Analyzing LTL Model Checking Techniques for Plan Synthesis and Controller Synthesis (Work in Progress)

Sylvain Kerjean, Froduald Kabanza, Richard St-Denis [1,2]

Département d'informatique
Université de Sherbrooke
Sherbrooke (Québec), Canada J1K 2R1

Sylvie Thiébaux [3]

National ICT Australia and Computer Science Laboratory
The Australian National University
Canberra ACT 0200 Australia

Abstract

In this paper, we present alternative means of handling invariances in reachability testing, either by formula progression or compilation into Büchi automata. These alternatives are presented in connection with three different applications of model checking: verification, plan synthesis as well as heuristic guidance of AI planning, and controller synthesis. We include results from benchmarks obtained from preparatory experiments with model checking using a family of LTL2Büchi translators and formula progression.

Key words: LTL2Büchi translators, formula progression, model checking, verification, synthesis.

1 Introduction

Model checking with Linear Temporal Logic (LTL) has demonstrated interesting potentials for program verification, plan synthesis, and controller synthesis problems. In these problems, we have a system modelled as a graph of states, representing its possible execution sequences. To validate the correctness of

[1] This Research is supported by grants from the Natural Sciences and Engineering Research Council of Canada.

[2] Email: {Sylvain.Kerjean,Froduald.Kabanza,Richard.St-Denis}@USherbrooke.ca

[3] Email: Sylvie.Thiebaux@anu.edu.au

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the system with respect to some behavior, we can express the behavior as an LTL formula and then use LTL model checking to perform the validation [24,10,12]. To control the system along desirable paths of a goal behavior, we can express the goal as an LTL formula and use LTL model checking to synthesize a plan of actions that indicates the action to execute at every point so that the system exhibits a behavior, possibly optimal, satisfying the goal [2,13,22]. The same technique can be used to implement search control strategies for a planning system, that is, rules of thumb about how plans should be efficiently computed in a particular application domain [3]. In order to prevent the system from engaging into undesirable behaviors, we can express desirable behaviors as an LTL formula and use LTL model checking to synthesize a controller that prohibits undesirable events [4,5].

The model checker underlying these program validation, plan synthesis, and controller synthesis frameworks can be implemented in different ways. One approach is to use an LTL2Büchi translator [17,8,11,12,9] to translate the LTL formula into an equivalent Büchi automaton and then use it on the fly to perform the verification process, plan synthesis process, or controller synthesis process. Another alternative, so far used in plan synthesis [2,13,22] and controller synthesis [4,5] is to progress the LTL formula along state sequences generated by the search engine that underlies the verification, plan synthesis, or controller synthesis program. While fundamentally rooted in the automata theoretic characterization of LTL formulas, formula progression is very simply described, abstracting away automata theoretic concepts. This simplicity presumably explains, at least partially, its appeal in AI planning, exemplified for instance by its adaptation to a non modal temporal logic, namely the Temporal Action Language in TALplanner [16]. In this context, the distinction between verification and plan or controller synthesis is equivalent to the distinction between formula compilation and formula progression.

The main question we want to address is to determine which of these two approaches (model checking via explicit LTL2Büchi translation vs formula progression) is most suitable in each context. Despite significant improvements over the last few years, LTL2Büchi translators still face a potential blow up. This may not be very dramatic for program verification since we are usually concerned with off-line validation processes, where run times in minutes are acceptable. In plan synthesis, for robotics applications for example [6], run times have to be in milliseconds. We may expect LTL2Büchi translators to be viable only for simple formulas, but whether this holds for more complex goals (e.g., fairness properties as in [9], but with quantification) or complex search control strategies for planning problem [2] has to be verified.

In plan synthesis, search control strategies are specified once for all, as part of the specification of the planning application domain. They remain unchanged for subsequent calls of the planner on given initial-state/goal problems [2,14]. In this case, being able to pre-compile LTL formulas into some kind of Büchi automaton would provide some savings by avoiding to re-

compute some sub-formula progressions, something that happens often with the formula progression algorithm. To a certain extent, this is what Kvarnström and Magnusson try to do in TALplanner [16], but using somewhat ad-hoc methods, which nevertheless lead to significant improvements. More specifically, a context is derived at each step from search control rules and the current state, and used in an inference procedure. This procedure reduces the possible values to assign to propositional variables or domain-quantified variables by using all classical tautologies and techniques of first-order predicate logic. This kind of partial compilation also permits to reduce the number of disjuncts and conjuncts in boolean formulas (as well as the size of finite domains to explore in case of quantified variables), because specifications are often redundant in the planning area. Thus, the size of a formula drastically shrinks during its progression.

A systematic LTL2Büchi translation seems a more principled approach of doing this. Unfortunately, TLPLAN search control strategies are specified using bounded-quantification. On one hand, even if this is equivalent to having finitely many instantiated, but overwhelmingly large LTL formulas, we fear that running an LTL2Büchi translator on such formulas may cause an impractical blow up. On the other hand, so far we do not know of any efficient LTL2Büchi translator for bounded-quantified LTL formulas. If it turns out that no such a translator exists, one of our goals will be to try developing one. If it exists, then we will have to implement it and compare it with formula progression.

Since we are still at the early stage of our investigation, only preliminary results are presented. More details on the background behind this investigation are, however, given, namely LTL syntax and semantics, formula progression algorithm, LTL2Büchi translators, and a deeper insight into the program verification, plan synthesis, and controller synthesis frameworks.

2 Linear Temporal Logic

We use LTL with bounded quantification. The language is defined by starting with a first-order language containing collections of constants, functions, and predicate symbols, along with variables, quantifiers, propositional constants $\top$ and $\bot$, and the usual connectives $\neg$ and $\wedge$. To these are added the unary GOAL modality, the unary temporal modality $\bigcirc$ (next), and the binary temporal modality U (until). Compounding under all those operators is unrestricted (in particular quantifying into temporal contexts is allowed), except that $\bigcirc$, U, and GOAL may not occur inside the scope of GOAL. The motivation for adding the GOAL modality is to distinguish between LTL goals to fulfill and search control strategies (about how to achieve goals) also expressed in LTL [2,3,14]. In the following, we assume that all variables are bounded.

LTL is interpreted over possibly infinite sequences of the form $\Gamma = \Gamma_0\Gamma_1\cdots$, where each Γ_i is a pair (s_i, G_i) consisting of a current world state s_i and a

set of current goal states G_i. The idea is that at stage Γ_i of the sequence we are in world state s_i and are trying to reach one of the states in G_i. The temporal modalities allow the expression of general properties of such sequences. Intuitively, an atemporal formula f means that f holds now (that is, at the first stage Γ_0 of the sequence); $\bigcirc f$ means that f holds next (that is, f, which may itself contain other temporal operators, is true of the suffix $\Gamma(1)$); $f_1 \,\mathsf{U}\, f_2$ means that f_2 will be true at some future stage and that f_1 will be true until then (that is f_2 is true of some suffix $\Gamma(i)$ and f_1 must be true of the all the suffixes starting before Γ_i); and $\mathsf{GOAL}(f)$ means that f is true in all the goal states G_i at the current stage, while an atemporal formula outside the scope of GOAL refers to the world state s_i at the current stage.

The goal component of a sequence is irrelevant in normal program verification problems and can be dropped. In the original TLPLAN implementation [3], the goal component is invariant (it is the final goal state one is trying to reach), therefore it can be dropped as well, and replaced by a global variable. In this case, the infinite sequences of states we consider consist of a finite sequence and of an idling final state. The recent extension of TLPLAN is more general and allows dynamically changing goal states and cyclic plans, requiring state sequences as just introduced for interpreting LTL formulas [14].

Formally, let a finite set D be the domain of discourse, which we assume constant across all world and goal states, and let Γ a sequence defined as above, with $\Gamma_i = (s_i, G_i)$. We say that a formula f is true of Γ (noted $\Gamma \models f$) iff $(\Gamma, 0) \models f$, where truth is defined recursively as follows:

- if f is an atomic formula, $(\Gamma, i) \models f$ iff $s_i \models f$ according to the standard interpretation rules for first-order predicate logic;
- $(\Gamma, i) \models \mathsf{GOAL}(f)$ iff for all $g_i \in G_i$, $g_i \models f$ according to the standard interpretation rules for first-order predicate logic;
- $(\Gamma, i) \models \forall x\ f$ iff $(\Gamma, i) \models f[x/d]$ for all $d \in D$, where $f[x/d]$ denotes the substitution of d's name for x in f;
- $(\Gamma, i) \models \neg f$ iff $(\Gamma, i) \not\models f$;
- $(\Gamma, i) \models f_1 \wedge f_2$ iff $(\Gamma, i) \models f_1$ and $(\Gamma, i) \models f_2$;
- $(\Gamma, i) \models \bigcirc f$ iff $(\Gamma, i+1) \models f$; and
- $(\Gamma, i) \models f_1 \,\mathsf{U}\, f_2$ iff there exists $j \geq i$ such that $(\Gamma, j) \models f_2$ and for all $k, i \leq k < j$, $(\Gamma, k) \models f_1$.

A bounded quantified formula, as used in TLPLAN, can be replaced by an equivalent one corresponding to finitely many conjunctions of instances of the formula obtained by replacing variables with the relevant objects in the quantification domain. But the resulting formula may not be as efficient as the original. In fact, TLPLAN implementation cleverly exploits the bounded quantification by making instantiations only when relevant.

It should be noted that this logic differs from QPTL [20] in which quantified variables are allowed to change interpretation from state to state.

3 Formula Progression Algorithm

When generating a plan using a search process (as is the case in TLPLAN), we can check plan prefixes on the fly and prune them as soon as violation is detected, using an incremental technique called *formula progression*, because it progresses or pushes the formula through the sequence Γ induced by the plan prefix.

The idea behind formula progression is to decompose an LTL formula f into a requirement about the present Γ_i, which can be checked straight away, and a requirement about the future that will have to hold of the yet unavailable suffix. That is, formula progression looks at Γ_i and f, and produces a new formula $\text{fprog}(\Gamma_i, f)$ such that $(\Gamma, i) \models f$ iff $(\Gamma, i+1) \models \text{fprog}(\Gamma_i, f)$. If Γ_i violates the part of f that refers to the present then $\text{fprog}(\Gamma_i, f) = \bot$ and the plan prefix can be pruned, otherwise, the prefix will be extended and the process will be repeated with Γ_{i+1} and the new formula $\text{fprog}(\Gamma_i, f)$. In our framework, where world states are paired with sets of goal states in Γ, i.e., $\Gamma_i = (s_i, G_i)$, $\text{fprog}(\Gamma_i, f)$ is defined by Algorithm 1.

Algorithm 1. Progression

$$
\begin{array}{lcl}
\text{fprog}(\Gamma_i, f) & = & \top \text{ iff } s_i \models f \text{ else } \bot, \text{ for } f \text{ atomic} \\
\text{fprog}(\Gamma_i, \mathsf{GOAL}(f)) & = & \top \text{ iff } g_i \models f \text{ for all } g_i \in G_i \text{ else } \bot \\
\text{fprog}(\Gamma_i, \forall x\ f) & = & \text{fprog}(\bigwedge_{d \in D} f[x/d]) \\
\text{fprog}(\Gamma_i, \neg f) & = & \neg \text{fprog}(\Gamma_i, f) \\
\text{fprog}(\Gamma_i, f_1 \wedge f_2) & = & \text{fprog}(\Gamma_i, f_1) \wedge \text{fprog}(\Gamma_i, f_2) \\
\text{fprog}(\Gamma_i, \bigcirc f) & = & f \\
\text{fprog}(\Gamma_i, f_1 \,\mathsf{U}\, f_2) & = & \text{fprog}(\Gamma_i, f_2) \vee (\text{fprog}(\Gamma_i, f_1) \wedge f_1 \,\mathsf{U}\, f_2)
\end{array}
$$

The function fprog runs in linear time in $|f| \times |G_i| \times |D|$. Progression can be seen as a delayed version of the LTL semantics, which is useful when the elements of the sequence Γ become available one at a time (as is the case with a search engine in a planning process), in that it defers the evaluation of the part of the formula that refers to the future to the point where the next element becomes available. However, since progression only applies to finite prefixes, it is only able to check safety properties, such as those involved in search control knowledge. In particular, it is unable to detect violation of liveness properties involved in temporally extended goals, as these can only be violated by an infinite sequence. Such properties will progress to $\top$ when satisfied, but will never progress to $\bot$. Handling such cases require an extension that explicitly checks for U formulas [13].

4 LTL2Büchi Translators

An alternative to formula progression would be to translate the formula into an equivalent Büchi automaton that accepts exactly sequences of states satisfied by the formula [24]. A Büchi automaton is a nondeterministic automaton over infinite words. The main difference with an automaton over finite words is that accepting a word requires an accepting state to be reached infinitely often, rather than just once.

Formally a Büchi automaton is a tuple $B = (\Sigma, Q, \Delta, q_0, Q_F)$, where Σ is a finite alphabet, Q is a finite set of automaton states, $\Delta : Q \times \Sigma \mapsto 2^Q$ is a nondeterministic transition function, q_0 is the start state of the automaton, and Q_F is the set of accepting states. A run of B over an infinite word $w = w_0 w_1 \ldots$ is an infinite sequence of automaton states $(q_0, q_1, \ldots)$ where $q_{i+1} = \Delta(q_i, w_i)$. The run is accepting if some accepting state occurs infinitely often in the sequence, that is, if for some $q \in Q_F$, there are infinitely many i's such that $q_i = q$. Word w is accepted by B if there exists an accepting run of B over w, and the language recognized by B is the set of words it accepts.

Given a temporally extended goal as an LTL formula f without GOAL modality and quantifiers, one can build a Büchi automaton recognizing exactly the set of infinite world state sequences that satisfy f. The transitions in this automaton are labelled with world states (i.e., $\Sigma = 2^P$, where P is the set of atomic propositions in f). In the worst case, the number of states in the automaton is exponential in the length of f. It is often small however for many practical cases.

The many methods to build such automata differ by the size of the result and the time needed to produce it. We are currently experimenting with the following translators.

SPIN – This is the translator used in the model checker SPIN [12].

ltl2ba – *ltl2ba* [9] is a very efficient translator, whose efficiency is based on using *very weak alternating automata*, which are in turn transformed in *generalized Büchi automata* (i.e., with multiple acceptance conditions on transitions instead of states). Finally the generalized Büchi automata are transformed in standard Büchi automata. Of course simplifications (suppression of inaccessible states, redundant transitions, and equivalent states) are applied at each step.

scheck – *scheck* [17] is an optimization of the Kupferman and Vardi algorithm [15]. The general idea is to start from an empty set of requirements and, by going backwards, compute all possible informative prefixes. An informative prefix is a sub-formula which can permit to determine why a formula failed on a given model. Such informative prefixes can be produced by an automaton derived form a set of sub-formulas. The optimization resides in restricting the sub-formulas in this set to all temporal sub-formulas, children of *next* operators, and possibly the top-level formula.

5 Program Verification, Plan and Controller Synthesis

5.1 Model Checking for Program Verification

In program verification, the general problem is to verify that a given program satisfies a given behavioral property or does not violate a given property (for example, mutual exclusion, absence of deadlock, or absence of livelocks). The general approach is as follows [24]. First, we model concurrent executions by their interleaving to obtain a state transition graph in which transitions model instructions and states represent program variables, flow control points, and program counters. Second, we express the property to be verified on the program as an LTL formula. Third, we translate the LTL formula into a Büchi automaton that accepts state sequences satisfying the formula. Finally, we combine Büchi automaton and the program state transition graph to determine state sequences that satisfy or violate the LTL property.

In practice, the combination of the Büchi automaton with the program transition graph is efficiently done on the fly [10]. The program transition graph is in practice an abstract model of a program in which only key variables are retained (typically variables implementing synchronization and communication among processes). It is automatically obtained from an abstract program modelling language (e.g., Promela in SPIN [12]) or a fully-fledged programming language (e.g., Java Path Finder that translates Java programs into equivalent Promela programs before verifying them [11]).

5.2 Model Checking for Plan Synthesis

In plan synthesis, the general problem is to determine the actions to be executed by an agent (e.g., a robot) at different points in time, in order to achieve a given purposeful task, that is, a goal in the usual artificial intelligence terminology. A planner is thus a program that, given a description of primitive agent's actions (e.g., moving a short distance for a robot, grasping an object, releasing an object being grasped, saying something or displaying a message on the robot's screen) and a goal, computes a plan. Usually the planner is just one component among others that participate in the overall control of the agent. In robotics for example, robot navigation processes and execution monitoring processes can be coordinated with a planner to achieve complex tasks [6].

Planning problems are naturally reduced to model checking problems, explicitly using an LTL2Büchi framework [7], or more implicitly using LTL formula progression. For deterministic planning problems, TLPLAN [2,3] and TALplanner [16] are some of the illustrative implementations of planners using formula progression. For nondeterministic problems without probabilities, SimPlan [13] is one illustrative example, whereas the work by Thiébaux *et al* [22] describes a system for nondeterministic domains with probabilities, using the framework of Markov decision processes.

5.3 Model Checking for Controller Synthesis

The problem in controller synthesis for discrete event systems is very similar to plan synthesis, except that usually one is interested in controlling a given plant (or agent) by just forbidding bad events, so that only legal behaviors remain allowed. From a planning perspective, this is intuitively equivalent to computing all legal plans, whereas in planning we usually want one of them, very often the optimal one. In terms of control theory, we often say that we want a maximal controller, that is, a controller allowing all satisfactory plans. Conditions to solve this problem and synthesis algorithms to automatically derive the maximally permissive controller constitute a part of the *Supervisory Control Theory* originally developed by Ramadge and Wonham [19]. This problem has also been considered as a problem of finding a winning strategy for finite or infinite games [1,18].

This problem can also be solved in the automata theoretic framework by modelling the plant as a state transition system, and the constraint to be imposed on the plant as a Büchi automaton (or, equivalently, as an LTL formula given that we have LTL2Büchi translators). Again a controller is obtained from a combination of the plant transition system and the Büchi automaton. Formula progression has also been adapted to this framework, leading to an algorithm for synthesizing controllers using MTL specifications [4,5]. Several other frameworks use a more direct automata theoretic setting (e.g., [23]).

6 Testing Environment

For the preliminary comparative results presented below, we adapted *lbtt* testing environment [21]. The purpose of this tool is to make automatic comparisons of LTL to Büchi translators in order to test the validity of specific implementations ([17,8,11,12,9]). This is done by model checking the resulting automata on state spaces, which are ordinary DAGS labelled by propositional symbols, with some parameterizable constraints on the paths.

Formulas can be automatically generated or read in a file, with two syntaxes: a prefix form or the Promela syntax [12]. The state spaces can only be randomly generated, but it is possible to impose some kind of path constraints:

random connected graph – The program will ensure that the graphs contain a state from which every other state in the graph can be reached.

random graph – Totally random spaces. It is however required that each state has at least one successor.

random paths – Randomly generated paths, in which the states of the state space are connected into a sequence, and the last state of the sequence is connected to a previous state on the path chosen using a uniform random distribution.

enumerated path – Exhaustively enumerate all paths having a given length and a given number of propositions in each state.

The format accepted is a specific one or the Promela never-claim syntax [12]. Of course all kinds of numeric parameters can be specified in a configuration file (e.g., number of formulas and state spaces generated, size of formulas and state spaces)

The tests for model checking via LTL2Büchi are carried out as follows: i) a formula and its negation are generated; ii) each translator is applied to these formulas; and iii) the Büchi automata are composed with the state space, then the accepting states of products are compared.

In order to use *lbtt* to measure the performance of our program *progress* against the performance of the LTL2Büchi+model checking, we integrated *progress* into *lbtt*.

7 Some Preliminary Experiments

So far we have experimental results from random paths and state spaces. A state space is generated from a set of five propositions (hence, the size of the state space may grow up to 2^5), except for the last experiment (10 propositions). Reported times are the average of 20 random generated formulas or state spaces. For LTL2Büchi translators, the time is the sum of translation time and model checking operation time (i.e., combining the Büchi automaton and the state space). For formula progression, it is just the time for progression since no construction of the automaton is involved. The tests are conducted on a SunOS 5.8 UltraSparc Sun-Fire-280R server. We included a timeout at two seconds, which explains the discontinuities in the graphs.

Figure 1 shows that the formula progression algorithm performs very well on large formula with middle-sized random state spaces, with the sole constraint that each state has a successor. The performance for *progress* is so efficient that its corresponding curve (in red) is barely distinguishable from the horizontal axis.

Figure 2 presents the results when we generate random paths instead of random graphs, that is all paths ends on a cycle. Results are very similar to the previous test.

Figure 3 involves the same test as previously, except we use a more complicated formula. It should be noted that the formula progression algorithm reaches the timeout of two seconds for sizes over 500, not because of a failure to compute the results, but to a failure on reading the state spaces (an implementation limitation that will be easily overcome in future experiments).

Figure 4 gives the results of an experiment inspired from [17]. Here the formula progression algorithm is very efficient due to the small state space, so that the time curve is not noticeable. It must be, however, noted that since random formulas are mostly false on random state spaces, we do not explore

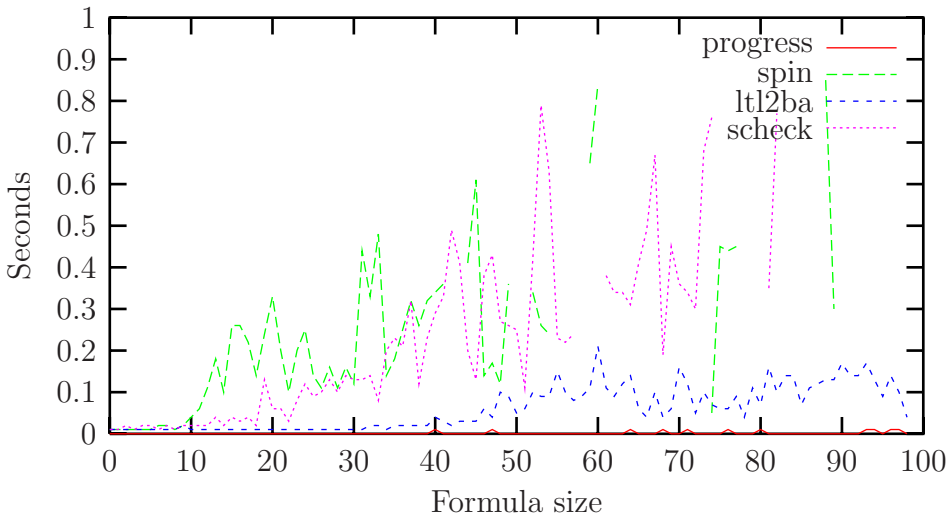

Fig. 1. 150 states (random graphs), formula size form 1 to 100

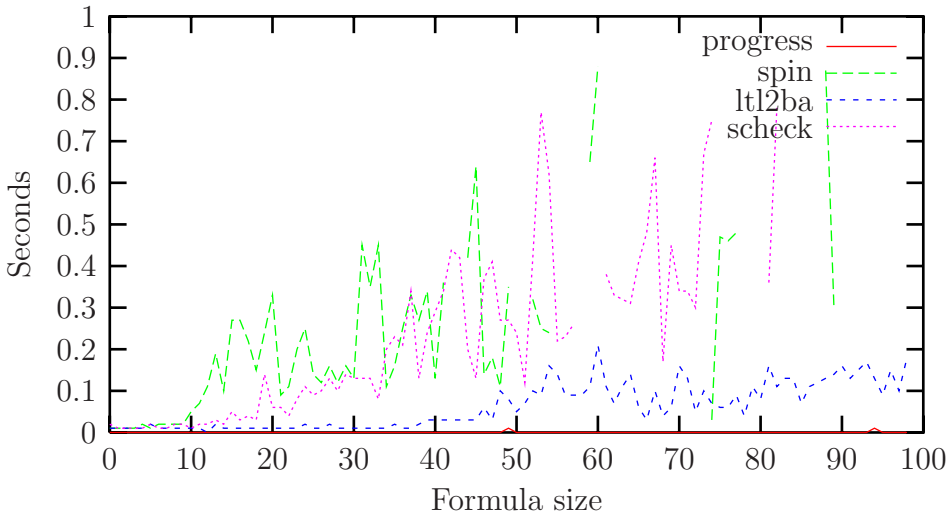

Fig. 2. 150 states (random paths), formula size form 1 to 100

much of the state space. The statistics obtained for *ltl2ba* (dashed blue line) and the formula progression algorithm (red line) are comparable. Again, the curves are barely distinguishable from the horizontal axis.

These results are still very preliminary and involve quite trivial test examples. While the LTL2Büchi translators are highly optimized, the current implementation of formula progression in the *lbtt* test environment is not. We did not not use the C TLPLAN implementation of formula progression. Instead, we found it easier to re-implement the algorithm from scratch for the *lbtt*. It is therefore possible that the good performances of formula progression observed so far, become even better after its optimization.

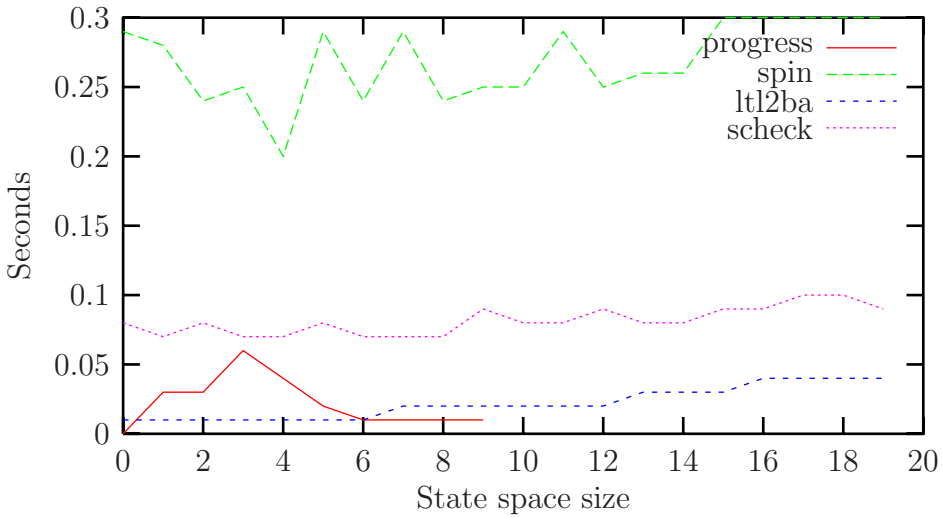

Fig. 3. State space size from 50 to 1000 (by 50), formula size = 20

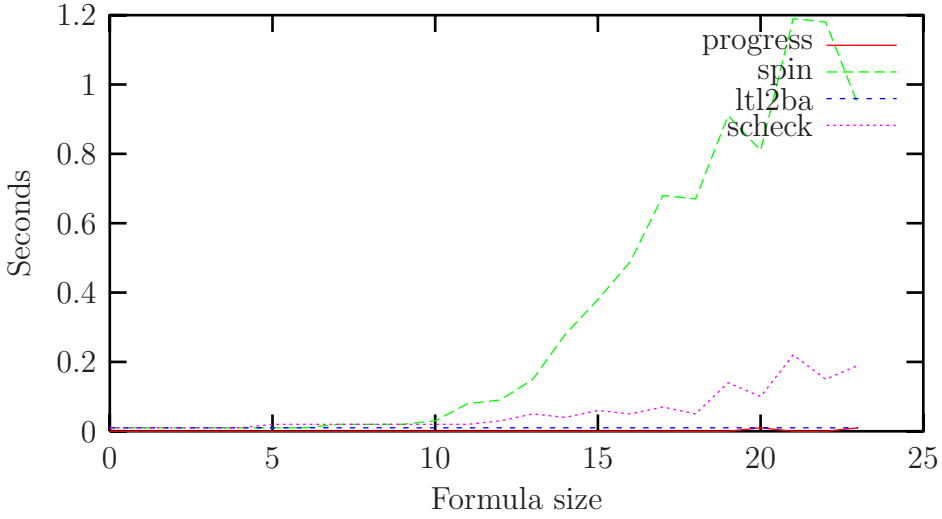

Fig. 4. Five states, formula size from 1 to 24

8 Conclusion

We are implementing benchmarks on more complex tests, randomly generated from the following domains:

catmouse This is an academic problem from the area of controller synthesis [19]. A cat and a mouse are in a maze consisting of some rooms separated by doors. Some of these doors can be controlled. The controller synthesis problem consists in finding a controller that closes key doors among controllable ones to keep the mouse safe (i.e., to guarantee that the cat will never catch it).

trains Another problem from the area of controller synthesis. Many trains are on sections of a circular track. They are only allowed to move one section forward or backward. The controller synthesis problem consists in finding a

controller (i.e., a disablement of events at different states) that ensures that no collision is possible or that there is always a minimum distance between trains.

blocksworld An academic domain from the area artificial intelligence planning [3]. Blocks are placed on an infinite table and can be held by a robot. Blocksworld planning problems consist in finding a plan that is a sequence of block stacking operations, that transforms a given stack configuration into a given goal configuration.

In the longer term, we intend to experiment with even more complex benchmark domains in the areas of program verification, plan synthesis, and controller synthesis and to look for an LTL2Büchi translator for bounded-quantified LTL formulas as used in TLPLAN.

Beside comparing different techniques of model checking and formula progression for planning, we expect this investigation to provide us with better implementations of plan synthesis algorithms, which combine model checking (assuming there are situations where an LTL formula to a Büchi automaton translator performs better) and formula progression (assuming there are situations where formula progression perform better). We have similar expectations with respect to controller synthesis.

References

[1] Asarin, E., O. Maler, and A. Pnueli, "Symbolic Controller Synthesis for Discrete and Timed Systems," Proc. Hybrid Systems II, LNCS **999** (1995), 1–20.

[2] Bacchus, F., and F. Kabanza, *Planning for Temporally Extended Goals*, Annals of Mathematics and Artificial Intelligence **22** (1998), 5–27.

[3] Bacchus, F., and F. Kabanza, *Using Temporal Logics to Express Search Control Knowledge for Planning*, Artificial Intelligence **116** (2000), 123–191.

[4] Barbeau, M., F. Kabanza, and R. St-Denis, "Synthesizing Plant Controllers Using Real-Time Goals," Proc. IJCAI (1995), 791–798.

[5] Barbeau, M., F. Kabanza, and R. St-Denis, *A Method for the Synthesis of Controllers to Handle Safety, Liveness, and Real-Time Constraints*, IEEE Trans. Automat. Contr. **43** (1998), 1543–1559.

[6] Beaudry, E., Y. Brosseau, C. Côté, C. Raïevsky, D. Létourneau, F. Kabanza, and F. Michaud, "Reactive Planning in a Motivated Behavioral Architecture," Proc. AAAI (2005), 1242–1247.

[7] De Giacomo, G., and M. Y. Vardi, "Automata-Theoretic Approach to Planning for Temporally Extended Goals," Proc. ECP (1999), 226–238.

[8] Fritz, C., "Constructing Büchi Automata from Linear Temporal Logic Using Simulation Relations for Alternating Büchi Automata," Proc. CIAA, LNCS **2759** (2003), 35–48.

[9] Gastin, P., and D. Oddoux, "Fast LTL to Büchi Automata Translation," Proc. CAV, LNCS **2102** (2001), 53–65.

[10] Gerth, R., D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly Automatic Verification of Linear Temporal Logic," Proc. PSTV (1995), 3–18.

[11] Giannakopoulou, D., and F. Lerda, "From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata," Proc. FORTE, LNCS **2529** (2002), 308–326.

[12] Holzmann, G. J., "The SPIN Model Checker: Primer and Reference Manual," Addison Wesley Professional (2003).

[13] Kabanza, F., M. Barbeau, and R. St-Denis, *Planning Control Rules for Reactive Agents*, Artificial Intelligence **95** (1997), 67–113.

[14] Kabanza, F., and S. Thiébaux, "Search Control in Planning for Temporally Extended Goals," Proc. ICAPS (2005), 130–139.

[15] Kupferman, O., and M. Y. Vardi, *Model Checking of Safety Properties*, Formal Methods in System Design **19** (2001), 291-314.

[16] Kvarnström, J., and M. Magnusson, *TALplanner in IPC-2002: Extensions and Control Rules*, Journal of Artificial Intelligence Research **20** (2003), 343–377.

[17] Latvala, T., "Efficient Model Checking of Safety Properties," Proc. 10th Int. SPIN Workshop, LNCS **2648** (2003), 74–88.

[18] Maler, O., A. Pnueli, and J. Sifakis, "On the Synthesis of Discrete Controllers for Timed Systems," Proc. of 12th Annual Symp. on Theoritical Aspects of Computer Science, LNCS, **900** (1995) 229–242.

[19] Ramadge, P. J. G., and W. M. Wonham, *The Control of Discrete Event Systems*, Proc. IEEE, **77** (1989), 81–98.

[20] Sistla, A. P., M. Y. Vardi, and P. Wolper, *The Complementation Problem for Büchi Automata with Applications to Temporal Logic*, Theoretical Computer Science **49** (1987), 217–237.

[21] Tauriainen, H., and K. Heljanko, *Testing LTL Formula Translation into Büchi Automata*, Int. J. Softw. Tools Technol. Transfer **4** (2002), 57–70.

[22] Thiébaux, S., C. Gretton, J. Slaney, D. Price, and F. Kabanza, *Decision-Theoretic Planning with non-Markovian Rewards*, to appear in Journal of Artificial Intelligence Research.

[23] Tripakis, S., and K. Altisen. "On-the-Fly Controller Synthesis for Discrete and Dense-Time Systems," Proc. FM, LNCS **1708** (1999), 233–252.

[24] Wolper, P., "On the Relation of Programs and Computations to Models of Temporal Logic," Proc. of Temporal Logic in Specification, LNCS **398** (1989), 75–123.