^*]\neg q$$

for $n \geq 0$. Note that $\alpha_n \equiv (a \cup b)^*$ for all $n \geq 1$ but not for $n = 0$. Hence, φ_n and ψ_n are valid for $n \geq 1$. However, in φ_n the nested Kleene stars occur inside a box formula which is a greatest fixpoint construct. In ψ_n they occur inside a diamond formula which makes it a least fixpoint construct. Since we are looking at validity, the involved deterministic automata need to trace ν-threads, and φ_n has a much richer ν-thread structure than ψ_n.

The times needed to generate and solve the resulting games as well as their sizes are presented in Fig. 3. A few aspects are worth noting. First of all, the sizes of the determinised thread-finding automata equal those of the original nondeterministic ones because of the structure of the formula: the latter are deterministic already. This shows that determinisation need not always be a problem in this approach. Also, note that one may expect φ_n to be harder to prove valid than ψ_n because of the richer thread structure. However, the simpler program inside the box operator leads to less branching in the tableaux which explains the better managability of those formulas. This, however, is not an artefact of the automata-theory involved but of the underlying tableaux. Hence, this benchmarking family shows that the supposedly difficult automata-theoretic determinisation may actually be much less of a problem than one faces using a tableau structure for satisfiability / validity.

An Example from the Model Checking Domain

MLSolver is also able to solve model checking problems by reading a given transition system and combining it with a formula specification that is to be verified in the transition system. We benchmark a simple fairness verification problem. States of a transition system modelling an *elevator* for n floors are of type

			Without Compaction			With Compaction		
	n	|TS|	|Game|	t_{generate}	t_{solve}	|Game|	t_{generate}	t_{solve}
FIFO	5	1,307	85,570	1.54s	0.81s	19,263	0.66s	0.20s
	6	9,028	606,730	14.59s	7.30s	138,308	5.64s	2.32s
	7	71,815	4,914,794	247.61s	127.51s	1,130,884	57.57s	27.14s
	8	645,352	†	†	†	10,370,665	1,465.59s	600.84s
LIFO	5	1,363	89,204	1.68s	0.94s	20,126	0.80s	0.32s
	6	9,288	624,637	16.02s	8.61s	142,720	7.30s	3.14s
	7	73,065	5,008,902	288.39s	88.12s	1,154,799	83.45s	39.59s
	8	651,168	†	†	†	10,505,651	2,342.61s	1,088.88s

Fig. 4. Runtime results on the example from the Model Checking Domain

$\{1,\ldots,n\} \times \{\mathsf{o},\mathsf{c}\} \times (\bigcup\{\mathit{Perm}(S) \mid S \subseteq \{1,\ldots,n\}\})$. The first component describes the current position of the elevator as one of the floors. The second component indicates whether the door is *open* or *closed*. The third component – a permutation of a subset of all available floors – holds the *requests*, i.e. those floors that should be served next. The transitions on these are as follows.

- At any moment, any request or none can be issued. For simplicity reasons, we assume that at most one floor is added to the requests per transition. Note that nondeterministically, no request can be issued, and a request for a certain floor that is already contained in the current requests does not change them.
- If the door is open then it is closed in the next step, the current floor does not change.
- If it is closed, the elevator moves one floor (up or down) into the direction of the first request. If the floor reached that way is among the requested ones, the door is opened and that floor is removed from the current requests. Otherwise, the door remains closed.

Proposition *isPressed* holds in any state s.t. the request list contains the number n, and *isAt* holds in a state where the current floor is n. We consider two different implementations of this elevator model: the first one stores requests in FIFO style, the second in LIFO style.

Both implementations are checked against the CTL* formula $\mathtt{A}(\mathtt{GF}\mathit{isPressed} \rightarrow \mathtt{GF}\mathit{isAt})$. Hence, this formula requires all runs of the elevator to satisfy the following fairness property: if the top floor is requested infinitely often then it is being served infinitely often. Note that the FIFO implementation encodes a positive instance of the model checking problem whereas FILO encodes a negative one.

The times needed to solve them as well as their sizes are presented in Fig. 4. It shows that this method is capable of doing model checking for non-trivial properties and large transition systems, here more than half a million states. The table does not show the sizes of the involved automata because they are independent of n since the formula expressing the desired correctness property is fixed, and the thread-finding automata only depend on the formula in CTL* model checking. The nondeterministic one has 8 states, the determinised one 27.

					Without Compaction			With Compaction		
Δ	n	$\lvert\Delta\rvert$	$\lvert$NBA$\rvert$	$\lvert$DPA$\rvert$	$\lvert$Game$\rvert$	t_{generate}	t_{solve}	$\lvert$Game$\rvert$	t_{generate}	t_{solve}
δ_n^0	1	51	143	3,529	12,679	0.57s	0.11s	1,071	0.37s	0.01s
	2	70	224	13,786	95,720	5.18s	1.18s	5,559	2.85s	0.08s
	3	89	321	67,743	928,931	71.00s	29.49s	65,079	39.81s	2.38s
	4	108	434	235,290	6,031,198	1,007.19s	611.74s	286,450	368.11s	61.35s
δ_n^1	4	83	74	35,591	89,652	7.86s	1.04s	8,853	4.37s	0.13s
	5	97	83	154,399	592,759	75.49s	14.24s	67,269	37.58s	3.00s
	6	111	92	265,252	929,756	155.37s	29.23s	86,237	80.35s	3.90s
	7	125	101	1,110,031	6,070,401	2,431.73s	895.97s	665,915	1,194.09s	43.86s
	8	139	110	1,768,900	†	†	†	772,587	2,601.78s	72.14s
δ_n^2	1	32	35	160	318	0.01s	0.00s	65	0.01s	0.00s
	2	46	59	2,968	8,673	0.38s	0.05s	1,114	0.42s	0.01s
	3	60	81	12,994	53,792	3.00s	0.42s	5,050	4.14s	0.08s

Fig. 5. Runtime results on Difficult Temporal Formulas

Difficult Temporal Formulas

It is well-known that limit closure – the fact that the limit of an infinite sequence of prefix-sharing paths in a transition system is again a path in this system – is one of the major problems in devising a decision procedure for CTL* [21]. This principle is expressible in CTL* as $LC^*(\phi,\psi) := \mathtt{AG}(\mathtt{E}\psi \to \mathtt{EX}((\mathtt{E}\varphi)\mathtt{UE}\psi)) \wedge \mathtt{E}\psi \to \mathtt{EG}((\mathtt{E}\varphi)\mathtt{UE}\psi)$ where φ and ψ are arbitrary (not necessarily state) formulas. CTL can express a restricted version of that: $LC(\psi) := \mathtt{AG}(\psi \to \mathtt{EX}\psi) \wedge \psi \to \mathtt{EG}\psi$. For the benchmarking, we consider the following families of formulas.

$$\delta_n^0 \;:=\; LC^*(\varphi_n,\psi_n) \qquad \delta_n^1 \;:=\; LC^*(\mathtt{tt},\psi_n) \qquad \delta_n^2 \;:=\; LC(\psi_n)$$

where $\varphi_n := \mathtt{G}(\bigvee_{i\leq n} \neg q_i)$, $\psi_0 := q_0$, $\psi_{2n+1} := q_{2n+1} \wedge \mathtt{X}\psi_{2n}$, and $\psi_{2n+2} := q_{2n+2} \vee \mathtt{X}\psi_{2n+1}$. It is reasonable to assume that these formulas are relatively difficult to prove valid.

The times needed to generate and solve the resulting games as well as their sizes are presented in Fig. 5.

5 Conclusion and Further Work

The implementation of MLSolver and some of the benchmarks show that the combined tableaux-automata way of satisfiability and validity solving for modal fixpoint logics is viable. Even difficult logics like CTL* and the modal μ-calculus can be tackled this way. However, the benchmarks also show a significant discrepancy between the time that is required to generate the parity games and the time that is required to solve them. There is no question that solving the games is not really the problem, but building the tableaux as well as the associated automata. The benchmarks particularly show that there are basically two difficulties in satisfiability and validity solving for such logics.

The first and most obvious difficulty is that of excluding branches with μ-threads. The automata-theoretic approach we follow here is theoretically elegant and appealing because it applies to a whole variety of logics, as opposed to ad-hoc solutions for one specific logic. The benchmarks reveal a great necessity for optimisations in the determinisation procedures, though. These are theoretically well-understood but practically not optimal yet. The reductions employed here would, for example, benefit from a built-in on-the-fly minimisation of the deterministic automata. It is not clear though, whether this is possible and how to do that.

Another difficulty which is not necessarily exhibited by the benchmarks presented here is propositional reasoning. It is easy to construct formulas that model binary counters for example for which the construction of parity games essentially transforms them into exponentially larger disjunctive or conjunctive normal form. Deciding these formulas is then difficult purely because of the size of the games.

It is planned to extend and optimise MLSolver in the furture in various ways. As mentioned above, LTL and CTL are currently being supported but only in a non-optimal way. Implementing separate modules for LTL and CTL is not difficult. This will also create a set-up which allows to quantify exactly the benefit of using co-Büchi over Büchi automata. There are also other logics (graded μ-calculus, probabilistic μ-calculus, etc.) for which this approach works in theory [4], and they can be implemented in MLSolver as well.

A significant disadvantage is also the creation of the entire parity game before it is being solved. This is in contrast to tableau-based solvers for example, and is done because so far there is only one algorithm for solving parity games which works on-the-fly, i.e. generates the game graph whilst solving it [25]. However, it turns out that in practice it is often much less efficient than global algorithms [31,30]. On the other hand, it remains to be seen whether or not the local algorithm may perform better on those graphs that represent satisfiability and validity problems, or whether or not the good global algorithms can be made to work on-the-fly.

Finally, there is another determinisation procedure for nondeterministic Büchi automata which is not based on tree-like states [13]. It remains to be seen whether this leads to more efficient determinisation and therefore quicker generation of parity games. Another way of avoiding such Safra-like determinisation constructions transforms the μ-thread recognising nondeterministic automata into, again, nondeterministic Büchi automata which are exponentially larger but can be used in this game setting instead of deterministic ones [11]. They are presumed to be easier to create than the deterministic ones which can be put to the test in this setting as well.

References

[1] F. Baader. Augmenting concept languages by transitive closure of roles: An alternative to terminological cycles. In *Proc. 12th Int. Joint Conf. on Artificial Intelligence, IJCAI'91*, pages 446–451. Morgan Kaufmann, 1991.

[2] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69(1):5–40, 2001.

[3] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL*. In *Proc. 10th Symp. on Logic in Computer Science, LICS'95*, pages 388–397, San Diego, CA, USA, 1995. IEEE.

[4] C. Cirstea, C. Kupke, and D. Pattinson. EXPTIME tableaux for coalgebraic mu-calculi. In *Proc. 18th Int. EACSL Annual Conference on Computer Science Logic, CSL'09*, volume ?? of *LNCS*. To appear.

[5] C. Dax, M. Hofmann, and M. Lange. A proof system for the linear time μ-calculus. In *Proc. 26th Conf. on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'06*, volume 4337 of *LNCS*, pages 274–285. Springer, 2006.

[6] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30:1–24, 1985.

[7] E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[8] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[9] O. Friedmann and M. Lange. Solving parity games in practice. In *Proc. 7th Int. Symp. on Automated Technology for Verification and Analysis, ATVA'09*, volume 5799 of *LNCS*, pages 182–196, 2009. To appear.

[10] G. De Giacomo and M. Lenzerini. Boosting the correspondence between description logics and propositional dynamic logics. In *Proc. of the 12th National Conference on Artificial Intelligence, AAAI'94*, pages 205–212. AAAI-Press/the MIT-Press, 1994.

[11] T. A. Henzinger and N. Piterman. Solving games without determinization. In *Proc. 20th Int. Conf. on Computer Science Logic, CSL'06*, volume 4207 of *LNCS*, pages 395–410. Springer, 2006.

[12] M. Jurdziński. Small progress measures for solving parity games. In *Proc. 17th Ann. Symp. on Theoretical Aspects of Computer Science, STACS'00*, volume 1770 of *LNCS*, pages 290–301. Springer, 2000.

[13] D. Kähler and Th. Wilke. Complementation, disambiguation, and determinization of Büchi automata unified. In *Proc. 35th Int. Coll. on Automata, Languages and Programming, ICALP'08*, volume 5125 of *LNCS*, pages 724–735. Springer, 2008.

[14] D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, December 1983.

[15] M. Lange and C. Stirling. Focus games for satisfiability and completeness of temporal logic. In *Proc. 16th Symp. on Logic in Computer Science, LICS'01*, Boston, MA, USA, 2001. IEEE.

[16] M. Lange and C. Stirling. Model checking games for branching time logics. *Journal of Logic and Computation*, 12(4):623–639, 2002.

[17] S. Miyano and T. Hayashi. Alternating finite automata on omega-words. *TCS*, 32(3):321–330, 1984.

[18] D. Niwiński and I. Walukiewicz. Games for the μ-calulus. *TCS*, 163:99–116, 1997.

[19] N. Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *Proc. 21st Symp. on Logic in Computer Science, LICS'06*, pages 255–264. IEEE Computer Society, 2006.

[20] A. Pnueli. The temporal logic of programs. In *Proc. 18th Symp. on Foundations of Computer Science, FOCS'77*, pages 46–57, Providence, RI, USA, 1977. IEEE.

[21] M. Reynolds. A tableau for bundled CTL*. *J. Log. Comput*, 17(1):117–132, 2007.

[22] S. Safra. Exponential determinization for ω-automata with strong-fairness acceptance condition (extended abstract). In *Proc. 24th Annual ACM Symposium on the Theory of Computing, STOC'92*, pages 275–282. ACM Press, 1992.

[23] S. Schewe. An optimal strategy improvement algorithm for solving parity and payoff games. In *Proc. 17th Ann. Conf. on Computer Science Logic, CSL'08*, volume 5213 of *LNCS*, pages 369–384. Springer, 2008.

[24] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the Association for Computing Machinery*, 32(3):733–749, 1985.

[25] P. Stevens and C. Stirling. Practical model-checking using games. In B. Steffen, editor, *Proc. 4th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'98*, volume 1384 of *LNCS*, pages 85–101. Springer, 1998.

[26] C. Stirling. Local model checking games. In *Proc. 6th Conf. on Concurrency Theory, CONCUR'95*, volume 962 of *LNCS*, pages 1–11. Springer, 1995.

[27] Balder ten Cate. The expressivity of XPath with transitive closure. In *Proc. 25th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems, PODS'06*, pages 328–337. ACM Press, 2006.

[28] M. Y. Vardi. A temporal fixpoint calculus. In ACM, editor, *Proc. Conf. on Principles of Programming Languages, POPL'88*, pages 250–259, NY, USA, 1988. ACM Press.

[29] M. Y. Vardi. *An Automata-Theoretic Approach to Linear Temporal Logic*, volume 1043 of *LNCS*, pages 238–266. Springer, 1996.

[30] J. Vöge and M. Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *Proc. 12th Int. Conf. on Computer Aided Verification, CAV'00*, volume 1855 of *LNCS*, pages 202–215. Springer, 2000.

[31] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *TCS*, 200(1–2):135–183, 1998.