Evaluation and Benchmarking for Robot Motion Planning Problems Using TuLiP

Nicholas Spooner *
Richard Murray, Necmiye Özay, Ufuk Topcu and Pavithra Prabhakar
Control and Dynamical Systems, California Institute of Technology

CDS Technical Report
28th September 2012

Abstract

Model checking is a technique commonly used in the verification of software and hardware. More recently, similar techniques have been employed to synthesize software that is correct by construction. TuLiP is a toolkit which interfaces with game solvers and model checkers to achieve this, producing a finite-state automaton representing a controller that satisfies the supplied specification. For motion planning in particular, a model checker may be employed in a deterministic scenario to produce a path satisfying a specification φ by checking against its negation $\neg\varphi$. If a counterexample is found, it will be a trace which satisfies φ. This was achieved in the TuLiP framework using the linear temporal logic (LTL) model checkers NuSMV and SPIN. A benchmark scenario based on a regular grid-world with obstacle and goal regions and reachability properties was devised, and extended to allow control of various complexity parameters, such as grid size, number of actors, specification type etc. Different measures of performance were explored, including CPU time, memory usage and path length, and the behavior of each checker with increasing problem complexity was analyzed using these metrics. The suitability of each checker for different classes and complexities of motion-planning problem was evaluated.

1 Introduction

Model checking and verification are important in many areas in both industry and research, especially in real-time embedded and safety-critical systems. Such systems pervade our everyday lives; for example, in hardware controllers in aviation or industrial plants, security and communication protocols in networks or switching technology in telephony. Conventional simulation and testing techniques are effective at finding flaws in these programs, but are necessarily incomplete for all but simple systems. Subtle errors in design may evade these techniques but can have disastrous consequences for reliability. In these systems, where in some cases human lives are potentially at risk, quality assurance of digital components is extremely important. In order to improve the effectiveness of error detection and correction, one can employ model checking; it is a useful alternative and aid to traditional debugging.

*N. Spooner, University of Cambridge, participating in the CamSURF programme with the Department of Control and Dynamical Systems, California Institute of Technology, Pasadena, CA 91125 USA ns507@cam.ac.uk

A natural extension to model-checking is synthesis: rather than manually writing a program, we instead write a specification and from that synthesise a program which is correct to that specification. The program is thereby correct by construction, which eliminates the need for debugging and verification, provided the model and specification are correct and assumptions about the environment are valid.

Both model-checking and synthesis require a method of formal specification of system and environment properties. The principal logics used for this are computation tree logic (CTL) and linear temporal logic (LTL). These can be used to specify the behaviour of an actor in a motion planning problem to an arbitrary complexity: for example, using temporal operators we can easily formulate tasks such as "never visit A or B until C is visited", or "visit A, B, and C in order". This flexibility gives temporal logic methods some advantage over traditional motion planning techniques.

This technique can be employed in robotic motion planning to synthesise plans satisfying particular conditions; in particular a model checker can be used to provide a trace satisfying a formula φ by producing a counterexample to $\neg\varphi$. We explore the usefulness of this technique in this report, by considering two widely-used model checkers: NuSMV [1] and SPIN [2]. Both employ different methods of checking, which give them very different performance characteristics.

The report will explain important concepts in Section 2. The specific problem considered is described in detail in Section 3, and we discuss benchmarking techniques in Section 4. The results are presented in Section 5.

2 Preliminaries

Let $G = (V, E)$ be a graph where $V = v_1, \ldots, v_k$ is the set of vertices and $E \subseteq V \times V$ is a symmetric relation representing the set of edges. In the context of motion planning, G represents a static environment where V is the set of region labels and E is the adjacency relation. In the context of this problem, we will assume that any agent may move instantaneously between adjacent vertices of G.

Let $TS = (S, \rightarrow, I, AP, L)$ be a transition system where $S = \{s_0, ..., s_n\}$ is a set of states, $\rightarrow \subseteq S \times S$ is the transition relation, $I \subseteq S$ is the set of possible initial states, AP is the set of atomic propositions and $L : S \rightarrow 2^{AP}$ is the labelling function on states, so that for a state $s \in S$, $L(s) = \{p \in AP \mid p \text{ satisfied in } s\}$. A path fragment of TS is a sequence of states $\pi = s_0 s_1 \ldots$, and may be infinite or end with a terminating state s_t, where $s_t \not\rightarrow$ (i.e. $s_t \notin dom(\rightarrow)$). The trace of π is defined $\text{trace}(\pi) = L(s_0)L(s_1)\ldots$, that is a sequence of sets of propositions satisfied in each state [3].

We consider a motion planning problem as a tuple $P = (TS, \varphi)$, where TS is a transition system as specified above and φ is an arbitrary LTL specification over its atomic propositions. In the simple case, $TS = (S, \rightarrow, I, AP, L)$ is constructed from a graph $G = (V, E)$ representing a static environment so that $S = V = \{v_0, ..., v_n\}$ where $n = |V|$, $\rightarrow = E$, $I \subseteq V$ is the set of possible initial regions, $AP = p_1, ..., p_n$, and $L = \{(s_i, p_i) \mid 0 < i < n, s_i \in S, p_i \in AP\}$, i.e. for each state s_i, $L(s_i) = p_i$ where each p_i is true in exactly one state.

This paper uses linear temporal logic (LTL) exclusively as the logic for specifications. LTL is defined using a small number of additional temporal operators in addition to the standard boolean operators ($\wedge$, $\vee$, etc.). Informally, these operators are:

- $\Box p$ "always", meaning that p is true in every state;

- $\Diamond p$ "eventually", meaning that p holds in some future state;
- $\bigcirc p$ "next", meaning that p holds in the next state;
- $p \,\mathcal{U}\, q$ "until", meaning that p holds in all states preceding some state in which q holds.

A path π satisfies an LTL formula φ ($\pi \models \varphi$) if the corresponding properties above hold of trace(π). (e.g. $\pi \models \Box p$ where trace(π) $= P_0 P_1 \ldots$, if for all P_i, $p \in P_i$.) For a more formal treatment of LTL, see [4].

3 Scenario

The scenario used in all benchmarks in this paper is based on a simple gridworld abstraction, used in many 2D motion planning problems [5, 6]. The gridworld is useful as a discrete abstraction of a static 2D environment, although it has some drawbacks due to its simplicity.

A gridworld is a regular arrangement of rectangular cells which partition a planar environment. Each cell is adjacent to four or fewer other cells; two cells are adjacent if and only if they share an edge. A robot must always occupy exactly one cell. Certain cells are marked as static obstacles (the black cells in Figure 1). No robot may occupy an obstacle cell. The cells marked 'I' are possible initial positions, and the cells marked 'G' are goal cells.

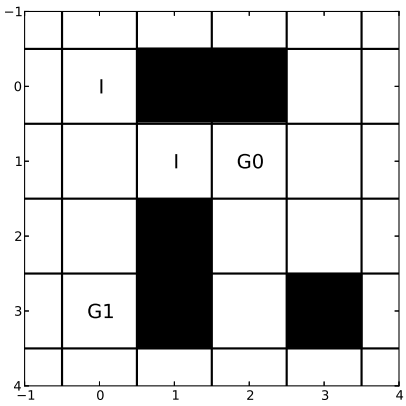

Figure 1: An example 4x4 grid world. Initial position cells are marked with an I; goals are cells G0 and G1.

A simple gridworld specification takes the form

$$\varphi = \bigwedge_i \Box \neg obs_i \wedge \bigwedge_i \Box \Diamond goal_i \tag{1}$$

where each obs_i is a proposition which is true if and only if the robot occupies obstacle cell i, and each $goal_i$ is a proposition which is true if and only if the robot occupies goal cell i. The proposition states that obstacle cells should never be visited, and all goal cells should be visited infinitely often in a satisfying path. This type of specification is used in most of the benchmarks.

The scenario is converted to a problem suitable for model checking by converting the adjacencies to a graph and constructing a transition system as detailed above. A detail of implementation is important here: during the conversion, the graph is pruned of edges representing forbidden transitions (i.e. those which move from an empty cell to an obstacle cell). This means that the safety subformula of (1) is redundant, leading to the simplified progress-only specification

$$\varphi = \bigwedge_i \Box\Diamond goal_i \tag{2}$$

The reduction to a specification with only a 'single' operator ($\Box\Diamond p$) allows some model checkers to run more efficiently. This preprocessing step also reduces the length of the specification significantly, which is helpful in reducing the complexity of the problem as will be seen later.

A specification of this kind produces a solution trace consisting of some prefix (may be zero-length) followed by a suffix which is repeated infinitely often. The suffix includes the states where each $goal_i$ is satisfied.

One more complex specification we consider is a robot which must visit each goal in sequence, so that its trace will be of the form:

$$\underbrace{s_1 \dots s_k}_{\text{prefix}} \underbrace{s_{k+1} \dots s_{g_1} \dots s_{g_n} \dots}_{\text{suffix}} s_{k+1} \dots$$

where s_{g_i} is the state where $goal_i$ is satisfied. The specification used to produce this behaviour is

$$\begin{aligned}
\varphi = & gnext_0 && \text{(initial)}\\
& \wedge \bigwedge_{0 \le i < n_g} \Big(\Box(gnext_i \rightarrow \Diamond goal_i) && \text{(goal progress)}\\
& \wedge \Box[gnext_i \rightarrow (\bigcirc gnext_i \vee (goal_i \wedge \bigcirc \neg gnext_{i+1}))] \Big) && \text{(transition)}\\
& \wedge \Box(gnext_{n_g} \rightarrow \bigcirc gnext_0) && \text{(reset)}\\
& \wedge \bigwedge_{0 \le i < n_g} \Big(\Box[\neg gnext_i \rightarrow \neg goal_i] \Big) && \text{(safety)}\\
& \wedge \Box\Diamond gnext_{n_g} && \text{(progress)}\\
& \wedge \Box \bigvee_{0 \le i \le n_g} \left(gnext_i \wedge \bigwedge_{\substack{0 \le j \le n_g \\ j \ne i}} \neg gnext_j \right) && \text{(mutex)}
\end{aligned} \tag{3}$$

In short, we introduce new propositional variables $gnext_i$ for $0 < i < n_g$ where n_g is the number of goals. $gnext_i$ is true if, on the current run, all goals $goal_j$ for $0 \le j < i$ have been visited; i.e. the next goal to visit is $goal_i$. The first goal to visit is $goal_0$ (initial); (goal progress) states that we eventually visit it. Whenever we make a step, either the next goal stays the same, or we have reached that goal and so we move to the next (transition). When all of the goals are reached ($gnext_{n_g}$) then we go back to $goal_0$ (reset); we do this infinitely often (progress). We never enter a goal cell unless it is the next goal (safety). Finally, there is only one next goal at any time (mutex).

Another important implementation detail relates to the input languages of NuSMV and SPIN. Here for mathematical simplicity we have defined $goal_i$ and $gnext_i$ as sets of boolean variables. In implementation, however, they are defined as integer variables of limited range, so that $goal_i \leftrightarrow cellID = j$, where j is the number assigned to the cell where $goal_i$ is satisfied, and $gnext_i \leftrightarrow goal = i$. This has some advantage in the size of the state space, but is mainly useful in that it implies mutual exclusion; it is not necessary to explicitly state (mutex) as above, because it is impossible for $gnext$ to hold two different values simultaneously.

The second major extension to the gridworld scenario is the multiple-robot case. This is simply placing multiple actors with the same dynamics and goals (but different initial positions) into the gridworld. Such a combination is represented by a tuple $R = (MP, \Phi)$ where $MP = (P_1, ..., P_n)$ is a tuple of individual motion planning problems and Φ is a global specification over all P_i. A solution to R is a tuple of equal-length paths $\sigma = (\pi_1, \ldots, \pi_n)$ such that each π_i is a solution for P_i and the path $\Pi = (\pi_1[1], \ldots, \pi_n[1]) \ldots (\pi_1[k], \ldots, \pi_n[k])$ over $S_1 \times \ldots \times S_n$, where k is the length of a path, satisfies Φ. We may assume that for any i,j $AP_i \cap AP_j = \emptyset$, that is no two problems share the same propositional symbols, which can be trivially achieved by renaming.

In the multiple-robot scenario we are typically interested in problems where robots may not simultaneously occupy the same cell (which in reality would cause a collision). To express this in LTL, we use the global specification

$$\Phi = \bigwedge_{i<n_r} \bigwedge_{j<n_c} cell_{i,j} \rightarrow \bigwedge_{k<n_r} \neg cell_{k,j} \tag{4}$$

where $cell_{i,j}$ is true iff robot i resides in cell j, and n_r and n_c are the numbers of robots and cells respectively.

This extension is particularly interesting because an increase in the number of transition systems sharply increases the complexity of the problem. The methods used to handle multiple interacting transition systems in the model checkers tested are unable to surmount that complexity, and the problem becomes intractable very quickly.

4 Benchmarking

The goal of this paper is the objective evaluation of different model checkers in the context of 2D motion planning. In order to achieve this, we require a stable benchmark environment which allows for easy adjustment of problem complexity and measurement of solution cost metrics. Such an environment was provided by the TuLiP temporal logic toolbox [7], a collection of tools for correct-by-construction synthesis of controllers. TuLiP provides the framework upon which these benchmarks were run.

Any benchmark consists of three distinct parts: a basic problem, a complexity metric and a performance metric. For the purposes of the paper, the basic problem is the gridworld as outlined above. The gridworld makes a strong basis for a benchmark as it is simple to generate at random with any chosen set of parameters. Using TuLiP we can generate gridworld models of arbitrary size and shape with varying obstacle density. It is also simple to generate special gridworlds with particular properties, as shown in Fig. 2. The structure of the gridworld limits its flexibility, however; the number of transitions entering or leaving any region is limited to four, and all transitions are symmetric. The gridworld problem (and a related problem, triangulation) are useful as intermediate discretizations of real continuous planning problems.

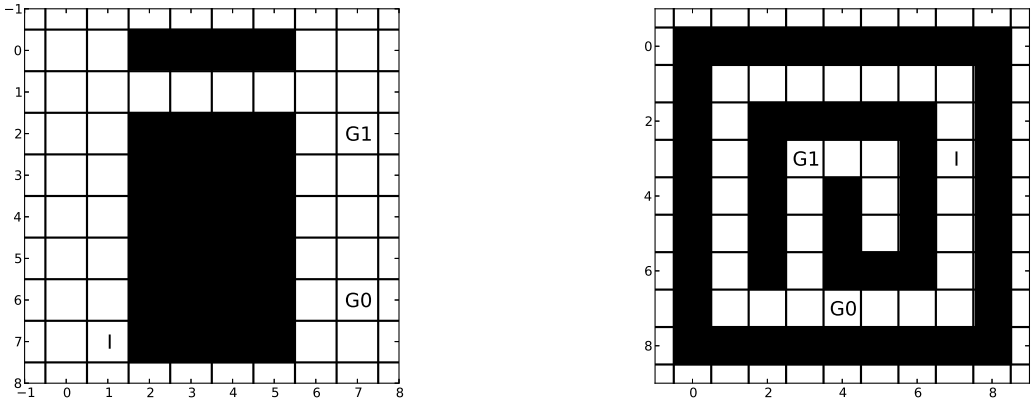

Figure 2: Examples of special gridworld types: a narrow passage (above) and a simple procedurally generated maze (below). Each has a one-cell corridor width, which requires more complex strategies in multi-robot scenarios.

We have devised a number of complexity measures; each one encapsulates some aspect of the difficulty or size of the problem. These include:

- Grid size: a simple and direct consequence of the chosen problem size, but not necessarily representative of solution difficulty.
- Transition system size: more generally applicable, and better correlated with the size of the problem presented to the model checker.
- Type of formula: for example, safety ($\square p$), liveness ($\square\diamond p$), persistence ($\diamond\square p$), GR(1) (a subset of LTL with an n^3 bound on synthesis, see [8]), or full LTL. Categorical, and so difficult to quantify its complexity.
- Number and size of discrete variables: directly related to the size of the state space (the number of distinct states is bounded by 2^l where l is the number of bits required to store all discrete variables).
- Number of distinct automata: Easily quantified but complexity increases very quickly, so not necessarily scalable. Does not account for the sizes of the automata.
- Minimum description length, Shannon entropy: information-theoretic properties of the problem description. It is not clear how these would be applied to the current gridworld problem.

We expect that for any solver, an increase in one of these measures will give a corresponding increase in the solution cost, although the size of that increase will vary significantly.

There are various possible measures of performance, each representing a different aspect of solution cost. We evaluate the suitability of each measure based on (i) consistency, so repeated runs under the same conditions give the same result, (ii) representativeness, so that the difference in performance is expressed proportionally to the difference in resulting value, and (iii) universal application, so that the benchmark can be used to compare different kinds of problem. Examples include:

- CPU time: the amount of time the process spends using the CPU. This is a good indicator of the amount of time taken to solve, and is more consistent than the real solve time. It is also universal.
- Memory usage: the maximum amount of memory used by the process while solving. Again, this is consistent, representative, and universal. It may, however, hide unreasonably long computation times.
- Tractability: whether or not a solver can, within the resource bounds of a system, solve a problem. This is affected more by the system than the solver, and is therefore not very useful.
- Path length: the length of the resulting solution path. Consistent and representative, but not necessarily universal in the sense of comparing types of problem.
- Proportion of state space explored: a checker which finds a solution after exploring a small proportion of the state space is clearly doing so efficiently. Consistent and universal, but not necessarily representative (the solver may explore slowly) and difficult to measure.

No single one of these can be considered more useful than the others, as the importance of these costs is highly dependent on the situation - for example, CPU time may be more important in situations where plans must be constructed quickly or repeatedly, but for a situation in which memory size is limited (such as embedded controllers) a decision may focus more on memory usage. Indeed it is often possible to trade off one for the other; for example, shorter solution paths can be found by some solvers at the expense of longer running time and greater memory usage (depth-first search).

5 Results

The results are presented as a series of comparisons between NuSMV and SPIN over different parameter spaces. The performance metrics we will consider most are CPU time and memory usage, since these have the most significant effect on the practicality of solving. All benchmarks are conducted on a server with two four-core 3.0GHz processors and 64GB of RAM (the CDS compute server 'Nessa', at time of writing).

The first benchmark result (Fig. 3) is simple but provides an interesting observation. While NuSMV is clearly superior to SPIN for reasonable problem sizes in terms of CPU time, and for small problems NuSMV leads on memory usage also, after around 1900 cells (about 43x43), the memory usage of NuSMV increases above that of SPIN. This is due to SPIN's depth-limited search behaviour; the depth to which SPIN explores the state space is to an extent independent of the problem size. It may be the case that in very large problems the depth limit has to be increased, leading to a corresponding increase in memory usage, but this was not explored. The length of the path produced is uniformly shorter under NuSMV than SPIN, but the large variation in path length for equally sized worlds (indicated by error bars) precludes much further analysis.

The second result (Fig. 4) shows clearly the effect of an increase in specification length on both solvers; recall that due to (2), the length of the specification is linear in the number of goals. The cost increase is not very pronounced under NuSMV, and both CPU and memory usage remain relatively constant throughout. For SPIN, however, there is a clear and rapid double-exponential increase which quickly makes testing large examples impractical (SPIN reaches only 5-goal worlds in this experiment, and would require very significant resources to attempt greater). We have,

therefore, that SPIN copes much better with an increased transition system size than with a longer specification; this result can help guide the creation of SPIN models. NuSMV can easily handle large specifications (most likely due to its internal BDD format), and is therefore of more use for complex tasks.

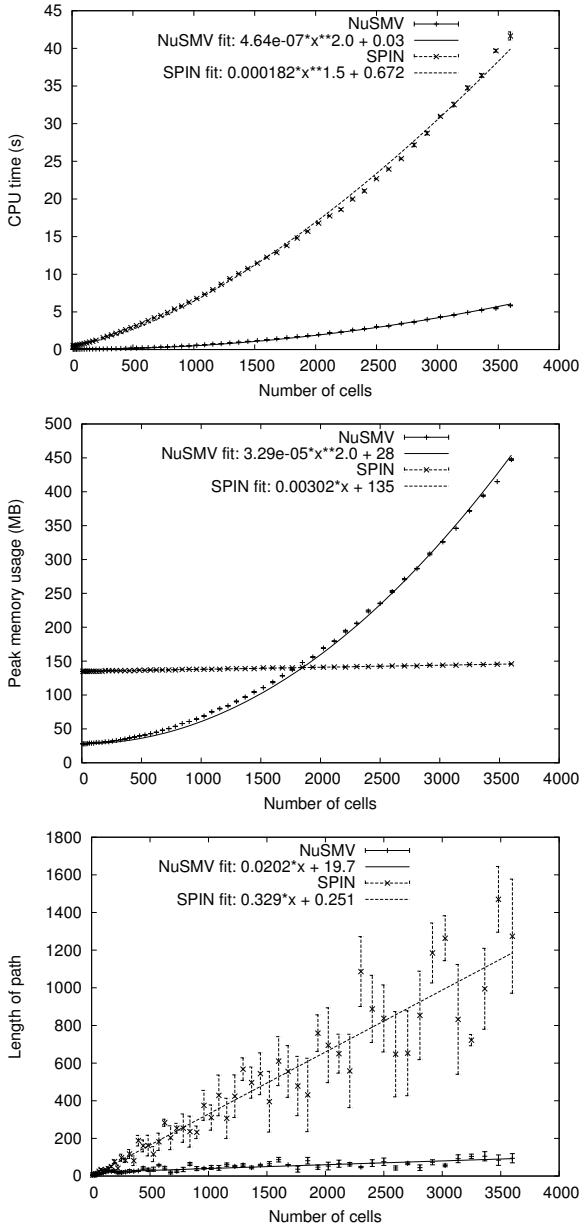

Figure 3: CPU time, memory usage and path length for simple square gridworld problems of different sizes, with one actor, one goal, a wall density of 0.25 and obstacles randomly placed, averaged over 8 randomly-generated worlds. The error bars represent $\pm\sigma_m$.

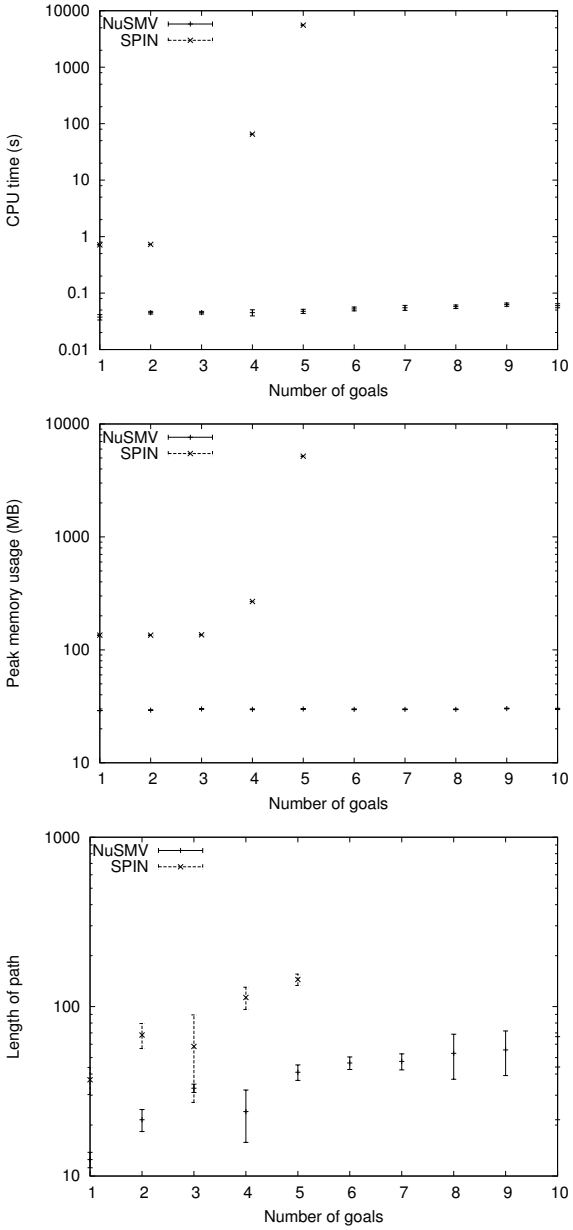

Figure 4: CPU time, memory usage and path length for simple square gridworld problems with different numbers of goals, each of size 10x10 with a single actor, averaged over 4 randomly-generated worlds per point. The error bars represent $\pm\sigma_m$.

Acknowledgments

With thanks to the Cambridge-Caltech Exchange (CamSURF) program, Lauren Stolper and Katherine Montgomery for their support, Caltech SFP for organizing the SURF program and the Caltech CDS group for accommodating me for the summer, in particular Necmiye Özay, Ufuk Topcu, Pavithra Prabhakar and Richard Murray who gave excellent guidance and proofread this report. Thanks also to Mihai Florian for helping to resolve a stack-overflow problem in SPIN.

References

[1] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," 2002, pp. 359–364.

[2] G. J. Holzmann, "The Model Checker SPIN," IEEE Trans. Softw. Eng., vol. 23, no. 5, pp. 279–295, May 1997.

[3] U. Topcu, N. Wongpiromsarn & R. M. Murray, Automata Theory, EECI 21 March 2011 `http://www.cds.caltech.edu/~utopcu/index.php/EECI2011:_Automata_Theory`

[4] C. B. and J.-P. Katoen and F. by K. G. Larsen, "Principles of Model Checking," May 2008.

[5] S. Morgan and M. S. Branicky, "Sampling-based planning for discrete spaces," in Intelligent Robots and Systems, 2004 IEEE/RSJ International Conference on, vol. 2, pp. 1938 – 1945, 2004.

[6] M. Kloetzer and C. Belta, "Automatic Deployment of Distributed Teams of Robots From Temporal Logic Motion Specifications," IEEE Transactions on Robotics, vol. 26, no. 1, pp. 48–61, Feb. 2010.

[7] Temporal Logic Planning (TuLiP) toolbox, `http://tulip-control.sourceforge.net/`, 11 Sep 2012.

[8] N. Piterman and A. Pnueli, "Synthesis of reactive(1) designs," in In Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI '06), 2006, pp. 364–380.