Proceedings of the

AIPS-02 Workshop on Planning via Model-Checking

Toulouse, France
April 23, 2002

Chairs:
Froduald Kabanza, *University of Windsor, Canada*
Sylvie Thiébaux, *The Australian National University, Australia*

Committee:
Alessandro Cimatti, *IRST, Italy*
Giuseppe de Giacomo, *University of Rome, Italy*
Robert Goldman, *Honeywell SRC, USA*
Patrick Haslüm, *University of Linköping, Sweden*
Rune Jensen, *Carnegie Mellon University, USA*

Foreword

Model checking is currently one of the hottest topics in computer science. It consists in comparing a model of a system against a logical requirement to discover inconsistencies. Traditionally, this idea has been used to verify the correctness of hardware circuits and network protocols. More recently, the same idea has been applied to planning with remarkable success, and has led to powerful planning systems such as MBP, MIPS, TALPLANNER, TLPLAN, and UMOP.

The 10 papers accepted for presentation at this workshop are representative of both the volume and diversity of the expansion of research on planning via model-checking. Their topics include extending the basic planning via model checking framework to handling partial observability, concurrency and resources, increasingly complex goals, as well as adversarial and multi-agent domains. It will be evident from this volume that planning via model checking largely intersects with other model-theoretic approaches to planning such as decision-theoretic planning, Markov decision processes, or controller synthesis, and relates to some recent frameworks for scheduling, model-based prediction, diagnosis, repair, and reconfiguration. Another important theme is symbolic heuristic search, using representations inherited from the verification community, such as binary or algebraic decision diagrams. It was the goal of this workshop to bring together these many strands of research, with a view to identifying not only their diversity but also the substantial unity of the emerging field.

We would like to thank the members of the program committee who worked hard in reviewing the papers on a very tight schedule. We are grateful to the AIPS workshop and local arrangement chairs, for their help with the organisation. Finally we are happy to acknowledge the contribution made by our sponsor PLANET, the European Network of Excellence in AI Planning.

Froduald Kabanza and Sylvie Thiébaux

Contents

Preemptive Job-Shop Scheduling using Stopwatch Automata
Yasmina Abdeddaïm, Oded Maler 7

Plan Validation for Extended Goals under Partial Observability (preliminary report)
Piergiorgio Bertoli, Alessandro Cimatti, Marco Pistore, Paolo Taverso 14

Solving Power Supply Restoration Problems with Planning via Symbolic Model-Checking
Piergiorgio Bertoli, Alessandro Cimatti, John Slaney, Sylvie Thiébaux 23

Solving Planning Problems Using Real-Time Model-Checking (Translating PDDL3 into Timed Automata)
Henning Dierks, Gerd Berhrmann, Kim Larsen 30

Symbolic Exploration in Two-Player Games: Preliminary Results
Stephan Edelkamp 40

Symbolic Heuristic Search for Factored Markov Decision Processes
Zhengzhu Feng, Eric Hansen 49

Planning Via Model-Checking in Dynamic Temporal Domains: Exploiting Planning Representations
Robert P. Goldman, David J. Musliner, Michael J.S. Pelican 55

Partial State Progression: An Extension to the Bacchus-Kabanza Algorithm,with Applications to Prediction and MITL Consistency
Patrick Haslüm 64

SetA*: An Efficient BDD-Based Heuristic Search Algorithm
Rune M. Jensen, Randal E. Bryant, Manuela M. Veloso 72

Probabilistic Plan Verification through Acceptance Sampling
Håkan L. S. Younes, David J. Mulsiner 81

Preemptive Job-Shop Scheduling using Stopwatch Automata*

Yasmina Abdeddaïm and **Oded Maler**
Verimag,Centre Equation, 2, av. de Vignate 38610 Gières, France
Yasmina.Abdeddaim@imag.fr Oded.Maler@imag.fr

Abstract

In this paper we show how the problem of job-shop scheduling where the jobs are preemptible can be modeled naturally as a shortest path problem defined on an extension of timed automata, namely stopwatch automata where some of the clocks might be freezed at certain states. Although general verification problems on stopwatch automata are known to be undecidable, we show that due to particular properties of optimal schedules, the shortest path in the automaton belongs to a finite subset of the set of acyclic paths and hence the problem is solvable. We present several algorithms and heuristics for finding the shortest paths in such automata and test their implementation on numerous benchmark examples.

Introduction

In (AM01) we have described a first step in a research programme intended to re-formulate scheduling problems using (timed) automata-based formalisms. Apart from the undeniable joy of re-inventing the wheel, this work is motivated by the belief that such automata provide timing problems with faithful state-based dynamic models on which a systematic study of semantic and computational problems can be done — the reader is referred to (AM01) for some of the motivation and background and to (AM99; AGP99; NTY00; NY01; BFH$^+$01) for other recent results in this spirit. In this framework the runs of the timed automaton correspond to feasible schedules and finding a time-optimal schedule amounts to finding the shortest path (in terms of elapsed time) in the automaton. In (AM01) we have shown how this works nicely for the job-shop scheduling problem which can be modeled by a certain class of *acyclic* timed automata, having finitely many qualitative[1] runs. Each such qualitative run is an equivalence class of a non-countable number of quantitative runs, but as we have shown, one of those (a "non-lazy" run which makes transitions as soon as possible) is sufficient to find the optimum over the whole class. These observations allowed us to apply efficient search algorithms over single configurations of clocks rather than work with zones.

In this work we extend these results to preemptible jobs, i.e. jobs that can use a machine for some time, stop for a while and then resume from where they stopped. Such situations are common, for example, when the machines are computers. While extending the framework of (AM01) to treat this situation we encounter two problems:

1. The corresponding class of automata goes beyond timed automata because clocks are stopped but not reset to zero when a job is preempted. General reachability problems for such stopwatch automata (also known as *integration graphs*) are known to be undecidable (C92; KPSY99).
2. Due to preemption and resumption, which corresponds to a loop in the underlying transition graph, the obtained automata are cyclic (unlike the non-preemptive case) and they have an *infinite* number of qualitative runs.

We will show however that these problems can be overcome for the class of stopwatch automata that correspond to preemtible job shop problems, and that efficient algorithms can be constructed.

The rest of the paper is organized as follows. In section 2 we give a short introduction to the preemptive job-shop scheduling problem including a fundamental property of optimal schedules. In section 3 we recall the definition of stopwatch automata and show how to transform a job-shop specification into such an automaton whose runs correspond to feasible schedules. In section 4 we describe efficient algorithms for solving the shortest-path problem for these automata (either exactly or approximately) and report the performance results of their prototype implementation on numerous benchmark examples.

Preemptive Job-Shop Scheduling

The Job-shop scheduling problem is a generic resource allocation problem in which common resources ("machines") are required at various time points (and for given durations) by different tasks. The goal is to find a way to allocate the resources such that all the tasks terminate as soon as possible. We consider throughout the paper a fixed set M of resources. A *step* is a pair (m, d) where $m \in M$ and $d \in \mathbb{N}$, indicating the required utilization of resource m for time duration d. A *job specification* is a finite sequence

$$J = (m_1, d_1), (m_2, d_2), \ldots, (m_k, d_k) \quad (1)$$

*This work was partially supported by the European Community Project IST-2001-35304 AMETIST `http://ametist.cs.utwente.nl`

[1]By a qualitative run of a timed automaton we mean a sequence of states and transitions without metric timing information.

of steps, stating that in order to accomplish job J, one needs to use machine m_1 for d_1 time, then use machine m_2 for d_2 time, etc.

Definition 1 (Job-Shop Specification) *Let M be a finite set of resources (machines). A job specification over M is a triple $J = (k, \mu, d)$ where $k \in \mathbb{N}$ is the number of steps in J, $\mu : \{1..k\} \to M$ indicates which resource is used at each step, and $d : \{1..k\} \to \mathbb{N}$ specifies the length of each step. A job-shop specification is a set $\mathcal{J} = \{J^1, \ldots, J^n\}$ of jobs with $J^i = (k^i, \mu^i, d^i)$.*

In order to simplify notations we assume that each machine is used exactly once by every job. We denote $\mathbb{R}_+$ by T, abuse $\mathcal{J}$ for $\{1, \ldots, n\}$ and let $K = \{1, \ldots, k\}$.

Definition 2 (Feasible Schedules) *Let $\mathcal{J} = \{J^1, \ldots, J^n\}$ be a job-shop specification. A feasible schedule for $\mathcal{J}$ is a relation $S \subseteq \mathcal{J} \times K \times T$ so that $(i, j, t) \in S$ indicates that job J^i is busy doing its jth step at time t and, hence, occupies machine $\mu^i(j)$. We let T^i_j be the set of time instants where job $i \in \mathcal{J}$ is executing its jth step, i.e. $T^i_j = \{t : (i, j, t) \in S\}$.*[2] *A feasible schedule should satisfy the following conditions:*

1. Ordering*: if $(i, j, t) \in S$ and $(i, j', t') \in S$ then $j < j'$ implies $t < t'$ (steps of the same job are executed in order).*
2. Covering*: For every $i \in \mathcal{J}$ and $j \in K$*
$$\int_{t \in T^i_j} dt \geq d^i(j)$$
(every step is executed).
3. Mutual Exclusion*: For every $i \neq i' \in \mathcal{J}$, $j, j' \in K$ and $t \in T$, if $(i, j, t) \in S$ and $(i', j', t) \in S$ then $\mu^i(j) \neq \mu^{i'}(j')$ (two steps of different jobs which execute at the same time do not use the same machine).*

Note that we allow a job to occupy the machine *after* the step has terminated. The *length* $|S|$ of a schedule is the supremal t over all $(i, j, t) \in S$. We say that a step j of job i is *enabled* in time t if $t \in \mathcal{E}^i_j = (\max T^i_{j-1}, \max T^i_j]$. The *optimal job-shop scheduling problem* is to find a schedule of a minimal length. This problem is known to be NP-hard (GJ79). From the relational definition of schedules one can derive the following commonly used definitions:

1. The *machine allocation function* $\alpha : M \times T \to \mathcal{J}$ stating which job occupies a machine at any time, defined as $\alpha(m, t) = i$ if $(i, j, t) \in S$ and $\mu^i(j) = m$.
2. The *task progress function* $\beta : \mathcal{J} \times T \to M$ stating what machine is used by a job is at a given time, defined as $\beta(i, t) = m$ if $(i, j, t) \in S$ and $\mu^i(j) = m$.

These functions are partial — a machine or a job might be idle at certain times.

Example 1: Consider $M = \{m_1, m_2, m_3\}$ and two jobs $J^1 = (m_1, 3), (m_2, 2), (m_3, 4)$ and $J^2 = (m_2, 5)$. Two schedules S_1 and S_2 appear in Figure 1. The length of S_1 is 9 and it is the optimal schedule. As one can see, at $t = 3$, J^1 preempts J^2 and takes machine m_2.

[2]We may assume further that T^i_j is can be decomposed into a countable number of left-closed right-open intervals.

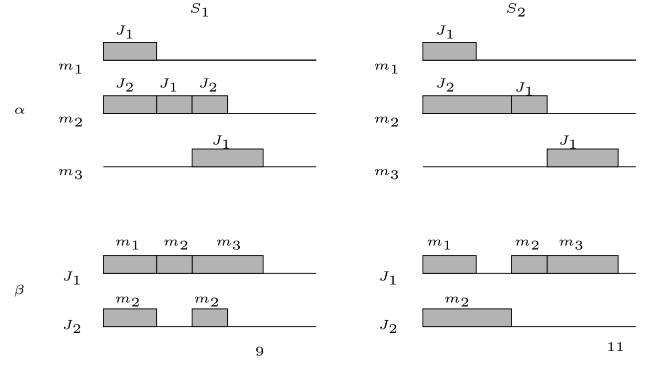


Figure 1: Two schedule S_1 and S_2 visualized as the machine allocation function α and the task progress function β.

We conclude this section with a reformulation of a well-known result concerning optimal preemptive schedules which will be used later. In essence this result formalizes the following two intuitive observations: *1) When jobs can be preempted and resumed at no cost, there is no reason to delay a step not being in a conflict with another. 2) Two jobs that keep on preempting each other do not contribute to the general progress.*

Definition 3 (Conflicts and Priorities) *Let S be a feasible schedule. We say that job i is in conflict with job i' on machine m in S (denoted by $i \nparallel_m i'$) when there are two respective steps j and j' such that $\mu^i(j) = \mu^{i'}(j') = m$ and $\mathcal{E}^i_j \cap \mathcal{E}^{i'}_{j'} \neq \emptyset$. We say that i has priority on m over a conflicting job i' (denoted by $i \prec_m i'$) if it finishes using m before i' does, i.e. $\sup T^i_j < \sup T^{i'}_{j'}$.*

Note that conflicts and priorities are always induced by a schedule S although S is omitted from the notation.

Definition 4 (Efficient Schedules) *A schedule S is efficient if for every job i and a step j such that $\mu^i(j) = m$, job i uses m during all the time interval $\mathcal{E}^i_j$ except for times when another job i' such that $i' \prec_m i$ uses it.*

The following is a folk theorem, whose roots go back at least to (J55) with some reformulation and proofs in, for example, (CP90; PB96).

Theorem 0.1 (Efficiency is Good) *Every preemptive job-shop specification admits an efficient optimal schedule.*

Sketch of Proof: The proof is by showing that every inefficient schedule S can be transformed into an efficient schedule S' with $|S'| \leq |S|$. Let I be the first interval when inefficiency occurs for job i and machine m. We modify the schedule by shifting some of the later use of m by i into I. If m was occupied during I by another job i' such that $i \prec_m i'$, we give it the time slot liberated by i. The termination of the step by i' is not delayed by this modification because it happens anyway after i terminates its step. ■

As an illustration consider the schedules appearing in Figure 2 with $J_1 \prec_m J_2 \prec_m J_3$ and where J_2 is enabled in the interval $[t_1, t_2]$. The first inefficiency in S_1 is eliminated in S_2 by letting J_2 use the free time slot before the arrival of

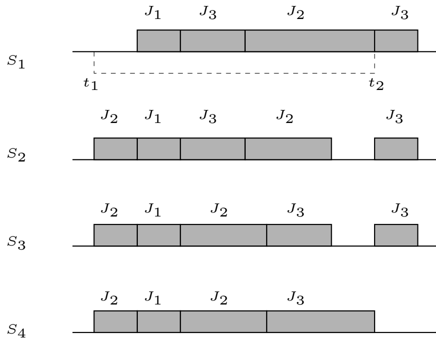


Figure 2: Removal of inefficiency, $J_1 \prec J_2 \prec J_3$.

J_1. The second inefficiency occurs when J_3 uses the machine while J_2 is waiting, and it is removed in S_3. The last inefficiency where J_3 is waiting while m is idle is removed in S_4.

This result reduces the set of candidates for optimality from the non-countable set of feasible schedules to the finite set of efficient schedules, each of which corresponds to a fixed priority relation.[3] There are potentially $kn!$ priority relations but only a fraction of those needs to be considered because when i and i' are never in conflict concerning m, the priority $i \prec_m i'$ has no influence on the schedule.

Stopwatch Automata

Timed automata (AD94) are automata augmented with continuous clock variables whose values grow uniformly at every state. Clocks are reset to zero at certain transitions and tests on their values are used as pre-conditions for transitions. Hence they are ideal for describing concurrent time-dependent behaviors. There are however situations, preemptive scheduling being among those, in which we need to measure the overall accumulated time that a systems spends in some state. This motivated the extension of the model to have clocks with derivative zero at certain states. Unlike timed automata, the reachability problem for these automata is undecidable (C92). Some sub-classes, *integration graphs*, were investigated in (KPSY99), where a decision procedure based on reducing the problem into linear constraint satisfaction was reported. Similar automata were studied in (MV94) and in (CL00) where an implementation of an approximate verification algorithm was described.

Definition 5 (Stopwatch Automaton) *A stopwatch automaton is a tuple $\mathcal{A} = (Q, C, s, f, \mathbf{u}, \Delta)$ where Q is a finite set of states, C is a finite set of n clocks, $\mathbf{u} : Q \to \{0,1\}^n$ assigns a constant slope to every state and Δ is a transition relation consisting of elements of the form (q, ϕ, ρ, q') where q and q' are states, $\rho \subseteq C$ and ϕ (the transition guard) is a boolean combination of formulae of the form $(c \in I)$ for some clock c and some integer-bounded interval I. States s and f are the initial and final states, respectively.*

[3]This might explain the popularity of priority-based approach in computer scheduling.

A *clock valuation* is a function $\mathbf{v} : C \to \mathbb{R}_+ \cup \{0\}$, or equivalently a $|C|$-dimensional vector over $\mathbb{R}_+$. We denote the set of all clock valuations by $\mathcal{H}$. A configuration of the automaton is hence a pair $(q, \mathbf{v}) \in Q \times \mathcal{H}$ consisting of a discrete state (sometimes called "location") and a clock valuation. Every subset $\rho \subseteq C$ induces a reset function $\mathrm{Reset}_\rho : \mathcal{H} \to \mathcal{H}$ defined for every clock valuation $\mathbf{v}$ and every clock variable $c \in C$ as

$$\mathrm{Reset}_\rho\, \mathbf{v}(c) = \begin{cases} 0 & \text{if} \quad c \in \rho \\ \mathbf{v}(c) & \text{if} \quad c \notin \rho \end{cases}$$

That is, Reset_ρ resets to zero all the clocks in ρ and leaves the others unchanged. We use $\mathbf{1}$ to denote the unit vector $(1, \ldots, 1)$, $\mathbf{0}$ for the zero vector and $\mathbf{u}_q$ for $\mathbf{u}(q)$, the derivative of the clocks at q.

A *step* of the automaton is one of the following:

- A discrete step: $(q, \mathbf{v}) \xrightarrow{0} (q', \mathbf{v}')$, where there exists $\delta = (q, \phi, \rho, q') \in \Delta$, such that $\mathbf{v}$ satisfies ϕ and $\mathbf{v}' = \mathrm{Reset}_\rho(\mathbf{v})$.
- A time step: $(q, \mathbf{v}) \xrightarrow{t} (q, \mathbf{v} + t\mathbf{u}_q)$, $t \in \mathbb{R}_+$.

A *run* of the automaton starting from $(q_0, \mathbf{v}_0)$ is a finite sequence of steps

$$\xi : \quad (q_0, \mathbf{v}_0) \xrightarrow{t_1} (q_1, \mathbf{v}_1) \xrightarrow{t_2} \cdots \xrightarrow{t_l} (q_l, \mathbf{v}_l).$$

The *logical length* of such a run is l and its *metric length* is $|\xi| = t_1 + t_2 + \cdots + t_l$. Note that discrete transitions take no time.

Next we construct for every job $J = (k, \mu, d)$ a timed automaton with one clock such that for every step j with $\mu(j) = m$ there are three states: a state $\overline{m}$ which indicates that the job is waiting to start the step, a state m indicating that the job is executing the step and a state $\tilde{m}$ indicating that the job is preempted after having started. Upon entering m the clock is reset to zero, and measures the time spent in m. Preemption and resumption are modeled by transitions to and from state $\tilde{m}$ in which the clock does not progress. When the clock value reaches $d(j)$ the automaton can leave m to the next waiting state. Let $\overline{M} = \{\overline{m} : m \in M\}$, $\tilde{M} = \{\tilde{m} : m \in M\}$ and let $\overline{\mu} : K \to \overline{M}$ and $\tilde{\mu} : K \to \tilde{M}$ be auxiliary functions such that $\overline{\mu}(j) = \overline{m}$ and $\tilde{\mu}(j) = \tilde{m}$ whenever $\mu(j) = m$.

Definition 6 (Stopwatch Automaton for a Job) *Let $J = (k, \mu, d)$ be a job. Its associated automaton is $\mathcal{A} = (Q, \{c\}, u, \Delta, s, f)$ with $Q = P \cup \overline{P} \cup \tilde{P} \cup \{f\}$ where $P = \{\mu(1), \ldots \mu(k)\}$,$\overline{P} = \{\overline{\mu}(1), \ldots, \overline{\mu}(n)\}$ and $\tilde{P} = \{\tilde{\mu}(1), \ldots, \tilde{\mu}(n)\}$. The slope is defined as $u_q = 1$ when $q \in P$ and $u_q = 0$ otherwise.*[4] *The transition relation Δ consists of the following types of tuples*

type	q	ϕ	ρ	q'	
1) begin	$\overline{\mu}(j)$	*true*	$\{c\}$	$\mu(j)$	$j = 1..k$
2) pause	$\mu(j)$	*true*	$\emptyset$	$\tilde{\mu}(j)$	$j = 1..k$
3) resume	$\tilde{\mu}(j)$	*true*	$\emptyset$	$\mu(j)$	$j = 1..k$
4) end	$\mu(j)$	$c \geq d(j)$	$\emptyset$	$\overline{\mu}(j+1)$	$j = 1..k-1$
end	$\mu(k)$	$c \geq d(k)$	$\emptyset$	f	

[4]Note that the slope at state $\overline{m}$ can be arbitrary because clock c is *inactive* in this state: it is reset to zero without being tested upon leaving $\overline{m}$.

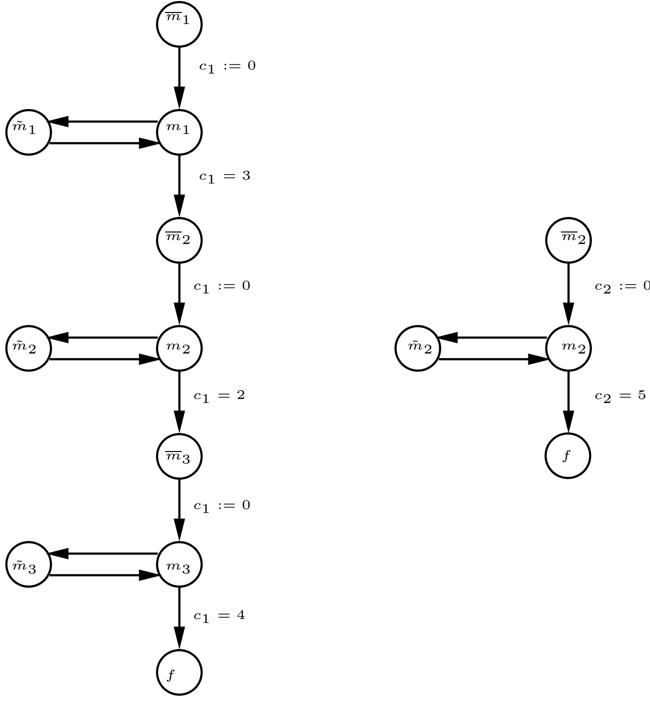


Figure 3: The automata corresponding to the jobs $J^1 = (m_1, 3), (m_2, 2), (m_3, 4)$ and $J^2 = (m_2, 5)$.

The initial state is $\overline{\mu}(1)$.

The automata for the two jobs in Example 1 are depicted in Figure 3.

For every automaton $\mathcal{A}$ we define a *ranking function* $g : Q \times \mathbb{R}_+ \to \mathbb{R}_+$ such that $g(q, v)$ is a lower-bound on the time remaining until f is reached from (q, v):

$$
\begin{aligned}
&g(f, v) = 0 \\
&g(\overline{\mu}(j), v) = \textstyle\sum_{l=j}^{k} d(l) \\
&g(\mu(j), v) = g(\overline{\mu}(j), v) - \min\{v, d(j)\} \\
&g(\tilde{\mu}(j), v) = g(\overline{\mu}(j), v) - \min\{v, d(j)\}
\end{aligned}
$$

In order to obtain the timed automaton representing the whole job-shop specification we need to compose the automata for the individual tasks. The composition is rather standard, the only particular feature is the enforcement of mutual exclusion constraints by forbidding global states in which two or more automata are in a state corresponding to the same resource m. An n-tuple $q = (q^1, \ldots, q^n) \in (M \cup \overline{M} \cup \tilde{M} \cup \{f\})^n$ is said to be *conflicting* if it contains two distinct components q^i and $q^{i'}$ such that $q^i = q^{i'} = m \in M$.

Definition 7 (Mutual Exclusion Composition) *Let $\mathcal{J} = \{J^1, \ldots, J^n\}$ be a job-shop specification and let $\mathcal{A}^i = (Q^i, C^i, u^i, \Delta^i, s^i, f^i)$ be the automaton corresponding to each J^i. Their mutual exclusion composition is the automaton $\mathcal{A} = (Q, C, \mathbf{u}, \Delta, s, f)$ such that Q is the restriction of $Q^1 \times \ldots Q^n$ to non-conflicting states, $C = C^1 \cup \ldots \cup C^n$, $s = (s^1, \ldots, s^n)$, $f = (f^1, \ldots, f^n)$. The slope $\mathbf{u}_q$ for a global state $q = (q^1, \ldots q^n)$ is $(u_{q^1}, \ldots, u_{q^n})$ and the transition relation Δ contains all the tuples of the form*

$$((q^1, \ldots, q^i, \ldots, q^n), \phi, \rho, (q^1, \ldots, p^i, \ldots, q^n))$$

such that $(q^i, \phi, \rho, p^i) \in \Delta^i$ for some i and the global states $(q^1, \ldots, q^i, \ldots, q^n)$ and $(q^1, \ldots, p^i, \ldots, q^n)$ are non-conflicting.

Part of the automaton obtained by composing the two automata of Figure 3 appears in Figure 4. We have omitted the *pause/resume* transitions for m_1 and m_3 as well as some other non-interesting paths.

A run of $\mathcal{A}$ is *complete* if it starts at $(s, \mathbf{0})$ and the last step is a transition to f. From every complete run ξ one can derive in an obvious way a schedule S_ξ such that $(i, j, t) \in S_\xi$ if at time t the ith component of the automaton is at state $\mu(j)$. The length of S_ξ coincides with the metric length of ξ.

Claim 1 (Runs and Schedules) *Let $\mathcal{A}$ be the automaton generated for the preemptive job-shop specification $\mathcal{J}$ according to Definitions 6 and 7. Then:*

1. *For every complete run ξ of $\mathcal{A}$, its associated schedule S_ξ is feasible for $\mathcal{J}$.*
2. *For every feasible schedule S for $\mathcal{J}$ there is a run ξ of $\mathcal{A}$ such that $S_\xi = S$.*

Corollary 1 (Preemptive Scheduling and Stopwatch Automata)

The optimal preemptive job-shop scheduling problem can be reduced to the problem of finding the shortest path in a stopwatch automaton.

The two schedules of Figure 1 correspond to the following two runs (we use the notation $\perp$ to indicate inactive clocks):

S_1 :
$(\overline{m}_1, \overline{m}_2, \perp, \perp) \xrightarrow{0} (m_1, \overline{m}_2, 0, \perp) \xrightarrow{0} (m_1, m_2, 0, 0) \xrightarrow{3}$
$(m_1, m_2, 3, 3) \xrightarrow{0} (\overline{m}_2, m_2, \perp, 3) \xrightarrow{0} (\overline{m}_2, \tilde{m}_2, \perp, 3) \xrightarrow{0}$
$(m_2, \tilde{m}_2, 0, 3) \xrightarrow{2} (m_2, \tilde{m}_2, 2, 3) \xrightarrow{0} (\overline{m}_3, \tilde{m}_2, \perp, 3) \xrightarrow{0}$
$(\overline{m}_3, m_2, \perp, 3) \xrightarrow{0} (m_3, m_2, 0, 3) \xrightarrow{2} (m_3, m_2, 2, 5) \xrightarrow{0}$
$(m_3, f, 2, \perp) \xrightarrow{2} (m_3, f, 4, \perp) \xrightarrow{0} (f, f, \perp, \perp)$

S_2 :
$(\overline{m}_1, \overline{m}_2, \perp, \perp) \xrightarrow{0} (m_1, \overline{m}_2, 0, \perp) \xrightarrow{0} (m_1, m_2, 0, 0) \xrightarrow{3}$
$(m_1, m_2, 3, 3) \xrightarrow{0} (\overline{m}_2, m_2, \perp, 3) \xrightarrow{2} (\overline{m}_2, m_2, \perp, 5) \xrightarrow{0}$
$(\overline{m}_2, f, \perp, \perp) \xrightarrow{0} (m_2, f, 0, \perp) \xrightarrow{2} (m_2, f, 2, \perp) \xrightarrow{0}$
$(\overline{m}_3, f, \perp, \perp) \xrightarrow{0} (m_3, f, 0, \perp) \xrightarrow{4} (m_3, f, 4, \perp) \xrightarrow{0}$
$(f, f, \perp, \perp)$

The job-shop automaton admits a special structure: ignoring the *pause* and *resume* transitions, the automaton is *acyclic* and its state-space admits a natural partial-order. It can be partitioned into levels according to the number of *begin* and *end* transitions from s to the state. There are no staying conditions (invariants) and the automaton can stay forever in any given state. Recall that in any automaton extended with auxiliary variables the transition graph might be misleading, because two or more transitions entering *the*

	state	action	new state	remark
1	$(\overline{m}, \overline{m})$	start 1	$(m, \overline{m})$	
2	$(\overline{m}, \tilde{m})$	start 1	$(m, \tilde{m})$	
3	$(\overline{m}, m)$	preempt 2	$(\overline{m}, \tilde{m})$	
4	$(\tilde{m}, \overline{m})$	resume 1	$(m, \overline{m})$	
5	$(\tilde{m}, \tilde{m})$	resume 1	$(m, \tilde{m})$	
6	$(\tilde{m}, m)$			(impossible)
7	$(m, \overline{m})$	(continue)	$(m, \overline{m})$	
8	$(m, \tilde{m})$	(continue)	$(m, \tilde{m})$	
9	(m, m)			(impossible)

Table 1: Resolving conflicts when $J_1 \preceq_m J_2$.

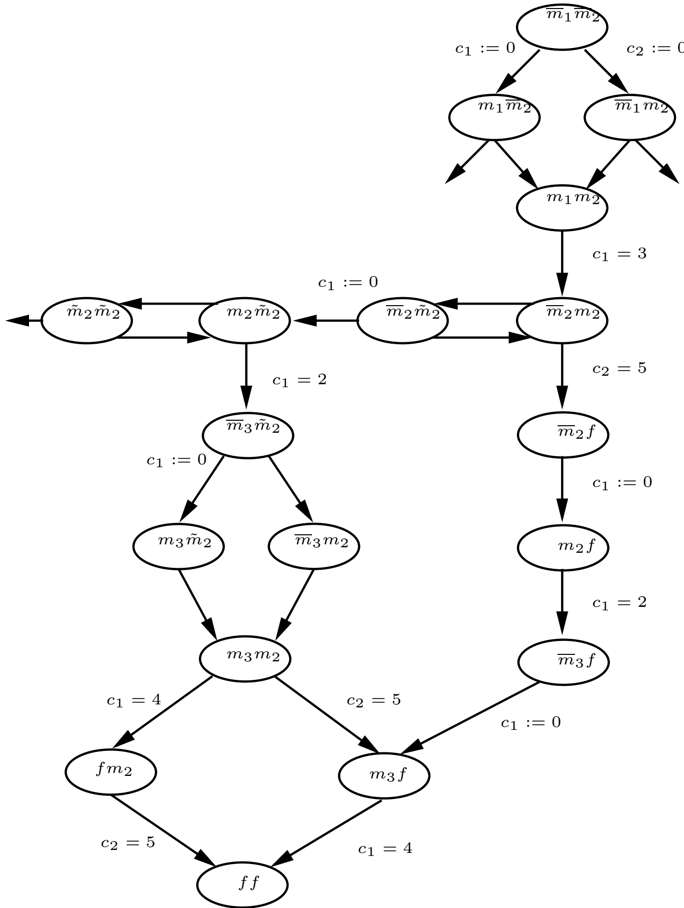


Figure 4: The global stopwatch automaton for the two jobs.

same discrete state, e.g. transitions to (m_3, f) in Figure 4, might enter it with different clock valuations, and hence lead to different continuations. Consequently, algorithms for verification and quantitative analysis might need to explore all the nodes in the unfolding of the automaton into a tree. Two transitions outgoing from the same state might represent a choice of the scheduler, for example, the two transitions outgoing from $(\overline{m}_2, m_2)$ represent the choice of whether or not to preempt J^2 and give machine m_2 to J^1. On the other hand some duplication of paths are just artifacts due to interleaving, for example, the two paths leading from $(\overline{m}_1, \overline{m}_2)$ to (m_1, m_2) are practically equivalent.

Another useful observation is that from every (preemptive or non-preemptive) job-shop specification $\mathcal{J}$ one can construct its reverse problem $\mathcal{J}'$ where the order of every individual job is reversed. Every feasible schedule for $\mathcal{J}'$ can be transformed easily into a feasible schedule for $\mathcal{J}$ having the same length. Doing a forward search on the automaton for $\mathcal{J}'$ is thus equivalent to doing a backward search on the automaton for $\mathcal{J}$.

Shortest Paths in Stopwatch Automata

In order to find shortest paths in stopwatch automata we will take advantage of Theorem 0.1 to restrict the search to runs whose corresponding schedules are efficient.

Definition 8 (Efficient Runs) *A run of a stopwatch automaton constructed according to Definitions 6 and 7 is efficient if all discrete transitions are taken as soon as they are enabled, and all conflicts are resolved according to a fixed priority relation.*

To be more precise, let J_1 and J_2 be two jobs which are in conflict concerning machine m and let J_1 be the one with the highest priority on m. Table depicts all the potential conflict situations and how they are resolved.

In situations 1, 2, 4, and 5 J_1 is waiting for the machine which is not occupied and so it takes it. Such situations could have been reached, for example, by a third job of higher priority releasing m or by J_1 finishing its prior step and entering $\overline{m}$. Situation 3 is similar but with J_2 occupying m and hence has to be preempted to reach situation 2. Situation 6, where J_1 is preempted and J_1 is executing, contradicts the priority and is not reachable. In situations 7 and

8, J_1 is executing and no preemption action is taken. Finally situation 9 violates mutual exclusion.

The restriction to efficient runs makes the problem decidable: we can just enumerate all priority relations, derive the schedules implied by each of them and compare their lengths. The search algorithm that we employ on the unfolding of the automaton generates priorities *on the fly* whenever two jobs come into conflict. In the example of Figure the first conflict is encountered in state $(\overline{m}_2, m_2)$ and from there we may choose between two options, either to continue with time passage or preempt J_2. in the first case we fix the priority $J_2 \prec J_1$ and let J_2 finish without considering preemption anymore while in the second case the priority is $J_1 \prec J_2$, we move to $(\overline{m}_2, \tilde{m}_2)$ and the transition back to $(\overline{m}_2, m_2)$ becomes forbidden. From there we can only continue to $(m_2, \tilde{m}_2)$ and let the time pass until J_1 releases m_2.

To formalize this we define a *valid successors* relation over tuples of the form $(q, \mathbf{x}, \Pi, \theta)$ where $(q, \mathbf{x})$ is a global configuration of the automaton, Π is a (partial) priority relation and θ is the total elapsed time for reaching $(q, \mathbf{x})$ from the initial state. When there are no immediate transitions enabled in $(q, \mathbf{x})$ we have

$$Succ(q, \mathbf{x}, \Pi, \theta) = \{(q, \mathbf{x} + t \cdot \mathbf{u}_q, \Pi, \theta + t)\}$$

where t is the minimal time until a transition becomes enabled, that is, the least t such that a guard on a transition from q is satisfied at $\mathbf{x} + t \cdot \mathbf{u}_q$.

When there are immediate transition enabled in $(q, \mathbf{x})$ we have

$$Succ(q, \mathbf{x}, \Pi, \theta) = L_1 \cup L_2 \cup L_3$$

where

$$L_1 = \{(q', \mathbf{x}', \Pi, \theta) : (q, \mathbf{x}) \xrightarrow{\tau} (q', \mathbf{x}')\}$$

for every immediate transition τ such that τ is non-conflicting or belongs to the job whose priority on the respective machine is higher than those of all competing jobs. In addition, if there is a conflict on m involving a new job i whose priority compared to job i^*, having the highest priority so far, has not yet been determined, we have

$$L_2 = \{(q, \mathbf{x}, \Pi \cup \{i^* \prec i\}, \theta)\}$$

and

$$L_3 = \{(q, \mathbf{x}, \Pi \cup \bigcup_{\{i' : i' \nparallel_m i\}} \{i \prec i'\}, \theta)\}.$$

The successor in L_2 represent the choice to prefer i^* over i (the priority of i relative to other waiting jobs will be determined only after i^* terminates), while S_3 represents the choice of preferring i over all other jobs.

Using this definition we can construct a search algorithm that explores all the efficient runs of $\mathcal{A}$.

Algorithm 1 (Forward Reachability for Stopwatch Automata)

Waiting:=$\{(s, \mathbf{0}, \emptyset, 0)\}$;
while *Waiting* $\neq \emptyset$; **do**
  Pick $(q, \mathbf{x}, \Pi, \theta) \in$ *Waiting;*
  For every $(q', \mathbf{x}', \Pi', \theta') \in Succ(q, \mathbf{x}, \Pi, \theta)$;
    Insert $(q', \mathbf{x}', \Pi', \theta')$ *into Waiting;*
  Remove $(q, \mathbf{x}, \Pi, \theta)$ *from Waiting*
end

The length of the shortest path is the least θ such that $(f, \mathbf{x}, \Pi, \theta)$ is explored by the algorithm.

This exhaustive search algorithm can be improved into a best-first search as follows (similar ideas were investigated in (BFH$^+$01)). We define an evaluation function for estimating the quality of configurations.

$$E((q_1, \ldots, q_n), (v_1, \ldots, v_n), \Pi, \theta) = \theta + \max\{g^i(q_i, v_i)\}_{i=1}^n$$

where g^i is the previously-defined ranking function associated with each automaton $\mathcal{A}^i$. Note that $\max\{g^i\}$ gives the most optimistic estimation of the *remaining* time, assuming that no job will have to wait. It is not hard to see that $E(q, \mathbf{x}, \Pi, \theta)$ gives a lower bound on the length of every complete run which passes through $(q, \mathbf{x})$ at time θ.

The following algorithm orders the waiting list of configurations according to their evaluation. It is guaranteed to produce the optimal path because it stops the exploration only when it is clear that the unexplored states cannot lead to schedules better than those found so far.

Algorithm 2 (Best-first Forward Reachability)

Waiting:=$\{(s, \mathbf{0}, \emptyset, 0)\}$;
Best:=∞
$(q, \mathbf{x}, F, \theta)$:= *first in Waiting;*
while *Best* $> E(q, \mathbf{x}, F, \theta)$
do
  $(q, \mathbf{x}, \Pi, \theta)$:= *first in Waiting;*
  For every $(q', \mathbf{x}', \Pi', \theta') \in Succ(q, \mathbf{x}, \Pi, \theta)$;
    if $q' = f$ **then**
      Best:=$\min\{$*Best*,$E((q', \mathbf{x}', \Pi', \theta'))\}$
    else
      Insert $(q', \mathbf{x}', \Pi', \theta')$ *into Waiting;*
  Remove $(q, \mathbf{x}, \Pi, \theta)$ *from Waiting*
end

Using this algorithm we were able the find optimal schedules of systems with up to 8 jobs and 4 machines (12^8 discrete states and 8 clocks). In order to treat larger problems we abandon optimality and use a heuristic algorithm which can quickly generate sub-optimal solutions. The algorithm is a mixture of breadth-first and best-first search with a fixed number w of explored nodes at any level of the automaton. For every level we take the w best (according to E) nodes, generate their successors but explore only the best w among them, and so on.

In order to test this heuristics we took 16 problems among the most notorious job-shop scheduling problems.[5] For each of these problems we have applied our algorithms for different choices of w, both forward and backward (it takes, on the average few minutes for each problem). In Table we compare our best results on these problems to the most recent results reported by Le Pape and Baptiste (PB96; PB97) where the problem was solved using state-of-the-art constraint satisfaction techniques. As the table shows, the results our first prototype are very close to the optimum.

[5]The problems are taken from `ftp://mscmga.ms.ic.ac.uk/pub/jobshop1.txt`

problem			non preempt	preemptive			
name	#j	#m	optimum	optimum	(PB96; PB97)	stopwatch	deviation
LA02	10	5	655	655	655	655	0.00 %
FT10	10	10	930	900	900	911	1.21 %
ABZ5	10	10	1234	1203	1206	1250	3.76 %
ABZ6	10	10	943	924	924	936	1.28 %
ORB1	10	10	1059	1035	1035	1093	5.31 %
ORB2	10	10	888	864	864	884	2.26 %
ORB3	10	10	1005	973	994	1013	3.95 %
ORB4	10	10	1005	980	980	1004	2.39 %
ORB5	10	10	887	849	849	887	4.28 %
LA19	10	10	842	812	812	843	3.68 %
LA20	10	15	902	871	871	904	3.65 %
LA21	10	15	1046	1033	1033	1086	4.88 %
LA24	10	15	936	909	915	972	6.48 %
LA27	10	20	1235	1235	1235	1312	5.87 %
LA37	15	15	1397	1397	1397	1466	4.71 %
LA39	15	15	1233	1221	1221	1283	4.83 %

Table 2: The results of our implementation on the benchmarks. Columns #j and #m indicated the number of jobs and machines, followed by the best known results for non-preemptive scheduling, the known optimum for the preemptive case, the results of Le Pape and Baptiste, followed by our results and their deviation from the optimum.

Conclusion

We have demonstrated that the automata-theoretic approach to scheduling can be extended to preemptive scheduling and can be applied successfully to very large systems. Future work will investigate the applicability of this approach to scheduling of periodic tasks in real-time systems. In retrospect, it looks as if the undecidability results for *arbitrary* stopwatch automata have been taken too seriously. Timed and stopwatch automata arising from specific application domains have additional structure and their analysis might turn out to be feasible (see also recent results in (FPY02)).

Acknowledgment: We thank Eugene Asarin and Stavros Tripakis for numerous useful comments.

References

Y. Abdeddaïm and O. Maler, Job-Shop Scheduling using Timed Automata in G. Berry, H. Comon and A. Finkel (Eds.), *Proc. CAV'01*, 478-492, LNCS 2102, Springer 2001.

K. Altisen, G. Goessler, A. Pnueli, J. Sifakis, S. Tripakis and S. Yovine, A Framework for Scheduler Synthesis, *Proc. RTSS'99*, 154-163, IEEE, 1999.

R. Alur and D.L. Dill, A Theory of Timed Automata, *Theoretical Computer Science* 126, 183-235, 1994.

E. Asarin and O. Maler, As Soon as Possible: Time Optimal Control for Timed Automata, *Proc. HSCC'99*, 19-30, LNCS 1569, Springer, 1999.

G. Behrmann, A. Fehnker T.S. Hune, K.G. Larsen, P. Pettersson and J. Romijn, Efficient Guiding Towards Cost-Optimality in UPPAAL, *Proc. TACAS 2001*, 174-188, LNCS 2031, Springer, 2001.

J. Carlier and E. Pinson, A Practical Use of Jackson's Preemptive Schedule for Solving the Job-Shop Problem, *Annals of Operations Research* 26, 1990.

F. Cassez and K.G. Larsen, On the Impressive Power of Stopwatches, in C. Palamidessi (Ed.) *Proc. CONCUR'2000*, 138-152, LNCS 1877, Springer, 2000.

K. Cerans, *Algorithmic Problems in Analysis of Real Time System Specifications*, Ph.D. thesis, University of Latvia, Riga, 1992.

E. Fersman, P. Pettersson and W. Yi, Timed Automata with Asynchronous Processes: Schedulability and Decidability, *TACAS 2002*, this volume, 2002.

M. R. Garey and D. S Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

J. R. Jackson, Scheduling a Production Line to Minimize Maximum Tardiness, Research Report 43, *Management Sciences Research Project*, UCLA, 1955.

A.S. Jain and S. Meeran, Deterministic Job-Shop Scheduling: Past, Present and Future, *European Journal of Operational Research* 113, 390-434, 1999.

Y. Kesten, A. Pnueli, J. Sifakis and S. Yovine, Decidable Integration Graphs, *Information and Computation* 150, 209–243, 1999.

J. McManis and P. Varaiya, Suspension Automata: A Decidable Class of Hybrid Automata, in D.L Dill (Ed.), *Proc. CAV'94*, 105-117, LNCS 818, Springer, 1994.

P. Niebert, S. Tripakis S. Yovine, Minimum-Time Reachability for Timed Automata, *IEEE Mediteranean Control Conference*, 2000.

P. Niebert and S. Yovine, Computing Efficient Operation Schemes for Chemical Plants in Multi-batch Mode, *European Journal of Control* 7, 440-453, 2001.

C. Le Pape and P. Baptiste, A Constraint-Based Branch-and-Bound Algorithm for Preemptive Job-Shop Scheduling, *Proc. of Int. Workshop on Production Planning and Control*, Mons, Belgium, 1996.

C. Le Pape and P. Baptiste, An Experimental Comparaison of Constraint-Based Algorithms for the Preemptive Job-shop Sheduling Problem, *CP97 Workshop on Industrial Constraint-Directed Scheduling*, 1997.

Plan Validation for Extended Goals under Partial Observability (preliminary report)

Piergiorgio Bertoli, Alessandro Cimatti, Marco Pistore, Paolo Traverso
ITC-IRST
Via Sommarive 18, 38050 Povo, Trento, Italy
{bertoli,cimatti,pistore,traverso}@irst.itc.it

Abstract

The increasing interest in planning in nondeterministic domains by model checking has seen the recent development of two complementary research lines. In the first, planning is carried out considering extended goals, expressed in the CTL temporal logic, but has been developed under the simplifying hypothesis of full observability. In the second, simple reachability goals have been tackled under the more general hypothesis of partial observability. The combination of extended goals and partial observability for nondeterministic domains is, to the best of our knowledge, an open problem, whose solution turns out to be by no means trivial.

This paper is a first step towards a full, principled merging of the two research lines, in order to be able to describe complex and significant goals over realistic domains. We make the following contributions. First, we define a general framework, encompassing both partial observability and temporally extended goals. Second, we define the K-CTL goal language, that extends CTL with a knowledge operator that allows to reason about the information that can be acquired at run-time. This is necessary to deal with partially observable domains, where limited "knowledge" about the domain state is a key issue. Then, we define a general mechanism for plan validation for K-CTL goals, based on the idea of monitor. A monitor plays the role of evaluating the truth of knowledge predicates, and allows us to fully exploit the CTL model checking machinery. This paper restricts to plan validation of K-CTL goals. However, it provides a solid basis for tackling the problem of planning for K-CTL goals under partial observability.

Introduction

Planning in nondeterministic domains has been devoted increasing interest, and different research lines have been developed. On one side, planning algorithms for tackling complex, temporally extended goals have been proposed in (Kabanza, Barbeau, & St-Denis 1997; Pistore & Traverso 2001). This research line is motivated by the fact that many real-life problems require temporal operators for expressing complex goals. For instance, it may be required, always avoiding a dangerous condition P, to achieve a certain condition G, and from there on to try to maintain a desirable condition H. This research activity is carried out under the assumption that the planning domain is fully observable. On the other side, in the works (Bertoli *et al.* 2001; Weld, Anderson, & Smith 1998; Bonet & Geffner 2000; Rintanen 1999) the hypothesis of full observability is relaxed in order to deal with realistic situations, where the whole status of the domain is by no means accessible to the plan executor. The key difficulty is in dealing with the uncertainty arising from the inability to determine precisely what the status of the domain will be at run-time. These approaches are however limited to the case of simple reachability goals.

Tackling the problem of planning for temporally extended goals under the assumption of partial observability is not trivial. The goal of this paper is to settle a basic but general framework that encompasses all the aspects that are relevant to deal with real-world domains and problems, which feature partial observability and extended goals, and gives a precise definition of the problem. This framework will be a basis for solving the problem in its full complexity.

We first provide a framework based on the Planning as Model Checking paradigm. We give a general notion of planning domain, in terms of finite state machine, where actions can be nondeterministic, and different forms of sensing can be captured. We define a general notion of plan, that is also seen as a finite state machine, with internal control points that allow to encode sequential, conditional, and iterative behaviors. The conditional behavior is based on sensed information, i.e., information that becomes available during plan execution. By connecting a plan and a domain, we obtain a closed system, that induces a (possibly infinite) computation tree, representing all the possible executions. Temporally extended goals are defined as CTL formulae. In this frameworks, the standard machinery of model checking for CTL temporal logic defines when a plan satisfies a temporally extended goal under partial observability. As a side result, this shows that a standard model checking tool can be applied as a black box to the validation of complex plans even in the presence of limited observability.

Unfortunately, this is by no means the end of the story: CTL is inadequate to express goals in presence of partial observability. Even in the simple case of conformant planning, i.e., when a reachability goal has to be achieved with no information available at run-time, CTL is not expressive enough. This is due to the fact that the basic propositions in CTL only refer to the status of the world, but do not take into account the aspects related to "knowledge", i.e., what is known at run-time. In fact, conformant planning is the

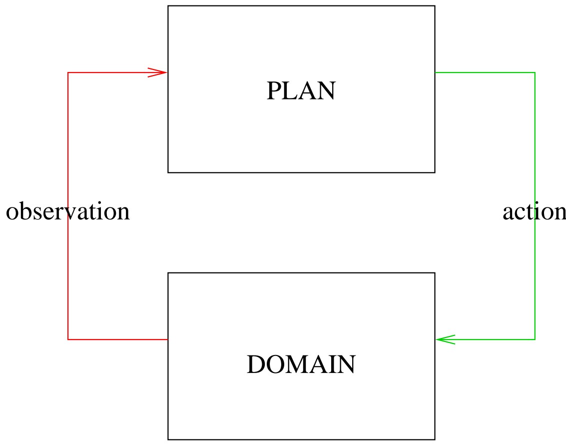


Figure 1: The general framework.

problem of finding a plan after which we *know* that a certain condition is achieved. In order to overcome this limitation, we define the K-CTL goal language, obtained by extending CTL with a knowledge operator, that allows to express knowledge atoms, i.e., what is known at a certain point in the execution. Then, we provide a first practical solution to the problem of checking if a plan satisfies a K-CTL goal. This is done by associating a given K-CTL goal with a suitable monitor, i.e., an observer system that is able to recognize the truth of knowledge atoms. Standard model checking techniques can be then applied to the domain-plan system enriched with the monitor.

The work presented in this paper focuses on setting the framework and defining plan validation procedures, and does not tackle the problem of plan synthesis. Still, the basic concepts presented in this paper formally distinguish what is known at planning time versus what is known at run time, and provide a solid basis for tackling the problem of plan synthesis for extended goals under partial observability.

The paper is structured as follows. First we provide a formal framework for partially observable, nondeterministic domains, and for plans over them. Then we incrementally define CTL goals and K-CTL goals; for each of those classes of goals, we describe a plan validation procedure. We wrap up with some concluding remarks and future and related work.

The Framework

The intuition underlying our framework is outlined in Figure 1. A domain is a generic system, possibly with its own dynamics, such as a power plant or an aircraft. The plan can control the evolutions of the domain by triggering *actions*. We assume that, at execution time, the state of the domain is only partially visible to the plan; the part of a domain state that is visible to the plan is called the *observation* of the state. In essence, planning is building a suitable plan that can guide the evolutions of the domain in order to achieve the specified goals.

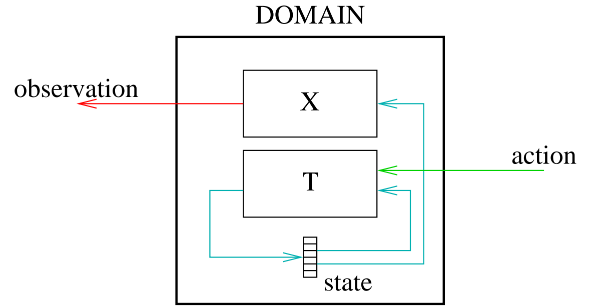


Figure 2: The model of the domain.

Planning Domains

A planning domain is defined in terms of its *states*, of the *actions* it accepts, and of the possible *observations* that the domain can exhibit. Some of the states are marked as valid *initial states* for the domain. A *transition function* describes how (the execution of) an action leads from one state to possibly many different states. Finally, an *observation function* defines what observations are associated to each state of the domain.

Definition 1 (planning domain) *A* nondeterministic planning domain with partial observability *is a tuple* $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{U}, \mathcal{I}, \mathcal{T}, \mathcal{X} \rangle$*, where:*

- $\mathcal{S}$ *is the set of* states.
- $\mathcal{A}$ *is the set of* actions.
- $\mathcal{U}$ *is the set of* observations.
- $\mathcal{I} \subseteq \mathcal{S}$ *is the set of* initial states*; we require* $\mathcal{I} \neq \emptyset$.
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow 2^{\mathcal{S}}$ *is the* transition function*; it associates to each current state* $s \in \mathcal{S}$ *and to each action* $a \in \mathcal{A}$ *the set* $\mathcal{T}(s,a) \subseteq \mathcal{S}$ *of next states; we require that for each* $s \in \mathcal{S}$ *there is some* $a \in \mathcal{A}$ *such that* $\mathcal{T}(s,a) \neq \emptyset$.
- $\mathcal{X} : \mathcal{S} \rightarrow 2^{\mathcal{U}}$ *is the* observation function*; it associates to each state* s *the set of possible observations* $\mathcal{X}(s) \subseteq \mathcal{U}$*; we require that for each* $s \in \mathcal{S}$, $\mathcal{X}(s) \neq \emptyset$.

A picture of the model of the domain corresponding to this Definition is given in Figure 2. Technically, a domain is described as a nondeterministic Moore machine, whose outputs (i.e., the observations) depend only on the current state of the machine, not on the input action. Uncertainty is allowed in the initial state and in the outcome of action execution. Also, the observation associated to a given state is not unique. This allows modeling noisy sensing and lack of information.

Notice that the definition tries to provide a general notion of domain, abstracting away from the language that is used to describe the domain. For instance, a planning domain is usually defined in terms of a set of *fluents* (or state variables), and each state corresponds to an assignment to the fluents. Similarly, the possible observations of the domain, that are primitive entities in the definition, can be presented by means of a set of *observation variables*, as in (Bertoli *et al.* 2001): each observation variable can be seen as an input port in the plan, while an observation is defined as a valuation to all the observation variables. The definition of

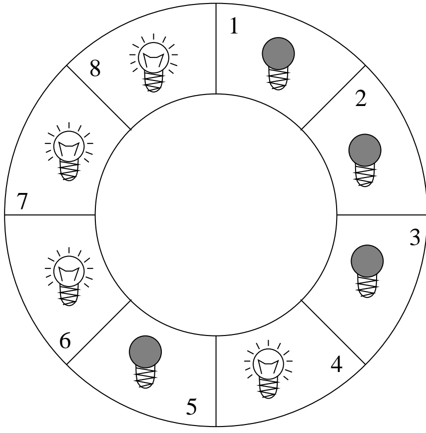


Figure 3: A simple domain.

planning domain does not allow for a direct representation of *action-dependent* observations, that is, observations that depend on the last executed action. However, these observations can be easily modeled by representing explicitly in the state of the domain (the relevant information on) the last executed action.

In the following example, that will be used throughout the paper, we will outline the different aspects of the defined framework.

Example 2 *Consider the domain represented in Figure 3. It consists of a ring of N rooms. Each room contains a light that can be on or off, and a button that, when pressed, switches the status of the light. A robot may move between adjacent rooms (actions* go-right *and* go-left*) and switch the lights (action* switch-light*).*

Uncertainty in the domain is due to an unknown initial room and initial status of the lights. Moreover, the lights in the rooms not occupied by the robot may be nondeterministically switched on without the direct intervention of the robot.

The domain is only partially observable: the rooms are indistinguishable, and, in order to know the current status of the light in the current room, the robot must perform a sense *action.*

A state of the domain is defined in terms of the following fluents:

- *fluent* room*, that ranges from 1 to N, describes in which room the robot is currently in;*
- *boolean fluents* light-on$[i]$*, for $i \in \{1, ...N\}$, describe whether the light in room i is on;*
- *boolean fluent* sensed*, describes whether last action was a* sense *action.*

Any state with fluent sensed *false is a possible initial state.*

The actions are go-left*,* go-right*,* switch-light*,* sense*, and* wait*. Action* wait *corresponds to the robot doing nothing during a transition (the state of the domain may change only due to the lights that may be turned on without the intervention of the robot). The effects of the other actions have been already described.*

The observation is defined in terms of observation variable light*. If fluent* sense *is true, then observation variable* light *is true if and only if the light is on in the current room. If fluent* sense *is false (no sensing has been done in the last action), then observation* light *may be nondeterministically true or false.*

The mechanism of observations allowed by the model presented in Definition 1 is rather general. It can model *no observability* and *full observability* as special cases. *No observability* (conformant planning) is represented by defining $\mathcal{U} = \{\bullet\}$ and $\mathcal{X}(s) = \{\bullet\}$ for each $s \in \mathcal{S}$. That is, observation $\bullet$ is associated to all states, thus conveying no information. *Full observability* is represented by defining $\mathcal{U} = \mathcal{S}$ and $\mathcal{X}(s) = \{s\}$. That is, the observation carries all the information contained in the state of the domain.

Plans and plan executions

Now we present a very general definition of plans, that encode sequential, conditional and iterative behaviors, and are expressive enough for dealing with partial observability and with extended goals. In particular, we need plans where the selection of the action to be executed depends on the observations, and on an "internal state" of the executor, that can take into account, e.g., the knowledge gathered during the previous execution steps. A plan is defined in terms of an *action function* that, given an observation and a *context* encoding the internal state of the executor, specifies the action to be executed, and in terms of a *context function* that evolves the context.

Definition 3 (plan) *A plan for domain $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{U}, \mathcal{I}, \mathcal{T}, \mathcal{X} \rangle$ is a tuple $\Pi = \langle \Sigma, \sigma_0, \alpha, \epsilon \rangle$, where:*

- *Σ is the set of* plan contexts.
- *$\sigma_0 \in \Sigma$ is the* initial context.
- *$\alpha : \Sigma \times \mathcal{U} \rightharpoonup \mathcal{A}$ is the* action function*; it associates to a plan context c and an observation o an action $\alpha(c, o)$ to be executed.*
- *$\epsilon : \Sigma \times \mathcal{U} \rightharpoonup \Sigma$ is the* context evolutions function*; it associates to a plan context c and an observation o a new plan context $\epsilon(c, o)$.*

A picture of the model of plans is given in Figure 4. Technically, a plan is described as a Mealy machine, whose outputs (the action) may depend on the inputs (the observation). Functions α and ϵ are deterministic (we do not consider nondeterministic plans), and can be partial, since a plan may be undefined on the context-observation pairs that are never reached during execution.

Example 4 *We consider two plans for the domain of Figure 3. According to plan Π_1, the robot moves cyclically trough the rooms, and turns off the lights whenever they are on. The plan is cyclic, that is, it never ends. The plan has three contexts E, S, and L, corresponding to the robot having just entered a room (E), the robot having sensed the light (S), and the robot being about to leave the room after having turned off the light (L) . The initial context is E. Functions*

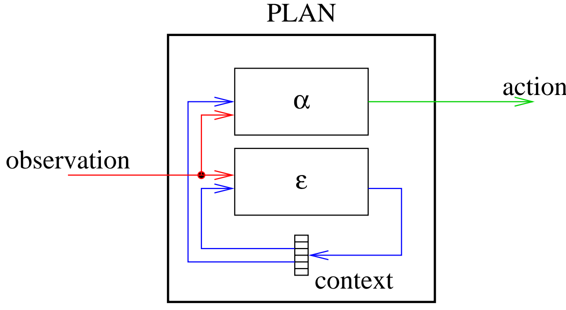


Figure 4: The model of the plan.

α and ϵ for Π_1 are defined by the following table:

c	o	$\alpha(c,o)$	$\epsilon(c,o)$
E	*any*	*sense*	S
S	*light* $= \top$	*switch-light*	L
S	*light* $= \bot$	*go-right*	E
L	*any*	*go-right*	E

In plan Π_2, the robot traverses all the rooms and turns on the lights; the robot stops once all the rooms have been visited. The plan has contexts of the form (E,i), (S,i), and (L,i), where i represents the number of rooms to be visited. The initial context is $(E, N{-}1)$, where N is the number of rooms. Functions α and ϵ for Π_2 are defined by the following table:

c	o	$\alpha(c,o)$	$\epsilon(c,o)$
(E,i)	*any*	*sense*	(S,i)
(S,i)	*light* $= \bot$	*switch-light*	(L,i)
$(S,0)$	*light* $= \top$	*wait*	$(L,0)$
$(S,i{+}1)$	*light* $= \top$	*go-right*	(E,i)
$(L,0)$	*any*	*wait*	$(L,0)$
$(L,i{+}1)$	*any*	*go-right*	(E,i)

Since both the plan and the domain are finite state machines, we can use the standard techniques for model checking synchronous compositions. We can describe the execution of a plan over a domain in terms of transitions between configurations that describe the state of the domain and of the plan. These concepts are formalized in the following definition.

Definition 5 (configuration) *A configuration for domain $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{U}, \mathcal{I}, \mathcal{T}, \mathcal{X} \rangle$ and plan $\Pi = \langle \Sigma, \sigma_0, \alpha, \epsilon \rangle$ is a pair (s,o,c) such that $s \in \mathcal{S}$, $o \in \mathcal{X}(s)$, and $c \in \Sigma$.*
Configuration (s,o,c) may evolve into configuration (s',o',c'), written $(s,o,c) \to (s',o',c')$, if:

- $s' \in \mathcal{T}(s, \alpha(c,o))$,
- $o' \in \mathcal{X}(s')$*, and*
- $c' = \epsilon(c,o)$.

Configuration (s,o,c) is initial if $s \in \mathcal{I}$ and $c = \sigma_0$.
The reachable configurations for domain $\mathcal{D}$ and plan Π are defined by the following inductive rules:

- *if (s,o,σ_0) is initial, then it is reachable;*
- *if (s,o,c) is reachable and $(s,o,c) \to (s',o',c')$, then (s',o',c') is also reachable.*

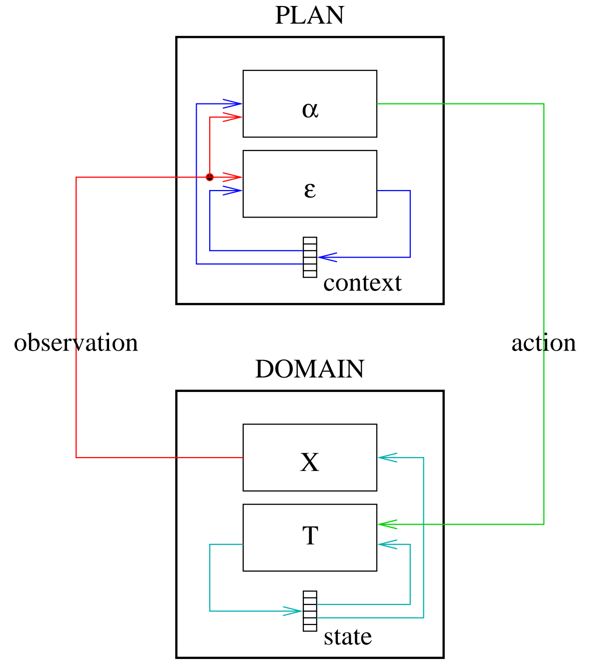


Figure 5: Plan execution.

Notice that we include the observations in the configurations. This is necessary in order to take into account the fact that more than one observation may correspond to the same state. On the other hand, we do not represent explicitly the action in the configuration, since it is a function of the context and of the observation.

We are interested in plans that define an action to be executed for each reachable configuration. These plans are called *executable*.

Definition 6 (executable plan) *Plan Π is executable on Domain $\mathcal{D}$ if the following condition holds for all the reachable configurations (s,o,c):*

- *$\alpha(c,o)$ and $\epsilon(c,o)$ are defined;*
- *$\mathcal{T}(s, \alpha(c,o)) \neq \emptyset$.*

The executions of a plan on a domain correspond to the synchronous executions of the two machines corresponding to the domain and the plan, as shown in Figure 5. At each time step, the flow of execution proceeds as follows. The execution starts from a configuration that defines the current domain state, observation, and context. First, based on the current context and observation, the plan determines the action to be executed (function α) and the next context (function ϵ). Then, the new state of the domain is determined by function $\mathcal{T}$ from the current state and action. Finally, the new observation is determined by applying nondeterministic function $\mathcal{X}$ to the new state. At the end of the cycle, the newly computed values for the domain state, the observation, and the context define the value of the new configuration. An execution of the plan is basically a sequence of subsequent configurations. Due to the nondeterminism in the domain, we may have an infinite number of different executions of a plan. We provide a finite presentation of these

executions with an *execution structure*, i.e, a Kripke Structure (Emerson 1990) whose set of states is the set of reachable configurations of the plan, and whose transition relation corresponds to the transitions between configurations.

Definition 7 (execution structure) *The execution structure corresponding to domain $\mathcal{D}$ and plan Π is the Kripke structure $K = \langle Q, Q_0, R\rangle$, where:*

- *Q is the set of reachable configurations;*
- *$Q_0 = \{(s, o, \sigma_0) \in Q : s \in \mathcal{I} \wedge o \in \mathcal{X}(s)\}$ are the initial configurations;*
- *$R = \{((s, o, c), (s', o', c')) \in Q^2 : (s, o, c) \to (s', o', c')\}$.*

Temporally extended goals: CTL

Extended goals are expressed with CTL formulas. CTL allows for temporal operators that state temporal conditions on plan executions. Moreover, CTL universal and existential path quantifiers allow us to specify requirements that take into account the fact that a plan may nondeterministically result in many different executions.

We assume that a set $\mathcal{B}$ of basic propositions is defined on domain $\mathcal{D}$. Moreover, we assume that for each $b \in \mathcal{B}$ and $s \in \mathcal{S}$, predicate $s \models_0 b$ holds if and only if basic proposition b is true on state s.

Definition 8 (CTL) *The goal language CTL is defined by the grammar:*

$$\begin{aligned} g &::= p \mid g \wedge g \mid g \vee g \mid \mathrm{AX}\, g \mid \mathrm{EX}\, g \\ &\quad\ \mathrm{A}(g \,\mathrm{U}\, g) \mid \mathrm{E}(g \,\mathrm{U}\, g) \mid \mathrm{A}(g \,\mathrm{W}\, g) \mid \mathrm{E}(g \,\mathrm{W}\, g) \\ p &::= b \mid \neg p \mid p \wedge p \end{aligned}$$

where b is a basic proposition.

CTL combines temporal operators and path quantifiers. "X", "U", and "W" are the "next time", "(strong) until", and "weak until" temporal operators, respectively. "A" and "E" are the universal and existential path quantifiers, where a path is an infinite sequence of states. They allow us to specify requirements that take into account nondeterminism. Intuitively, the formula $\mathrm{AX}\, g$ ($\mathrm{EX}\, g$) means that g holds in every (in some) immediate successor of the current state. $\mathrm{A}(g_1 \,\mathrm{U}\, g_2)$ ($\mathrm{E}(g_1 \,\mathrm{U}\, g_2)$) means that for every path (for some path) there exists an initial prefix of the path such that g_2 holds at the last state of the prefix and g_1 holds at all the other states along the prefix. The formula $\mathrm{A}(g_1 \,\mathrm{W}\, g_2)$ ($\mathrm{E}(g_1 \,\mathrm{W}\, g_2)$) is similar to $\mathrm{A}(g_1 \,\mathrm{U}\, g_2)$ ($\mathrm{E}(g_1 \,\mathrm{U}\, g_2)$) but allows for paths where g_1 holds in all the states and g_2 never holds. Formulas $\mathrm{AF}\, g$ and $\mathrm{EF}\, g$ (where the temporal operator "F" stands for "future" or "eventually") are abbreviations of $\mathrm{A}(\top \,\mathrm{U}\, g)$ and $\mathrm{E}(\top \,\mathrm{U}\, g)$, respectively. $\mathrm{AG}\, g$ and $\mathrm{EG}\, g$ (where "G" stands for "globally" or "always") are abbreviations of $\mathrm{A}(g \,\mathrm{W}\, \perp)$ and $\mathrm{E}(g \,\mathrm{W}\, \perp)$, respectively. A remark is in order: even if $\neg$ is allowed only in front of basic propositions, it is easy to define $\neg g$ for a generic CTL formula g, by "pushing down" the negations: for instance $\neg \mathrm{AX}\, g \equiv \mathrm{EX}\, \neg g$ and $\neg \mathrm{A}(g_1 \,\mathrm{W}\, g_2) \equiv \mathrm{E}(\neg g_2 \,\mathrm{U} (\neg g_1 \wedge \neg g_2))$.

Goals as CTL formulas allow us to specify different interesting requirements on plans. Let us consider first some examples of *reachability goals*. $\mathrm{AF}\, g$ ("reach g") states that a condition should be guaranteed to be reached by the plan, in spite of nondeterminism. $\mathrm{EF}\, g$ ("try to reach g") states that a condition might possibly be reached, i.e., there exists at least one execution that achieves the goal. A reasonable reachability requirement that is stronger than $\mathrm{EF}\, g$ is $\mathrm{A}(\mathrm{EF}\, g \,\mathrm{W}\, g)$: it allows for those execution loops that have always a possibility of terminating, and when they do, the goal g is guaranteed to be achieved.

We can distinguish among different kinds of *maintainability goals*, e.g., $\mathrm{AG}\, g$ ("maintain g"), $\mathrm{AG}\, \neg g$ ("avoid g"), $\mathrm{EG}\, g$ ("try to maintain g"), and $\mathrm{EG}\, \neg g$ ("try to avoid g"). For instance, a robot should never harm people and should always avoid dangerous areas. Weaker requirements might be needed for less critical properties, like the fact that the robot should try to avoid to run out of battery.

We can *compose reachability and maintainability goals*. $\mathrm{AF}\, \mathrm{AG}\, g$ states that a plan should guarantee that all executions reach eventually a set of states where g can be maintained. For instance, an air-conditioner controller is required to reach eventually a state such that the temperature can then be maintained in a given range. Alternatively, if you consider the case in which a pump might fail to turn on when it is selected, you might require that "there exists a possibility" to reach the condition to maintain the temperature in a desired range ($\mathrm{EF}\, \mathrm{AG}\, g$). As a further example, the goal $\mathrm{AG}\, \mathrm{EF}\, g$ intuitively means "maintain the possibility of reaching g".

Reachability-preserving goals make use of the "until operators" ($\mathrm{A}(g_1 \,\mathrm{U}\, g_2)$ and $\mathrm{E}(g_1 \,\mathrm{U}\, g_2)$) to express reachability goals while some property must be preserved. For instance, an air-conditioner might be required to reach a desired temperature while leaving at least n of its m pumps off.

Notice that in all examples above, the ability of composing formulas with universal and existential path quantifiers is essential. Logics that do not provide this ability, like LTL (Emerson 1990), cannot express these kinds of goals[1].

Given an execution structure K and an extended goal g, we now define when a goal g is true in (s, o, c), written $K, (s, o, c) \models g$ by using the standard semantics for CTL formulas over the Kripke Structure K.

Definition 9 (semantics of CTL) *Let K be a Kripke structures with configurations as states. We extend $\models_0$ to propositions as follows:*

- *$s \models_0 \neg p$ iff not $s \models_0 p$;*
- *$s \models_0 p \wedge p'$ iff $s \models_0 p$ and $s \models_0 p'$.*

We define $K, q \models g$ as follows:

- *$K, q \models p$ iff $q = (s, o, c)$ and $s \models_0 p$.*
- *$K, q \models g \wedge g'$ iff $K, q \models g$ and $K, q \models g'$.*
- *$K, q \models g \vee g'$ iff $K, q \models g$ or $K, q \models g'$.*
- *$K, q \models \mathrm{AX}\, g$ iff for all q', if $q \to q'$ then $K, q' \models g$.*

[1] In general, CTL and LTL have incomparable expressive power (see (Emerson 1990) for a comparison). We focus on CTL since it provides the ability of expressing goals that take into account nondeterminism.

- $K, q \models \mathrm{EX}\, g$ *iff there is some* q' *such that* $q \rightarrow q'$ *and* $K, q' \models g$.
- $K, q \models \mathrm{A}(g \,\mathrm{U}\, g')$ *iff for all* $q = q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \cdots$ *there is some* $i \geq 0$ *such that* $K, q_i \models g'$ *and* $K, q_j \models g$ *for all* $0 \leq j < i$.
- $K, q \models \mathrm{E}(g \,\mathrm{U}\, g')$ *iff there is some* $q = q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \cdots$ *and some* $i \geq 0$ *such that* $K, q_i \models g'$ *and* $K, q_j \models g$ *for all* $0 \leq j < i$.
- $K, q \models \mathrm{A}(g \,\mathrm{W}\, g')$ *iff for all* $q = q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \cdots$, *either* $K, q_j \models g$ *for all* $j \geq 0$, *or there is some* $i \geq 0$ *such that* $K, q_i \models g'$ *and* $K, q_j \models g$ *for all* $0 \leq j < i$.
- $K, q \models \mathrm{E}(g \,\mathrm{W}\, g')$ *iff there is some* $q = q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \cdots$ *such that either* $K, q_j \models g$ *for all* $j \geq 0$, *or there is some* $i \geq 0$ *such that* $K, q_i \models g'$ *and* $K, q_j \models g$ *for all* $0 \leq j < i$.

We define $K \models g$ *iff* $K, q_0 \models g$ *for all the initial configurations* $q_0 \in Q_0$ *of* K.

Plan validation

The definition of when a plan satisfies a goal follows.

Definition 10 (plan validation for CTL goals) *Plan* Π *satisfies CTL goal* g *on domain* $\mathcal{D}$, *written* $\Pi \models_{\mathcal{D}} g$, *if* $K \models g$, *where* K *is the execution structure corresponding to* $\mathcal{D}$ *and* Π.

In the case of CTL goals, the *plan validation* task amounts to CTL model checking. Given a domain $\mathcal{D}$ and a plan Π, the corresponding execution structure K is built as described in Definition 7 and standard model checking algorithms are run on K in order to check whether it satisfies goal g. This simple consideration has an important consequence: the problem of plan validation under partial observability can be tackled with standard model checking machinery, and, more importantly, with model checking tools.

We describe now some goals for the domain of Figure 3. We recall that the initial room of the robot is uncertain, and that light can be turned on (but not off) without the intervention of the robot.

Example 11 *The first goal we consider is*

$$\mathrm{AF}\,(\neg \textit{light-on}[3]),$$

which requires that the light of room 3 is eventually off. Plan Π_1 *satisfies this goal: eventually, the robot will be in room 3 and will turn out the light if it is on.*

There is no plan that satisfies to following goal:

$$\mathrm{AF}\,\mathrm{AG}\,(\neg \textit{light-on}[3]),$$

which requires that the light in room 3 is turned off and stays then off forever. This can be only guaranteed if the robot stays in room 3 forever, and it is impossible to guarantee this condition in this domain: due to the partial observability of the domain, the robot does never know he is in room 3.

Plan Π_1 *satisfies the following goal*

$$\bigwedge_{i \in 1,\ldots,N} \mathrm{AG}\,\mathrm{AF}\,(\neg \textit{light-on}[i]),$$

which requires that the light in every room is turned off infinitely often. On the other hand, it does not satisfy the following goal

$$\mathrm{AG}\,\mathrm{AF} \bigwedge_{i \in 1,\ldots,N} (\neg \textit{light-on}[i]),$$

which requires that the lights in all the rooms are off at the same time infinitely often. Indeed, the nondeterminism in the domain may cause light to turn on at any time.

While plan Π_1 *does not guarantee that all the lights will be eventually off, it always leaves open the possibility that such a configuration will be eventually reached. That is, plan* Π_1 *satisfies the following goal*

$$\mathrm{AG}\,\mathrm{EF} \bigwedge_{i \in 1,\ldots,N} (\neg \textit{light-on}[i]),$$

which asserts that in each moment (AG *) there is the possibility of reaching (*EF *) the desired configuration.*

Finally, consider the goal

$$\mathrm{AG}\,\mathrm{AF} \bigwedge_{i \in 1,\ldots,N} (\textit{light-on}[i]),$$

which requires that the lights in all the rooms are on *at the same time infinitely often. It is satisfied by plan* Π_2*: once all the rooms have been explored, and the lights have been turned on, they will stay on forever.*

Goals over knowledge: K-CTL

Unfortunately, under the hypothesis of partial observability, CTL is not adequate to express many interesting goals. Consider for instance the first goal in Example 11. Notice that the robot will never "know" when condition $\neg$light-on[3] holds. In fact, the robot cannot detect when it is in room 3, and once that room is left, the light can be turned on again. The inadequacy of CTL is related with the limited knowledge that the plan execution has to face at run-time, because of different forms of uncertainty (e.g., in the initial condition, and in the execution of actions) that can not be ruled out by the partial observability. In order to tackle this problem, in this section we extend CTL with a knowledge operator $\mathbf{K}\, p$. Goal $\mathbf{K}\, p$ expresses the fact that the executor *knows*, or *believes*, that all the possible current states of the domain, that are compatible with the past history and the past observations, satisfy condition p. This allows, for instance, for expressing reachability under partial observability, by stating a goal of the kind $\mathrm{AF}\,\mathbf{K}\, g$.

Definition 12 (K-CTL) *The goal language K-CTL is defined by the grammar:*

$$\begin{aligned} g &::= p \mid \mathbf{K}\, p \mid g \wedge g \mid g \vee g \mid \mathrm{AX}\, g \mid \mathrm{EX}\, g \\ &\quad\ \mathrm{A}(g \,\mathrm{U}\, g) \mid \mathrm{E}(g \,\mathrm{U}\, g) \mid \mathrm{A}(g \,\mathrm{W}\, g) \mid \mathrm{E}(g \,\mathrm{W}\, g) \\ p &::= b \mid \neg p \mid p \wedge p \end{aligned}$$

where b *is a basic proposition.*

In order to define when a plan satisfies a given K-CTL goal, we have to extend the execution structure with an additional piece of information, called *belief state*. A belief state is a set of possible candidate states of the domain that we cannot distinguish given the past actions and the observations collected so far.

Definition 13 (bs-configuration) *A bs-configuration for domain $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{U}, \mathcal{I}, \mathcal{T}, \mathcal{X} \rangle$ and plan $\Pi = \langle \Sigma, \sigma_0, \alpha, \epsilon \rangle$ is a pair (s, o, c, bs) such that $s \in \mathcal{S}$, $o \in \mathcal{X}(s)$, $c \in \Sigma$, and $bs \in 2^{\mathcal{S}}$. We require:*

- *$s \in bs$ (the current state must belong to the belief state);*
- *if $\bar{s} \in bs$ then $o \in \mathcal{X}(\bar{s})$ (the states in the belief state must be compatible with the observed output).*

Bs-configuration (s, o, c, bs) may evolve into bs-configuration $(s', o'c', bs')$, written $(s, o, c, bs) \rightarrow (s', o'c', bs')$, if:

- *$s' \in \mathcal{T}(s, \alpha(c, o))$,*
- *$o' \in \mathcal{X}(s')$,*
- *$c' = \epsilon(c, o)$, and*
- *$bs' = \{\bar{s}' : \exists \bar{s} \in bs.\ \bar{s}' \in \mathcal{T}(\bar{s}, a) \wedge o' \in \mathcal{X}(\bar{s}')\}$.*

Bs-configuration (s, o, c, bs) is initial if $s \in \mathcal{I}$, $c = \sigma_0$, and $bs = \{\bar{s} \in \mathcal{I} : o \in \mathcal{X}(\bar{s})\}$.
The reachable bs-configurations are defined by trivially extending Definition 5.

Definition 14 (semantics of K-CTL) *Let K be a Kripke structures with bs-configurations as states. We define $K, q \models g$ by extending Definition 9 as follows:*

- *$K, q \models \mathbf{K}\, p$ iff $q = (s, o, c, bs)$ and $\bar{s} \models_0 p$ for all $\bar{s} \in bs$.*

We define $K \models g$ iff $K, q_0 \models g$ for all the initial configurations q_0 of K.

Plan validation for K-CTL goals

Also in the case of K-CTL, the definition of when a plan satisfies a goal is reduced to model checking.

Definition 15 (plan validation for K-CTL goals) *Plan Π satisfies K-CTL goal g on domain $\mathcal{D}$, written $\Pi \models_{\mathcal{D}} g$, if $K \models g$, where K is the bs-execution structure corresponding to $\mathcal{D}$ and Π.*

We consider now an additional set of K-CTL goals for the example.

Example 16 *In Example 11 we have seen that plan Π_1 satisfies goal $\mathrm{AF}\,(\neg \textit{light-on}[3])$. However, it does not satisfy goal*

$$\mathrm{AF}\,\mathbf{K}\,(\neg \textit{light-on}[3]).$$

In fact, this goal cannot be satisfied by any plan: due to the uncertainty on the room occupied by the robot, there is no way to "know" when the light in room 3 is turned off.

Goal

$$\mathrm{AF}\,\mathbf{K}\,(\textit{light-on}[3]),$$

instead, is satisfied by Π_2. Even if it is not possible to know when the robot is turning on the light in room 3, we "know" for sure that the light is on once the robot has visited all the rooms. Plan Π_2 satisfies also the more complex goal

$$\bigwedge_{i \in 1,\ldots,N} \mathrm{AF}\,\mathbf{K}\,(\textit{light-on}[i]).$$

According to Definition 15, the problem of checking whether a plan satisfies a K-CTL goal g is reduced to model checking formula g on the bs-execution structure corresponding to the plan. While theoretically sound, this approach is not practical, since the number of possible belief states for a given planning domain is exponential in the number of its states. This makes the exploration of a bs-execution structure infeasible for non-trivial domains.

In order to overcome this limitation, in this section we introduce a different approach for plan validation. This approach is based on the concept of *monitor*. A monitor is a machine that observes the execution of the plan on the domain and reports a belief state, i.e., a set of possible current states of the domain. Differently from the belief states that appear in a bs-configuration, the belief states reported by the monitor may be a super-set of the states that are compatible with the past history. As we will see, it is this possibility of approximating the possible current states that makes monitors usable in practice for validating plans.

Definition 17 (monitor) *A monitor for a domain $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{U}, \mathcal{I}, \mathcal{T}, \mathcal{X} \rangle$ is a tuple $\mathcal{M} = \langle \mathcal{MS}, m_0, \mathcal{MT}, \mathcal{MO} \rangle$, where:*

- *$\mathcal{MS}$ is the set of states of the monitor.*
- *$m_0 \in \mathcal{MS}$ is the initial state of the monitor.*
- *$\mathcal{MT} : \mathcal{MS} \times \mathcal{U} \times \mathcal{A} \rightharpoonup \mathcal{MS}$ is the transition function of the monitor; it associates to state m of the monitor, observation o, and action a, an updated state of the monitor $m' = \mathcal{MT}(m, o, a)$.*
- *$\mathcal{MO} : \mathcal{MS} \times \mathcal{U} \rightarrow 2^{\mathcal{S}}$ is the output function of the monitor; it associates to each state m of the monitor and observation o the corresponding belief state $\mathcal{MO}(m, o)$.*

Definition 18 (m-configuration) *A m-configuration for domain $\mathcal{D}$, plan Π and monitor $\mathcal{M}$ is a tuple (s, o, c, m) such that $s \in \mathcal{S}$, $o \in \mathcal{X}(s)$, $c \in \Sigma$, and $m \in \mathcal{MS}$.*
M-configuration (s, o, c, m) may evolve into m-configuration (s', o', c', m'), written $(s, o, c, m) \rightarrow (s', o', c', m')$, if:

- *$s' \in \mathcal{T}(s, \alpha(c, o))$,*
- *$o' \in \mathcal{X}(s')$,*
- *$c' = \epsilon(c, o)$, and*
- *$m' = \mathcal{MT}(m, o, \alpha(c, o))$.*

M-configuration (s, o, c, m) is initial if $s \in \mathcal{I}$, $c = \sigma_0$, and $m = m_0$.
The reachable m-configurations are defined by trivially extending Definition 5.

We say that a monitor is *correct* for a given domain and plan if the belief state reported by the monitor after a certain evolution contains *all* the states that are compatible with the observation gathered during the evolution. In the following definition, this property is expressed by requiring that there are no computations along which a state of the domain is reached that is not contained in the belief state reported by the monitor.

Definition 19 (correct monitor) *Monitor $\mathcal{M}$ is correct for domain $\mathcal{D}$ and plan Π if the following conditions holds for all the reachable m-configurations (s, o, c, m):*

- $s \in \mathcal{MO}(m, o)$;
- $\mathcal{MT}(m, o, \alpha(c, o))$ *is defined.*

From now on we consider only correct monitors.

We now define when a triple domain-plan-monitor satisfies a given K-CTL goal g. We start by defining the Kripke structure corresponding by the synchronous execution of the machines corresponding to domain, plan, and monitor.

Definition 20 (m-execution structure) *The m-execution structure corresponding to domain $\mathcal{D}$, plan Π, and monitor $\mathcal{M}$ is the Kripke structure $K = \langle Q, Q_0, R\rangle$, where:*

- *Q is the set of reachable m-configurations;*
- *Q_0 are the initial m-configurations;*
- $R = \{((s, o, c, m), (s', o, c', m')) \in Q^2 : (s, o, c, m) \rightarrow (s', o', c', m')\}$.

The validity of a K-CTL formula on a m-execution structure K is defined as in Definition 14, with the exception of the case of goals $\mathbf{K}\, p$, where:

- *$K, q \models \mathbf{K}\, p$ iff $q = (s, o, c, m)$ and $\bar{s} \models_0 p$ for all $\bar{s} \in \mathcal{MO}(m, o)$.*

Definition 21 (plan validation using monitors) *Plan Π satisfies K-CTL goal g on domain $\mathcal{D}$ according to monitor $\mathcal{M}$, written $\Pi \models_{\mathcal{D},\mathcal{M}} g$, if $K \models g$, where K is the m-execution structure corresponding to $\mathcal{D}$, Π, and $\mathcal{M}$.*

The possibility of using monitors for plan validation is guaranteed by the following Theorem.

Theorem 22 *Plan Π satisfies K-CTL goal g on domain $\mathcal{D}$ if and only if there is a correct monitor $\mathcal{M}$ for $\mathcal{D}$ and Π such that $\Pi \models_{\mathcal{D},\mathcal{M}} g$.*

The proof of this theorem is simple. For the *if* implication, it is sufficient to notice that, for any reachable m-configuration (s, o, c, m), the output $\mathcal{MO}(m, o)$ of a correct monitor is a super-set of the belief states that are compatible with the evolutions leading to the m-configuration. The condition for the validity of knowledge goals given in Definition 20 is stronger than the condition given in Definition 14. Therefore, if $\Pi \models_{\mathcal{D},\mathcal{M}} g$, then $\Pi \models_{\mathcal{D}} g$.

In order to prove the *only if* implication, we introduce *universal monitors*.

Definition 23 (universal monitor) *The universal monitor $\mathcal{M}_{\mathcal{D}}$ for domain $\mathcal{D}$ is defined as follows:*

- *$\mathcal{MS} = 2^{\mathcal{S}}$ are the belief states of $\mathcal{D}$.*
- *$m_0 = \mathcal{I}$.*
- *$\mathcal{MT}(bs, o, a) = \{\bar{s}' : \exists \bar{s} \in bs.\ o = \mathcal{X}(\bar{s}) \wedge \bar{s}' = \mathcal{T}(\bar{s}, a)\}$.*
- *$\mathcal{MO}(bs, o) = \{\bar{s} \in bs : o \in \mathcal{X}(\bar{s})\}$.*

The universal monitor of a domain traces the precise evolution of the belief states, that is, it does not lose any information. One can check that the belief state reported by this monitor for a given m-configuration coincides with the belief state of the bs-configuration corresponding to the same computation. Therefore, $\Pi \models_{\mathcal{D}} g$ if and only if $\Pi \models_{\mathcal{D},\mathcal{M}_{\mathcal{D}}} g$. Since the universal monitor is correct, this is sufficient to prove the *only if* implication of Theorem 22.

The possibility of losing some of the information on the current belief state makes monitors very convenient for plan validation. In many practical cases, monitors are able to represent in a very compact way the aspects of the evolution of belief states that are relevant to the goal being analyzed. Consider for instance the extreme case of a K-CTL goal g that does not contain any $\mathbf{K}\, p$ sub-goal — so it is in fact a CTL goal. In order to apply Definition 15, we should trace the exact evolution of belief states, which may lead to an exponential blowup w.r.t. the size of the domain. Theorem 22, on the other hand, allows us to prove a plan correct against a very simple monitor: the monitor with a single state that is associated to the belief state $bs = \mathcal{S}$, independently from the observation. This monitor traces no information at all on the belief states, which is possible since no knowledge goal appears in g (compare with Definition 9).

Another, less extreme example of the advantages of using monitors for plan validation is the following.

Example 24 *Consider the execution of Π_2 on the domain of Figure 3. The belief states corresponding to the different steps of the execution are rather complex. They have to take into account that, after i rooms have been visited by the robot, we know that there are i consecutive rooms with the light on, but that we do not know which are these rooms. For instance, after two rooms have been visited, the belief state is the following:*

$$(\textit{room} = 1 \wedge \textit{light-on}[1] \wedge \textit{light-on}[N]) \vee$$
$$(\textit{room} = 2 \wedge \textit{light-on}[2] \wedge \textit{light-on}[1]) \vee$$
$$(\textit{room} = 3 \wedge \textit{light-on}[3] \wedge \textit{light-on}[2]) \vee \cdots$$

Most of the information of these belief states is useless for most of the goals. Consider for instance goals

$$\mathrm{AF}\, \mathbf{K}\, (\textit{light-on}[3])$$

or

$$\bigwedge_{i \in 1,\ldots,N} \mathrm{AF}\, \mathbf{K}\, (\textit{light-on}[i]).$$

The only relevant information for proving that these goals are satisfied by plan Π_2 is that, once the robot has visited all the rooms, all the lights are on. A suitable monitor for these goals is the following. It has two states, m_0 and m_1, with m_0 the initial state, and m_1 corresponding to the termination of the exploration. The transition function and the output of the monitor are defined by the following table:

m	o	a	$\mathcal{MT}(m, o, a)$	$\mathcal{MO}(m, o)$
m_0	*any*	*wait*	m_1	$\top$
m_0	*any*	$\neq$ *wait*	m_0	$\top$
m_1	*any*	*wait*	m_1	$\bigwedge_{i \in 1,\ldots,N} (\textit{light-on}[i])$

According to Definition 21, the problem of proving that plan Π satisfies K-CTL goal g on domain $\mathcal{D}$ according to monitor $\mathcal{M}$ is reduced to model checking goal g on the m-execution structure corresponding to the synchronous execution of $\mathcal{D}$, Π, and $\mathcal{M}$. In order to conclude that Π satisfies goal g, however, we have to prove that monitor $\mathcal{M}$ is correct.

The correctness of a monitor can also be proved using model checking techniques. Indeed, it corresponds to prove that the following formula is true on the m-execution structure:

$$\mathrm{AG}\,(s \in \mathcal{MO}(m, o)).$$

We conclude the section by remarking that in this paper we have not yet addressed the problem of defining a suitable monitor for checking plan validation. In practice, it may be very difficult to decide what information on the belief states has to be traced by the monitor. Intuitively, the problem amounts to identifying an abstraction of the universal monitor that is sufficient for proving that the plan satisfies a given the goal. Although the use of incremental abstraction refinement techniques can be envisaged, this is currently an open problem. In the specific case the plan is synthesized by an algorithm, however, a proof of the correctness of the plan is built implicitly during the search. In this case, a monitor could be produced by the algorithm itself, by generating a sort of *proof-carrying plan*.

Concluding remarks

This paper is a first step towards planning for temporally extended goals under the hypothesis of partial observability. We defined the basic framework and introduced the K-CTL language, that combines the ability of expressing temporally extended constraints with the ability to predicate over uncertainty aspects. Then, we introduced the notion of monitor, and defined correctness criteria that can be used in practice to validate plans against K-CTL goals.

The issue of "temporally extended goals", within the simplified assumption of full observability, is certainly not new. However, most of the works in this direction restrict to deterministic domains, see for instance (de Giacomo & Vardi 1999; Bacchus & Kabanza 2000). A work that considers extended goals in nondeterministic domains is described in (Kabanza, Barbeau, & St-Denis 1997). Extended goals make the planning problem close to that of automatic synthesis of controllers (see, e.g., (Kupferman, Vardi, & Wolper 1997)). However, most of the work in this area focuses on the theoretical foundations, without providing practical implementations. Moreover, it is based on rather different technical assumptions on actions and on the interaction with the environment.

On the other side, partially observable domains has been tackled either using a probabilistic Markov-based approach (see (Bonet & Geffner 2000)), or within a framework of possible-world semantics (see, e.g., (Bertoli *et al.* 2001; Weld, Anderson, & Smith 1998; Rintanen 1999)). These works do not go beyond the possibility of expressing more than simple reachability goals. An exception is (Karlsson 2001), where a linear-time temporal logics with a knowledge operator is used to define search control strategies in a progressive probabilistic planner. The usage of a linear-time temporal logics and of a progressive planning algorithm makes the approach of (Karlsson 2001) quite different in aims and techniques from the one discussed in this paper.

Future steps of this work will include the definition of a procedure for synthesizing monitors from K-CTL goals, and the investigation of planning procedures for planning for extended goals under partial observability. The synthesis of monitors appears to be related to the problem of generating supervisory controllers for the diagnosis of failures. We are investigating whether the techniques developed by the diagnosis community (see, e.g., (Sampath *et al.* 1996)) can be applied to the synthesis of monitors. The main challenge for obtaining a planning procedure appears to be the effective integration of the techniques in (Bertoli, Cimatti, & Roveri 2001; Bertoli *et al.* 2001; Pistore & Traverso 2001) that make effective use of (extensions of) symbolic model checking techniques, thus obtaining a practical implementation based on symbolic model checking techniques.

References

Bacchus, F., and Kabanza, F. 2000. Using Temporal Logic to Express Search Control Knowledge for Planning. *Artificial Intelligence* 116(1-2):123–191.

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2001. Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In *Proc. IJCAI'01*.

Bertoli, P.; Cimatti, A.; and Roveri, M. 2001. Heuristic Search + Symbolic Model Checking = Efficient Conformant Planning. In *Proc.* 7th *International Joint Conference on Artificial Intelligencof (IJCAI-01)*. AAAI Press.

Bonet, B., and Geffner, H. 2000. Planning with Incomplete Information as Heuristic Search in Belief Space. In *Proc. AIPS 2000*, 52–61.

de Giacomo, G., and Vardi, M. 1999. Automata-theoretic approach to Planning with Temporally Extended Goals. In *Proc. ECP'99*.

Emerson, E. A. 1990. Temporal and modal logic. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier.

Kabanza, F.; Barbeau, M.; and St-Denis, R. 1997. Planning Control Rules for Reactive Agents. *Artificial Intelligence* 95(1):67–113.

Karlsson, L. 2001. Conditional progressive planning under uncertainty. In *Proc.* 7th *International Joint Conference on Artificial Intelligencof (IJCAI-01)*. AAAI Press.

Kupferman, O.; Vardi, M.; and Wolper, P. 1997. Synthesis with incomplete information. In *Proc. ICTL'97*.

Pistore, M., and Traverso, P. 2001. Planning as Model Checking for Extended Goals in Non-deterministic Domains. In *Proc. IJCAI'01*.

Rintanen, J. 1999. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intellegence Research* 10:323–352.

Sampath, M.; Sengupta, R.; Lafortune, S.; Sinnamohideen, K.; and Teneketzis, D. 1996. Failure Diagnosis Using Discrete-Event Models. *IEEE Transactions on Control Systems Technology* 4(2):105–124.

Weld, D.; Anderson, C.; and Smith, D. 1998. Extending Graphplan to Handle Uncertainty and Sensing Actions. In *Proc. AAAI'98*, 897–904.

Solving Power Supply Restoration Problems with Planning via Symbolic Model Checking

Piergiorgio Bertoli(1), **Alessandro Cimatti**(1), **John Slaney**(2), **Sylvie Thiébaux**(2)

(1) IRST – Istituto per la Ricerca Scientifica e Tecnologica, Trento, Italy. email: {bertoli, cimatti}@irst.itc.it
(2)Computer Sciences Lab, The Australian National University, Canberra, Australia. email: {John.Slaney, Sylvie.Thiebaux}@anu.edu.au

Abstract

The past few years have seen a flurry of new approaches for planning under uncertainty, but their applicability to real-world problems is yet to be established since they have been tested only on toy benchmark problems. To fill this gap, the challenge of solving power supply restoration problems with existing planning tools has recently been issued. This requires the ability to deal with incompletely specified initial conditions, fault conditions, unpredictable action effects, and partial observability in real-time. This paper reports a first response to this nontrivial challenge, using the approach of planning via symbolic model-checking as implemented in the MBP planner. We show how the problem can be encoded in MBP's input language, and report very promising experimental results on a number of significant test cases.

Introduction

It has long been recognized that real-world planning requires coping with uncertainty arising from exogenous events, nondeterministic actions, and partial observability. Accordingly, a wide range of approaches for planning under uncertainty have been proposed; see (Bertoli *et al.* 2001b; Bonet & Geffner 2000; Hansen & Feng 2000; Kabanza, Barbeau, & St-Denis 1997; Majercik & Littman 1999) for recent examples. While theoretically well-founded, these approaches have only been tested on very artificial examples, and, at a practical level, there is no evidence that they will scale up to address interesting problems. Therefore, one of the most significant outstanding tasks for the field is to demonstrate the applicability of these approaches to realistic problems, identify their bottlenecks, and suggest improvements.

Recently, Thiébaux and Cordier made a concrete proposal in this direction (Thiébaux & Cordier 2001). They recast the problem of power supply restoration (PSR) as a benchmark for planning under uncertainty, and issued the challenge of solving PSR problems with existing planning tools. PSR consists in planning actions to reconfigure a faulty power distribution network, with a view to resupplying the customers affected by the faults. Due to sensor and actuator uncertainty, the location of the faulty areas and the current network configuration are only partially observable. This results in a tradeoff between acting to achieve a suitable configuration, and acting (intrusively) for the purpose of acquiring additional information. This tradeoff is typical not only of problems arising in planning for partially observable domains, but also of diagnosis, repair, and reconfiguration problems (Friedrich & Nejdl 1992; Sun & Weld 1993; Baral, McIlraith, & Son 2000).

In this paper, we examine the applicability of the framework of Planning via Symbolic Model Checking to the PSR problem. We focus on the approach implemented in the MBP planner (Bertoli *et al.* 2001b), which makes aggressive use of Binary Decision Diagrams, and generates conditional plans which are guaranteed to achieve the goal. We show how the PSR problem can be encoded in MBP's input language $\mathcal{AR}$ (Giunchiglia, Kartha, & Lifshitz 1997), and discuss the bottlenecks induced by alternative encoding styles. Using our preferred encoding, we experiment with the rural network described in (Thiébaux & Cordier 2001), as well as with artificial networks with simple topologies. For the rural network, we show that MBP solves problems involving a realistic degree of uncertainty in better than real-time. With the simpler topologies, we study the effects of the network size and of the degree of uncertainty on MBP's run time. The results show that MBP is able to successfully exploit the network topology. They also highlight that the degree of uncertainty induces an easy-hard-easy pattern when the number of faults is known, and an easy-hard pattern when the number of faults is unknown.

This paper is organized as follows. First we outline the PSR benchmark, then we introduce the planning as model checking paradigm and discuss the modeling of PSR in $\mathcal{AR}$. We step on reporting the experimental results, and conclude with some remarks about related and future work.

The PSR Benchmark

We start with a short statement of the power supply restoration problem given in (Thiébaux & Cordier 2001), to which we refer for a more detailed description. As shown in Figure 1, a power distribution system is a network of electric lines connected by switching devices (the small squares in the figure) and fed by circuit-breakers (represented by large squares). Switching devices and circuit-breakers are connected to at most two lines, and have two possible positions: closed or open (open devices, e.g. SD8, are white in the figure). When a circuit-breaker is closed, it supplies power to the network, and this power propagates downstream until an

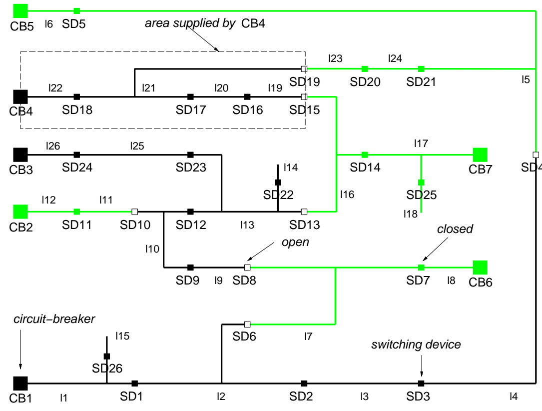


Figure 1: Rural Power Distribution System (from (Thiébaux & Cordier 2001))

open switching device stops the propagation. The positions of the devices are initially set so that each circuit-breaker feeds a given area of the network (e.g. the area fed by CB4 is boxed in the figure), with no line being fed by more than one circuit-breaker. In the figure, gray and dark are used to distinguish adjacent areas.

Permanent faults can affect one or more lines of the network. When a line is faulty, the circuit-breaker feeding this line opens in order to protect the rest of its area from overloads. As a result, not just the faulty line but the entire area is left without power. The supply restoration problem consists in reconfiguring the network by opening and closing devices so as to electrically isolate the faulty lines and resupply a maximum of non-faulty lines on the lost areas. This must be done within minutes. For instance, suppose that l20 becomes faulty. This leads CB4 to open and the boxed area to be without power. Assuming that the location of the fault and the current network configuration are known, an adequate restoration plan would be to open SD16 and SD17 to isolate the faulty line, to close SD15 to have CB7 resupply l19, and to re-close CB4 to resupply the others.

Opening and closing operations are the only available actions in the benchmark. A first source of uncertainty is that these actions can fail and that failures are not always observable. More specifically, in normal operation the actuator of the prescribed switching device executes the requested action and sends a positive notification. However, the actuator sometimes fails to alter the device's position and sends a negative notification – this is called the "out of order" mode – or fails to alter the position but still sends a positive notification – this is called the "liar" mode. Both abnormal modes are permanent. Clearly, only the out of order mode is directly observable via the notification.

In addition to action-triggered notifications, two types of action-independent sensing information are continuously provided. Firstly, each device is equipped with a position detector which, when in normal mode, indicates the device's current position. Unfortunately, the position detector can be "out of order" for an indeterminate time, during which it does not return any information. Because of the liar actuator mode, the position of an operated device cannot be known with certainty while its position detector remains out of order. It is then difficult to know whether faults have been correctly isolated.

Secondly, each switching device is equipped with a fault detector which senses the presence of faults. In normal operation, as long as the device is fed, its fault detector indicates whether there exists a fault downstream of it on the area. If the device is not fed, its fault detector keeps the status it had when last fed. For example, if l20 is faulty, only the fault detectors of SD17 and SD18 should indicate a fault downstream. Then CB4 should open and the fault information returned by the devices on its area should remain the same until they are fed again. If position detectors were reliable, this scheme would be sufficient to locate single faults, as well as multiple faults on different areas. Unfortunately, fault detectors sometimes do not return any information at all ("out of order mode"), or can lie and return the negation of the correct status ("liar mode"). Both modes are permanent and again only the out of order mode is directly observable.

It follows that several fault location hypotheses are consistent with the observations, and that each of them corresponds to an assumption about the modes (normal or liar) of the detectors. The same applies to positions. The only hope of gaining sufficient information to invalidate a hypothesis is to change the network configuration and compare the new sensing information with the predicted one. This results in a tradeoff between acting to resupply and acting to reduce uncertainty, which is typical of partially observable domains.

Planning via Symbolic Model Checking

In this section we briefly outline the Planning via Symbolic Model Checking (PSMC) approach, which we use to solve PSR problems. In PSMC, planning domains are represented as nondeterministic finite state automata (see (Bertoli *et al.* 2001b) for formal definitions). Modeling a domain involves the two following steps. Firstly, the state and dynamics of the system (e.g. in the PSR, the plant to be restored) are specified. A state of the domain is an assignment to a set of variables (e.g. whether a switch is open or closed, or whether a line is faulty or not). The effects of (possibly nondeterministic) actions are modeled by relating a starting state and an action with one or more target states. Secondly, the information actually available at execution is characterized by means of observation variables that are assigned values depending on the state of the system. For instance, in the PSR it is impossible to directly observe whether a line is fed or faulty, but it is possible to observe the status of devices via (possibly untrustworthy) detectors.

In planning under uncertainty and with partial observability, the problem is to reach a given goal condition (e.g. feed every non-faulty line that can be fed) starting from a given, possibly uncertain, initial condition. A solution to the problem is a *strong plan* that, when executed, guarantees that all possible executions starting from any initial state will reach a goal state. In general, the search space of planning under partial observability can be seen as an AND-OR graph in the space of belief states, i.e. sets of states that represent uncertain situations. OR nodes represent alternate choices among

possible actions and observations, and AND nodes represent the effect of observations. While an action maps a belief state into a belief state, an observation conveys information by splitting the belief state into smaller belief states, one per each possible observation value; since every possible result of observing must be taken into account, this originates and AND branching. Plan formation amounts to finding an AND-OR subtree within the search space.

MBP (Bertoli *et al.* 2001a) is a general purpose planner that provides for different styles of planning, e.g. conformant, strong and strong cyclic planning, and extended goals, allowing for partial observability and uncertainty. One of its strengths lies in the use of Binary Decision Diagrams (BDDs) to represent and manipulate belief states. BDDs are compact data structures for the representation and manipulation of propositional formulae. In particular, in MBP the effect of actions is efficiently computed by means of symbolic relational operations on BDDs.

Formalizing the PSR in MBP

We model the PSR domain using MBP's input language, an extension of the $\mathcal{AR}$ language (Giunchiglia, Kartha, & Lifshitz 1997). $\mathcal{AR}$ allows for describing domains where fluents may be inertial or not, where actions may feature preconditions, conditional effects and uncertain effects, and where observations may or may not be triggered by an action. We take the problem description in stages: network topology, network states, problem dynamics, and observations.

Network topology. The basic elements of the network are circuit-breakers, switching devices and lines. By device, we intend either a breaker or a switch. A device d has two sides d^+, d^-. By convention, a breaker b has side b^- attached to the power supply, and b^+ attached to the network. If d^s is a device side, we indicate by $d^{\overline{s}}$ its complementary side. A connection either links two device sides via a line, or consists of a "hanging" line from a device side to earth (see e.g. l15 in Figure 1). Thus, a side-to-side connection is a triple $\langle \delta; l; \delta' \rangle$, and a hanging connection is a pair $\langle \delta; l \rangle$, where δ and δ' are device sides, $\delta \neq \delta'$, and l is a line.

A supply network is a 4-tuple $N = \langle B, S, L, C \rangle$ where B is a set of breakers, S is a set of switches, L is a set of lines and C is a set of connections over B, S and L, satisfying the conditions:

1. Each side of a switch in S occurs in some connection in C.
2. For each breaker $b \in B$, b^+ occurs in some connection in C and b^- in no connection in C.
3. Every line in L occurs in some connection in C.
4. No device side is incident on more than one line: if $\langle \delta; l[; \delta'] \rangle$ and $\langle \delta; l'[; \delta''] \rangle$ occur in C then $l = l'$.
5. The connection relation is transitive and symmetric: $\langle \delta; l; \delta' \rangle \in C \Longrightarrow \langle \delta'; l; \delta \rangle \in C$ $\{\langle \delta; l; \delta' \rangle, \langle \delta'; l; \delta'' \rangle\} \subseteq C, \delta \neq \delta'' \Longrightarrow \langle \delta; l; \delta'' \rangle \in C$

To describe the phenomena of power propagation we use the notion of a *path*: a sequence of connected devices and lines in the network, starting from a breaker and ending with a line. Formally, a path P of N is a sequence $b^-, b^+, l_1[, d_i^{s_i}, d_i^{\overline{s_i}}, l_i]^*$ where each subsequence $[\delta, l, \delta'] \in P$ corresponds to a side-to-side connection $\langle \delta; l; \delta' \rangle \in C$, and the tail $[\delta, l]$ of P corresponds either to a hanging connection $\langle \delta; l \rangle$ of C, or to a side-to-side connection $\langle \delta, l, \delta' \rangle$ in C. We indicate with $\mathsf{dev}(P)$ the set of devices in P, and with $\mathsf{lin}(P)$ the set of lines in P. A path is acyclic iff it does not contain duplicate lines, cyclic otherwise. A cyclic path P is minimal iff no prefix of P is a cyclic path. We define $\mathsf{AP}(N)$ as the set of all acyclic paths of network N, and $\mathsf{CP}(N)$ as the set of all minimal cyclic paths of N. Reasoning about N amounts to determining the properties of $\mathsf{AP}(N) \cup \mathsf{CP}(N)$.

Network state. The state of a supply network is described by:

- The position of each device, modeled by a dynamic predicate $\mathtt{closed}(d)$, defined on $B \cup S$. We say that a path P is active iff it brings power to every line in $\mathsf{lin}(P)$, which it does when every device in P is closed: $\mathtt{active}(P) = \forall d \in \mathsf{dev}(P) : \mathtt{closed}(d)$.

  We say that P is active upon closing d iff closing d leads to P being active: $\mathtt{active_upon_closing}(P, d) = \forall d' \in \mathsf{dev}(P) : ((d' = d) \vee \mathtt{closed}(d'))$.
- The permanent modes of the lines (faulty or not), modeled by a predicate $\mathtt{faulty}(l)$, statically defined on L.
- The fault-status of each device, i.e., whether there was a fault downstream of the device when it was last fed. It is modeled by a dynamic predicate $\mathtt{affected_when_last_fed}(d)$ which is set every time d is part of an active path whose last line is faulty, and is reset when d is fed with no fault downstream.
- The permanent modes of the devices' actuators, fault detectors and position detectors. These are modeled by predicates $\mathtt{AC_correct}(d)$, $\mathtt{AC_liar}(d)$, $\mathtt{FD_liar}(d)$, $\mathtt{PD_correct}(d)$, $\mathtt{FD_correct}(d)$, statically defined on $B \cup S$.

In MBP, statically defined predicates whose value is known in the given problem can be compiled away as DEFINEs. Dynamic predicates, and static predicates whose value is not known, are fluents and build the actual state of the domain. Under the current assumptions, every such fluent is inertial.

Dynamics and observations. Two kinds of phenomena may affect the state of the network: (a) user-induced actions on a device d may affect d's position, and (b) faults and power propagation may affect the fault-status of various devices and the positions of breakers. Their effects can be described as follows:

a1 When opening [closing] a device d, if the actuator of d is correct, d opens [resp. closes]. Otherwise, it keeps its current position.

b1 If there exists an active path P whose last line is faulty and $d \in P$, then the fault-status of d is set. If d is a breaker, it opens; otherwise, it keeps its current position.

b2 If d is in an active path, but in no active path ending in a faulty line, then the fault-status of d is reset. d keeps its current position.

b3 If no active path P exists such that $d \in P$, then position and fault-status of d are unchanged.

Several options are possible for modeling the above dynamics. We identified two main classes of modeling styles. In the first class, the combined effects of (a) and (b) are computed as a "one-step" consequence of each user-triggered action. In the second class, the effects of (a) and (b) are considered in turn. Thus, the model execution interleaves the effects of actions with those of fault propagation, represented by adding a fictitous `propagate` action. Experimental evaluation of the two classes of modelings showed the "one-step" models to be more efficient on average, so we focus on their description.

In the "one-step" modeling class, the description of an action upon a device d_0 must consider the effects on every device d as follows:

1. as a consequence of closing d_0, a non-active path P, whose last line is faulty, becomes active, and d belongs to P. In this case the fault-status of d is set. If d is a breaker, it opens; otherwise it keeps its current position. We say that closing d_0 has affected d.
2. d_0 does not affect d and, as a consequence of closing d_0, a non-active path P, whose last device is d, becomes active. The device d_0 is said to have fed d. In this case, the fault-status of d is reset. d's position is unchanged.
3. as a consequence of closing d_0, neither case (1) nor case (2) applies. In this case, closing d_0 has no effect on d.
4. As a consequence of opening d_0, an active path becomes inactive. This does not change the position or fault-status of any device d other than d_0 itself.
5. As a consequence of opening [closing] d_0, d_0 opens [resp. closes] if its actuator is correct, unless d_0 is a breaker whose closing affects itself (in which case it reopens, see (1)).

The definitions above are directly translated into conditional CAUSE statements in $\mathcal{AR}$. For instance, given an appropriate propositional $\mathcal{AR}$ definition for `closing_SD17_affects_CB4`, the effect of closing switch SD17 upon breaker CB4's position (see b1) is described as follows:

```
CAUSES  act = close_SD17
        next(closed_CB4) := 0
        IF
           AC_correct_SD17 &
           closing_SD17_affects_CB4;
```

In addition, we must take into account the possibility of situations where breakers feed cyclic paths, or in which devices are fed both ways. In these situations, the direction of the electricity flow cannot uniquely be established (unless additional physical data are modeled); thus, the status of fault sensors is not uniquely determined. Although a deployed system would have to be extended somewhat to model these eventualities, in the benchmark, they must be prevented from arising. This is easily achieved by determining "cycle-causing" and "multiple-feed-causing" conditions for any device, and preconditioning the action of closing of a device to the absence of such conditions. We omit details, for reasons of space.

Modeling sensing is straightforward, and independent of the modeling of actions. The observation returned by the sensors and actuators of a device depend on their mode and on the actual fault-status and position of the device. For instance, the position detector of CB1 signalling that CB1 is open is captured by the boolean observation:

```
OBSERVE says_open_CB1:
        PD_correct_CB1 & !closed_CB1;
```

We designed a tool to automatically generate MBP models following the ideas above, starting from a description of the network topology. Both "one-step" models and interleaved models can be generated via this tool.

Solving PSR Problems in MBP

We used MBP to test our approach against different topologies, experimenting with different modeling styles, and considering some different search strategies. We discuss these aspects in turn.

In terms of modeling styles, we experimented with the "one-step" style, and two interleaved models: a "strictly interleaved" model where `propagate` takes place after any user-induced action, and a "loosely interleaved" model where `propagate` takes place only after an action affects a device. Textual descriptions of interleaved models are more compact and convenient for a human reader, since the effects of fault propagation are described locally to the `propagate` action. However, this advantage has no parallel in the construction of the automaton representing the domain, nor in the search times. In fact, in preliminary experiments the "one-step" model performed considerably better. We used interleaved models for testing our modeling ideas, since they facilitate debugging, but we adopted he "one-step" modelings for the tests described below.

We adopted a forward heuristic search algorithm in MBP see (P.Bertoli, A.Cimatti, & Roveri), which makes it possible to encode search strategies either by incorporating control knowledge into the action descriptions, or by adding adhoc heuristics. We used the following simple ideas which appear to be generic to the benchmark rather than specific to the topologies we considered: (a) do not open a device which is fed or which has previously been fed, (b) favor closing actions over opening actions. Idea (a) is obvious, given that a fed device cannot be incident on a faulty line. Idea (b) comes from the fact that `open` actions are only useful to isolate faults. while `close` actions either lead towards the goal by feeding more lines or give the planner information by unexpectedly refeeding a fault.

The first topology we considered was a simple linear one, making it easy to test scalability by varying the network size and the reliability of lines, sensors and actuators. Then we

considered a still "simple" but slightly more complex network allowing for a greater variety of configurations, and experimented by varying the reliability of the lines. Finally, we considered a realistic problem taken from ((Thiébaux & Cordier 2001)), based on the topology of Figure 1. Every experiment has been run on a 700 Mhz Pentium III Linux machine with 6 GBytes of RAM, but in no case more than 140 MBytes of RAM have been used by MBP.

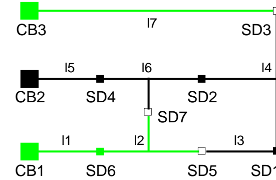
Linear topology. For the linear topology, we first considered the problem of restoring supply given that (a) *exactly* n, and (b) *at most* n of the lines are faulty, starting from a state where all devices are open and all sensors and actuators are reliable, but the locations of the faults are unknown. Figure 3 shows the results for problem (a), considering linear topologies of size varying from 5 to 20 lines, plotted against the exact percentage of faulty lines.

An easy-hard-easy pattern emerges. It is not too hard to see why this might happen: if there is no fault the problem is trivial; if there are faults, the lines between the first and last faults are essentially irrelevant because they can never be fed, so the more faults there are the smaller the number of lines causing work. Problem (b), by contrast, showed no such cost peak: when only the maximum number of faults is known, the problem difficulty is roughly constant for any maximum number of faults greater than zero (search times are always below 7 seconds). This is because the difficulty of problems with up to n faults is roughly the sum of those with exactly k for $k \leq n$, so it does not depend much on n because this sum is dominated by the first few values. In summary, the results of the experiments with the linear topology are intuitively explicable, but it is important to note that MBP achieves these results, as it shows that the planner is able to exploit the structure of the problem.

To experiment with varying reliability of the actuators and sensors, we selected one of the problems above: a linear network of 9 lines, with at most 6 faults. We independently considered that:

1. at least (exactly) n fault detectors are reliable;
2. at least (exactly) n actuators are reliable;
3. at least (exactly) n position detectors are reliable;
4. at most (exactly) n fault detectors are liars;
5. at most (exactly) actuators are liars.

Again, in the initial situation all devices are open; we try to feed lines that are reachable through reliable devices. Fig.4 shows the results; $\text{FD}_{c(=)}, \text{AC}_{c(=)}, \text{PD}_{c(=)}, \text{FD}_{l(=)}, \text{AC}_{l(=)}$ refer to problem 1,2,3,4,5 respectively, in both their versions. As in the case of faulty lines, problems become more constrained and therefore easier if it is known that most devices are unreliable. Of course, the combination of faults upon lines and "issues" upon sensors/actuators originate more complex problems, leading to higher search times than the previous ones.



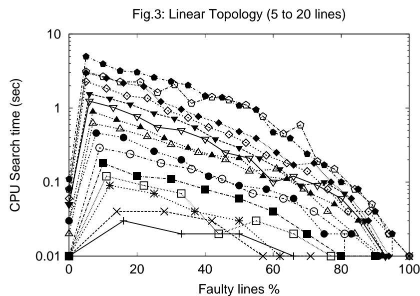
Simple. Here we considered the topology above and an experimental setting similar to those we used upon the linear topology, varying the (maximum/exact) fault percentage F of lines. The results appear very similar to those for lines, and show that MBP is able to exploit more complex structures. Considering exactly n faults, the easy-hard-easy pattern reaches its top when $F \approx 40\%$, with 1.2 seconds of search time. Considering up to n faults, problems are increasingly hard up to $F \approx 50\%$; the higher search times are around 13 seconds. Of course, higher times are expected due to the fact that the topology is more complex than a simple line. The results are presented in Fig.5.

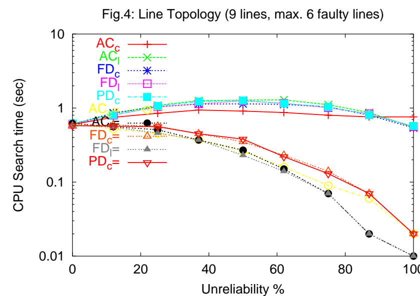
Rural. Here we consider an example taken from (Thiébaux & Cordier 2001), see fig. 1. The initial situation is that in the figure, except that CB1 is open; no fault sensors signal a fault, and every line/device/sensor is known to be correct apart from: lines l3 and l15, the fault sensors of SD1, SD2, SD3, SD26, the actuator of SD26, the position detector of SD26. For these devices, no hypothesis is made. Supply restoration must feed every feedable line. Here, this means that, if the actuator of SD26 is correct, then every non-faulty line must be fed; if the actuator of SD26 is not working and l15 is faulty, then neither l15 nor l1 can be fed.

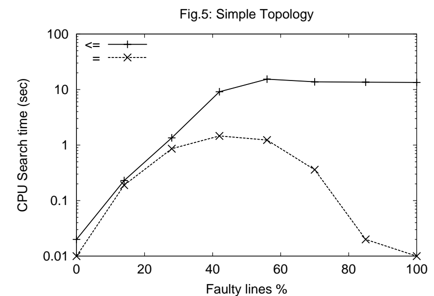
The textual description of the one-step model is 8.5 megabytes; however, it takes MBP only 2 seconds to parse, and 28 seconds to construct the automata for the machine. Once this is done, a plan is found in 1.2 seconds, using approximately 140 megabytes of RAM. Vital to this is the fact that a "good" ordering for the variables is automatically established by MBP; this is achieved by a reusable off-line pre-computing which depends on the model (but not on the problem), and thus has to be performed only once for a given topology. In this case, precomputing takes approximately 33 minutes. The plan is presented in fig. 2, and seems a proof of the feasibility of the model-based planning approach to this kind of problems, given the fact that the network and problem are not far from real-life networks.

Conclusion, Related and Future Work

This paper demonstrates that the planning via symbolic model checking paradigm, as implemented in the MBP planner, is able to solve realistic supply restoration problems. We have developed a systematic representation of the dynamics of the plant as a finite state automaton in MBP's input language, modeled fault conditions, and reformulated power supply restoration as a problem of planning under partial observability. A key difficulty of the planning task is the need to intertwine diagnosis (identifying the causes of the problem) with the search for corrective actions. An experimental evaluation shows that MBP is able to manage some basic plant configurations, and to automatically produce strong








```
close_SD4;                 -- feed every line
if says_fault_SD4 then -- l3 and/or l15 faulty
   open_SD3;               -- MBP assumes
                           -- l15 ok and
   open_SD2;               -- isolates l3
   close_CB5;              -- refeed...
   close_SD26;
   close_SD6;
   if says_fault_SD6 then -- l15 faulty
      open_SD1;              -- MBP assumes
                             -- l3 ok and
      close_SD2;             -- isolates l15
      close_CB6;             -- refeed...
      if says_fault_SD6 then -- l3 and l15
                                -- faulty
         open_SD26;             -- try isolate
                                -- l15
         open_SD2;              -- isolate l3
         close_CB1;             -- ...refeed.
         close_SD8;
      else
         open_SD26;             -- l3 ok: try
                                -- isolate l15
         close_CB1;             -- ...refeed.
```

Figure 2: Plan for Rural Network problem

plans for problems when different degrees of information are available. The underlying symbolic machinery of Binary Decision Diagrams confirms its ability to compactly store and efficiently traverse the automaton.

Although the model-based diagnosis community has investigated similar power supply restoration problems in the context of distribution and transport networks, to our knowledge, MBP is the first general-purpose system able to cope with the presence of uncertainty in such problems. For instance, SyDRe, the supply restoration system in (Thiébaux *et al.* 1996) is able to handle the full PSR benchmark including sensor and actuator uncertainty, but is entirely domain-specific. Supply restoration of power transmission systems using a general-purpose diagnosis and planning engine has been studied e.g. in (Friedrich & Nejdl 1992), but a crucial difference with the PSR benchmark is that observations and actions are assumed to be reliable. Another work related to ours is the application of the model-based reactive planner Burton to spacecraft engine reconfiguration (Williams & Nayak 1997). Noticeably, Burton's compilation of a transition system into prime implicants is related to the compilation into BDDs performed by MBP, but again Burton does not handle partial observability nor actions with uncertain effects.

This work is the first step towards the integration of MBP into a complex real-world domain such as PSR. In the future, we will investigate the potential for an on-line integration as in (Thiébaux *et al.* 1996), interleaving probability-based diagnosis, planning and execution modules. Indeed, for real-world problems trying to construct strong plans considering every possible contingency is overkill (even when feasible). Interleaving planning with action might have a dramatic impact on the search times, by restricting the search to only that part of the belief space which is admissible given the current observations.

In the future, we will also investigate ways to extend and improve MBP. On one side, we would like to be able to express temporally extended goals (M.Pistore & P.Traverso 2001), that can be more realistic than reachability goals, for partially observable domains. On the other side, we intend to identify ways to cut the search space (e.g. by means of user-defined strategies (Bacchus & Kabanza 2000), or by automatically-detected heuristics, where promising results have been shown in (P.Bertoli & A.Cimatti 2001) for conformant planning.

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logic to express search control knowledge for planning. *Art. Int.* 116(1-2).

Baral, C.; McIlraith, S.; and Son, T. 2000. Formulating diagnostic problem solving using an action language with narratives and sensing. In *Proc. KR*.

Bertoli, P.; Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2001a. MBP: a Model Based Planner. In *Proc. of the IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*.

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2001b. Planning in nondeterministic domains under partial observability. In *Proc. IJCAI*, 473–478.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *Proc. AIPS*, 52–61.

Friedrich, G., and Nejdl, W. 1992. Choosing observations

and actions in model-based diagnosis-repair systems. In *Proc. KR*, 489–498.

Giunchiglia, E.; Kartha, N.; and Lifshitz, V. 1997. Representing action: indeterminacy and ramifications. *Art. Int.* 95:409–443.

Hansen, E., and Feng, Z. 2000. Dynamic programming for POMPDs using a factored state representation. In *Proc. AIPS*.

Kabanza, F.; Barbeau, M.; and St-Denis, R. 1997. Planning control rules for reactive agents. *Artificial Intelligence* 95:67–113.

Majercik, S., and Littman, M. 1999. Contingent planning under uncertainty via stochastic satisfiability. In *Proc. AAAI*, 549–556.

M.Pistore, and P.Traverso. 2001. Planning as model checking for extended goals in non-deterministic domains. In *Proc. IJCAI'01*.

P.Bertoli, and A.Cimatti. 2001. Improving heuristics for planning as search in belief space. In *Proc. AIPS'02*. To appear.

P.Bertoli; A.Cimatti; and Roveri, M. Conditional planning under partial observability as heuristic-symbolic search in belief space. In *Proc. ECP'01*.

Sun, Y., and Weld, D. 1993. Beyond simple observation: Planning to diagnose. In *Proc. AAAI*, 182–187.

Thiébaux, S., and Cordier, M.-O. 2001. Supply restoration in power distribution systems — a benchmark for planning under uncertainty. In *Proc. ECP*, 85–95.

Thiébaux, S.; Cordier, M.-O.; Jehl, O.; and Krivine, J.-P. 1996. Supply restoration in power distribution systems — a case study in integrating model-based diagnosis and repair planning. In *Proc. UAI*, 525–532.

Williams, B., and Nayak, P. 1997. A reactive planner for a model-based executive. In *Proc. IJCAI*, 1178–1185.

Solving Planning Problems Using Real-Time Model Checking (Translating PDDL3 into Timed Automata)

Henning Dierks
University of Oldenburg
Germany

Gerd Behrmann
BRICS, Aalborg University
Denmark

Kim G. Larsen
BRICS*, Aalborg University
Denmark

Abstract

We present a translation for the variant PDDL 3 of PDDL (Planning Domain Definition Language) into Timed Automata. The advantage of having such a translation is the availability of tool support by model-checkers. We present a case study in which we apply a version of UPPAAL that has been extended for the search of cost-optimal solutions.

Introduction

Scheduling and planning are subjects traditionally studied from different perspectives by the Operation Research (OR) and the Artificial Intellingence (AI) communitities. Typically, OR models tend to reduce these problem to well-defined static optimisation problems to be solved by mathematical programming techniques. AI models emphasise complex logical structure and dynamic interactions between actions and has paid less attention to quantitative aspects. Recently, model-checking techniques developed within the area of computer aided verification have been applied succesfully to the planning (e.g. (Traverso, Veloso, & Giunchiglia 2000; Edelkamp & Helmert 2000)): planning domains are formalized as finite-state automata, goals are expressed as temporal logical formula, and planning is performed with model checking techniques, in particular symbolic model checking using ROBDDs[1].

For planning problems with explicit constraints on action durations, formal verification tools for real-time and hybrid systems, such as UPPAAL (Larsen, Petterson, & Wang Yi 1997) and KRONOS (Yovine 1997), have been applied to solve realistic scheduling problems (Fehnker 1999). The basic common idea of these works is to reformulate the scheduling problem as a reachability problem for so-called timed automata (Alur & Dill 1994) which may then be solved by the verification tools. In particular, the efficient symbolic data structures (BDDs (Bryant 1986), DBMs (Dill 1989), NDD (Asarin, Maler, & Pnueli 1997), CDDs (Behrmann *et al.* 1999), DDDs (Wang 2000)) developed for representing and manipulating the continuous (and infinite) state-space of timed automata are crucial to the success of this approach. In addition, the verification engines of the tools have been extended with guiding and pruning heuristics in order that (time-) optimal or near-optimal solutions (plans) may be found without necessarily exhaustive exploration of the state-space.

PDDL (Planning Domain Definition Language) has been introduced in (McDermott & the AIPS-98 Planning Competition Committee 1998) as common problem-specification language for the AIPS-98 planning competition. The purpose of PDDL is to describe the nature of a domain by specifying its entities and actions that may have effects on the domain. These effects can change the state of the domain.

In order to handle domains with time and numbers PDDL has been extended hierarchically (Fox & Long 2001b; 2001a): The original PDDL is called PDDL level 1; the first extension by numeric effects represents level 2. That means in PDDL at level 2 it is possible to handle functional expressions and effects may change the values of those. The next extension (level 3) allows to specify *durations* for actions. Further extensions introduce duration-dependent effects for durative actions (level 4) and continuous effects for durative actions (level 5).

In this paper we will concentrate on PDDL at level 3 (PDDL_3). In particular, we will offer an *automatic* translation from PDDL_3 domain specifications into networks of timed automata suited for the real-time verification tool UPPAAL. Thus, the entire collection of data structures and heuristic search-algorithms developed within the framework of UPPAAL become available to any planning problem describable within PDDL_3. Beyond PDDL_3 such a translation cannot be done because of the restricted expressiveness of timed automata. The reason is that by duration-dependent effects we can sum up durations. Thus, we need "stopwatches" and it is known that timed automata with stopwatches are more expressive than timed automata (Cassez & Larsen 2000). Moreover, reachability is undecidable for this class of automata.

PDDL

In Fig. 1 an example of a domain description in PDDL at level 3 is given. It describes a variation of a classical planning problem. We have some jugs and we are able to fill

*BRICS: Basic Research in Computer Science, Centre of the Danish National Research Foundation

[1]In 2000, the BDD-based tool MIPS by Stefan Edelkamp and Malte Helmert, Freiburg University, was awarded for "distinguished Performance" at AIPS.

them, to empty them or to pour the contents of one jug into another one. To do so we need one hand for filling or emptying a jug and two hands for pouring. The goal is to reach certain amounts of water within in the jugs by these operations. The $PDDL_3$ code in Fig. 1 contains the specification of the *domain*.

It starts with the name “Timed_Jugs” of the domain and some requirements. These requirements specify the syntactical features that will be used within the domain description. For example, the requirement “durative-actions” indicates that the domain will specify actions with durations. The “types” requirements allows for typing the entities. In the next line we specify the types that are in use within the domain: “jug”. Then we specify a predicate “`(used ?j - jug)`” that signals whether a given jug is already in use by an operation. This avoids simultaneous filling and emptying of the same jug, etc.

After that we specify functional expressions:

- “`(hands)`” declares a nullary function representing the number of hands that are currently available for operations.
- “`(capacity ?j -jug)`” declares a function that assigns to each jug a numerical value representing the maximum capacity of the jug.
- “`(contents ?j -jug)`” declares a function that assigns to each jug a numerical value representing the current contents of water in the jug.
- In our variation of the jug problem we assume that we have to pay a price for water. “`(price)`” declares a nullary function representing the price we have paid at the moment.

The specification of the domain finally declares the actions that may take place:

- The first action “`fill`” shall fill a given jug to its maximum capacity. Hence, it has one parameter namely the jug to be filled. The “`:duration`” field specifies the duration of the action to be the difference between the maximum capacity and the current content. By the “`:condition`” field we can define conditions for the action to happen. In this case we require that at least one hand is free for this action and that the jug is not filled already. Finally, we specify in the “`:effect`” field what will happen. At the beginning of the filling we occupy a hand. At the end of the filling we stop occupying the hand. Moreover, we want the jug to be filled and the price we have to pay for the water is added to “`price`”.
- The next durative actions “`empty`” empties a given jug. The duration of this action corresponds directly to the current contents of the jug. We require that the jug is not empty when the action starts and one hand is free for this operation. The effect of “`empty`” occupies a hand from the start until the end of the action. It also sets the contents of the jug to 0. Note that we do not pay a price for emptying a jug. We only consume time.
- Action “`pour`” represents the pouring of the contents in a jug j_1 into another jug j_2 provided that j_1 contains

```
(define (domain Timed_Jugs)
 (:requirements :typing :fluents
   :conditional-effects
   :durative-actions)
 (:types jug)
 (:predicates (used ?j - jug))
 (:functions
    (hands)
    (capacity ?j - jug)
    (contents ?j - jug)
    (price)  ; we have to pay a price for water
 )

 ; Note that all actions take time
 ; corresponding to the amount of water
 ; that is moved.

(:durative-action fill
 :parameters (?j - jug)
 :duration (= ?duration (- (capacity ?j) (contents ?j)))
 :condition (at start (and (not (used ?j))
                          (>= hands 1)
                          (< (contents ?j) (capacity ?j))))
 :effect
   (and (at start (used ?j))
        (at start (decrease hands 1))
        (at end (not (used ?j)))
        (at end (increase hands 1))
        (at end (assign (contents ?j) (capacity ?j)))
        (at end (increase (price) (- (capacity ?j)
                                     (contents ?j)))))
    ; water is expensive!
 )

 ; all other actions do not increase the price
 ; they only take time (which might be expensive,too)

(:durative-action empty
 :parameters (?j - jug)
 :duration (= ?duration (contents ?j))
 :condition (at start (and (not (used ?j))
                          (>= hands 1)
                          (> (contents ?j) 0)))
 :effect (and (at start (used ?j))
              (at start (decrease hands 1))
              (at end (not (used ?j)))
              (at end (increase hands 1))
              (at end (assign (contents ?j) 0)))
 )

(:durative-action pour
 :parameters (?j1 ?j2 - jug)
 :duration (= ?duration (- (capacity ?j2) (contents ?j2)))
 :condition
    (at start (and (not (used ?j1))
                   (not (used ?j2))
                   (>= hands 2)
                   (> (capacity ?j2) (contents ?j2))
                   (> (contents ?j1)
                      (- (capacity ?j2) (contents ?j2)))))
 :effect
   (and (at start (used ?j1))
        (at start (used ?j2))
        (at start (decrease hands 2))
        (at end (not (used ?j1)))
        (at end (not (used ?j2)))
        (at end (increase hands 2))
        (at end (decrease (contents ?j1)
                          (- (capacity ?j2) (contents ?j2))))
        (at end (assign (contents ?j2) (capacity ?j2))))
 )

(:durative-action pour2
 :parameters (?j1 ?j2 - jug)
 :duration (= ?duration (contents ?j1))
 :condition
    (at start (and (not (used ?j1))
                   (not (used ?j2))
                   (>= hands 2)
                   (>  (contents ?j1) 0)
                   (<= (contents ?j1)
                       (- (capacity ?j2) (contents ?j2)))))
 :effect
    (and (at start (used ?j1))
         (at start (used ?j2))
         (at start (decrease hands 2))
         (at end (not (used ?j1)))
         (at end (not (used ?j2)))
         (at end (increase hands 2))
         (at end (assign (contents ?j1) 0))
         (at end (increase (contents ?j2) (contents ?j1))))
 ))
```

Figure 1: Jug domain

enough water to fill j_2 completely. Therefore the condition requires that j_1 contains more water than the difference between capacity and contents of j_2. Since we need two hands for this operation the condition checks whether those are available. The effect of the action is that we pour water from j_1 into j_2 such that j_2 gets filled completely. The contents of both jugs is adjusted appropriately. Moreover, two hands are occupied during the operation. The time that this action takes depends on the amount of water that is moved.

- Action "pour2" represents the pouring of the whole contents of a jug j_1 into another jug j_2 provided that j_1 contains not more water than the remaining capacity of jug j_2. Hence, the condition of this actions requires that j_1 contains not more water than the difference between capacity and contents of j_2. If the action is finished, j_1 will be empty and j_2 will contain the contents of both jugs before the action. Again, two hands are needed and the duration of "pour2" corresponds directly to the amount of water that changes the jug.

The "used" flags are set appropriately.

The description is not complete since no concrete problem is given so far. This is done by a problem specification like the following:

```
(define (problem jugs1)
  (:domain Timed_Jugs)
  (:objects jug1 jug2 - jug
  )

  ; the classic problem

  (:init (= hands 2)
         (not (used jug1))
         (not (used jug2))
         (= (capacity jug1) 5)
         (= (capacity jug2) 3)
         (= (contents jug1) 0)
         (= (contents jug2) 0)
  )

  (:goal (and (= hands 2)
              (= (contents jug1) 1)))

  ; we have to pay for time AND water!

  (:metric
      minimize (+ total-time price))
)
```

In the specification of a problem we have to define the domain of the problem and the static set of objects of the problem. In this case we declare two jugs named "jug1" and "jug2". Afterwards initialisation of functional expressions may happen. Here we set the capacity of "jug1" to 5 and the capacity of "jug2" to 3. Moreover, we define that both jugs are empty initially. Finally, we declare the goal, a formula speaking about the state space. In this case we simply want to have 1 unit of water in the first jug. We can also specify in which way the solution should be optimal. In our example we are looking for the minimal solution regarding the sum of time and price for water.

The optimal solution is the following plan:

action	time units	cost units
fill(jug2)	3	3
pour2(jug2,jug1)	3	0
fill(jug2)	3	3
pour(jug2,jug1)	2	0
empty(jug1)	5	0
pour2(jug2,jug1)	1	0

Note that the plan executes the actions subsequently. However, it is possible that several actions are executed in parallel. This is allowed when the effects of the actions do not interfere.

UPPAAL

UPPAAL[2] is a modeling, simulation, and verification tool for real-time systems modeled as networks of timed automata (Alur & Dill 1994) extended with data types such as bounded integer variables, arrays etc. For a thorough description of UPPAAL see (Larsen, Petterson, & Wang Yi 1997).

Figure 2 shows an UPPAAL model consisting of two parallel timed automata **P** and **Q** with two and three locations respectively (i.e. **S0** to **S2** and **T0** to **T1**). They use the two clocks x and y, an action channel a, and an integer variable i. Initially all clocks and integer variable values are zero.

When automaton **P** takes the transition from **S0** to **S1** the integer variable i is incremented by one. For **P** to go from location **S1** to **S2** the clock x is required to be exactly 5 (by the guard x==5). On the same transition, the automaton must also synchronize on channel a with automata **Q**, as the edge is labeled with the action a!. Furthermore, automaton **Q** can not delay in location **T0** for more than 42 time units because of the location invariant y<=42.

In general, the edges of component timed automata are decorated with *guards* and *resets*. The guards express conditions on the values of clocks, integer and array variables that must be satisfied in order for the edge to be taken. Here — for reasons of decidability — the constraints on clock values are restricted to bounds on individual clock variables and bounds on differences between clock variables. When taking an edge clock and data variables may be subject to simple manipulations in terms of resets and updates of the form $w := e$. If w is a clock variable e is — again for reasons of decidability — restricted to non-negative integer expressions over bounded data variables. If w is a data variable, e can be any expression of the proper type.

As already indicated automata components may communicate either via global data variables, or using communica-

[2]See the web site http://www.uppaal.com/.

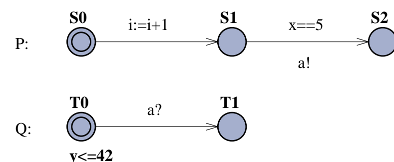


Figure 2: A simple UPPAAL model.

tion channels, which offer a method for two-process synchronization. Now, communication channels may be declared as being *urgent* in which case delay is prevented in situations where two components are already able to synchronize on the given (urgent) channel. Also, locations may be declared as either *urgent* or *committed*. Whenever some component automaton is in an urgent location no further delay is permitted, and some enabled edge of some component must be taken. For committed locations, the edge taken must furthermore be from the component itself. All three notions (urgent channel, urgent locations and committed locations) provide means for ensuring that behaviour progresses before given upper time bounds.

UPPAAL can check reachability and invariance properties of boolean combination of automata locations, and clocks and integers constraints. In UPPAAL, `E<>`ϕ expresses that it is possible to reach a state satisfying ϕ. Dually, `A[]`ϕ expresses invariance of ϕ. For example, property `E<>`**Q.T1** specifies that automata **Q** can reach location **T1**. The property `A[]`(**y**>**42** `imply` **Q.T1**) states that automata **Q** is always operating in location **T1** when clock `y` is greater than 42. The two properties are both satisfied in the model.

In more recent work (Behrmann *et al.* 2001; Larsen *et al.* 2001) the timed automata model as well as the underlying verification engine of UPPAAL have been extended to support computation of *optimal* reachability with respect to various cost criteria. The optimal plans that we will compute in this paper will be based on the work in (Behrmann *et al.* 2001). Here the timed automata model has been extended with discrete costs on edges and the optimality criteria consist in minimizing either the total accumulated time (for reaching a goal state) or the total accumulated discrete cost or the sum of these two. In the model of 2 the minimum time of reaching the location **T1** is clearly 5. Assuming that the edge from **S0** to **S1** has cost 3, and the edge from **S1** to **S2** has cost 4, the minimum discrete cost for reaching **T1** is 7. The minimum combined time and discrete cost for reaching **T1** is clearly 12. Note, that in this simple example the three optimality criteria were met by the same trace(s). In general this will not be the case. The version of UPPAAL reported on in (Behrmann *et al.* 2001; Larsen *et al.* 2001) offer various mechanisms for guiding and pruning the search for optimal reachability and has applied succesfully on a number of scheduling problems (e.g. jobshop scheduling, air-craft landing).

Translation

In this section we construct a system of Timed Automata that represent a semantics for PDDL_3 in the following sense:

Theorem 1 *If $\mathcal{S}$ is a PDDL_3 problem specification, then $\mathcal{TA}(\mathcal{S})$ satisfies the following properties:*

- *Each finite trace of $\mathcal{TA}(\mathcal{S})$ that reaches the state $planner.goal$ describes a valid plan for $\mathcal{S}$.*
- *For each valid plan for $\mathcal{S}$ there exists a finite trace of $\mathcal{TA}(\mathcal{S})$ that reaches the state $planner.goal$*

For the PDDL_3 specification $\mathcal{S}$ we need the following variables and automata for the TA semantics $\mathcal{TA}(\mathcal{S})$:

- We introduce a Timed Automaton $planner$ with states $start$, $work$, $goal$ and $dead$. State $start$ is the initial state.
- We introduce an auxiliary Timed Automaton $grabber$ with the initial state $idle$. There is only one transition in $grabber$ from $idle$ to $idle$ with guard $true$ and synchronisation $violation?$ on the urgent channel $violation$.
- We introduce an auxiliary clock c_0.
- For each initialised ground instance $expr$ of a functional expression we add two global variables: $expr$ and $expr'$.
- For each ground instance $expr$ of a relational expression we add a global variable: $expr$.
- For each ground instance of a durative action da we add:
  - A clock c_{da} to measure the duration of the action.
  - Variables lb_{da} and ub_{da} to remember lower and upper time bounds if given by the specification.
  - A boolean flag $active_{da}$ that signals whether the action is currently executed.

The rest of the construction adds transitions for the automata.

Initialisation

First we add a transition that initialises all variables that are introduced. The problem specification of $\mathcal{S}$ describes initial values by this grammar:

$$\begin{aligned} < init > &::= \texttt{(:init } < init-el >^{+} \texttt{)} \\ < init-el > &::= \texttt{(not (} < grnd_rexpr > \texttt{))} \\ &\quad | \texttt{(} < grnd_rexpr > \texttt{)} \\ &\quad | \texttt{(= (} < grnd_fexpr > \texttt{)} < value > \texttt{)} \end{aligned}$$

Hence, we add a transition from state $start$ to state $work$ of $planner$ with guard $true$ and a set of assignments that is constructed from the $< init-el >$ parts of the initialisation in $\mathcal{S}$:

- A `(not (`$< grnd_rexpr >$`))` term induces an assignment of the form $grnd_rexpr := 0$
- A `(`$< grnd_rexpr >$`)` term induces an assignment of the form $grnd_rexpr := 1$
- A `(= (`$< grnd_fexpr >$`)` $< value >$`)` term induces an assignment of the form $grnd_fexpr := value$

In the jug example we would get the following set of assignments:

$$\begin{aligned} hands &:= 2, \\ used(jug1) &:= 0, \\ used(jug2) &:= 0, \\ capacity(jug1) &:= 5, \\ capacity(jug2) &:= 3, \\ contents(jug1) &:= 0, \\ contents(jug2) &:= 0 \end{aligned}$$

Goal

The goal is given in the problem specification of $\mathcal{S}$. It is basically a formula over the functional and relational expressions with a usual syntax. Quantifiers over objects are allowed and they can be converted into equivalent formulas without quantifiers since the number of objects is finite. The details how to translate formulas in PDDL$_3$ syntax to guards for Timed Automata are straightforward and omitted here. We denote this translation of a formula ϕ into a guard for Timed Automata as $g(\phi)$.

For $\mathcal{TA}(\mathcal{S})$ we add a transition from state $work$ to state $goal$ of the planner with the guard $g(\phi)$ where ϕ is the goal of $\mathcal{S}$. In the jug example we would get this transition:

$$work \xrightarrow{contents(hands)==2 \wedge contents(jug1)==1} goal$$

Actions

PDDL$_3$ distinguishes between actions and durative actions. The latter may take time where actions happen instantaneously. In the jug example it would make sense to specify the empty action as non-durative action since emptying a jug could be done very quickly. In this case the specification of empty without hands and used could look like the following:

```
(:action empty
 :parameters (?j - jug)
 :precondition
    (and (> (contents ?j) 0))
 :effect
    (and (assign (contents ?j) 0))
)
```

A non-durative action may have parameters and carries a precondition and an effect. The precondition is a formula ϕ whereas the effect is a conjunction of terms with the following form:[3]

$$(grnd_rexpr),$$
$$(\texttt{not}(grnd_rexpr)) \text{ or}$$
$$(op\ grnd_fexpr\ term)$$

where op can be `assign`, `increase`, `decrease`, `scale-up` or `scale-down`.

The meaning of the two first kinds of terms is to set a relational expression to true or false, respectively. The meaning of the third kind of such a term is

$$grnd_fexpr' = term$$
$$grnd_fexpr' = grnd_fexpr + term$$
$$grnd_fexpr' = grnd_fexpr - term$$
$$grnd_fexpr' = grnd_fexpr * term$$
$$grnd_fexpr' = grnd_fexpr / term$$

respectively. That means that the new values are determined by the old values. The effect

```
(and (done obj1)
     (assign x y)
     (assign y x)
     (increase z (+ x y))
)
```

exchanges the values for the nullary functional expressions x and y and increases the value of z by the sum of x and y. Moreover, the value of the predicate $done(obj1)$ is set to true.

The values of expressions that do not appear as $grnd_fexpr$ or $grnd_rexpr$ are unchanged. Note that effects are not always well-defined. For example,

```
(and (assign x 3)
     (assign x y)
     (increase x 2)
)
```

is only well-defined if $y = 3$ and $x = 1$ holds before.

```
(and (not (done obj1))
     (done obj1)
)
```

is not well-defined.

A non-durative action is executable iff the precondition holds *and* the effect is well-defined.

We define for auxiliary function a for effects:

$$a(grnd_rexpr)$$
$$\stackrel{\text{df}}{=} grnd_rexpr := 1$$
$$a(\texttt{not}(grnd_rexpr))$$
$$\stackrel{\text{df}}{=} grnd_rexpr := 0$$
$$a(\texttt{assign}\ grnd_fexpr\ term)$$
$$\stackrel{\text{df}}{=} grnd_fexpr' := b(term)$$
$$a(\texttt{increase}\ grnd_fexpr\ term)$$
$$\stackrel{\text{df}}{=} grnd_fexpr' := grnd_fexpr + b(term)$$
$$a(\texttt{decrease}\ grnd_fexpr\ term)$$
$$\stackrel{\text{df}}{=} grnd_fexpr' := grnd_fexpr - b(term)$$
$$a(\texttt{scale-up}\ grnd_fexpr\ term)$$
$$\stackrel{\text{df}}{=} grnd_fexpr' := grnd_fexpr * b(term)$$
$$a(\texttt{scale-down}\ grnd_fexpr\ term)$$
$$\stackrel{\text{df}}{=} grnd_fexpr' := grnd_fexpr / b(term)$$

[3] Note that PDDL$_3$ allows quantification in preconditions and all-quantification in effects. As stated above this can be translated into equivalent formulas resp. effects since the number of objects is finite. Within effects so-called "conditional effects" are allowed, too. The syntax is (when ϕ *eff*). It is clear that an action with a conditional effect can be translated into two actions: The first has the additional precondition ϕ and executes *eff*, the second has the additional precondition $\neg\phi$ and does not execute *eff*. Hence, it suffices for our translation into Timed Automata to consider only actions without these syntatical features.

where $b(term)$ is the straightforward translation from terms in PDDL$_3$ syntax into terms in Timed Automata syntax. The auxiliary function c is given as:

$$c(grnd_rexpr) \stackrel{\text{df}}{=} \varepsilon$$
$$c(\texttt{not}(grnd_rexpr)) \stackrel{\text{df}}{=} \varepsilon$$
$$c(op\ grnd_fexpr\ term) \stackrel{\text{df}}{=} grnd_fexpr := grnd_fexpr'$$

We translate a non-durative action with precondition ϕ and effect `(and (`$t_1 \ldots t_n$`))` as follows:

- If there are $i \neq j$ such that
$$t_i = (grnd_rexpr) \text{ and}$$
$$t_j = (\texttt{not}(grnd_rexpr)),$$
then the effect is not well-defined. In this case we do nothing.
- Otherwise, we introduce a transition from state $work$ to itself with guard
$$g(\phi) \wedge \bigwedge_{1 \leq i < j \leq n} wdc(t_i, t_j)$$
where $wdc(i, j)$ is a well-definedness condition:
$$wdc(t_i, t_j) \stackrel{\text{df}}{=} true \text{ if } lhs(a(t_i)) \neq lhs(a(t_j))$$
$$wdc(t_i, t_j) \stackrel{\text{df}}{=} (rhs(a(t_i)) = rhs(a(t_j)) \text{ otherwise}$$
That means that if there are two different t_k assigning to the same functional expression, then we introduce a test that the result is equal. This must be done to ensure that only well-defined effects are executed.
The assignments of the new transitions are given as:
$$a(t_1), \ldots, a(t_n), c(t_1), \ldots, c(t_n)$$

The translation of the modified empty action for jug1 would be

$$work \xrightarrow[contents(jug1)':=0, contents(jug1):=contents(jug1)']{contents(jug1)>0} work$$

The action

```
(:action example
 :parameters () ; no parameters
 :precondition (and (done)
                    (not (finished)))
 :effect
    (and (finished)
         (assign x 7)
         (assign y 0)
         (increase x (+ y 2))
    )
)
```

would introduce a transition

$$work \xrightarrow[finished:=1, x':=7, y':=0, x':=x+(y+2), x:=x', y:=y', x:=x']{done \wedge \neg finished \wedge 7 == x+(y+2)} work$$

which can be obviously simplified to

$$work \xrightarrow[finished:=1, x':=7, y':=0, x:=x', y:=y']{done \wedge \neg finished \wedge 7 == x+y+2} work$$

Durative Actions

Durative actions are actions where the execution may take time. Like non-durative actions they have conditions for execution and effects. Moreover, constraints on the duration of the execution are specifiable.

Conditions have to be conjunctions of the following formulas:

```
(at start (φ1))
(over all (φ2))
(at end (φ3))
```

where ϕ_i are formulas as in the case of non-durative actions. Effects are also splitted into effects at the start of the action and at the end of the action:

```
(at start (eff))
(at end (eff))
```

The meaning of the time-specifiers for the condition are:

at start The formula has to hold when the execution of the durative action starts.

over all The formula has to hold after execution of the effects at the start of the action. It has to be valid as long as the execution of the actions takes place.

at end The formula has to hold at the time where the execution of the action finishes.

Effects are executed according to their time specification.[4]

Duration constraints are conjunctions of the following formulas:

```
(at start dc)
(at end dc)
```

where dc is

```
(= ?duration fexp)
(<= ?duration fexp)
(>= ?duration fexp)
```

fexp is a functional expression. Note that the time specification is necessary since the evaluation of *fexp* may lead to different results at the beginning and the end of the action.[5]

Let us assume that a ground instance da of a durative action has the following condition:

```
(and (at start (φ1))
     (over all (φ2))
     (at end (φ3)))
```

The effects are

$$\texttt{(and (at start } t_1^s \ldots t_k^s\texttt{)}$$
$$\texttt{(at end } t_1^e \ldots t_m^e\texttt{)}$$

[4]As in the case of non-durative actions PDDL$_3$ accepts durative actions with extended formulas and effects which can be reduced similarly to durative actions as described in this paper.

[5]In Fig. 1 duration constraints appear *without* time specification because the default time specifier is `at start`.

and duration constraints are given by

$$(\texttt{and } (ts_1\,(op_1 \texttt{ ?duration } grnd_fexp_1))\ \ldots \\ (ts_n\,(op_n \texttt{ ?duration } grnd_fexp_n))$$

where $grnd_fexp_i$ are grounded functional expressions, $op_i \in \{\texttt{=}, \texttt{<=}, \texttt{>=}\}$, and $ts_i \in \{\texttt{at start}, \texttt{at end}\}$.

Then we translate such a durative action as follows:

- If there are $i \neq j$ such that

  $$t_i^{ts} = (grnd_rexpr) \text{ and}$$
  $$t_j^{ts} = (\texttt{not}(grnd_rexpr)),$$

  with $ts \in \{s, e\}$ then the effect is not well-defined. In this case we do nothing.
- Otherwise, we introduce the following transitions:
  - A loop for state $work$ with guard

    $$g(\phi_1) \wedge \bigwedge_{1 \leq i < j \leq k} wdc(t_i^s, t_j^s) \wedge active_{da} == 0$$

    and assignments

    $$a(t_1^s), \ldots, a(t_k^s), c(t_1^s), \ldots, c(t_k^s),$$
    $$active_{da} := 1, c_{da} := 0,$$
    $$lb_{da} := \max\{grnd_fexp_i | ts_i = \texttt{at start}, op_i \in \{\texttt{=}, \texttt{>=}\}\}$$
    $$ub_{da} := \min\{grnd_fexp_i | ts_i = \texttt{at start}, op_i \in \{\texttt{=}, \texttt{<=}\}\}$$
  - A loop for state $work$ with guard

    $$g(\phi_3) \wedge \bigwedge_{1 \leq i < j \leq m} wdc(t_i^e, t_j^e) \wedge active_{da} == 1$$
    $$\wedge\, c_{da} \leq ub_{da} \wedge c_{da} \geq lb_{da}$$
    $$\wedge\, c_{da} \leq \min\{grnd_fexp_i | ts_i = \texttt{at end}, op_i \in \{\texttt{=}, \texttt{<=}\}$$
    $$\wedge\, c_{da} \leq \max\{grnd_fexp_i | ts_i = \texttt{at end}, op_i \in \{\texttt{=}, \texttt{>=}\}$$

    and assignments

    $$a(t_1^e), \ldots, a(t_m^e), c(t_m^e), \ldots, c(t_m^e),$$
    $$active_{da} := 0$$
  - A transition from state $dead$ to state $dead$ with guard

    $$\neg g(\phi_2) \wedge active_{da} == 1$$

    and *synchronisation on the urgent channel* $violation$:

    $$violation!$$

Hence, for a successful execution of a durative action the Timed Automaton has to fire two transition for the start and the end of the action. At the start the automaton has to check the start condition ϕ_1 and to execute the start effects which have to be well-defined. The flag $active_{da}$ symbolises the status of the action. It is true iff the action is currently executed. When the action is started we reset the clock c_{da} to measure the duration of the execution. Moreover, we save the bounds on the duration that have to be evaluated at the beginning of the action.

To finish action da the Timed Automaton has to fire the second transition. Here we check the condition ϕ_3 and the well-definedness of the effects at the end of da. The transition shall only be firable iff da is under execution. Hence, we test the flag $active_{da}$ appropriately. Finally, we test whether the duration requirements are met by the corresponding comparisons.

The third transition ensures that during execution of da the condition ϕ_2 always holds. As soon as $\neg\phi_2$ is true during execution this transition is firable. Since it is a synchronisation on an urgent channel it will be executed eventually. The resulting effect is that this computation will never reach state $goal$ because there is no transition from $dead$.

Since we do not want to reach state $goal$ if a durative action is under execution we extend the guard for the transition from $work$ to $goal$ by a test whether all durative actions are inactive, ie.

$$work \xrightarrow{g(\phi) \wedge \bigwedge_{da} active_{da} == 0} goal$$

Simultaneous Actions

The semantics of PDDL_3 allows to execute more than one action (or end-points of durative actions) at the same point of time. That means that this set A of actions (or end-points) happen simultaneously. This is possible if

- all conditions are satisfied,
- the evaluation of a condition is not affected by an effect of another action in A, and
- the conjunction of the effect is well-defined.

To simplify the analysis of these conditions a weak notion of interference has been introduced in (Fox & Long 2001b). Only sets of non-interfering actions (or end-points) are allowed in the semantics. Interference can be determined by a simple syntactical analysis. Let a and b be actions (or end-points). They are interfering iff one of the following conditions is true.

- The (pre-)condition of a contains a ground instance of a relational expression $grnd_rexpr$ that is set or reset by the effects of b (or vice versa).
- The effects of a set a ground instance of a relational expression $grnd_rexpr$ that is *reset* by the effects of b (or vice versa).
- The effects of a change a value of a ground instance of a functional expression $grnd_fexpr$ that appears rhs in an assignment of b (or vice versa).
- The effects of a and b change the value of a ground instance of a functional expression $grnd_fexpr$ and one of this changes is non-additive.

In the Timed Automaton semantics for PDDL_3 specifications we have to introduce a new transition for each set $A \stackrel{\text{df}}{=} \{a_1, \ldots, a_k\}$ of non-interfering actions (or end-points). Let t_a be the transition introduced for an action $a \in A$. Then the transition for the simultaneous action set A carries the conjunction of all guards:

$$\bigwedge_{a \in A} guard(t_a)$$

The assignment of the transition is the sequence of all assignments:

$$assignments(t_{a_1}), \ldots, assignments(t_{a_k})$$

Due to the interference conditions the order of the assignments can be chosen arbitrarily.

In the model of Timed Automata it is not necessary that some time passes between transitions. That means that we have a two-dimensional time domain because within a certain point of time *ordered sequences* of transitions can fire. However, the time model of PDDL_3 does not allow such a behaviour, it is required that time passes between execution of sets of actions. Hence, we extend for each loop in state $work$ the guard by the condition $c_0 > 0$ and the assignments by $c_0 := 0$. Then it is guaranteed that some time has to pass before execution of a set of actions.

Case Study

We have implemented a prototype based on a parser for PDDL[6]. The input of this tool is a PDDL specification containing a domain and a problem. If the parsing is successful, then the tool constructs a file that contains the translation of the PDDL specification as described in Sect. Translation[7].

Note that the translation of PDDL_3 specification as explained so far does not consider the definition of a metric if given in the problem (cf. Section PDDL). For planning problems a version of UPPAAL has been developed in which it possible to specify costs of both passing time and transitions (Behrmann *et al.* 2001). With this option the prototype is able to handle four different possibilities:

- no metric is given: In this case the result of the translation is not modified.
- `(:metric minimize total-time)`: In this case the consumption of time shall be minimal. Hence, we specify that the costs increase uniformly with the time.
- `(:metric minimize price)`: We assume that `price` is a nullary function that is only increased by effects of the PDDL_3 specification. Here, all increasing effects are translated into proper costs for the corresponding transitions.
- `(:metric minimize (+ total-time price))`: This is the accumulation of both types of costs. We translate this by both increasing the cost uniformly with the time and increasing the cost properly when `price` is increased.

As case study we consider the domain of the timed jugs again. However, with slightly more difficult problem specification. We introduce two more jugs with capacity 7 resp. 9 and finally we would like to have 2 water units in the first jug and the second jug and 6 water units in the fourth jug:

[6]available at http://www.dur.ac.uk/d.p.long/competition.html.

[7]There are some modifications that save variables resp. clocks. Moreover, UPPAAL only allows formulas as guards which are constructed by comparisons and conjunction. Hence, we have to construct the disjunctive normalform of a given guards and introduce a transition for each conjunction in this normalform.

```
(define (problem jugs1)
  (:domain Timed_Jugs)
  (:objects jug1 jug2 jug3 jug4 - jug
  )

  ; the classic problem

  (:init (= hands 2)
         (not (used jug1))
         (not (used jug2))
         (not (used jug3))
         (not (used jug4))
         (= (capacity jug1) 5)
         (= (capacity jug2) 3)
         (= (capacity jug3) 7)
         (= (capacity jug4) 9)
         (= (contents jug1) 0)
         (= (contents jug2) 0)
         (= (contents jug3) 0)
         (= (contents jug4) 0)
  )

  (:goal (and (= hands 2)
              (= (contents jug1) 2)
              (= (contents jug2) 2)
              (= (contents jug4) 6)))

  ; we have to pay for time AND water!

  (:metric
      minimize (+ total-time price))
)
```

With two hands and equivalent costs for time and water the optimal plan is[8]

		contents			
action	t	1	2	3	4
f_2, f_3	0	0	0	0	0
	3		3		
$2 \to 4$	7			7	
$3 \to 1$	10		0		3
$1 \to 2$	15	5		2	
$2 \to 4$	18	2	3		
$3 \to 2$	21		0		6
	23		2	0	

It consumes the minimal amount of water (10) and takes 23 time units to execute. However, if we change the metric in a way that water is for free, a different plan is optimal:

		contents			
action	t	1	2	3	4
f_1, f_4	0	0	0	0	0
f_2	5	5			
	8		3		
$2 \to 3$	9				9
$4 \to 2$	12		0	3	
$2 \to 3$	15		3		6
$1 \to 2$	18		0	6	
$2 \to 3$	21	2	3		
	22		2	7	

In this case we can save one time unit by the cost of 7 water units. Note that the PDDL semantics requires a time distance between subsequent durative actions in this case because they are interfering. However, the distances can be chosen arbitrarily small such that the plans above can be executed within all intervals longer than 23 (resp. 22) time

[8]We use these abbreviations: f_i: fill jug i and $i \to j$: pouring from jug i into jug j.

units. For the simplicity of the presentation we omit these distances here.

We can also consider the same problem but with more hands than two. If we assume that three hands are available and water is as expensive as time, then the optimal plan is:

action	t	contents 1	2	3	4
f_2, f_3	0	0	0	0	0
$2 \to 4$	3		3		
	6		0		3
$3 \to 1$	7			7	3
$1 \to 2$	12	5		2	
$2 \to 4$	15	2	3		
$3 \to 2$	18		0		6
	20		2	0	

Hence, the third hand would save 3 time units here. Note that this plan is not feasible with two hands but very similar to the first plan above. If water is for free, we can even save more time:

action	t	contents 1	2	3	4
f_1, f_2	0	0	0	0	0
f_4	2				
	3		3		
$2 \to 3$	5	5			
$1 \to 2$	8		0	3	
$2 \to 3$	11	2	3		9
$4 \to 2$	14		0	6	
$2 \to 3$	17	2	3		6
	18		2	7	

In case of four hands (more than four hands are not useful for four jugs) we get the following plans. In the case of expensive water we get the same solution as with three hands. In the case where water is for free, the system computes a solution that takes only 16 time units:

action	t	contents 1	2	3	4
f_1, f_2, f_4	0	0	0	0	0
$2 \to 3$	3		3		
	5	5			
$1 \to 2$	6	5	0	3	
$4 \to 1, 2 \to 3$	9	2	3		9
$1 \to 2$	12	5	0	6	6
$4 \to 2$	15	2	3		
$2 \to 3$	16		2	7	

The execution times of the model-checking are as follows:[9]

problem	time	
2 hands, exp. water		24 s
2 hands, free water		55 s
3 hands, exp. water	1 m	58 s
3 hands, free water	8 m	40 s
4 hands, exp. water	6 m	56 s
4 hands, free water	21 m	25 s

[9]The execution times for the translation are neglectable.

These results have been established by a special feature in the cost optimizing version of UPPAAL: The tool has a list of the reachable symbolic states that are still waiting for being checked. In this version of UPPAAL it is possible to order this list in a very useful way. The order of the list is given by a function $heur$ on the state space. For example, a simple and useful definition would be $heur(s) := cost(s)$ for each state s where $cost(s)$ describes the minimum cost to reach s. The result of this heuristic is that the tool explores the cheapest state first.

However, the situation of our case study is even better: We defined $heur(s)$ as follows

$$heur(s) := cost(s) + \frac{1}{2} \sum_{i=1,2,4} |goal(i) - content(i, s)|$$

where $goal(i)$ decribes the desired contents of the ith jug and $content(i)$ is the current content of the ith jug in state s. Thus, we assign to state the minimal costs to reach this state *plus* estimated remaining costs to reach the goal. Typically such estimates are difficult to find. However, the function above is a rough, simple lower bound for the remaining costs. This heuristic prefers the most promising states in the search order.

The positive effect of a good heuristic function can be seen from the following execution times of UPPAAL when we use

$$heur(s) := cost(s)$$

instead:

problem	time	
2 hands, exp. water		55 s
2 hands, free water	1m	18 s
3 hands, exp. water	20 m	14 s
3 hands, free water	20 m	38 s
4 hands, exp. water	$>$ 90 m	
4 hands, free water	81 m	40 s

Conclusion

In this paper we have described a translation from $PDDL_3$ into the modelling formalism (cost-decorated) timed automata model thus allowing cost- and time-optimal solutions to PDDL$-$3 planning problems to be re-formulated as cost-optimizing reachability problems for timed automata. Initial experiments with the translations in combination with the real-time model checker UPPAAL demonstrate the feasibility of this approach.

The experiments reported on in the current version of this paper do not fully exploit *all* the guiding and pruning heuristics developed for the cost-optimizing version of UPPAAL (Behrmann *et al.* 2001). In particular, all our experiments have been using a so-called *minimum cost* search order, which has the property that the first solution found is guaranteed to be the optimal one. However, in several cases it is often useful to apply alternative (heuristic/random) search orders to quickly obtain an upper bound on the cost, which in the continued search for better solutions may be used to prune the search space. In (Behrmann *et al.* 2001) 15 Lawrence Job Shop problems were considered. Here the

minimum-cost search order failed on all instances to find (the optimal) solutions in 30 minutes, whereas the solutions found within just 60 seconds by random search with pruning were close to the optimal (in fact identical to the optimal in 11 cases).

As for long-term future research we want to consider alternative translations from PDDL_3 to UPPAAL as the particular translation chosen will clearly have a major impact on the final performance. The current translation is rather monolithic resulting in a system with only two timed automata, but with heavy use of discrete variables. This calls for identification of good symbolic data structures for simultaneous representation of *both* discrete and continuous parts of the state space. However, despite various suggestions (Asarin, Maler, & Pnueli 1997; Behrmann *et al.* 1999; Wang 2000) this is as yet an open research problem (on which we — as others — are working). An alternative translation would opt for more timed automata and fewer discrete variables (e.g. one timed automata for each object in the problem instance), in the hope of being able to exploit the (independence) structure of such a model. However, it is still an open research topic how to extend methods like partial-order-reduction to the setting of time.

References

Alur, R., and Dill, D. 1994. A theory of timed automata. *TCS* 126:183–235.

Asarin, E.; Maler, O.; and Pnueli, A. 1997. Data–structures for the Verification of timed automata. In *Proc. of the Int. Workshop on Hybrid and Real-Time Systems*.

Behrmann, G.; Larsen, K. G.; Pearson, J.; Weise, C.; and Yi, W. 1999. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *Proc. of the 11*th *Int. Conf. on Computer Aided Verification*, Lecture Notes in Computer Science. Springer–Verlag.

Behrmann, G.; Fehnker, A.; Hune, T.; Larsen, K.; Petterson, P.; and Romijn, J. 2001. Efficient Guiding Towards Cost-Optimality in Uppaal. In Margaria, T., and Wang Yi., eds., *Proceedings of TACAS'01*, volume 2031 of *LNCS*, 174–188. Springer.

Bryant, R. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* C-35(8):677–691.

Cassez, F., and Larsen, K. 2000. The Impressive Power of Stopwatches. In *International Conference on Concurrency Theory (CONCUR)*, number 1877 in LNCS, 138–152. Springer.

Dill, D. 1989. Timing Assumptions and Verification of Finite-State Concurrent Systems. In Sifakis, J., ed., *Proc. of Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, 197–212. Springer–Verlag.

Edelkamp, S., and Helmert, M. 2000. On the Implementation of Mips. In *Artificial Intelligence Planning and Scheduling (AIPS), Workshop on Decision-Theoretic Planning*, 18–25. AAAI press.

Fehnker, A. 1999. Scheduling a Steel Plant with Timed Automata. In *Proc. of the* 6th *International Conference on Real-Time Computing Systems and Applications*. IEEE Computer Society Press.

Fox, M., and Long, D. 2001a. PDDL+ level 5: An Extension to PDDL2.1 for Modelling Planning Domains with Continuous Time-dependent Effects. Technical report, University of Durham.

Fox, M., and Long, D. 2001b. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. Technical report, University of Durham.

Larsen, K.; Behrmann, G.; Brinksma, E.; Fehnker, A.; Hune, T.; Pettersson, P.; and Romijn, J. 2001. As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata. In Berry, G.; Comon, H.; and Finkel, A., eds., *Proceedings of CAV 2001*, number 2102 in LNCS, 493–505. Springer.

Larsen, K.; Petterson, P.; and Wang Yi. 1997. Uppaal in a nutshell. *Software Tools for Technology Transfer* 1(1+2):134–152.

McDermott, D., and the AIPS-98 Planning Competition Committee. 1998. PDDL – The Planning Domain Definition Language. Technical report. Available at: www.cs.yale.edu/homes/dvm.

Traverso, P.; Veloso, M.; and Giunchiglia, F. 2000. Model-Theoretic Approaches to Planning. In *Workshop of the 5th International Conference of AI Planning and Scheduling (AIPS'00)*. AAAI press.

Wang, F. 2000. Efficient Data Structure for Fully Symbolic Verification of Real-Time Software Systems. In *Proc. of the* 6th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *LNCS*. Springer.

Yovine, S. 1997. Kronos: a verification tool for real-time systems. *Software Tools for Technology Transfer* 1(1+2):123–133.

Symbolic Exploration in Two-Player Games: Preliminary Results

Stefan Edelkamp

Institut für Informatik
Albert-Ludwigs-Universität
Georges-Köhler-Allee, Geb. 51
D-79110 Freiburg
eMail: edelkamp@informatik.uni-freiburg.de

Abstract

In this paper symbolic exploration with binary decision diagrams (BDDs) is applied to two-player games to improve main memory consumption for reachability analysis and game-theoretical classification, since BDDs provide a compact representation for large set of game positions. A number of examples are evaluated: *Tic-Tac-Toe*, *Nim*, *Hex*, and *Four Connect*. In *Chess* we restrict the considerations to the creation of endgame databases. The results are preliminary, but the study puts forth the idea that BDDs are widely applicable in game playing and provides a universal tool for people interested in quickly solving practical problems.

Introduction

BDDs have encountered AI for different purposes. The most apparent area is (especially non-deterministic) *planning* (Cimatti, Roveri, & Traverso 1998), since the spaces of planning problems tend to be very large. AI-Planning can be casted as *Model Checking* (Clarke, Grumberg, & Peled 2000), where it has been observed that BDDs can be used to concisely express sets of states in a state transition system. *Symbolic reachability analysis* traverses each state within the system and applies to transition systems with more than 10^{20} states. Even verifying temporal properties as specified in different logics according to additionally labeled states is possible for moderate system sizes.

BDDs have also been applied to single-agent problems like the $(n^2 - 1)$–Puzzle and Sokoban (Edelkamp & Reffel 1998). The applied search algorithm BDDA* enjoys recent research interests of the groups Hansen, Zhou, and Feng (ADDA*), or Jensen, Bryant and Veloso (SetA*). However, due to the lack of refined symbolic information the results for $(n^2 - 1)$–Puzzle and Sokoban are currently too weak to compete with heuristic single state-space search techniques.

As game playing in AI contributes to both AI-planning and AI-search, this paper considers two- and multi-player games in exploring large sets of game positions with moderate node sizes of the corresponding BDD data structure. It is structured as follows. In the next two sections we briefly address BDD basics and two-player zero-sum games with complete information. Afterwards we introduce the symbolic exploration technique with respect to the example of *Tic-Tac-Toe*. We show how to encode the problem and how to perform reachability analysis and game-theoretical classification. In the experimental section we additionally address games like *Nim*, *Hex*, and *Four Connect*, and explain how to compute symbolic endgame databases for *Chess*.

BDDs

Binary decision diagrams (Bryant 1985) have been introduced in the context of hardware verification and are a graphical representation of Boolean functions. More precisely, a *binary decision diagram (BDD)* to represent a Boolean function f is a directed rooted acyclic graph with one or two terminal nodes, labeled with 0 and 1, and internal nodes with out-degree two. A Boolean variable from the variable set of the represented function f is associated to every internal node. The outgoing edges of node k are labeled with *low*(k) and *high*(k). The *interpretation* of a BDD for f is f_{low} if the variable at the root node is zero and f_{high} otherwise, where f_{low} and f_{high} are itself interpreted as BDDs.

For a compact representation of f two rules for reducing the graph representation are available. *Rule 1* deletes a node with all outgoing edges if there exists another node with the same labeling and same successors. *Rule 2* deletes a node k if *low*(k) is equal to *high*(k). A BDD is *reduced* if neither Rule 1 nor Rule 2 can be applied anymore. It is *ordered*, if on every path in the graph a total ordering of the variables is preserved. Since reduced, ordered BDDs are a unique representation of Boolean functions, with the term BDD we always refer to reduced and ordered BDDs.

The main advantage with respect to a truth table is that in practice BDDs tend to have tractable (polynomial) size even if the represented set of all satisfying assignments is intractable (exponential). This is due to the fact that a directed acyclic graph commonly represents exponentially many paths. A further advantage is that given two BDD-representations for functions f and g the logical operations $f \wedge g$, $f \vee g$ and $\neg f$ can be executed efficiently and for the existential ($\exists$) and universal ($\forall$) quantification the efficient graph representation positively influences the execution time.

Two-Player Zero-Sum Games

A two-player zero-sum game (with perfect information) is given by a set of states S, move-rules to modify states and

two players, called Player 0 and Player 1. Since one player is active at a time, the entire state space of the game is $Q = S \times \{0, 1\}$. A game has an initial state and some predicate *goal* to determine whether the game has come to an end. We assume that every path from the initial state to a final one is finite. For the set of goal states $G = \{s \in Q \mid goal(s)\}$ we define an evaluation function $v : G \to \{-1, 0, 1\}$, -1 for a lost position, 1 for a winning position, and 0 for a drawing one. This function is extended to $\hat{v} : Q \to \{-1, 0, 1\}$ asserting a game theoretical value to each state in the game.

Let $L(i)$ be the set of lost positions for Player i, $i \in \{0, 1\}$. The set $L(1)$ can be recursively calculated as follows: $i)$ All lost position s for Player 1 are contained in $L(1)$, i.e., $\{g \in G \mid v(g) = -1\} \subseteq L(1)$. $ii)$ If for each move of Player 1 in state s there exists a move of Player 2 to a state in $L(1)$ then s itself is in $L(1)$. The set $L(2)$ of lost positions for Player 2, is defined analogously. These definitions assume optimal play for both players, where each player chooses a move that maximizes his objective, e.g. Player 2 has to choose a move that forces Player 1 remain in $L(1)$ and so does Player 1 in the analogous case.

Let R be the set of all reachable states, with respect to the initial position and the rules of the game, then $D = R \setminus (L(1) \cup L(2))$ is the set of draw games.

Constructing the above sets $L(1)$ $L(2)$ and D together with their game theoretical value in a backward traversal of the state space of the game is referred to as *classification* or *retrograde analysis*.

A general introduction to *Two-Person Game Theory* is provided by (Rapoport 1966).

Exploration and Classification Algorithms

We exemplify the algorithmic considerations to compute the set of reachable states and the game theoretical value for large sets of states in the game *Tic-Tac-Toe* and distinguish the two players by denoting *White* for Player 1 and *Black* for Player 2.

Tic-Tac-Toe is a pencil-and-paper two-player game. It is played on a 3×3–Grid, with alternating ticks of the players. The winner of the game has to complete either a row, a column or a diagonal with his own ticks. Obviously, the size of the state space is bounded by $3^9 = 19,683$ states, since each field is either unmarked or marked with Player 1 or 2. A complete enumeration shows that there is no winning strategy (for either side); the game is a draw.

Encoding of States and Transitions

To encode a state s all positions are indexed as Figure 1 visualizes.

1	1	1
2		
	2	

1	2	3
4	5	6
7	8	9

Figure 1: a) A State in Tic-Tac-Toe b) Labeling of the Board

We devise two predicates: $Occ(s, i)$ being 1 if position i is occupied, and $Black(s, i)$ evaluating to 1 if the position i, $1 \leq i \leq 9$, is marked by Player 2. This results in a total state encoding length of 18 bits. All final positions in which Player 1 has lost are defined by enumerating all rows, columns and the two diagonals as follows.

$$\begin{aligned}
&\mathit{WhiteLost}(s) = \\
&(\mathit{Occ}(s,1) \wedge \mathit{Occ}(s,2) \wedge \mathit{Occ}(s,3) \wedge \\
&\mathit{Black}(s,1) \wedge \mathit{Black}(s,2) \wedge \mathit{Black}(s,3)) \vee \\
&(\mathit{Occ}(s,4) \wedge \mathit{Occ}(s,5) \wedge \mathit{Occ}(s,6) \wedge \\
&\mathit{Black}(s,4) \wedge \mathit{Black}(s,5) \wedge \mathit{Black}(s,6)) \vee \\
&(\mathit{Occ}(s,7) \wedge \mathit{Occ}(s,8) \wedge \mathit{Occ}(s,9) \wedge \\
&\mathit{Black}(s,7) \wedge \mathit{Black}(s,8) \wedge \mathit{Black}(s,9)) \vee \\
&(\mathit{Occ}(s,1) \wedge \mathit{Occ}(s,4) \wedge \mathit{Occ}(s,7) \wedge \\
&\mathit{Black}(s,1) \wedge \mathit{Black}(s,4) \wedge \mathit{Black}(s,7)) \vee \\
&(\mathit{Occ}(s,2) \wedge \mathit{Occ}(s,5) \wedge \mathit{Occ}(s,8) \wedge \\
&\mathit{Black}(s,2) \wedge \mathit{Black}(s,5) \wedge \mathit{Black}(s,8)) \vee \\
&(\mathit{Occ}(s,3) \wedge \mathit{Occ}(s,6) \wedge \mathit{Occ}(s,9) \wedge \\
&\mathit{Black}(s,3) \wedge \mathit{Black}(s,6) \wedge \mathit{Black}(s,9)) \vee \\
&(\mathit{Occ}(s,1) \wedge \mathit{Occ}(s,5) \wedge \mathit{Occ}(s,9) \wedge \\
&\mathit{Black}(s,1) \wedge \mathit{Black}(s,5) \wedge \mathit{Black}(s,9)) \vee \\
&(\mathit{Occ}(s,3) \wedge \mathit{Occ}(s,5) \wedge \mathit{Occ}(s,7) \wedge \\
&\mathit{Black}(s,3) \wedge \mathit{Black}(s,5) \wedge \mathit{Black}(s,7))
\end{aligned}$$

The predicate *BlackLost* is defined analogously. In order to specify the transition relation we fix a *Frame* denoting that in the transition from state s to s' besides the move in the actual grid cell j nothing else will be changed.

$$\begin{aligned}
&\mathit{Frame}(s,s',j) = \\
&(\mathit{Occ}(s,j) \wedge \mathit{Occ}(s',j)) \vee \\
&(\neg\mathit{Occ}(s,j) \wedge \neg\mathit{Occ}(s',j)) \wedge \\
&(\mathit{Black}(s,j) \wedge \mathit{Black}(s',j)) \vee \\
&(\neg\mathit{Black}(s,j) \wedge \neg\mathit{Black}(s',j))
\end{aligned}$$

These predicates are superimposed to express that with respect to board position i the status of every other cell is preserved.

$$\mathit{Frame}(s,s',i) = \bigwedge_{1 \leq i \neq j \leq 9} \mathit{Frame}(s,s',j)$$

Now we can express the relation of a black move with origin s and successor s'. As a precondition we have that one cell i is not occupied and the effects of the operator are that in state s' cell i is occupied and black.

$$\begin{aligned}
&\mathit{BlackMove}(s,s') = \\
&\bigvee_{1 \leq i \leq 9} \neg\mathit{Occ}(s,i) \wedge \mathit{Black}(s',i) \wedge \\
&\qquad \mathit{Occ}(s',i) \wedge \mathit{Frame}(s,s',i)
\end{aligned}$$

The predicate *WhiteMove* is defined analogously. To devise the encoding of all moves in the transition relation $Trans(s, s')$ we address one additional bit $Move(s)$ for each state s, denoting the truth of Player 2's turn, as follows.

$$
\begin{aligned}
&Trans(s, s') = \\
&(\neg Move(s) \land \neg WhiteLost(s) \land \\
&WhiteMove(s, s') \land Move(s')) \lor \\
&(Move(s) \land \neg BlackLost(s) \land \\
&BlackMove(s, s') \land \neg Move(s'))
\end{aligned}
$$

There are two cases. If it is Player 2's turn and if he is not already lost, execute all black moves. The next move is a white one. The other case is interpreted as follows. If Player 1 has to move, and if he is not already lost, execute all possible white moves and continue with a black one.

Reachability Analysis

Symbolic reachability analysis traverses the entire state space that is reachable from the initial position. Essentially reachability analysis corresponds to a symbolic breadth-first search traversal, which successively takes the set *From* of all positions in the current iteration and applies the transition relation to find the set of all *New* positions in the next iteration. For the iteration to be completed we further need a procedure *Replace* to change the nodes labeling from the association with respect to s' back to an association with respect to s. Iteration is aborted if no new position is available. The union of all new positions is stored in the set *Reached*. All sets are represented by BDDs according to the given encoding. The implementation is depicted in Fig 2.

procedure *Reachable*
    $Reach \leftarrow From \leftarrow Start(s')$
    do
        $To \leftarrow Replace(From, s', s)$
        $To \leftarrow \exists s\ (Trans(s, s') \land To(s))$
        $From \leftarrow New \leftarrow To \land \neg Reach$
        $Reach \leftarrow Reach \lor New$
    while *New*

Figure 2: Calculating the Set of Reachable Positions.

Game-Theoretical Classification

As stated above, two-player games with perfect information are classified iteratively. Therefore, in opposite to reachability analysis the direction of the search process is *backwards*. Fortunately, backward search causes no problem, since the representation of all moves has already been defined as a relation.

Assuming optimal play and starting with all goal situations according to one player – here Black's lost positions – all previous winning positions – here White's winning positions – are computed. A position is lost for Player 2 if all moves lead to an intermediate winning position in which white can force a move back to a lost position.

$$
\begin{aligned}
&BlackLose(s) = \\
&BlackLost(s) \lor \forall s'\ (Trans(s, s') \Rightarrow \\
&\quad (\exists s''\ Trans(s', s'') \land BlackLost(s'')))
\end{aligned}
$$

Note that the choice of the operators $\land$ for existential quantification and $\Rightarrow$ for universal quantification are crucial.

procedure *Classify*
    $WhiteWin \leftarrow false$
    $BlackLose \leftarrow From \leftarrow BlackLost(s)$
    do
        $To \leftarrow Replace(From, s, s')$
        $To \leftarrow \exists s'\ (Trans(s, s') \land To(s'))$
        $To \leftarrow To \land \neg Reach \land \neg Move(s)$
        $WhiteWin \leftarrow WhiteWin \lor To$
        $To \leftarrow Replace(To, s, s')$
        $To \leftarrow \forall s'\ (Trans(s, s') \Rightarrow To(s'))$
        $To \leftarrow To \land Move(s)$
        $From \leftarrow New \leftarrow To \land \neg BlackLose$
        $BlackLose \leftarrow BlackLose \lor New$
    while *New*

Figure 3: Determining the set of white winning and black lost positions.

The pseudo-code for symbolic classification is shown in Fig. 3. The algorithm *Classify* starts with the set of all final lost postitions for Black, and alternates beween the set of positions that in which black (at move) will loose and positions in which white (at move) can win, assuming optimal play. In each iteration each player moves once; corresponding to two quantification in the analysis. The executed Boolean operations are exactly those established in the recursive description above.

One important issue of the pseudo-code is the explicit attachment of the player to move, since this information might not be available in the backward traversal. Furthermore, the computations can be restricted to the set of reachable states through conjuncts with its BDD representation. We summarize that given a suitable state encoding *Config*, for symbolic exploration and classification in a specific two-player game the programmer has to implement the procedures of the following interface.

1. *Start(Config)*: Definition of the initial state for reachability analysis.
2. *WhiteLost(Config)*: Final lost positions for white.
3. *BlackLost(Config)*: Final lost position for black.
4. *WhiteMove(Config, Config)*: Transition relation for white moves.
5. *BlackMove(Config, Config)*: Transition relation for black moves.

Complete Exploration

In this section we experiment with implementations to the given interface specification for simple two-player games in

C/C++ with J. Lind-Nielsen's BDD library *BuDDy*, Release 1.6 (Lind-Nielsen 1999).

Tic-Tac-Toe

Table 1 depicts the growths of the number of states and the number of BDD nodes in the reachability analysis and the classification algorithm with respect to an increasing search depth d in Tic-Tac-Toe. The statistics for the BDD structure for *BlackLose* and *WhiteWin* are interleaved. Instead of the expected $9!/5!4! = 126$ newly generated states in the last iteration we obtain 78 states. This is due to the fact that we stop state enumeration if a goal has been found.

	Reachable		BlackLose		WhiteWin	
d	n	s	n	s	n	s
0	19	1	90	626		
1	37	10			532	1498
2	81	82	772	1034		
3	111	334			825	1838
4	152	1090	997	1098		
5	175	2350			1002	1986
6	344	3870				
7	525	5010				
8	652	5400				
9	656	5478				

Table 1: Reachability analysis and classification in the game *Tic-Tac-Toe*. The numbers of BDD nodes and the sizes of the represented sets of reachable states, white winning and black lost positions are given.

The number of represented states in a BDD corresponds to the number of paths to the one sink. The number of reachable lost positions for black is 626, whereas we have 316 lost positions for white. The number of eventually lost positions for black accumulates to 1098. The number of white winning positions is 1986 and, as expected, the start is not won for white.

Four Connect

This popular game is played on a vertical rack with 6 rows and 7 columns. Alternatively, black and white pieces are inserted into one pile falling down onto the existing ones. The state space is obviously bounded by 3^{42} and we experiment with encoding of two bits for each of the 42 positions, ordered from bottom to top and right to left. When encoding the height of each column for occupancy $42 + 7 \cdot 3 = 63$ bits suffice, for a state space of at most 10^{18} states. Tables 2 and 3 depict the results of the reachability analysis. *Four Connect* has been proven to be a win for the first player in optimal play using a knowledge-based approach (Allis 1998) and minimax-based proof number search (PNS) (Allis 1994), that introduces the third value *unknown* into the game search tree evaluation. PNS has a working memory requirement linear in the size of the search tree, while $\alpha\beta$ requires only memory linear to the depth of the tree. Proof-Set Search is a recent improvement to PNS, that trades node explorations for a higher memory consumption (Müller 2001b).

Due to space limitation we have not yet verified Allis' result. So far, we have only succeeded in full BDD-classifications for 4-Connect up to a 4×5 board, which are all draws.

	Reachable		New	
l	n	s	n	s
0	85	1	157	7
1	169	8	322	49
2	334	57	502	238
3	582	295	901	1,120
4	979	1,415	1,476	4,263
5	1,630	5,678	2,426	16,422
6	2,728	22,100	3,977	54,859
7	4,542	76,959	6,347	186,389
8	7,436	263,348	10,260	567,441
9	12,246	830,789	15,669	1.73e+06
10	19,353	2.56e+06	23,127	4.81e+06
11	29,064	7.38e+06	28,152	1.34e+07
12	39,394	2.08e+07	37,494	3.43e+07
13	51,978	5.51e+07	41,081	8.83e+07
14	65,097	1.43e+08	54,072	2.09e+08
15	80,717	3.53e+08	59,908	5.01e+08
16	96,554	8.55e+08	74,105	1.11e+09
17	113,616	1.96e+09	78,847	2.48e+09
18	129,004	4.45e+09	92,083	5.17e+09
19	145,584	9.63e+09	96,704	1.08e+10
20	160,009	2.04e+10	109,838	2.11e+10
21	175,720	4.16e+10	114,229	4.14e+10

Table 2: Reachability analysis of the game *Four Connect*. The numbers of BDD nodes and represented reached and new states for each step l in the exploration are depicted.

Hex

Hex is a classical board game invented by the Danish mathematician Hein. The book (Browne 2000) provides a comprehensive report on the history of the game and advanced playing strategies. The board is a hexagonal tiling of n rows and m columns. Usually $m = n$, with 11x11 being the widely accepted standard board size. The rules are simple: players take turns placing a piece of their color on an unoccupied location. The game is won when one player establishes an unbroken chain of ther pieces connection their sides of the board. Since the game can never result in a draw it is easy to prove that the game is won for the first player to move, since otherwise he can adopt the winning strategy of the second player to win the game. Nevertheless the proof is not constructive such that we are still left with the problem to determine the game theoretical value of all intermediate positions. The state space of *Hex* is bounded by 3^{n^2} as each point may exists in either of three states *Empty*, *White* or *Black*. The current state-of-the-art program *Hexy* uses a quite unusual approach electrical circuit theory to combine the influence of sub-positions (virtual connections) to larger and larger ones (Anshelevich 2000).

The binary encoding for *Hex* is similar to the previous example, with two bits per field, since the players are only allowed to set their pieces. Table 4 displays the results of

l	Reachable n	Reachable s	New n	New s
22	188,617	8.31e+10	125,290	7.61e+10
23	201,421	1.59e+11	127,098	1.39e+11
24	210,410	2.99e+11	133,653	2.40e+11
25	220,472	5.39e+11	135,216	4.13e+11
26	227,188	9.52e+11	140,897	6.64e+11
27	234,464	1.61e+12	141,266	1.06e+12
28	238,101	2.68e+12	144,497	1.59e+12
29	241,280	4.27e+12	141,112	2.35e+12
30	239,985	6.62e+12	137,731	3.25e+12
31	239,253	9.88e+12	133,574	4.42e+12
32	234,583	1.43e+13	127,470	5.57e+12
33	228,654	1.98e+13	121,470	6.83e+12
34	219,173	2.67e+13	111,600	7.70e+12
35	207,433	3.44e+13	101,312	8.31e+12
36	191,440	4.27e+13	83,287	8.11e+12
37	176,475	5.08e+13	73,051	7.36e+12
38	160,640	5.82e+13	55,880	5.75e+12
39	144,188	6.39e+13	44,667	3.82e+12
40	127,134	6.78e+13	29,267	1.87e+12
41	109,287	6.96e+13	15,147	5.33e+11
42	90,269	7.02e+13	0	0

Table 3: Reachability analysis of the game *Four Connect* (cont.)

reachability analysis applied for this encoding. Time consumption in all cases was within 1 minute on a 450 MHz Pentium, and space in the order of 128 MByte was sufficient for exploration.

n	l	Transition Relation n	Reachable n	Reachable s
3	9	171	144	6,046
4	16	311	424	1.01e+07
5	25	491	1,000	1.61e+11
6	36	711	2,034	2.40e+16
7	49	971	3,724	3.30e+22
8	64	1,271	6,304	4.15e+29
9	81	1,611	10,044	4.78e+37
10	100	1,991	15,250	5.00e+46
11	121	2,411	22,264	4.76e+56
12	144	2,871	31,464	4.11e+67

Table 4: Final results of the reachability analysis for the game *Hex* scaled with parameter n denoting an $n \times n$ board. The final depth of the analysis l, the BDD-size of the transition relation and the numbers of BDD nodes and represented states for a complete exploration are presented.

The challenging question is to how to encode the end of the game, i.e. the connection of two opponent sides by the respective color of the players. We proceed as follows. All paths from one node a to a node b of length l are generated in a Divide-and-Conquer style by recursively determining all paths from a to an intermediate node k of length $\lfloor l/2 \rfloor$ and all paths from k to b of length $\lceil l/2 \rceil$. To avoid re-computations of BDDs we memorize calculated results in a 3-dimensional table $T[a][b][l]$. With 210 BDD nodes for $n = 3$, 424 for $n = 4$, and 7206 for $n = 5$ the BDD representations of the goal predicates is small, but the enumeration of all paths is very time consuming, such that we have verified correct classification only for these cases.

Nim

Nim is another folklore two-player game. We consider the very simple situation of one stack of disks, where each player is allowed to take one, two, or three of them. Note that the n-stack Nim problem reduces to a 1-stack problem by applying combinatorial game theory (Berlekamp, Conway, & Guy 1982). The situation is lost if the resulting vector is zero. The game is lost for the player, who faces the empty stack. The optimal strategy is to enforce a situation with $3k + 1$ disks, $k \geq 1$, and the opponent to move. One encoding is very simple. We assign a bit b_i for each disk d_i, $1 \leq i \leq n$. Therefore, a move (for either side) is represented by the conjunction $b_i(s) \wedge b_j(s) \wedge b_k(s) \wedge \neg b_i(s') \wedge \neg b_j(s') \wedge \neg b_k(s') \wedge \mathit{Frame}(s, s', i, j, k)$, for $1 \leq i, j, k \leq n$, where $\mathit{Frame}(s, s', i, j, k)$ asserts that every bit except the bits at i, j and k is unchanged. This encoding, however, scales linearly with the number of disks. Therefore, a concise representation requires the binary encoding of natural numbers in finite domain. This reduces the encoding length from $n+1$ bits (the additional bit denotes the player to move) to $\lceil \log n \rceil + 1$. The set of reachable states in the case $n = 2$ and the first encoding are `00`-*white moves*, `00`-*black moves*, `01`-*black moves*, `10`-*black moves*, and `11`-*white moves*. In the second encoding this reduces to the set `0`-*white moves*, `0`-*black moves*, `1`-*black moves*, and `2`-*white moves*.

Table 5 gives the number of reachable states in the state space and the size of their BDD representation for both cases. While in the first encoding the represented set grows exponential ($\sim 2^{n+1}$ states) the BDD representation scale linearly ($2n + 1$ nodes). For the second encoding the growth of the symbolic representation is still smaller than the represented set of $2n$ states, e.g for $n = 10,000$ only 19 BDD nodes represent the total of 20,000 states. Note that explicit and symbolic classification is also trivial for this game and verifies the above game-theoretical values of the start states.

Endgame Databases in Chess

Several current challenges in single-agent search like Sokoban, Rubik's Cube and the $(n^2 - 1)$-Puzzle can be solved with State-of-the-Art implementations and use a common data structure for improving the lower bound estimate on the solution length: The pattern database, in which sub-positions are stored together with their optimal solution length in a relaxed problem space. In two-player games, pattern databases are generated to determine the game-theoretical value for endgames. Pattern databases can be casted as representations of Boolean functions. Instead of computing the value of a function f for a given input from scratch, a representation of f is stored in a table. For the input the pattern value can be retrieved in a simple table lookup.

	Unary Encoding		Binary Encoding	
n	n	s	n	s
5	11	58	5	10
10	21	2,037	6	20
15	31	65,520	6	30
20	41	2.09e+06	8	40
25	51	6.71e+07	7	50
30	61	2.14e+09	7	60
35	71	6.87e+10	8	70
40	81	2.14e+09	10	80

Table 5: The game *Nim* with an either unary or binary encoding of natural numbers. The numbers of BDD nodes and represented states for the entire set of reachable states are provided. In the binary encoding equivalent states have been merged.

Chess is one of the oldest games known to mankind. The book (Heinz 2000) provides a computer-chess primer and new results of computer-chess. Chess has advanced from the *drosophila* of AI to one of its main successes, resulting in the defeat of the human-world champion in a tournament match. Some combinatorial chess problems like the number of 33,439,123,484,294 complete Knight's tours have already been solved with BDDs (Löbbing & Wegener 1996). Due to the complexity of chess we are restricted to the construction of endgame databases. Not only the size of the chess board but also the numerous available moves lead to intractable large dat base for all states and transitions. Early work on endgame databases is surveyed in (van den Herik & Herschberg 1986). Nowadays, Edward's table-bases and Thomson's databases are most important to the chess community. The major compressions schemes for positions without pawns (since these situations contain only totally reversible moves) use symmetries along the diagonal, horizontal and vertical middle axes of the chess board. Since the 3-fold symmetry allows for the confinement of one piece in a triangular region of 10 squares, the obtained reduction ratio is about $(64 - 10)/64 = 84.38\%$.

Example Setting

As a first example we consider a board with two opposing Kings and one white Queen. In the encoding of the moves for the kings we restrict successor generation to squares not threatened by another figure. Movements of the queen are more complicated. The Queen can freely move on rows, columns and diagonals as long as no other figure is present. Removal is only allowed if a figure of the opposite color is encountered at the destination square. However, captures in this simple problem instance simply terminates the game since the remaining game is either a draw (two opposing Kings) or a definite win for white. Opposite to Tic-Tac-Toe in each move two squares are changed for each move. Therefore the *Frame* has to be adequately enlarged with this further parameter.

The simple encoding with three bits for each square denoting occupancy, color and figure type, together with one bit for the player to move, yields 193 bits in total, a number for which symbolic exploration turns out to be intractable. The BDD sizes simply exhaust main memory. A more suitable encoding of board positions is a binary representation of the occupied squares for the two kings and the queen. Since six bits per figure suffice to encode 64 squares this gives a total of 19 bits.

In a specialized implementation of the example problem the chess configuration *ChessConfig* is realized by three simple predicates: $qw(i, j)$ for row *i* and column *j* returns a BDD, evaluating to 1, if and only if the white queen is positioned at (i, j). The procedures $kw(i, j)$ and $ks(i, j)$ yield analogous BDD representations for the black and white king, respectively. The constructor includes a switch for the different variable sets in the situations prior and after move commitment. The class *Chess* then implements all above methods, while we have added two methods *BlackThreatened* and *WhiteTreatened* that indicate, if a given square is available for the player black and white, respectively.

General Setting

In a more general case of arbitrary figures on the board exploration is based on the BDD representation of the figures' positions $At(i, j)$, $1 \leq i, j \leq 8$. This implementation scans the endgame database description and the initial state for reachability analysis in the command line.

We distinguish between white and black occupancy of squares, since according to the color of the pieces the target position might lead to a capture. For Queens, Bishops and Rooks all intermediate positions within one move are blocked. Therefore, to specify a move beside the precalculated *Frame* we pre-compute two vectors *EmptyWhite* and *EmptyBlack* representing the emptiness of each square. *EmptyWhite* (*EmptyBlack*) w.r.t. (i, j) evaluates to 1, if no white (black) figure is currently located at position (i, j). For example, all Knight jumps in the direction up-up-left are characterized as follows.

$$\bigvee_{2<i\leq 8, 1<j\leq 8} At(s,i,j) \wedge At(s',i-2,j-1) \wedge \\ \mathit{EmptyWhite}(s,i-2,j-1) \wedge Frame(s,s')$$

A capture is found by querying variables equality for each pair of figures on the board. This predicate simply assigns that the binary value of the board positions of different figures coincide. Figures that have been captured are placed onto an extension of the board.

In order to specify the *Check* and *Mate* positions, we apply the classification algorithm for one and two plies (half-move). A position is *Mate* if even in perfect play the foreign king is definitely captured in one move and a position is *Check* if the king can be taken assuming a void move of the opponent. Therefore, a *Check-Mate* is the conjunct of the former two predicates. Table 6 gives our preliminary results. For these simple endgame studies the game can be terminated if one of the player takes a figure of the opponent. In general we encounter a database entry of a smaller game.

Reachability and classification took less than 30 seconds on a 450 MHz Pentium. The number of represented states in

	Check-Mate		Reachable	
	n	s	n	s
KQK	385	256	75	516,096
KRK	297	216	75	516,096
KRRK	8,829	13,0720	794	3.19e+07
KRNK	2,595	12,751	954	3.19e+07
KNNK	804	236	1,050	3.19e+07
KNBK	519	156	1,148	1.59e+07
KBBK	1,295	1,056	641	8.12e+06

Table 6: Endgame databases in *Chess* according to different situations. The letter N abbreviates Knight, K denotes King Q Queen, R Rook, B Bishop.

the final set of reachable states exceeds the BDD representation by magnitudes. In classifying the example KBRK, the set for states eventually won for the first player has had 312'932 elements with a BDD representation of 15'460 nodes, which corresponds to a saving of about 95 %; the larger the number of states, the better the gain by BDD representation.

In all cases backward analysis is more time consuming than forward exploration. This observation not necessarily reflects BDD sizes but to the number of sub-functions encountered in existential and universal quantification. Moreover, the number of subproblems is related to the number of represented states. Therefore, backward iterations corresponds to a sizable amount of work if the number of represented states in the goal predicate is large.

After a successful classification the optimal strategy for Player 1 can be obtained by memorizing the sets of states *WhiteWins* in each iteration. If Player 2 has performed his move a simple conjunction of the available successors with the *WhiteWins* gives the next winning position.

Endgame database queries can be decided in time linear to the encoding length. Therefore, BDD endgame databases can compete with hashing schemes to query the game-theoretical value in ordinary endgame databases.

The symbolic representation of a BDD B transforms to an explicit one by extracting and deleting one represented state after another. A satisfying path p in the B is extracted as a BDD P and B is updated to $B \wedge \neg P$ until B represents false. One application is a print routine for the PDB.

A Framework for Multi-Player Games

Let Q be the set of all states and for $G = \{s \in Q \mid \mathit{goal(s)}\}$ we define an evaluation function $v : G \to \{0, 1, \ldots, k\}$, with k being the number of players and the value $i > 0$ corresponds to a definite win for Player i.

As in the two-player scenario this function is extended to $\hat{v} : Q \to \{0, 1, \ldots, k\}$ by induction. The set of winning positions for player i, i.e. all positions $s \in Q$ with $\hat{v}(s) = i$, is calculated as follows.

$$
\begin{aligned}
&\mathit{PlayerWin}(i, s) = \\
&\mathit{PlayerWon}(i, s) \vee \\
&\quad \forall s^{(1)} \, (Trans(s^{(0)}, s^{(1)}) \Rightarrow \\
&\quad\quad (\forall s^{(2)} \, Trans(s^{(1)}, s^{(2)}) \Rightarrow \\
&\quad\quad\quad \ldots \\
&\quad\quad\quad (\forall s^{(k)} \, Trans(s^{(k-1)}, s^{(k)}) \Rightarrow \\
&\quad\quad\quad\quad (\exists s^{(k+1)} \, Trans(s^{(k)}, s^{(k+1)}) \wedge \\
&\quad\quad\quad\quad\quad \mathit{PlayerWin}(i, s^{(k+1)}) \ldots)
\end{aligned}
$$

Other Domains and Their Encoding

Due to the depth and diversity of the research in the area of game-playing (Schaeffer 2000) we can only indicate possible applications of BDD-exploration and classification for a complete exploration.

Nine-Men-Morris has been solved with huge pattern databases (Gasser 1993), in which every situation (after the initial setting) has been asserted to its game-theoretical value. The outcome of a complete search is that the game is a draw. A straightforward encoding yields 48 bits for $3 \cdot 8$ placements. Since Gasser's result has never been verified BDDs are a mean for reconsidering the result. Since each cyclic layer of the *Nine-Men-Morris* graph can be furtherly compressed to 12 bits, we find a better encoding of 36 bits. However, involved encodings might lead to a more complicated transition relations.

In its time the retired human-computer world-champion *Checkers* program *Chinnock* has performed its intense endgame database computations (up the 7 and 8 pieces) on various high-end computers in the US and Canada. All checker positions involving 8 or fewer pieces on the board, result in a total of 443,748,401,247 positions. In a simple encoding Checkers requires $32 \cdot \lceil \log 5 \rceil = 96$ bits. As above an improved encoding might get closer to the reasonable bound of $\lceil \log 5^{32} \rceil = 75$ bits. Although this seems tractable for a BDD engineer, experiments with the Fifteen-Puzzle in a concise encoding of 64 bits has shown that reachability analysis easily exhausts 500 MByte of memory with symbolic breadth first search. Therefore, for the application of presented approach in Checkers we suggest to increase the bound on endgame computations.

The size of the search space in *Go* (19×19 variant) has been estimated at 10^{170} positions ($3^{19 \cdot 19} \approx 10^{172}$), and is probably the biggest of all popular board games. The resulting binary encodings of over 500 bits are definitely too large for a complete symbolic exploration. Go has been addressed by different strategies. One important approach (Müller 1995) with exponential savings in some endgame situations uses a Divide-and-Conquer method based on combinatorial game theory in which some board situations are split into a sum of local games of tractable size. In these sub-searches BDD databases might be advantageous. The general strategy of partial order bounding (Müller 2001a) to propagate relative evaluations in the tree has been shown to be effective in Go endgames; it applies to all minimax search algorithms such as $\alpha\beta$ and PNS.

One multi-player game is *Halma/Chineese Checkers* with its star-like game board introduced at the end of the 19th century. It can be played with up to 6 players. The goal of the game is to move all own pieces to the opposing side of the board by sliding single pieces to adjacent places or jump-

ing over adjacent pieces if the destination is free. Chaining of jumps is allowed and reveals the tactics of the game. One obvious encoding considers 3 bits for each of the 121 board positions to denote the occupancy of the pieces such that this game is likely to be too complex to be solved with current BDD technology.

Conclusion and Outlook

To the authors' knowledge, this paper is one of the first manuscripts on using BDDs to classify two-player games. The only other work we are aware of is (Baldamus *et al.* 2002), that applies a Model Checker to solve American Checkers problems.

In the experiments we highlighted possible memory savings for the complete exploration in two simple problems *Nim* and *Tic-Tac-Toe* and a medium-size problem *Four Connect*. With *Hex* and *Chess* we gave examples, where the binary encodings of the moves and the ending is not trivial. The results are preliminary, but lead to drastic savings in the considered problem spectrum. The gap often corresponds to several magnitudes. We generalized the approach to the the multi-player scenario which especially in card-games attracts several researchers nowadays (cf. (Ginsberg 1999) for an example).

The results are preliminary. In large instances to *Hex* and in *Four Connect* we have not yet determined the game-theoretical value of all states for larger problem instances with the symbolic traversal of the search space. Moreover, the domain of chess gives only rather trivial results (King + 1 or 2 pieces vs. King), which state spaces have already fully been explored by many computer-chess programmers. Last but not least, the indicated application of the methods in *Nine-Man-Morris*, *Checkers* and *Halma* is quite speculative and yet not been implemented. However, these limitation are not necessarily problem-inherent, and symbolic representation and exploration is promising to enrich the portfolio of game playing programs. Moreover, the simplicity and generality of the approach can serve as an interface for specifying simple two-player games with optimal play. The next two options to improve the performance are static or dynamic variable ordering schemes, which usually have a significant influence on the space and time complexity, and partitioning techniques of the search space to bypass bottlenecks in the symbolic exploration process. As in the case of chess, game playing problem instances are often redundant with respect to different automorphisms to be exhibited by combined reduction schemes and advanced data structures.

Since reachability analysis is in practice easier and faster than classification, BDDs might support the evaluation of successors in active play as follows. Take the successor state and evaluate the game all the way down to the end and draw statistics on how often the goal situation is met for both players. The successor with the best score for one player is searched first. Enumeration bases on recent progress in game playing, since its successful variant is *random sampling* that has been applied in *Bridge*, where *Monte Carlo Sampling* determines the hands of the opponents (Ginsberg 1999) and to *Backgammon*, where a *Roll-Out* reveals the current strength of the game (Tesauro 1995).

Another reason why counting might be an advantage to $\alpha\beta$ pruning in min-max search is that this algorithm is sensible to the number of leaf nodes responsible for the root evaluation even though alpha-beta tends to hide errors at leaf nodes. This problem has been addressed by conspiracy number search (CNS) (McAllester 1988). The basic idea of CNS is to search the tree in a manner that at least $c > 1$ leaf values have to change in order to change the root one. CNS has been successfully applied to chess by (Schaeffer 1989) and (Lorenz 2000).

As a final side remark, note that two player exploration joins many features with adversarial universal planning for multi-agent domains in which a set of uncontrollable agents may be adversarial to the planner (Jensen, Veloso, & Bowling 2001).

Acknowledgment The author would like to thank DFG for the support in the projects Ot 64/13-2 and Ed 74/2-1.

References

Allis, V. 1994. *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. Dissertation, Department of Computer Science, Limburg, Maastrich.

Allis, V. 1998. A knowledge-based approach to connect-four. the game is solved: White wins. Master's thesis, Vrije Univeriteit, The Netherlands.

Anshelevich, V. V. 2000. The game of hex: An automatic theorem proving approach to game programming. In *National Conference on Artificial Intelligence (AAAI)*, 189–194.

Baldamus, M.; Schneider, K.; Wenz, M.; and Ziller, R. 2002. Can american checkers be solved by means of symbolic model checking? *Electronic Notes in Theoretical Computer Science* 43.

Berlekamp, E. R.; Conway, J. H.; and Guy, R. K. 1982. *Winning ways (book)*. Academic Press, vol.I and II, 850 pages.

Browne, C. 2000. *Hex Strategy: Making the right connections*. A K Peters.

Bryant, R. E. 1985. Symbolic manipulation of boolean functions using a graphical representation. In *ACM/IEEE Design Automation Conference*, 688–694.

Cimatti, A.; Roveri, M.; and Traverso, P. 1998. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *National Conference on Artificial Intelligence (AAAI)*, 875–881.

Clarke, E. M.; Grumberg, O.; and Peled, D. A. 2000. *Model Checking*. MIT Press.

Edelkamp, S., and Reffel, F. 1998. OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)*, 81–92.

Gasser, R. 1993. *Harnessing Computational Resources for Efficient Exhaustive Search*. Ph.D. Dissertation, ETH Zürich.

Ginsberg, M. 1999. Step toward an expert-level bridge-playing program. In *IJCAI*, 584–589.

Heinz, E. A. 2000. *Scalable Search in Computer Chess*. Vierweg.

Jensen, R. M.; Veloso, M. M.; and Bowling, M. H. 2001. Obdd-based optimistic and strong cyclic adversarial planning. In *European Conference on Planning (ECP)*.

Lind-Nielsen, J. 1999. Buddy: Binary decision diagram package, release 1.7. Technical Univeristy of Denmark. jln@itu.dk.

Löbbing, M., and Wegener, I. 1996. The number of night's tours equals 33,439,123,484,294 - counting with binary decision diagrams. *The Electronic Journal of Combinatorics*.

Lorenz, U. 2000. Controlled conspriracy-2 search. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, 466–478.

McAllester, D. A. 1988. Conspiricy-number-search for min-max searching. *Artificial Intelligence* 35:287–310.

Müller, M. 1995. *Computer Go as a sum of local games*. Ph.D. Dissertation, ETH Zürich.

Müller, M. 2001a. Partial order bounding: A new approach to game tree search. *Artificial Intelligence*.

Müller, M. 2001b. Proof set search. Technical Report TR01-09, University of Alberta.

Rapoport, A. 1966. *Two-Person Game Theory*. Dover.

Schaeffer, J. 1989. Conspiracy numbers. *Advances in Computer Chess 5 (Eds. Beal, D.F)* 199–217.

Schaeffer, J. 2000. The games computer (and people) play. In Zelkowitz, M., ed., *Advances in Computers*, volume 50, 189–266. AcademicPress.

Tesauro, G. 1995. Temporal difference learning and te-gammon. *Communication of the ACM* 38(3):58–68.

van den Herik, H. J., and Herschberg, I. 1986. A data base on data bases. *ICCA Journal* 29–34.

Symbolic LAO* Search for Factored Markov Decision Processes

Zhengzhu Feng
Computer Science Department
University of Massachusetts
Amherst MA 01003

Eric A. Hansen
Computer Science Department
Mississippi State University
Mississippi State MS 39762

Abstract

We describe a planning algorithm that integrates two approaches to solving Markov decision processes with large state spaces. It uses state abstraction to avoid evaluating states individually. And it uses forward search from a start state, guided by an admissible heuristic, to avoid evaluating all states. These approaches are combined in a novel way that exploits symbolic model-checking techniques and demonstrates their usefulness in solving decision-theoretic planning problems.

Introduction

Markov decision processes (MDPs) have been adopted as a framework for AI research in decision-theoretic planning. Classic dynamic programming (DP) algorithms solve MDPs in time polynomial in the size of the state space. However, the size of the state space grows exponentially with the number of features describing the problem. This "state explosion" problem limits use of the MDP framework, and overcoming it has become an important topic of research.

Over the past several years, approaches to solving MDPs that do not rely on complete state enumeration have been developed (Boutilier *et al.* 1999). One approach exploits a feature-based (or factored) representation of an MDP to create state abstractions that allow the problem to be represented and solved more efficiently. Another approach limits computation to states that are reachable from the starting state(s) of the MDP. In this paper, we show how to combine these two approaches. Moreover, we do so in a way that demonstrates the usefulness of symbolic model-checking techniques for decision-theoretic planning.

There is currently great interest in using symbolic model-checking to solve AI planning problems with large state spaces. This interest is based on recognition that the problem of finding a plan (i.e., a sequence of actions and states leading from an initial state to a goal state) can be treated as a reachability problem in model checking. The planning problem is solved symbolically in the sense that reachability analysis is performed by manipulating sets of states, rather than individual states, which alleviates the state explosion problem. Earlier work has shown that symbolic model checking can be used to perform deterministic and nondeterministic planning (Cimatti *et al.* 1998). Nondeterministic planning is similar to decision-theoretic planning in that it considers actions with multiple outcomes, allows plan execution to include conditional and iterative behavior, and represents a plan as a mapping from states to actions. However, decision-theoretic planning is more complex than nondeterminsitic planning because it associates probabilities and rewards with state transitions. The problem is not simply to construct a plan that can reach the goal, but to find a plan that maximizes expected value.

The first use of symbolic model checking for decision-theoretic planning is the SPUDD planner (Hoey *et al.* 1999), which solves factored MDPs using dynamic programming. Although it does not use reachability analysis, it uses symbolic model-checking techniques to perform dynamic programming efficiently on sets of states. Our paper builds heavily on this work, but extends it in an important way. Whereas dynamic programming solves an MDP for all possible starting states, we use knowledge of the starting state(s) to limit planning to reachable states. In essence, we show how to improve the efficiency of the SPUDD framework by using symbolic reachability together with dynamic programming. Our algorithm can also be viewed as a generalization of heuristic-search planners for MDPs, in which our contribution is to show how to perform heuristic search symbolically for factored MDPs. The advantage of a heuristic search approach over dynamic programming is that heuristic search can focus planning resources on the relevant parts of the state space.

Factored MDPs and decision diagrams

A Markov decision process (MDP) is defined as a tuple (S, A, P, R) where: S is a set of states; A is a set of actions; P is a set of transition models $P^a : S \times S \rightarrow [0, 1]$, one for each action, specifying the transition probabilities of the process; and R is a set of reward models $R^a : S \rightarrow \Re$, one for each action, specifying the expected reward for taking action a in each state. We consider MDPs for which the objective is to find a policy $\pi : S \rightarrow A$ that maximizes total discounted reward over an indefinite (or infinite) horizon, where $\gamma \in [0, 1)$ is the discount factor.

In a factored MDP, the set of states is described by a set of random variables $\mathbf{X} = \{X_1, \ldots, X_n\}$, where $\mathbf{x} = \{x_1, \ldots, x_n\}$ denotes a particular instantiation of the variables, corresponding to a unique state. Without loss of generality, we assume state variables are Boolean. Because the

set of states $S = 2^{\mathbf{X}}$ grows exponentially with the number of variables, it is impractical to represent the transition and reward models explicitly as matrices. Instead we follow Hoey *et al.*(1999) in using algebraic decision diagrams to achieve a more compact representation.

Algebraic decision diagrams (ADDs) are a generalization of binary decision diagrams (BDDs), a compact data structure for Boolean functions used in symbolic model checking. A decision diagram is a data structure (corresponding to an acyclic directed graph) that compactly represents a mapping from a set of Boolean state variables to a set of values. A BDD represents a mapping to the values 0 or 1. An ADD represents a mapping to any discrete set of values (including real numbers). Hoey *et al.* (1999) describe how to represent the transition and reward models of a factored MDP compactly using ADDs. We adopt their notation and refer to their paper for details of this representation. Let $\mathbf{X} = \{X_1, \ldots, X_n\}$ represent the state variables at the current time and let $\mathbf{X}' = \{X'_1, \ldots, X'_n\}$ represent the state variables at the next step. For each action, an ADD $P^a(\mathbf{X}, \mathbf{X}')$ represents the transition probabilities for the action. Similarly, the reward model $R^a(\mathbf{X})$ for each action a is represented by an ADD. The advantage of representing mappings from states to values as ADDs is that the complexity of operators on ADDs depends on the number of nodes in the diagrams, not the size of the state space. If there is sufficient regularity in the model, ADDs can be very compact, allowing problems with large state spaces to be represented and solved efficiently. Symbolic model checking has been used to verify hardware and software systems with up to 10^{100} states. It also shows promise in scaling up AI planning algorithms to cope with large state spaces.

Symbolic LAO* algorithm

To solve factored MDPs, we describe a symbolic generalization of the LAO* algorithm (Hansen & Zilberstein 2001). LAO* is an extension of the classic search algorithm AO* that can find solutions with loops. This makes it possible for LAO* to solve MDPs, since a policy for an infinite-horizon MDP allows both conditional and cyclic behavior. Like AO*, LAO* has two alternating phases. First, it expands the best partial solution (or policy) and evaluates the states on its fringe using an admissible heuristic function. Then it performs dynamic programming on the states visited by the best partial solution, to update their values and possibly revise the best current solution. The two phases alternate until a complete solution is found, which is guaranteed to be optimal.

AO* and LAO* differ in the algorithms they use in the dynamic programming step. Because AO* assumes an acyclic solution, it can perform dynamic programming in a single backward pass from the states on the fringe of the solution to the start state. Because LAO* allows solutions with cycles, it relies on an iterative dynamic programming algorithm (such as value iteration or policy iteration). In organization, the LAO* algorithm is similar to the "envelope" dynamic programming approach to solving MDPs (Dean *et al.* 1995). It is also closely related to RTDP (Barto *et al.* 1995), which is an on-line (or "real time") search algorithm for MDPs, in contrast to LAO*, which is an off-line search algorithm.

We call our generalization of LAO* a symbolic search algorithm because it manipulates sets of states, instead of individual states. In keeping with the symbolic model-checking approach, we represent a set of states S by its characteristic function χ_S, so that $s \in S \iff \chi_S(s) = 1$. We represent the characteristic function of a set of states by an ADD. (Because its values are 0 or 1, we can also represent a characteristic function by a BDD.) From now on, whenever we refer to a set of states, S, we implicitly refer to its characteristic function, as represented by a decision diagram.

In addition to representing sets of states as ADDs, we represent every element manipulated by the LAO* algorithm as an ADD, including: the transition and reward models; the policy $\pi : S \rightarrow A$; the state evaluation function $V : S \rightarrow \Re$ that is computed in the course of finding a policy; and an admissible heuristic evaluation function $h : S \rightarrow \Re$ that guides the search for the best policy. Even the discount factor γ is represented by a simple ADD that maps every input to a constant value. This allows us to perform all computations of the LAO* algorithm using ADDs.

Besides exploiting state abstraction, we want to limit computation to the set of states that are reachable from the start state by following the best policy. Although an ADD effectively assigns a value to every state, these values are only relevant for the set of reachable states. To focus computation on the relevant states, we introduce the notion of *masking* an ADD. Given an ADD D and a set of relevant states U, masking is performed by multiplying D by χ_U. This has the effect of mapping all irrelevant states to the value zero. We let D_U denote the resulting *masked ADD*. (Note that we need to have U in order to correctly interpret D_U). Mapping all irrelevant states to zero can simplify the ADD considerably. If the set of reachable states is small, the masked ADD often has dramatically fewer nodes. This in turn can dramatically improve the efficiency of computation using ADDs.[1]

Our symbolic implementation of LAO* does not maintain an explicit search graph. It is sufficient to keep track of the set of states that have been "expanded" so far, denoted G, the *partial value function*, denoted V_G, and a *partial policy*, denoted π_G. For any state in G, we can "query" the policy to determine its associated action, and compute its successor states. Thus, the graph structure is implicit in this representation. Note that throughout the whole LAO* algorithm, we only maintain one value function V and one policy π. V_G and π_G are implicitly defined by G and the masking operation.

Symbolic LAO* is summarized in Table 1. In the following, we give a more detailed explanation.

[1] Although we map the values of irrelevant states to zero, it does not matter what value they have. This suggests a way to simplify a masked ADD further. After mapping irrelevant states to zero, we can change the value of a irrelevant state to any other non-zero value whenever doing so further simplifies the ADD.

policyExpansion(π, S^0, G)
1. $E = F = \emptyset$
2. $from = S^0$
3. REPEAT
4. $to = \bigcup_a Image(from \cap S^a_\pi, P^a)$
5. $F = F \cup (to - G)$
6. $E = E \cup from$
7. $from = to \cap G - E$
8. UNTIL ($from = \emptyset$)
9. $E = E \cup F$
10. $G = G \cup F$
11. RETURN (E, F, G)

valueIteration(E, V)
12. $saveV = V$
13. $E' = \bigcup_a Image(E, P^a)$
14. REPEAT
15. $V' = V$
16. FOR each action a
17. $V^a = R^a_E + \gamma \sum_{E'} P^a_{E \cup E'} V'_{E'}$
18. $M = \max_a V^a$
19. $V = M_E + saveV_{\overline{E}}$
20. residual $= \|V_E - V'_E\|$
21. UNTIL stopping criterion met
22. $\pi = extractPolicy(M, \{V^a\})$
23. RETURN $(V, \pi, residual)$

LAO*($\{P^a\}, \{R^a\}, \gamma, S^0, h, threshold$)
24. $V = h$
25. $G = \emptyset$
26. $\pi = 0$
27. REPEAT
28. $(E, F, G) = policyExpansion(\pi, S^0, G)$
29. $(V, \pi, residual) = valueIteration(E, V)$
30. UNTIL ($F = \emptyset$) AND (residual $\leq$ threshold)
31. RETURN (π, V, E, G).

Table 1: Symbolic LAO* algorithm.

Policy expansion

In the policy expansion step of the algorithm, we perform reachability analysis to find the set of states F that are not in G (i.e., have not been "expanded" yet), but are reachable from the set of start states, S^0, by following the partial policy π_G. These states are on the "fringe" of the states visited by the best policy. We add them to G and to the set of states $E \subseteq G$ that are visited by the current partial policy. This is analogous to "expanding" states on the frontier of a search graph in heuristic search. Expanding a partial policy means that it will be defined for a larger set of states in the dynamic-programming step.

Symbolic reachability analysis using decision diagrams is widely used in VLSI design and verification. Our policy-expansion algorithm is similar to the traversal algorithms used for sequential verification, but is adapted to handle the more complex system dynamics of an MDP. The key operation in reachability analysis is computation of the *image* of a set of states, given a transition function. The image is the set of all possible successor states. To perform this operation, it is convenient to convert the ADD $P^a(\mathbf{X}, X')$ to a BDD $T^a(\mathbf{X}, X')$ that maps state transitions to a value of one if the transition has a non-zero probability, and otherwise zero. The image computation is faster using BDDs than ADDs. Mathematically, the image is computed using the relational-product operator, defined as follows:

$$Image_{\mathbf{X}'}(S, T^a) = \exists \mathbf{x} \left[T^a(\mathbf{X}, X') \wedge \chi_S(\mathbf{X}) \right].$$

The conjunction $T^a(\mathbf{X}, X') \wedge \chi_S(\mathbf{X})$ selects the set of valid transitions and the existential quantification extracts and unions the successor states together. Both the relational-product operator and symbolic traversal algorithms are well studied in the symbolic model checking literature, and we refer to that literature for details about how this is computed, for example, (Somenzi 1999).

The *image* operator returns a characteristic function over $\mathbf{X}'$ that represents the set of reachable states *after* an action is taken. The assignment in line 4 implicitly converts this characteristic function so that it is defined over $\mathbf{X}$, and represents the current set of states ready for the next expansion.

Because a policy is associated with a set of transition functions, one for each action, we need to invoke the appropriate transition function for each action when computing successor states under a policy. For this, it is useful to represent the partial policy π_G in another way. We associate with each action a the set of states for which the action to take is a under the current policy, and call this set of states S^a_π. Note that $S^a_\pi \cap S^{a'}_\pi = \emptyset$ for $a \neq a'$, and $\cup_a S^a_\pi = G$. Given this alternative representation of the policy, line 4 computes the set of successor states following the current policy using the *image* operator.

Dynamic programming

The dynamic-programming step of LAO* is performed using a modified version of the SPUDD algorithm. The original SPUDD algorithm performs dynamic programming over the entire state space. We modify it to focus computation on reachable states, using the idea of masking. Masking lets us perform dynamic programming on a subset of the state space instead of the entire state space. The pseudocode in Table 1 assumes that dynamic programming is performed on E, the states visited by the best (partial) policy. This has been shown to lead to the best performance of LAO*, although a larger or smaller set of states can also be updated (Hansen & Zilberstein 2001). Note that all ADDs used in the dynamic-programming computation are masked to improve efficiency.

Because π_G is a partial policy, there can be states in E with successor states that are not in G, denoted E'. This is true until LAO* converges. In line 13, we identify these states so that we can do appropriate masking. To perform dynamic programming on the states in E, we assign admissible values to the "fringe" states in E', where these values come from the current value function. Note that the value function is initialized to an admissible heuristic evaluation function at the beginning of the algorithm.

With all components properly masked, we can perform dynamic programming using the SPUDD algorithm. This is

summarized in line 17. The full equation is

$$V^a(\mathbf{X}) = R^a_E(\mathbf{X}) + \gamma \sum_{E'} P^a_{E \cup E'}(\mathbf{X}, X') \cdot V'_{E'}(\mathbf{X}').$$

The masked ADDs R^a_E and $P^a_{E \cup E'}$ need to be computed only once for each call to $valueIteration()$ since they don't change between iterations. Note that the product $P^a_{E \cup E'} \cdot V'_{E'}$ is effectively defined over $E \cup E'$. After the summation over E', which is accomplished by existentially abstracting away all post-action variables, the resulting ADD is effectively defined over E only. As a result, V^a is effectively a masked ADD over E, and the maximum M at line 18 is also a masked ADD over E.

The residual in line 20 can be computed by finding the largest absolute value of the ADD $(V_E - V'_E)$. We use the masking subscript here to emphasize that the residual is computed only for states in the set E. The masking operation can actually be avoided here since at this step, $V_E = M$, which is computed in line 18, and V'_E is the M from the previous iteration.

Dynamic programming is the most expensive step of LAO*, and it is usually not efficient to run it until convergence each time this step is performed. Often a single iteration gives the best performance. After performing value iteration, we extract a policy in line 22 by comparing M against the action value function V^a (breaking ties arbitrarily):

$$\forall s \in E \;\; \pi(s) = a \;\text{ if }\; M(s) = V^a(s).$$

The symbolic LAO* algorithm returns a value function V and a policy π, together with the set of states E that are visited by the policy, and the set of states G that have been "expanded" by LAO*.

Convergence test

At the beginning of LAO*, the value function V is initialized to the admissible heuristic h that overestimates the optimal value function. Each time the value iteration is performed, it starts with the current values of V. Hansen and Zilberstein (2001) show that these values decrease monotonically in the course of the algorithm; are always admissible; and converge arbitrarily close to optimal. LAO* converges to an optimal or ϵ-optimal policy when two conditions are met: (1) its current policy does not have any unexpanded states, and (2) the error bound of the policy is less than some predetermined threshold. Like other heuristic search algorithms, LAO* can find an optimal solution without visiting the entire state space. The convergence proofs for the original LAO* algorithm carry over in a straightforward way to symbolic LAO*.

Experimental results

Table 2 compares the performance of symbolic LAO* and SPUDD on the factory examples (f to f6) used by Hoey *et al.*(1999) to test the performance of SPUDD, as well as additional test examples (r1 to r4) that correspond to randomly-generated MDPs. Because the performance of LAO* depends on the starting state, the results for LAO* are an average for 50 random starting states. Experiments were performed on a Sun UltraSPARC II with a 300MHz processor and 2 gigabytes of memory.

We compared symbolic LAO* and SPUDD on additional examples (r1 to r4) because many of the state variables in the factory examples (f to f6) represent resources that cannot be affected by any action. As a result, we found that only a small number of states are reachable from any starting state. The additional examples are structured so that every state variable can be changed by some action. As a result, for examples r1 to r4, all the random starting states we generated can reach the entire state space.

To create an admissible heuristic function, we performed ten iterations of an approximate value iteration algorithm similar to APRICODD (St-Aubin *et al.* 2000). Value iteration is started with an initial admissible value function created by assuming the maximum reward is received each step, and it improves this value function each iteration. These first few iterations are very fast because the ADD representing the value function is compact, especially when approximation is used. The time used to compute the heuristic for these experiments is between 2% and 8% of the running time of SPUDD on these examples.

LAO* achieves its efficiency by focusing computation on a subset of the state space. The column labelled $|E|$ shows the average number of states visited by an optimal policy, beginning from a random start state. Clearly, the factory examples have an unusual structure, since an optimal policy for these examples visits very few states. The numbers are much larger for the random MDPs, although they still show that an optimal policy visits a small part of the state space. The column labeled $reach$ shows the average number of states that can be reached from the starting state, by following any policy. The column labelled $|G|$ is important because it shows the number of states "expanded" by LAO*. These are states for which a backup is performed at some point in the algorithm, and this number depends on the quality of the heuristic. The better the heuristic, the fewer states need to be expanded before finding an optimal policy. The gap between $|E|$ and $reach$ reflects the potential for increased efficiency using heuristic search, instead of simple reachability analysis.

The columns labeled "nodes" and "leaves", under LAO* and SPUDD respectively, compare the size of the final value function returned by LAO* and SPUDD. give the number of nodes in the final value function ADD returned by LAO* and SPUDD. The columns under "nodes" gives the number of nodes in the respective value function ADDs, and the columns under "leaves" give the number of leaves. Because LAO* focuses computation on a subset of the state space, it finds a much more compact solution (which translates into increased efficiency).

The last four columns compare the running times of LAO* and SPUDD. The total running time of LAO* is broken down into two parts; the column "expand" shows the average time for policy expansion and the column "DP" shows the average time for value iteration. These results show that value iteration consumes most of the running time. This is in keeping with a similar observation about the original LAO* algorithm for flat state spaces. The time for value iteration includes the time for masking. For this set of examples, masking takes between 0.5% and 21% of the running

Example			Reachability Results			Size Results				Timing Results			
			LAO*			LAO*		SPUDD		LAO*			SPUDD
	$\lvert S \rvert$	$\lvert A \rvert$	$\lvert E \rvert$	$\lvert G \rvert$	reach	nodes	leaves	nodes	leaves	expand	DP	total	total
f	2^{17}	14	5.4	104.7	190.4	55.0	5.4	1220	246	0.27	6.4	6.7	34.5
f0	2^{19}	14	5.3	61.9	131.7	61.0	5.2	1597	246	0.15	3.5	3.7	46.2
f1	2^{21}	14	4.1	54.1	106.7	54.3	4.1	3101	327	0.13	3.6	3.7	101.6
f2	2^{22}	14	4.0	65.6	124.5	53.4	3.9	3101	327	0.22	4.1	4.4	105.0
f3	2^{25}	15	4.3	59.3	135.9	73.8	4.4	9215	357	0.16	4.7	4.9	289.1
f4	2^{28}	15	4.2	49.4	124.9	78.2	4.1	22170	527	0.14	5.0	5.2	645.3
f5	2^{31}	18	4.8	218.4	508.6	83.0	4.1	44869	1515	1.15	35.9	37.3	1524.2
f6	2^{35}	23	9.2	1418.7	2385.6	106.4	4.5	169207	3992	13.52	771.5	792.6	7479.5
r1	2^{15}	20	134.3	413.7	2^{15}	65.5	11.5	1288	406	0.24	16.8	17.1	539.0
r2	2^{20}	25	3014.3	3281.1	2^{20}	181.4	19.3	15758	4056	0.46	53.6	54.0	12774.3
r3	2^{20}	30	10442.8	33322.6	2^{20}	6240.2	2190.4	9902	4594	57.71	1678.5	1738.1	10891.7
r4	2^{35}	30	383.9	383.9	2^{35}	77.4	3.5	NA	NA	0.05	7.2	7.4	> 20hr

Table 2: Performance comparison of LAO* and SPUDD.

time of value iteration. The final two columns show that the time it takes LAO* to solve a problem, given a specific starting state, is much less than the time it takes SPUDD to solve the problem. The running time of LAO* is correlated with $|G|$, the number of states expanded during the search, which in turn is affected by the starting state, the reachability structure of the problem, and the accuracy of the heuristic function.

Related work

Use of symbolic model checking in heuristic search has been explored by Edelkamp and Reffel (1998), who describe a symbolic generalization of A* that can solve deterministic planning problems. They show that symbolic search guided by a heuristic significantly outperforms breadth-first symbolic search.

In motivation, our work is closely related to the framework of *structured reachability analysis*, which exploits reachability analysis in solving factored MDPs (Boutilier *et al.* 1998). However, there are important differences. The symbolic model-checking techniques we use differ from the approach used in that work, which is derived from GRAPHPLAN. More importantly, their concept of reachability analysis is weaker than the approach adopted here. In their framework, states are considered irrelevant if they cannot be reached from the start state by following *any policy*. By contrast, our approach considers states irrelevant if they cannot be reached from the start state by following *an optimal policy*. To recognize states that cannot be reached by following an optimal policy, our algorithm gradually expands a partial policy, guided by an admissible heuristic. Use of an admissible heuristic to limit the search space is characteristic of heuristic search, in contrast to simple reachability analysis. As Table 2 shows, LAO* evaluates much less of the state space than simple reachability analysis. The better the heuristic, the smaller the number of states it examines.

Conclusion

We have described a symbolic generalization of LAO* that solves factored MDPs using heuristic search. The algorithm improves on the efficiency of dynamic programming by using an admissible heuristic to focus computation on the reachable parts of the state space. The stronger the heuristic, the greater the focus and the more efficient a planner based on this approach. An important topic not explored in this paper is how to design good admissible heuristics for factored MDPs. We will address this question in future work.

Acknowledgments Support for this work was provided in part by NSF grant IIS-9984952 and by NASA grant NAG-2-1463.

References

Barto, A.; Bradtke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72:81–138.

Boutilier, C.; Brafman, R. I.; and Geib, C. 1998. Structured reachability analysis for Markov decision processes. In *Proceedings of the 14th International Conference on Uncertainty in Artificial Intelligence (UAI-98).*

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research* 11:1–94.

Cimatti, M.; Roveri, M.; and Traverso, P. 1998. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 875 – 881.

Dean, T.; Kaelbling, L.; Kirman, J.; and Nicholson, A. 1995. Planning under time constraints in stochastic domains. *Artificial Intelligence* 76:35–74.

Edelkamp, S., and Reffel, F. 1998. OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)*, 81–92.

Hansen, E., and Zilberstein, S. 2001. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129:35–62.

Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Articial Intelligence*, 279–288.

Somenzi, F. 1999. Binary decision diagrams. In Broy, M., and Steinbruggen, R., eds., *Calculational System Design*, volume 173 of *NATO Science Series F: Computer and Systems Sciences*. IOS Press. 303–366.

St-Aubin, R.; Hoey, J.; and Boutilier, C. 2000. APRICODD: Approximate policy construction using decision diagrams. In *Proceedings of NIPS-2000*.

Planning Via Model Checking in Dynamic Temporal Domains: Exploiting Planning Representations

Robert P. Goldman and **David J. Musliner** and **Michael J. S. Pelican***
Honeywell Technology Center
3660 Technology Drive, Minneapolis, MN 55418, USA
{goldman, musliner, pelican}@@htc.honeywell.com

Introduction

We are developing autonomous, flexible control systems for mission-critical applications such as Uninhabited Aerial Vehicles (UAVs) and deep space probes. These applications require hybrid real-time control systems, capable of effectively managing both discrete and continuous controllable parameters to maintain system safety and achieve system goals. Using the CIRCA architecture for adaptive real-time control systems (Musliner, Durfee, & Shin 1993; 1995; Musliner *et al.* 1999), these controllers are synthesized *automatically* and dynamically, on-line, *while the platform is operating*. Unlike many other AI planning systems, CIRCA's automatically-generated control plans have strong temporal semantics and provide safety guarantees, ensuring that the controlled system will avoid all forms of mission-critical failure.

To support on-line reconfiguration, CIRCA uses concurrently-operating planning (controller synthesis) and control (plan-execution) subsystems. The Controller Synthesis Module (CSM) uses models of the world (plant and environment) to automatically synthesize hard real-time safety-preserving controllers (plans). Concurrently a separate Real-Time Subsystem (RTS) executes the controllers, enforcing response time guarantees. The concurrent operation means that the computationally expensive methods used by the CSM will not violate the tight timing requirements of the controllers.

At the last AIPS workshop on model-theoretic approaches to planning, we discussed how the CIRCA CSM used a model-checker to plan CIRCA's hard real-time controllers (Goldman, Musliner, & Pelican 2000). In this approach, our forward (Musliner, Durfee, & Shin 1995) or divide-and-conquer (Goldman *et al.* 1997) planning search used an external model-checking program[1] to verify the correctness of the plan. The verifier was used incrementally on plan fragments, after each decision during plan construction.

We found that the verifier was the primary bottleneck in this architecture. Accordingly, over the past two years, our efforts have concentrated on improving the performance of the verification component, and on "opening up" communication between the verification and planning components. To improve the performance of CIRCA plan verification, we have built a new, CIRCA-Specific Verifier (CSV). The CSV exploits features of the CIRCA action representation and execution semantics to significantly simplify the verification process and improve verifier performance; by two orders of magnitude for large examples. This paper will primarily concern itself with the implemented CSV, which has proven itself in practice.

We start by introducing the CIRCA CSM and its action representation, a STRIPS-style notation augmented with timing information. Then we outline the CSM planning algorithm, pointing out the role played by timed automaton verification. Next we explain how to formulate the execution semantics of the CIRCA model as a construction of sets of timed automata. The timed automaton model provides the semantics, but does not provide a practical approach for verification. We describe methods for model-checking that exploit CIRCA's implicit, transition-based, state space representation. We conclude with a comparison to related work in controller synthesis and AI planning and future directions in our work.

CIRCA

The CIRCA architecture is intended to provide intelligent control to autonomously-operating systems.[2] To do this, CIRCA must operate at multiple time scales. At the coarsest scale, CIRCA must be able to reason about the profile of a mission as a whole. For example, if CIRCA is operating an Uninhabited Combat Aerial Vehicle (UCAV), its mission-level planning must be able to reason about issues like fuel use and navigation to its goal. At a lower level, CIRCA must have a controller that is able to react to threats and opportunities that arise in its immediate environment. For example, when targeted by enemy radar, the CIRCA-controlled UCAV must carry out countermeasures (e.g., release chaff) and initiate evasive maneuvers. Furthermore, CIRCA must

*This material is based upon work supported by DARPA/ITO and the Air Force Research Laboratory under Contract No. F30602-00-C-0017.

[1]Initially KRONOS. (Yovine 1997)

[2]CIRCA has been applied to real-time planning and control problems in several domains including mobile robotics, simulated autonomous aircraft, space probe challenge problems (Musliner & Goldman 1997) and controlling a fixed-wing model aircraft (Atkins *et al.* 1998).

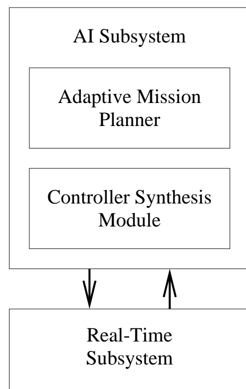


Figure 1: The CIRCA architecture combines AI planning and verification with real-time execution.

guarantee that these reactions will be taken in time. It is not enough to *eventually* release chaff; CIRCA must inspect its environments for threats sufficiently often, and must react to those threats within specified time bounds.

CIRCA employs two strategies to manage this complex task. First, its mission planner decomposes the mission into more manageable subtasks that can be planned in detail. Second, CIRCA itself is decomposed into two concurrently-operating subsystems (see Fig. 1): an *AI Subsystem* (AIS) reasons about high-level problems that require powerful but potentially unbounded computation, while a separate *real-time subsystem* (RTS) reactively executes the AIS-generated plans and enforces guaranteed response times. The AIS contains the CSM, which is the focus of this paper, as well as the mission planner and some support modules that are not discussed here.

The Controller Synthesis Module (CSM) bridges mission-level planning and reactive control. It takes descriptions of a phase of a system mission and automatically synthesizes a set of reactions that maintain the system's safety and move it towards its goals. While the RTS is executing this set of reactions (controller), the CSM will be working to generate controllers for other phases of the mission.

The Controller Synthesis Module

CIRCA's CSM plans real-time reactive discrete controllers that guarantee system safety when run on CIRCA's Real-Time Subsystem (RTS). The CSM builds reactive discrete controllers that observe the system state and some features of its environment and take appropriate control actions. The CSM takes in a description of the processes in the system's environment, represented as a set of time-constrained transitions that modify world features. Discrete states of the system are modeled as sets of feature-value assignments.

CIRCA's transitions are similar to STRIPS actions, having preconditions and postconditions, but also have timing information. Furthermore, not all of the transitions are controllable. Fig. 2 shows several transitions taken from a problem where CIRCA is to control the Cassini spacecraft in Saturn Orbital Insertion (Gat 1996; Musliner & Goldman 1997). This figure also includes the initial state description.

The CSM reasons about transitions of three types:

Action transitions represent actions performed by the RTS. These parallel the operators of a conventional planning system. Associated with each action is a worst case execution time, an *upper bound* on the delay before the action occurs.

Temporal transitions represent uncontrollable processes, some of which may need to be preempted. See the following section for the definition of "preemption" in this context. Associated with each temporal transition is a *lower bound* on its delay. Transitions whose lower bound is zero are referred to as "events," and are handled specially for efficiency reasons.

Reliable temporal transitions represent continuous processes that may need to be employed by the CIRCA agent. For example, when CIRCA turns on an Inertial Reference Unit it initiates the process of warming up that equipment; the process will complete after some delay. Reliable temporal transitions have both upper and lower bounds on their delays.

The use of planner-style transition representations significantly differentiates the CIRCA CSM from discrete and timed controller synthesis approaches developed in control theory and theoretical computer science. As in conventional planning, we find that vast sub-spaces of the state space are unreachable, either because of the control regime, or because of consistency constraints. The effect is only made greater when we must incorporate reasoning about time into the planning problem. The use of an implicit representation, together with a constructive search algorithm, allow us to avoid enumerating the full state space. The transition-centered representation allows us to conveniently represent processes that extend over multiple states. For example, a single transition (e.g., warming up a piece of equipment) may be extended over multiple discrete states. A similar representational convenience is often achieved by multiplying together many automata, but expanding the product construction restores the state explosion. later in this paper we show how the transition-based implicit representation can be exploited in a verifier.

CSM Algorithm

The CIRCA planning problem can be posed as *choosing a control action for each reachable discrete state (feature-value assignment) of the system.* Note that this controller synthesis problem is simpler than the general problem of synthesizing controllers for timed automata. In particular, CIRCA's controllers are memoryless and cannot reference clocks. This restriction has two advantages: first, it makes the synthesis problem easier and second, it allows us to ensure that the controllers we generate are actually realizable in the RTS.

Since the CSM focuses on generating *safe* controllers, a critical issue is making failure states unreachable. In controller synthesis, this is done by the process we refer to as *preemption*. A transition t is preempted in a state s iff some other transition t' from s must occur before t could possibly occur. The CSM achieves preemption by choosing a control

```
;; Turning on an Inertial Reference Unit (IRU).
ACTION start_IRU1_warm_up
   PRECONDITIONS: '((IRU1 off))
   POSTCONDITIONS: '((IRU1 warming))
   DELAY: <= 1

;; The process of the IRU warming.
RELIABLE-TEMPORAL warm_up_IRU1
   PRECONDITIONS: '((IRU1 warming))
   POSTCONDITIONS: '((IRU1 on))
   DELAY: [45 90]

;; Sometimes the IRUs break without warning.
EVENT IRU1_fails
   PRECONDITIONS: '((IRU1 on))
   POSTCONDITIONS: '((IRU1 broken))

;; If engine is burning while active IRU
;; breaks, we must quickly fix problem before
;; the spacecraft gets too far out of control.
TEMPORAL fail_if_burn_with_broken_IRU1
   PRECONDITIONS: '((engine on)(active_IRU IRU1)
                    (IRU1 broken))
   POSTCONDITIONS: '((failure T))
   DELAY: >= 5
```

```
    (failure F)
    (engine off)
    (IRU1 off)
    (IRU2 off)
    (active_IRU none)
```

Initial State Description

Figure 2: Example transition descriptions and initial state description given to CIRCA CSM.

action that is fast enough that it is guaranteed to occur before the transition to be preempted.[3]

The controller synthesis algorithm is as follows:

1. Choose a state from the set of reachable states (at the start of controller synthesis, only the initial state(s) is(are) reachable).
2. For each uncontrollable transition enabled in this state, choose whether or not to preempt it. Transitions that lead to failure states *must* be preempted.
3. Choose a control action or `no-op` for that state.
4. Invoke the verifier to confirm that the (partial) controller is safe.
5. If the controller is *not* safe, use information from the verifier to direct backtracking.
6. If the controller *is* safe, recompute the set of reachable states.
7. If there are no unplanned reachable states (reachable states for which a control action has not been chosen), terminate successfully.
8. If some unplanned reachable states remain, loop to step 1.

During the course of the search algorithm, the CSM will use the verifier module after each assignment of a control action (see step 4). This means that the verifier will be invoked before the controller is complete. At such points we use the verifier as a conservative heuristic by treating all unplanned states as if they are "safe havens." Unplanned states are treated as absorbing states of the system, and any verification traces that enter these states are regarded as successful. Note that this process converges to a sound and complete verification when the controller synthesis process is complete. When the verifier indicates that a controller is *unsafe*, the CSM will query it for a path to the distinguished failure state. The set of states along that path provides a set of candidate decisions to revise. The CSM uses backjumping (Gaschnig 1979) to exploit this information in its search.

Two remarks are worth making. The first is that the search described here is *not* made blindly. We use a domain-independent heuristic, providing limited lookahead, to direct the search. We do not have space to describe that heuristic here; it is based on one developed for AI planning (McDermott 1999). Without heuristic direction, even small synthesis problems can be too challenging. The second is that we have developed an alternative method of search that works by divide-and-conquer rather than reasoning forward (Goldman *et al.* 1997). For many problems, this supplies a substantial speed-up. Again, we do not have space to discuss this approach in depth here.

Modeling for Verification

The CSM algorithm described above operates entirely in the discrete domain of the timed problem. This ensures that the controllers may be easily implemented automatically. However, a path-dependent computation is required to determine how much time remains on a transition's delay when it applies to two or more states on a path. The CSM uses a timed automaton verification system to ensure that the controllers the CSM builds are safe. In this section, we discuss a formal model of the RTS, expressed in terms of timed automata. The following section describes how to reason about this model efficiently.

Execution Semantics

The controllers of the CIRCA RTS are not arbitrary pieces of software; they are intentionally very limited in their computational power. These limitations serve to make controller synthesis computationally efficient and make it simpler to build an RTS that provides timing guarantees. The controller generated by the CSM is compiled into a set of *Test-Action Pairs* (TAPs) to be run by the RTS. Each TAP has a boolean test expression that distinguishes between states where a particular action is and is not to be executed. Note that these test expressions do not have access to any clocks. A sample TAP for the Saturn Orbit Insertion domain is given in Fig. 3.

The set of TAPs that make up a controller are assembled into a loop and scheduled to meet all the TAP deadlines. Note that in order to meet deadlines, this loop may contain multiple copies of a single TAP. The deadlines are computed from the delays of the transitions that the control actions must preempt.

[3]Note that in some cases a reliable temporal transition, e.g., the warming up of the backup IRU, can be the transition that preempts a failure.

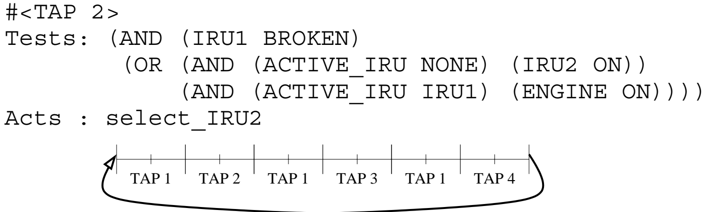


Figure 3: A sample Test-Action Pair and TAP schedule loop from the Saturn Orbit Insertion problem.

Timed Automata

Now that we have a sense of the execution semantics of CIRCA's RTS, we briefly review the modeling formalism, timed automata, before presenting the model itself. For those not concerned with the details of the semantic contruction, it will be sufficient to know that timed automata are nondeterministic finite automata augmented with timing information. This timing information dictates upper and lower bounds on state transitions. When reasoning about complex systems, or systems embedded in an environment, it is convenient to describe sets of automata, that can be assembled into a product automaton. The behaviors of such automata can be synchronized by *labels*, or *events*, on their edges. The formal details follow.

Definition 1 (Timed Automaton (Daws *et al.* 1996).) *A timed automaton A is a tuple $\langle \mathcal{S}, s^i, \mathcal{X}, \mathcal{L}, \mathcal{E}, \mathcal{I} \rangle$ where $\mathcal{S}$ is a finite set of locations; s^i is the initial location; $\mathcal{X}$ is a finite set of clocks; $\mathcal{L}$ is a finite set of labels; $\mathcal{E}$ is a finite set of edges; and $\mathcal{I}$ is the set of invariants. Each edge $e \in \mathcal{E}$ is a tuple (s, L, ψ, ρ, s') where $s \in \mathcal{S}$ is the source, $s' \in \mathcal{S}$ is the target, $L \subseteq \mathcal{L}$ are the labels, $\psi \in \Psi_{\mathcal{X}}$ is the* guard, *and $\rho \subseteq \mathcal{X}$ is a clock reset. Timing constraints ($\Psi_{\mathcal{X}}$) appear in guards and invariants and clock assignments. In our models, all clock constraints are of the form $c_i \leq k$ or $c_i > k$ for some clock c_i and integer constant k. Guards dictate when the model* may *follow an edge, invariants indicate when the model* must *leave a state. In our models, all clock resets re-assign the corresponding clock to zero; they are used to start and reset processes. The state of a timed automaton is a pair: $\langle s, C \rangle$. $s \in \mathcal{S}$ is a location and $C : \mathcal{X} \rightarrow \mathbf{Q} \geq 0$ is a clock valuation, that assigns a non-negative rational number to each clock.*

It often simplifies the representation of a complex system to treat it as a product of some number of simpler automata. The labels $\mathcal{L}$ are used to synchronize edges in different automata when creating their product.

Definition 2 (Product Automaton) *Given two automata A_1 and A_2, $A_1 = \langle \mathcal{S}_1, s_1^i, \mathcal{X}_1, \mathcal{L}_1, \mathcal{E}_1, \mathcal{I}_1 \rangle$ and $A_2 = \langle \mathcal{S}_2, s_2^i, \mathcal{X}_2, \mathcal{L}_2, \mathcal{E}_2, \mathcal{I}_2 \rangle$, their product A_p is $\langle \mathcal{S}_1 \times \mathcal{S}_2, s_p^i, \mathcal{X}_1 \cup \mathcal{X}_2, \mathcal{L}_1 \cup \mathcal{L}_2, \mathcal{E}_p, \mathcal{I}_p \rangle$, where $s_p^i = (s_1^i, s_2^i)$ and $\mathcal{I}(s_1, s_2) = \mathcal{I}(s_1) \wedge \mathcal{I}(s_2)$. The edges are defined by:*

1. *for $l \in \mathcal{L}_1 \cap \mathcal{L}_2$, for every $\langle s_1, l, \psi_1, \rho_1, s_1' \rangle \in \mathcal{E}_1$, and $\langle s_2, l, \psi_2, \rho_2, s_2' \rangle \in \mathcal{E}_2$, $\mathcal{E}_p$ contains $\langle (s_1, s_2), l, \psi_1 \cup \psi_2, \rho_1 \cup \rho_2, (s_1', s_2') \rangle$.*

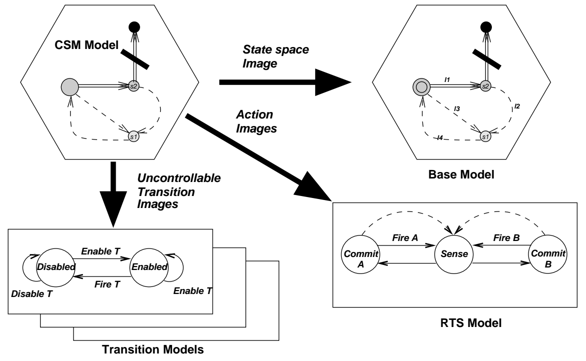


Figure 4: The verifier model and its relation to the CSM model.

2. *for $l \in \mathcal{L}_1 \setminus \mathcal{L}_2$, for $\langle s_1, l, \psi_1, \rho_1, s_1' \rangle \in \mathcal{E}_1$ and $s_2 \in \mathcal{S}_2$, $\mathcal{E}_p$ contains $\langle (s_1, s_2), l, \psi_1, \rho_1, (s_1', s_2) \rangle$. Likewise for $l \in \mathcal{L}_2 \setminus \mathcal{L}_1$.*

Modeling CIRCA with Timed Automata

We give the semantics of CSM models in terms of sets of interacting timed automata. The use of multiple automata permits us to accurately and elegantly capture the interaction of multiple, simultaneously operating processes. In this construction, there is one *base machine*, the locations of which correspond to the states of the CSM model. The base machine captures the overall state of the system and its environment. The base machine interacts with a number of *uncontrollable transition machines*, one for each of the uncontrollable transitions, and an automaton that represents the CIRCA agent's RTS. This construction is summarized in Fig. 4.

The effects of the various transitions are captured by the labels on the edges of the various machines. Labels synchronizing transition automata and the RTS with the base machine ensure that the base machine state reflects the effect of the transitions. Labels synchronizing the transition automata and the base machine ensure that the state of the transition machines accurately indicate whether or not a given process is enabled in a particular system state. Finally, labels synchronizing the base machine with the RTS machine cause the RTS machine to correctly model the planner's assignment of control actions to world states.

In the remaining parts of this section, we give the details of the semantic construction. The casual reader can skip this material. The important thing to know is that the semantics, while offering a basis for plan verification, is not suitable for direct use. The following section will describe how our CIRCA-Specific Verifier directly interprets the semantics, bypassing the product construction, to achieve computational savings.

The starting point of the translation is the CIRCA plan-graph, constructed by the CIRCA CSM:

Definition 3 (Plan Graph)
$\mathcal{P} = \langle S, E, \vec{F}, \vec{V}, \phi, I, T, \iota, \eta, p, \pi \rangle$ *where*

1. *S is a set of states.*

2. *E is a set of edges.*
3. *$\vec{F} = [f_0 ... f_m]$ is a vector of features (in a purely propositional domain, these will be propositions).*
4. *$\vec{V} = [\mathcal{V}_0 ... \mathcal{V}_m]$ is a corresponding vector of sets of values ($\mathcal{V}_i = \{v_{i0} ... v_{ik_i}\}$) that each feature can take on.*
5. *$\phi : S \mapsto \vec{V}$ is a function mapping from states to unique vectors of value assignments.*
6. *$I \subset S$ is a distinguished subset of initial states.*
7. *$T = U \cup A$ is the set of transitions, made up of an uncontrollable (U) subset, the temporals and reliable temporals, and a controllable (A) subset, the actions. Each transition, t, has an associated delay (Δ_t) lower and upper bound: $lb(\Delta_t)$ and $ub(\Delta_t)$. For temporals $ub(\Delta_t) = \infty$, for events $lb(\Delta_t) = 0, ub(\Delta_t) = \infty$.*
8. *ι is an interpretation of the edges: $\iota : E \mapsto T$.*
9. *$\eta : S \mapsto 2^T$ is the* enabled *relationship — the set of transitions enabled in a particular state.*
10. *$p : S \mapsto A \cup \epsilon$ (where ϵ is the "action" of doing nothing) is the actions that the CSM has planned. Note that p will generally be a partial function.*
11. *$\pi : S \mapsto 2^U$ is a set of preemptions the CSM expects.*

For every CIRCA plan graph, $\mathcal{P}$, we construct a timed automaton model, $\theta(\mathcal{P})$. $\theta(\mathcal{P})$ is the product of a number of individual automata. There is one automaton, which we call the *base model*, that models the feature structure of the domain. There is an *RTS model* that models the actions of the CIRCA agent. Finally, for every uncontrollable transition, there is a separate timed automaton modeling that process. Proper synchronization ensures that the base machine state reflects the effect of the transitions and that the state of the other automata accurately indicate whether or not a given process will (may) be underway.

Definition 4 (Translation of CIRCA Plan Graph) *$\theta(\mathcal{P}) = \beta(\mathcal{P}) \times \rho(\mathcal{P}) \times \prod_{u \in U(\mathcal{P})} \upsilon(u)$ where $\beta(\mathcal{P})$ is the base model; $\rho(\mathcal{P})$ is the RTS model; and $\upsilon(u)$ is the automaton modeling the process that corresponds to uncontrollable transition u.*

Definition 5 (Base model) *$\beta(\mathcal{P}) = \langle \theta(S), \{l^0\}, \emptyset, \Sigma(\mathcal{P}), \theta_E(\mathcal{P}), I_\top \rangle$ where:*

1. *$\theta(S) = \{\theta(s) \mid s \in S\} \cup \{l^{\mathcal{F}}, l^0\}$ is the image under θ of the state set of $\mathcal{P}$. This image contains a location for each state in $\mathcal{P}$, as well as a distinguished failure location, $l^{\mathcal{F}}$, and initial location, l^0.*
2. *$\Sigma(\mathcal{P})$ is the label set; it is given as Definition 6.*
3. *$\theta_E(\mathcal{P})$ is the edge set of the base model. It is given as Definition 7.*

Note that there are no clocks in the base machine; all timing constraints will be handled by other automata in the composite model. Thus, the invariant for each state in this model is simply $\top$. We have notated this vacuous invariant as $I_\top$. Similarly, all of the edges have a vacuous guard. The labels of the translation model ensure that the other component automata synchronize correctly.

Definition 6 (Label set for $\theta(\mathcal{P})$)

$$\begin{aligned}\Sigma(\mathcal{P}) \quad = \quad & \{\mathbf{e}_u, \mathbf{d}_u, \mathbf{f}_u \mid u \in U\} \cup & (1)\\ & \{\mathbf{c}_a, \mathbf{f}_a \mid a \in A\} \cup & (2)\\ & \{\mathbf{r}\} & (3)\end{aligned}$$

The symbols in (1) are used to synchronize the automata for uncontrollable transitions with the base model. The symbols in (2) together with the distinguished reset symbol **r** are used to synchronize the automaton modeling the RTS with the base model. The base model edge set, $\theta_E(\mathcal{P})$, captures the *effect* on the agent and environment of the various transitions.

Definition 7 (Base model edge set) *$\theta_E(\mathcal{P})$ is made up of the following subsets of edges. The clock resets of these transitions are all $\emptyset$, and the guards are all $\top$, so we have omitted them.*

$$\begin{array}{ll}(1) & \{\langle l^0, \theta_\sigma(\mathit{init}, s), \theta(s)\rangle \mid s \in I\}\\ (2) & \{\langle \theta(s), \{\mathbf{f}_u\}, l^{\mathcal{F}}\rangle \mid s \in S, u \in \pi(s)\}\\ (3) & \{\langle \theta(s), \{\mathbf{f}_u\} \cup \theta_\sigma(u, s'), \theta(s')\rangle \mid \\ & \quad s \in S, u \notin \pi(s), s' \in u(s)\}\\ (4) & \{\langle \theta(s), \{\mathbf{c}_a\}, s\rangle \mid s \in S, a = p(s)\}\\ (5) & \{\langle \theta(s), \{\mathbf{f}_a\} \cup \theta_\sigma(a, s'), \theta(a(s))\rangle \mid \\ & \quad s \in S, a \in \eta(s), s' \in a(s)\}\\ (6) & \{\langle \theta(s), \{\mathbf{f}_a\}, l^{\mathcal{F}}\rangle \mid s \in S, a \notin \eta(s)\}\end{array}$$

Edge set (1) is merely a set of initialization edges, that carry the base model from its distinguished single initial location to the image of each of the initial states of $\mathcal{P}$. (2) takes the base model to its distinguished failure location, $l^{\mathcal{F}}$, when a preemption fails. (3) captures the effects of the uncontrollable transitions the CSM didn't preempt. (4) synchronizes with the RTS transitions that capture the RTS committing to execute a particular action (i.e., the test part of the TAP). (5) captures the effects of a successfully-executed action. (6) captures a failure due to a race condition. Event sets $\theta_\sigma(t, s)$ are used to capture the effects on the various processes of going to s by means of t.

Definition 8

$$\theta_\sigma(t, s) = \{\mathbf{e}_u \mid u \in \eta(s)\} \cup \{\mathbf{d}_u \mid u \neq t \wedge u \notin \eta(s)\} \cup \{\mathbf{r}\}$$

The symbol set $\theta_\sigma(t, s)$ contains an enable symbol for each u enabled in s, and a disable symbol for each u not enabled in s. The addition of the symbol **r** ensures that the RTS machine will "notice" the state transition.

There will be one automaton, $\upsilon(u)$ for every uncontrollable transition, u. Each such model will have two states, enabled, e_u, and disabled, d_u, and transitions for enabling, disabling, and firing: $\mathbf{e}_u, \mathbf{d}_u$, and $\mathbf{f}_u$, respectively (see Fig. 4). It will also have a clock, c_u, and the guards and invariants will be derived from the timing constraints on u:

Definition 9 (Uncontrollable Transition Automata)

$$\upsilon(u) = \langle \{e_u, d_u\}, d_u, \{c_u\}, \{\mathbf{e}_u, \mathbf{d}_u, \mathbf{f}_u\}, E(\upsilon(u)), I\rangle$$

$$\begin{aligned}E(\upsilon(u)) \quad = \quad \{ & \langle d_u, \{\mathbf{d}_u\}, \top, \emptyset, d_u\rangle,\\ & \langle d_u, \{\mathbf{e}_u\}, \top, c_u := 0, e_u\rangle,\\ & \langle e_u, \{\mathbf{e}_u\}, \top, \emptyset, e_u\rangle, \langle e_u, \{\mathbf{d}_u\}, \top, \emptyset, d_u\rangle,\\ & \langle e_u, \{\mathbf{f}_u\}, c_u \geq \mathit{lb}(\Delta_u), \emptyset, d_u\rangle \ \}\end{aligned}$$

$$I(e_u) = c_u \leq \mathit{ub}(\Delta_u) \text{ and } I(d_u) = \top$$

The model of the RTS, ρ, contains all of the planned actions in a single automaton. Execution of each planned action is captured as a two stage process: first the process of committing to the action (going to the state c_a), and then the action's execution (returning to s_0 through transition $\mathbf{f}_a$).

Definition 10 (RTS Model)

$$\begin{aligned}
\rho \;=\; & \langle \{s_0\} \cup \{c_a \mid a \in p\}, s_0, \{c_{RTS}\}, \\
& \{\mathbf{r}\} \cup \{\mathbf{c}_a, \mathbf{f}_a \mid a \in p\}, \\
& \{\langle s_0, \{\mathbf{c}_a\}, c_{RTS} \leq 0, c_{RTS}{:=}0, c_a\rangle, \\
& \;\langle c_a, \{\mathbf{f}_a\}, c_{RTS} \geq ub(\Delta_a), c_{RTS}{:=}0, s_0\rangle, \\
& \;\langle c_a, \{\mathbf{r}\}, c_{RTS} < ub(\Delta_a), c_{RTS}{:=}0, s_0\rangle, \\
& \;\langle c_a, \{\mathbf{r}\}, c_{RTS} < ub(\Delta_a), \emptyset, c_a\rangle \mid a \in p\} \\
& \{I(c_a) = c_{RTS} \leq ub(\Delta_a), I(s_0) = c_{RTS} \leq 0\}\rangle
\end{aligned}$$

Note that the RTS model is not a deterministic timed automaton, because of the final two transitions in Definition 10. These two transitions capture what happens when there is an uncontrollable transition, $s \xrightarrow{u} s'$, that changes the world state, before the RTS has executed the reaction planned for state s. One of two things will happen:

1. The uncontrolled transition happens before the TAP's test has executed, and the TAP is never triggered. Effectively, the RTS doesn't "know" it was ever in state s.
2. The uncontrolled transition occurs after the TAP for state s has started running. In this case, the RTS will have "decided to" execute the action for state s, but the action will actually take place in state s' (or possibly even a successor state, if the environment is sufficiently fast-moving).

The latter case is a hazard, and the verifier should check for it. See the discussion of safety violations below.

The RTS is actually a deterministic piece of software. The nondeterminism here captures our uncertainty about the exact course of execution of the TAP loop, and the exact separation between TAPs that provide a particular reaction. We will not know the exact separation until the controller synthesis process is complete, and the TAPs have been compiled and scheduled. Therefore, we make the conservative assumption that a hazard might occur.

There are two classes of safety violations the verifier must detect. The first is a failure to successfully preempt some nonvolitional transition. This case is caught by transitions (2) of Definition 7. The second is a race condition: here the failure is to plan a for state s but not complete it before an uncontrolled process brings the world to another state, s', that does not satisfy the preconditions of a. The latter case is caught by transitions (6) of Definition 7.[4]

Exploiting the Model in Verification

A direct implementation of the above model will suffer a state space explosion. To overcome this, we have built a CIRCA-Specific Verifier (CSV) able to exploit CIRCA's implicit state-space representation. The CSV constructs its timed automata, *both the individual automata and their product*, in the process of computing reachability. This on-the-fly computation relies on the factored representation of the discrete state space and on the limitations of CIRCA's RTS.

The efficiency gains from our factored state representation come in the computation of successor states. A naive implementation of the search would compute all of the locations (distinct discrete states) of the timed automaton up front, but many of those might be unreachable. We compute the product automaton lazily, rather than before doing the reachability search, thus constructing only reachable states.

The individual automata, as well as their product, are computed on-the-fly. The timed automaton formalism permits multiple automata to synchronize in arbitrary ways. However, CIRCA automata synchronize in only limited ways. There will be only one "primary transition" that occurs in any state of the CIRCA product automaton: either a controlled transition that is part of the RTS automaton, or a single uncontrolled transition. Thus we may dispense with component transitions and their labels.

The transitions that synchronize with the primary transition are of three types:

1. updates to the world automaton, recording the effect (the postconditions) of the primary jump on the discrete state of the world;
2. enabling and disabling jumps that set the state of uncontrolled transitions in the environment;
3. a jump that has the effect of activating the control action planned for the new state.

Accordingly, we can very efficiently implement a lazy successor generation for a set of states $S = \langle s, \mathbf{C}\rangle$, where s is a discrete state and $\mathbf{C}$ is a symbolic representation of a class of clock valuations, in our case a difference-bound matrix. When one needs to compute the successor locations for the location s, one need only compute a single outgoing edge for the RTS transition and make one outgoing edge for each uncontrollable transition.

Making the outgoing edges is a matter of (again lazily) building the successor locations and determining the clock resets for the edge. The clocks that must be reset are: (a) For each uncontrolled transition that is enabled in the successor location, but not enabled in the source location, s, add a clock reset for the corresponding transition; (b) If the action planned for the successor location is different from the action planned for the source location, reset the action clock. These computations are quite simple to make and much easier than computing the general product construction.

Our experimental results show that the CSV substantially improves performance over KRONOS (Yovine 1997) and also over a conventional model checker (denoted "RTA") that we built into CIRCA before developing the CSV. Table 1 contains comparison data between the conventional verifiers and the CSV, for two different search strategies. The problems are available at: `http://www.htc.honeywell.com/projects/ants/`; the precise scenario definitions can be found using the key in Table 3. The columns marked "forward," correspond

[4] Checking for the race condition is not fully implemented in our current version; its implementation is in progress as of this writing.

Scenario	Size	Forward			DAP		
		Kronos	RTA	CSV	Kronos	RTA	CSV
1	1920	9288	190	188	22431	483	417
2	72	6777	173	124	7070	385	309
3	100	4765	114	97	4399	783	385
4	560	5619	138	156	5599	366	288
5	3182592	∞	∞	∞	∞	∞	16278
6	40304	16983	762	568	506897	3035	1349
7	191232	258166	23030	25194	12919	4102	1833
8	191232	14637	652	533	436450	1157849	79855
9	991232	231769	21923	15474	∞	∞	2254
10	448512	∞	1063321	466631	∞	∞	5661
11	411136	∞	1064518	444657	∞	∞	5571
12	193536	37500	2585	1568	321626	3382	1626
13	129024	56732	3453	2933	77022	9958	1218
14	4592	16025	478	427	20220	1251	1036
15	7992	4680	183	176	75941	7672	5568
16	768	11535	426	337	13983	859	621
17	120	5730	100	368	5695	754	680
18	2880	16425	1349	1102	28922	2484	1669
19	192	6474	170	117	5715	331	308
20	768	9016	303	246	6870	564	416

Table 1: Comparison of run times with different search strategies (Forward and DAP), timed automaton verifier (RTA) versus CIRCA-Specific Verifier (CSV). Times are given in milliseconds.

to the algorithm described in this paper. The columns marked "DAP" correspond to the divide-and-conquer alternative (Goldman *et al.* 1997). The times, given in milliseconds, are for runs of the CSM on a Sun UltraSparc 10, SPARC v.9 processor, 440 MHz, with 1 gigabyte of RAM. An ∞ indicates a failure to find an automaton within a 20 minute time limit (i.e., $t > 1,200,000$).

To give a sense of the raw size of the problems, the "Size" column presents a worst-case bound on the number of *discrete* states for the final verification problem of each scenario. This value is computed by multiplying the number of possible CSM world model states (for the base model) times the number of transition model states ($2^{|U|}$) times the number of RTS model states ($|A| + 1$).

Using the forward search strategy, the CSV is faster on 16 out of 20 scenarios. Using DAP, the CSV is faster on all 20 trials. The probability of these occurring, if the CSV and the conventional verifier were equally likely to win on any given trial, is .0046 and .000019, respectively. Table 1 indicates a speed-up of two orders of magnitude on the larger scenarios, numbers 9-11, using DAP.

Table 2 shows the state space reductions achieved by exploiting the implicit representation. This table compares the total number of states visited by each verifier in the course of controller synthesis.

A few facts should be noted: A verifier will be run many times in the course of synthesizing a controller. To minimize this, a number of cheaper tests filter controller synthesis choices in advance of verification, in order to avoid verification search whenever possible. The comparison is only with KRONOS used *as a component of the CSM*, not KRONOS as a general verification tool. Finally, the computations done by KRONOS and RTA are of a special-purpose product model that is slightly simpler and *less* accurate than the CSV's model.

Related Work

Asarin, Maler, Pneuli and Sifakis (AMPS) (1995; 1995) independently developed a game-theoretic method of synthesizing real-time controllers. They view the problem as trying to "force a win" against the environment, by guaranteeing that all traces of system execution will pass through (avoid) a set of desirable (undesirable) states. Unlike the game-theoretic algorithms, the CSM algorithm works starting from an initial state and building forward by search. The game-theoretic algorithms, on the other hand, use a fixpoint operation to find a controllable subspace, starting from unsafe states (or other synthesis failures). Another difference is that the CSM algorithm heavily exploits its implicit state space representation. Because of these features, for many problems, the CSM algorithm is able to find a controller without visiting large portions of the state space.

The AMPS work stopped at the design of the algorithm and derivation of complexity bounds; to our knowledge it was not implemented. More recently, the AMPS algorithm has been implemented for the special case of automatically synthesizing schedulers based on Petri Net designs (Altisen *et al.* 1999), using the KRONOS model-checking program. Our work differs in being aimed at a different class of control problems, involving controlling devices in an active environment. We also differ in constructing purely reactive

Scenario	Forward RTA	Forward CSV	DAP RTA	DAP CSV
1	30	30	30	33
2	34	34	33	33
3	15	15	15	15
4	18	18	18	18
5	153147	229831	170325	3069
6	122	120	500	54
7	2826	5375	631	83
8	146	131	301165	24065
9	4361	4799	163133	259
10	219885	129329	184972	871
11	219885	129329	189184	871
12	585	513	509	99
13	685	675	1782	93
14	106	106	141	142
15	17	17	1054	1389
16	117	101	131	116
17	27	27	29	25
18	284	290	355	269
19	18	18	18	18
20	63	60	33	34

Table 2: Comparison of state spaces explored with different search strategies (Forward and DAP), timed automaton verifier (RTA) versus CIRCA-specific verifier (CSV). Units are verifier state objects, i.e., a location $\times$ a difference-bound matrix.

Scenario	Scenario name
1	DC
2	FORWARD-DC
3	NO-TIME
4	PREEMPT
5	PUMA
6	PUMA-NO-EMERG-EVENT
7	PUMA-NO-FALL
8	PUMA-NO-FALL-FAST-EMERG
9	PUMA-NO-TABLE-REP
10	PUMA-NO-UNKNOWN
11	PUMA-NO-UNKNOWN-FAST-PACKER
12	PUMA-NO-UNKNOWN-NO-EMERG-TTF
13	RT-SATURN-ORBIT-INSERTION
14	SATURN-ORBIT-INSERTION
15	TABLE
16	TIME-OUT
17	UCAV-FIRE-AT-TARGET
18	UCAV-RADAR-CHAFF
19	UCAV-RADAR-ONLY
20	UCAV-RADAR-ONLY2

Table 3: Scenario key. The scenario names can be used to find the corresponding scenario definitions at `http://www.htc.honeywell.com/projects/ants/`.

(memoryless) controllers.

Tripakis and Altisen (TA) (1999) have independently developed a controller synthesis algorithm for discrete and timed systems, that also uses forward search with on-the-fly generation of the state space. Note that on-the-fly synthesis has been part of the CIRCA system since its conception in the early 1990s (Musliner, Durfee, & Shin 1993; 1995). TA's on-line synthesis has some different features from ours. They allow for multiple control actions in a single state, and they allow the controller to consult clocks. TA's implicit representation of the state space is based on composition of automata, as opposed to our feature and transition approach. We hope to compare performance of CIRCA and a recent implementation of the TA algorithm (Tripakis 2002).

TA do not have a fully on-the-fly algorithm for timed controller synthesis. Their algorithm requires the initial computation of a quotient graph of the automata, in turn requiring a full enumeration of the discrete state space. The disadvantages of such an approach can be seen by considering the state space sizes of some examples, given in Table 1. We do not need to pre-enumerate the quotient, since we build only a clockless reactive controller and so can use the cruder time abstraction, which we compute on-the-fly. Note that this means that there are some controllers that TA (and AMPS) can find, that we cannot. However, clockless reactive controllers are easy to implement automatically, and this is not true of controllers that employ clocks. Also, and again because of the size of the state space, we use heuristic guidance in our state space search.

The "planning as model checking" (Giunchiglia & Traverso 1999) approach is similar to work on game-

theoretic controller synthesis, but limited to purely discrete systems.

Kabanza (1996)'s SIMPLAN is very similar to our CSM. However, SIMPLAN adopts a discrete time model and uses domain-specific heuristics. SIMPLAN incorporates time imposing a system-wide clock and progressing the controller one "tick" at a time. In control problems with widely varying time constants, this approach leads to state-space explosion; we have adopted region-based (equivalence class) model-checking techniques to minimize this state explosion.

Conclusions

In this paper, we have presented the CIRCA controller synthesis algorithm, provided a timed automaton model for CIRCA CSM problems, and shown how a CIRCA-Specific Verifier (CSV) algorithm can exploit the features of the model. The CSV shows dramatic speed-up over a general-purpose verification algorithm. While our model was developed for CIRCA, it is a general model for supervisory control of timed automata, and could readily be used in other applications.

Acknowledgments

This material is based upon work supported by DARPA/ITO and the Air Force Research Laboratory under Contract No. F30602-00-C-0017. Thanks to our anonymous reviewers for their suggestions.

References

Altisen, K.; Goessler, G.; Pnueli, A.; Sifakis, J.; S.Tripakis; and S.Yovine. 1999. A framework for scheduler synthesis. In *Proceedings of the 1999 IEEE Real-Time Systems Symposium (RTSS '99)*. Phoenix, AZ: IEEE Computer Society Press.

Asarin, E.; Maler, O.; and Pneuli, A. 1995. Symbolic controller synthesis for discrete and timed systems. In Antsaklis, P.; Kohn, W.; Nerode, A.; and Sastry, S., eds., *Proceedings of Hybrid Systems II*. Springer Verlag.

Atkins, E. M.; Miller, R. H.; VanPelt, T.; Shaw, K. D.; Ribbens, W. B.; Washabaugh, P. D.; and Bernstein, D. S. 1998. Solus: An autonomous aircraft for flight control and trajectory planning research. In *Proceedings of the American Control Conference (ACC)*, volume 2, 689–693.

Daws, C.; Olivero, A.; Tripakis, S.; and Yovine, S. 1996. The tool Kronos. In *Hybrid Systems III*.

Gaschnig, J. 1979. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University.

Gat, E. 1996. News from the trenches: An overview of unmanned spacecraft for AI. In Nourbakhsh, I., ed., *AAAI Technical Report SSS-96-04: Planning with Incomplete Information for Robot Problems*. American Association for Artificial Intelligence.

Giunchiglia, F., and Traverso, P. 1999. Planning as model-checking. In *Proceedings of ECP-99*. Springer Verlag.

Goldman, R. P.; Musliner, D. J.; Krebsbach, K. D.; and Boddy, M. S. 1997. Dynamic abstraction planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, 680–686. Menlo Park, CA: American Association for Artificial Intelligence.

Goldman, R. P.; Musliner, D. J.; and Pelican, M. S. 2000. Using model checking to plan hard real-time controllers. In *Proceedings of AIPS Workshop on Model-Theoretic Approaches to Planning*.

Kabanza, F. 1996. On the synthesis of situation control rules under exogenous events. In Baral, C., ed., *Theories of Action, Planning, and Robot Control: Bridging the Gap*, number WS-96-07, 86–94. AAAI Press.

Maler, O.; Pneuli, A.; and Sifakis, J. 1995. On the synthesis of discrete controllers for timed systems. In Mayr, E. W., and Puech, C., eds., *STACS 95: Theoretical Aspects of Computer Science*. Springer Verlag. 229–242.

McDermott, D. 1999. Using regression-match graphs to control search in planning. *Artificial Intelligence* 109(1 – 2):111–159.

Musliner, D. J., and Goldman, R. P. 1997. CIRCA and the Cassini Saturn orbit insertion: Solving a prepositioning problem. In *Working Notes of the NASA Workshop on Planning and Scheduling for Space*.

Musliner, D. J.; Goldman, R. P.; Pelican, M. J.; and Krebsbach, K. D. 1999. SA-CIRCA: Self-adaptive software for hard real time environments. *IEEE Intelligent Systems* 14(4):23–29.

Musliner, D. J.; Durfee, E. H.; and Shin, K. G. 1993. CIRCA: a cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man and Cybernetics* 23(6):1561–1574.

Musliner, D. J.; Durfee, E. H.; and Shin, K. G. 1995. World modeling for the dynamic construction of real-time control plans. *Artificial Intelligence* 74(1):83–127.

Tripakis, S., and Altisen, K. 1999. On-the-fly controller synthesis for discrete and dense-time systems. In Wing, J.; Woodcock, J.; and Davies, J., eds., *Formal Methods 1999*, volume I of *Lecture Notes in Computer Science*. Berlin: Springer Verlag. 233–252.

Tripakis, S. 2002. personal communication.

Yovine, S. 1997. Kronos: A verification tool for real-time sytems. In *Springer International Journal of Software Tools for Technology Transfer*, volume 1.

Partial State Progression: An Extension to the Bacchus-Kabanza Algorithm, with Applications to Prediction and MITL Consistency

P@trik Haslum
Department of Computer Science, Linköping University
pahas@ida.liu.se

Abstract

The Metric Interval Temporal Logic (MITL) progression algorithm invented by Bacchus and Kabanza (1996) is extended to work also for the case when the duration and content of states are only partially specified (constrained). This is motivated by an approach to prediction, but also leads to a new, tableaux style, algorithm for deciding the consistency of an MITL formula. The algorithm is not yet fully developed.

1. Introduction

Metric Interval Temporal Logic (MITL) (Alur, Feder, & Henzinger 1996), like LTL (Emerson 1990), is a tense-modal logic expressing properties of infinite sequences of propositional states. The logics differ in that state sequences in MITL models are *timed*: each state in the sequence has a starting and ending timepoint, which maps to an underlying real timeline, and modal operators in MITL are qualified with constraints on state durations.

For applications working with MITL, the progression algorithm of Bacchus and Kabanza (1996) is a powerful tool. The algorithm "pushes" the truth criteria for an MITL formula forwards over the states of a timed model, one at a time: input is an MITL formula ϕ, and a state q with duration $D(q)$, and the algorithm returns an MITL formula ϕ' such that ϕ is T in q *iff* ϕ' is T in the succeeding state.

Of state succeeding q, nothing needs to be known: the returned formula is a condition that can be checked for contradiction, for truth in the succeeding state when it becomes known, or further progressed. The progression algorithm as presented, however, requires all properties of the state q, in particular the duration $D(q)$, to be known.

Motivated by the use of MITL to represent knowledge in a predictive model, I have extended the progression algorithm to work with only partial knowledge about the duration of state q, expressed by a set of constraints on variables representing state starting and ending times. As it turns out, the further extension to working with only partial knowledge about the state q itself is fairly straightforward, and potentially applicable to the problem MITL consistency checking.

The next two sections introduce background material: first, the approach to prediction that motivated this line of inquiry, then in more detail MITL and Bacchus' and Kabanza's progression algorithm. Section *4* presents the extension to partial knowledge about state timing.

Section *5*, finally, describes the further extension to the progression algorithm and how it may be applied to MITL consistency. This is still work in progress: the consistency checking algorithm presented is sound, but not complete.

2. Background I: Prediction as a Knowledge Representation Problem

Prediction is a central component in many control and reasoning tasks, *e.g.* state estimation/diagnosis, interpretation of sensor data (most notably the problem of reidentification), tracking control and planning. All of prediction relies on *models*: it is only with the knowledge encoded in a model, whatever form it takes, that a predictive system can conclude anything stronger than a tautology.

In a previous paper (Haslum 2001), I argued that there are for any application involving prediction many possible "model designs", by which I mean broadly the ontology, representation, acquisistion and computational methods associated with a model, and for the comparative investigation of different alternatives. As a case study, I have designed two different solutions to a prediction problem encountered in the WITAS UAV project[1]: one is a discrete event model and uses a schema-like representation of "normality", while the other is based on a Markov process. Both models use continuous time.

The role of MITL is in the first model design, as a language to formulate *expectations*, expressing beliefs about what will hold true, over time, in the normal case. Expectations are arranged in a heirarchy reflecting the strength of belief. A possible future development is represented by a timed sequence of events, finite since the prediction horizon is bounded. By checking which formulas of the hierarchy, if any, the development necessarily violates, it is assigned a "normality rating", representing a measure of the percieved relative likelihood of that development occuring.

[1]The WITAS project studies architectures and techniques for intelligent autonomous systems, centered around an unmanned aerial vehicle (UAV) for traffic surveillance. For an overview, see *e.g.* (Doherty *et al.* 2000) or `http://www.ida.liu.se/ext/witas/`. The problem considered is to predict the movements of a vehicle in a road network, and appears in the context of the task of planning a search strategy.

To make enumeration of even finite developments possible, it is necessary to adopt a constraint based representation of state starting and ending times, since events are distributed along a dense time line. This, and the use of progression as the tool to check for formula violation in finite developments, motivated the first step in extending the progression algorithm.

3. Background II: Timed Automata, MITL and Progression

Timed automata and MITL both have their roots in the research area of formal specification and verification of reactive hardware/software systems. Introductions can be found in *e.g.* Alur (1999) and Emerson (1990).

Timed Automata Timed automata (Alur & Dill 1994) are essentially finite state automata, augmented with time constraints of two kinds: a transition can have a time window in which it is possible to make the transition and a state can have a maximal time that the system may remain in the state before it has to exit by some transition.

Let $\mathbb{R}^+$ denote real numbers $\geqslant 0$, with a special symbol ∞ for infinity.

Definition 1 (Timed Automaton)
A timed automaton, $A = (Q, R, C, L)$, consists of a set of *states* Q, a *transition relation*

$$R \subseteq \Sigma \times Q \times Q \times \mathbb{R}^+ \times \mathbb{R}^+$$

where Σ is some set of *event* labels, a *state constraint* function $C : Q \longrightarrow \mathbb{R}^+$, and state labelling function $L : Q \longrightarrow 2^P$, where P is some set of propositional symbols (often the set of states is 2^P, *i.e.* a state is defined by its properties).

As usual, if $(a, q, q', t, t') \in R$, the system may transit from state q to q' in response to the event a, but only in the time interval $[t, t']$ relative to the time that the system entered q (time constraints of the first kind), and the system may remain in state q for a time at most equal to $C(q)$ (time constraints of the second kind)[2].

Like a finite automaton accepts a set of strings over its alphabet, a timed automaton accepts a set of histories.

Definition 2 (Development)
A development is a sequence of alternating states and events marking state transitions, $d = q_0, a_0, q_1, a_1, \ldots$, with an associated function $T : d \longrightarrow \mathbb{R}^+$ that tells the starting time of each state, such that

(*i*) for $i \geqslant 0$, there exists $t, t' \in \mathbb{R}^+$ such that $R(a_i, q_i, q_{i+1}, t, t')$ and $T(q_i)+t \leqslant T(q_{i+1}) \leqslant T(q_i)+t'$, and

(*ii*) for $i \geqslant 0$, $T(q_{i+1}) \leqslant T(q_i) + C(q_i)$.

[2]The normal way to define timed automata is to augment standard automata with a set of real-valued "clock variables", and express time constraints in a language of inequalities (Alur 1999). Definition 1 is less general, but it is sufficient for MITL satisfiability and simplifies some of what follows.

The time interval through which state q_i lasts is $[T(q_i), T(q_{i+1}))$, *i.e.* closed at the beginning and open at the end[3]. The duration of a state q_i is denoted $D(q_i) = T(q_{i+1}) - T(q_i)$.

Two additional properties are usually required of a timed automaton: *executability*, which is the requirement that any finite prefix satisfying conditions (*i*) and (*ii*) of definition 2 can be extended to an infinite development, and *non-zenoness*, which is the requirement that the automaton does not make an infinite number of transitions in finite time.

Even when only finite development prefixes starting in a specific state q_0 are considered, the set of possible developments is uncountable, since the starting time of any state in a development can change by an arbitrarily small amount. For finite developments to be enumerable, a more compact representation has to be adopted: a set of developments that differ only on state starting times are represented by a single sequence of states and events, $d = q_0, a_0, \ldots, q_n$, and a set of constraints on the starting times $T(q_0), \ldots, T(q_n)$, managed in a temporal constraint network (TCN). Throughout, the TCN is assumed to be *simple*, *i.e.* containing only upper and lower bounds on the difference between pairs of temporal variables. This ensures that there are efficient algorithms for checking the consistency of the TCN, and for extracting minimal and maxmimal bounds on variable differences (Dechter, Meiri, & Pearl 1991; Brusoni, Console, & Terenziani 1995).

Metric Interval Temporal Logic

The Metric Interval Temporal Logic (MITL) is a so called "tense modal logic", and was developed as a language for specifying properties of real-time, reactive systems (Alur, Feder, & Henzinger 1996).

Definition 3 (MITL Syntax)
The language of propositional MITL consists of a set of atoms P, propositional connectives and four temporal operators: $\Box_{[t,t']}\varphi$ (*always* φ), $\Diamond_{[t,t']}\varphi$ (*eventually* φ), $\bigcirc_{[t,t']}\varphi$ (*next* φ) and $\varphi\,\mathcal{U}_{[t,t']}\,\psi$ (φ *until* ψ). The intervals adjoined to the operators express metric temporal restrictions, and take point values in $\mathbb{R}^+$.

Formulas in MITL are evaluated over an infinite timed development (d, T). Since MITL formulas only reference present and future states, any suffix of a development is, for the purpose of evaluating formulas, also a development: thus a formula holds in a state q_i, if it holds in the development suffix beginning with q_i.

Definition 4 (MITL Semantics)
Let d^i denote the suffix of d starting with the ith state.

A formula φ not containing any temporal operator holds in d^i iff φ evaluates to T in the state q_i. The truth conditions for temporal formulas are

[3]This choice is rather arbitrary: the reverse convention could be made as well.

- $\Box_{[t,t']}\varphi$ holds in d^i iff φ holds in every d^k such that the intersection of time intervals $[T(q_k), T(q_{k+1}))$ and $[T(q_i) + t, T(q_i) + t']$ is non-empty (note that $k \geqslant i$ is implied by the fact that $t, t' \geqslant 0$ and that time increases along a development).
- $\Diamond_{[t,t']}\varphi$ holds in d^i iff there exists a q_k such that $[T(q_k), T(q_{k+1}))$ and $[T(q_i)+t, T(q_i)+t']$ intersect and φ holds in d^k.
- $\bigcirc_{[t,t']}\varphi$ holds in d^i if φ holds in q_{i+1} and $T(q_i) + t \leqslant T(q_{i+1}) \leqslant T(q_i) + t'$.
- $\varphi \mathcal{U}_{[t,t']} \psi$ holds in d^i iff there exists a q_k such that $[T(q_k), T(q_{k+1}))$ and $[T(q_i) + t, T(q_i) + t']$ intersect, ψ holds in d^k and φ holds for all d^j with $i \leqslant j < k$.

Connectives are interpreted as in ordinary logic.

It should be clear that the definitions can be extended to allow open or half-open operator time intervals, but to avoid an explosion of cases in definitions and algorithms, only closed intervals are considered in the remainder of the paper.

The MITL Progression Algorithm

The MITL progression algorithm provides a way to evaluate MITL formulas "incrementally" over a finite prefix of a development. Thus, the algorithm does not always return TRUE or FALSE, but often a condition to be further progressed through remaining states in the development.

Algorithm 5 (MITL Progression)
Let φ and q be the input formula and state, respectively, and φ' the returned formula. The progresion algorithm works recursively, by cases depending on the form of the input formula:

(*i*) If φ contains no temporal operators, $\varphi' =$ TRUE if φ is true in q and $\varphi' =$ FALSE if not (note that TRUE and FALSE are formula constants, not truth values).

(*ii*) If φ combines one or more subformulas with a propositional connective, φ' is the result of likewise combining the result of progressing each subformula. For example, if $\varphi = \alpha \wedge \beta$ then $\varphi' = \alpha' \wedge \beta'$.

(*iii*) If $\varphi = \bigcirc_{[t,t']}\psi$, then

(*iii.a*) if $D(q) < t$ or $t' < D(q)$, then $\varphi' =$ FALSE, and

(*iii.b*) if $t \leqslant D(q) \leqslant t'$, then $\varphi' = \psi$.

(*iv*) If $\varphi = \Box_{[t,t']}\psi$, then

(*iv.a*) if $D(q) < t$, then $\varphi' = \Box_{[t-D(q),t'-D(q)]}\psi$,

(*iv.b*) if $t \leqslant D(q) \leqslant t'$, then $\varphi' = \psi' \wedge \Box_{[0,t'-D(q)]}\psi$, and

(*iv.c*) if $t' < D(q)$, then $\varphi' = \psi'$,
where ψ' is the result of progressing ψ.

(*v*) If $\varphi = \Diamond_{[t,t']}\psi$, then

(*v.a*) if $D(q) < t$, then $\varphi' = \Diamond_{[t-D(q),t'-D(q)]}\psi$,

(*v.b*) if $t \leqslant D(q) \leqslant t'$, then $\varphi' = \psi' \vee \Diamond_{[0,t'-D(q)]}\psi$,

(*v.c*) if $t' < D(q)$, then $\varphi' = \psi'$,
where ψ' is the result of progressing ψ.

(*vi*) if $\varphi = \chi \mathcal{U}_{[t,t']} \psi$, then

(*vi.a*) if $D(q) < t$, then $\varphi' = \chi' \wedge (\chi \mathcal{U}_{[t-D(q),t'-D(q)]} \psi)$,

(*vi.b*) if $t \leqslant D(q) \leqslant t'$, $\varphi' = \psi' \vee (\chi' \wedge (\chi \mathcal{U}_{[0,t'-D(q)]} \psi))$ and

(*vi.c*) if $t' < D(q)$, then $\varphi' = \psi'$,
where χ' and ψ' are the result of progressing χ and ψ, respectively.

4. The Extended Progression Algorithm

Algorithm 5 assumes that the duration of the input state, $D(q)$, is a number, but as explained above sets of developments have to be represented by a combination of state/event sequence and time constraints to achieve enumerability. Consequently, the progression algorithm has to take as input a set of time constraints, C, and return the set of *all* possible progressions, $\{(\varphi'_1, C'_1), \ldots, (\varphi'_k, C'_k)\}$, where each C'_i is a set of additional time constraints consistent with C and φ'_i is the result of progressing the input formula φ under constraints $C \cup C'_i$.

The extension is conceptually straightforward, though somewhat complicated in practice. Notice that for each temporal operator algorithm 5 branches depending on the duration of the input state, and that the returned formula is built up according to the recursive path of branches taken. In general, an MITL formula defines a tree structure of possible progressions, with time constraints associated to the branches and resulting formulas at the leaves.

For the formal definition of the progression tree, a special form of each temporal operator has to be introduced: a *relative interval* is written $[X : t, t']$, where X is a TCN variable and $t, t' \in \mathbb{R}^+$, and interpreted as $[X + t, X + t']$. Relative temporal operators are obtained by adjoining a relative interval to any of the standard temporal operators.

Definition 6 (Progression Tree)
For an MITL formula φ and a state q with starting and ending times denoted by variables $X_S(q)$ and $X_E(q)$, the *progression tree* $\mathcal{T}_P(\varphi)$ is defined as follows: edges in the tree are labeled with constraints (involving $X_S(q)$, $X_E(q)$, and possibly other TCN variables) and leaf nodes are labeled with formulas.

(*i*) If φ is a state formula (*i.e.* contains no temporal operators), $\mathcal{T}_P(\varphi)$ consists only of a leaf, labeled with TRUE if φ holds in q and FALSE if not.

(*ii*) $\mathcal{T}_P(\neg\alpha)$ is constructed from $\mathcal{T}_P(\alpha)$ by replacing every leaf label α' with $\neg\alpha'$.

(*iii*) To construct $\mathcal{T}_P(\alpha \wedge \beta)$, start with $\mathcal{T}_P(\alpha)$. For every leaf, let α' be the leafs label: replace the leaf with a copy of $\mathcal{T}_P(\beta)$, and within that copy replace every leaf label β' by $\alpha' \wedge \beta'$. The construction is illustrated in figure 1.
The construction of $\mathcal{T}_P(\alpha \vee \beta)$ is analogous.

(*iv*) $\mathcal{T}_P(\bigcirc_{[t,t']}\psi)$ has three branches:

(*iv.a*) a branch labeled $X_E(q) - X_S(q) < t$ leads to a leaf labeled by FALSE,

(*iv.b*) a branch labeled $X_E(q) - X_S(q) > t'$, also to a leaf labeled FALSE, and

(*iv.c*) a branch labeled $t \leqslant X_E(q) - X_S(q) \leqslant t'$ leads to the root of $\mathcal{T}_P(\psi)$.

(*v*) $\mathcal{T}_P(\Box_{[X:t,t']}\psi)$ also has three branches:

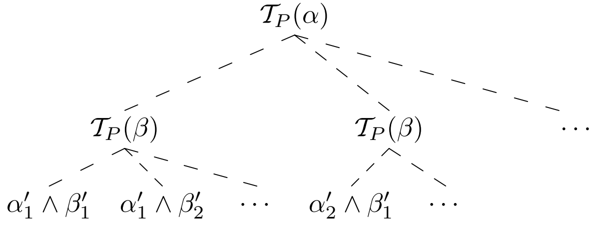


Figure 1: Construction of $\mathcal{T}_P(\alpha \wedge \beta)$

(*v.a*) at the first branch, labeled $X_E(q) - X < t$, is a leaf labeled $\Box_{[X:t,t']}\psi$,

(*v.b*) at the second branch, labeled $t \leqslant X_E(q) - X \leqslant t'$, is a copy of $\mathcal{T}_P(\psi)$, and within that copy every leaf label ψ' is replaced with $(\Box_{[X:t,t']}\psi) \wedge \psi'$, and

(*v.c*) at the third branch, labeled $t' < X_E(q) - X$, is an unmodified copy of $\mathcal{T}_P(\psi)$.

(*vi*) $\mathcal{T}_P(\Box_{[t,t']}\psi)$ is equal to $\mathcal{T}_P(\Box_{[X_S(q):t,t']}\psi)$.

(*vii*) $\mathcal{T}_P(\Diamond_{[X:t,t']}\psi)$ has three branches:

(*vii.a*) at the first branch, labeled $X_E(q) - X < t$, is a leaf labeled $\Diamond_{[X:t,t']}\psi$,

(*vii.b*) at the second branch, labeled $t \leqslant X_E(q) - X \leqslant t'$, is a copy of $\mathcal{T}_P(\psi)$ with every leaf label ψ' replaced by $(\Diamond_{[X:t,t']}\psi) \vee \psi'$, and

(*vii.c*) at the third branch, labeled $t' < X_E(q) - X$, is a copy of $\mathcal{T}_P(\psi)$.

(*viii*) $\mathcal{T}_P(\Diamond_{[t,t']}\psi)$ is equal to $\mathcal{T}_P(\Diamond_{[X_S(q):t,t']}\psi)$.

(*ix*) $\mathcal{T}_P(\chi\mathcal{U}_{[X:t,t']}\psi)$ has three branches:

(*ix.a*) at the first branch, labeled $X_E(q) - X < t$, is a leaf labeled $\chi\mathcal{U}_{[X:t,t']}\psi$,

(*ix.b*) at the second branch, labeled $t \leqslant X_E(q) - X \leqslant t'$, is $\mathcal{T}_P(\psi)$, but every leaf, with label ψ', is replaced a copy of $\mathcal{T}_P(\chi)$, and in this copy, every leaf label χ' is replaced by $(\chi' \wedge (\chi\mathcal{U}_{[X:t,t']}\psi)) \vee \psi'$, and

(*ix.c*) at the third branch, labeled $t' < X_E(q) - X$, is only $\mathcal{T}_P(\psi)$.

(*x*) $\mathcal{T}_P(\chi\mathcal{U}_{[t,t']}\psi)$ equals $\mathcal{T}_P(\chi\mathcal{U}_{[X_S(q):t,t']}\psi)$.

The construction of the progression tree parallels progression algoritm 5, but time constraints are captured in the edge labels instead of the bounds of intervals adjoined to temporal operators in the formulas labelling the leafs. The introduction of relative temporal operators serves to ensure that all time constraints found in the progression tree remain simple. The algorithm for progression now becomes a simple matter of tree traversal:

Algorithm 7 (Extended MITL Progression)
Let φ and q be the input MITL formula and state, respectively, and C a set of (simple) time constraints.

Construct $\mathcal{T}_P(\varphi)$. For every leaf l in the progression tree, collect the set of constraints C_l found along the path to l: if $C \cup C_l$ is consistent, (φ'_l, C'_l) is included in the set of progressions returned.

The method of traversing the progression tree is not important, as long as every leaf at the end of a consistent path is eventually found. The most efficient method appears to be to search the tree depth first and check the set of constraints incrementally, at each node.

Example 1 Consider the formula $\varphi = \Box_{[5,9]} \bigcirc_{[0,4]} p$. The progression tree is shown in figure 2. If the input set of constraints is $C = \{0 < X_E - X_S < 7\}$, there are two consistent solution paths:

(*a*) $X_E - X_S < 5$, with the result $\Box_{[X_S:5,9]} \bigcirc_{[0,4]} p$.

(*b*) $5 \leqslant X_E - X_S \leqslant 9$, $X_E - X_S > 4$, with the result FALSE.

$X_E - X_S > 9$ and $X_E - X_S < 0$ both contradict C, while $0 \leqslant X_E - X_S \leqslant 4$ is inconsistent with the constraint $5 \leqslant X_E - X_S$ labelling the branch above.

Note that solutions returned by the algorithm may contain relative temporal operators: these only make sense in the context of a particular constraint set, which may be viewed as conjoined to the returned formula. Consequently, if the formula of a returned solution (φ', C') is to be further progressed through another state, the whole constraint set, $C \cup C'$, must be passed as input to the next progression.

5. MITL Consistency

Alur *et al.* (1996) present a decision procedure for the consistency of an MITL formula, but with several restrictions on the formula: there is no *next* operator, the time intervals adjoined to temporal operators must have rational constant endpoints, and must be non-singular, *i.e.* not of the form $[t, t]$. The restriction to non-singular intervals is necessary, since the consistency problem is provably undecidable if singular intervals are allowed (Alur & Henzinger 1994).

The algorithm constructs a timed automaton accepting exactly the developments that are models of the formula. The automaton is then checked for emptiness (*i.e.* whether it accepts at least one development), *e.g.* using the procedure of Alur and Dill (1994).

Suppose that the progression algorithm is extended, so that the input state q may be unspecified *w.r.t.* propositions as well as starting and ending time, and the solutions returned contain constraints on both. Then, if progressing an MITL formula through a sequence of completely unspecified states (except for the constraint $D(q) \geqslant 0$, for each state), results in TRUE, any sequence of states that satisfies the set of constraints collected along the way is a model for the formula. This idea leads to what is essentially a tableaux algorithm for MITL consistency.

The decision procedure for (non-real time) Linear Temporal Logic (LTL) is also tableaux based, but constructs a Büchi automaton equivalent to the formula to be checked (Wolper 1989; Gerth *et al.* 1995). A variant described by

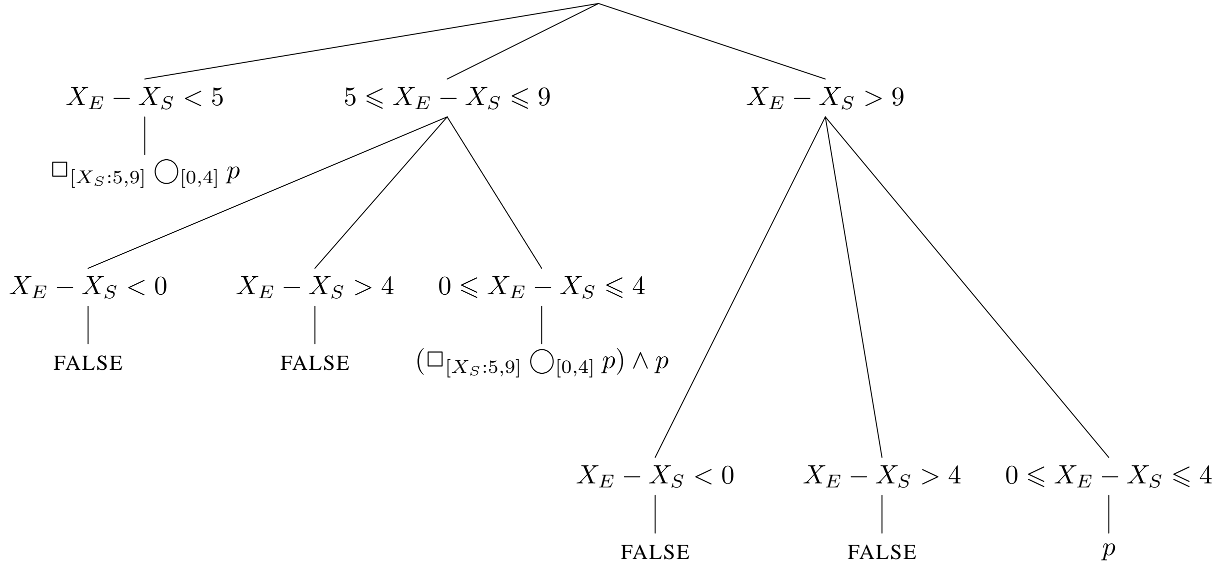


Figure 2: $\mathcal{T}_P(\Box_{[5,9]} \bigcirc_{[0,4]} p)$

Schwendimann (1998) is remarkably similar to the algorithm developed here, except it is formulated in terms of inference rules rather than progression, and applies of course only to LTL.

Partial State Progression

For the progression algorithm to work with partially or completely unspecified states requires only a small change in the progression tree: case (*i*) is replaced by

(*i'*) For a single proposition p, $\mathcal{T}_P(p)$ has two branches, labeled by $p = \mathrm{T}$ and $p = \mathrm{F}$, ending in leafs labeled TRUE and FALSE, respectively.

The branch labels are constraints on the value of p, while the leaf labels are formula constants. Because there are no disjunctions, the set of propositional constraints essentially corresponds to a partial assignment, so consistency can be easily determined.

Tableaux Construction

The tableaux is a tree constructed by repeated progression of the MITL formula to be checked for consistency. Each node in the tree is labeled by a formula and a constraint set. The tree also represents a set of developments: each node corresponds to a state, and each path from the root downwards to a development.

Definition 8 (Tableaux Tree)
Let φ be an MITL formula. The *tableaux tree* of φ, $\mathcal{T}_T(\varphi)$ is defined inductively by:

(*i*) The root, r, is labeled by φ and $\{X_E(r) - X_S(r) \geqslant 0\}$.

(*ii*) Let n be a node labeled with formula φ_n and constraint set C_n. For each solution (φ', C') found by progressing φ_n with input constraints C_n, n has a successor node, n', labeled with φ' and $C_n \cup C' \cup \{X_S(n') = X_E(n), X_E(n') - X_S(n') \geqslant 0\}$.

Because constraints are only passed down in the tree, variables can be named by the depth of the node only, *i.e.* the root node starts at X_0 and ends at X_1, its successors all start at X_1 and end at X_2, *etc.* The constraint $X_{i+1} - X_i \geqslant 0$, placed on every node, ensures that the duration of the corresponding state is non-negative.

Example 2 Figure 3 shows the tableaux tree for the formula $\Box_{[5,9]} \bigcirc_{[0,4]} p$ (formula labels only). Nodes marked "zeno cycle" correspond to non-executable developments, as explained in the next section.

Definition 9 (Closed, Satisfying)
A node n in $\mathcal{T}_T(\varphi)$ is *closed* iff (*a*) the formula labeling the node is TRUE or FALSE, or (*b*) the constraint set stored in the node is inconsistent. A node is *satisfying* if it is labeled with TRUE.

To decide the consistency of a formula φ, $\mathcal{T}_T(\varphi)$ is searched for a satisfying node: if one is found, φ is consistent, and the sequence of states along the path from the root to the node is a model. The subtree beneath a closed node does not have to be examined, and if every leaf node is closed and not satisfying, φ is inconsistent.

Closing Cycles

For many MITL formulas, the tableaux tree contains infinite branches without a closing node, and thus a straightforward search in the tableaux tree may fail to terminate. Consider for example $\Box_{[0,\infty]} \Diamond_{[0,10]} p$: this formula generates, among others, an infinite sequence of progressions identical to the

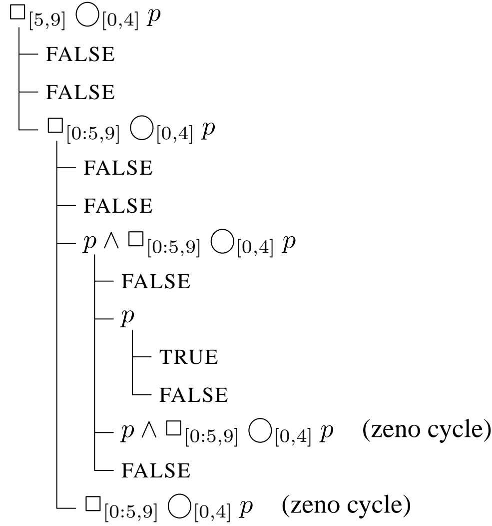


Figure 3: $\mathcal{T}_T(\Box_{[5,9]} \bigcirc_{[0,4]} p)$

input formula (except for the $\Box$ operator being relative to X_0).

If a node n is labeled by the same formula as a node n' found along the path from the root to n, and the constraint set of the ancestor node n' subsumes, *i.e.* is less restrictive than, that of n, then all the possible successors of n are already contained in the subtree beneath n', and it should not be necessary to search further below n. Thus, definition 9 must be amended:

Definition 10 (Cycle, Zeno Cycle)
A node n in $\mathcal{T}_T(\varphi)$ for which there exists an ancestor node n' labeled by an identical formula and such that the constraint set of n' subsumes that of n, is a *cycle*. A cycle node is also closed.

If in addition, the maximal value of $X_S(n) - X_S(r)$, where r is the root, is bounded by the set of constraints (*i.e.* not infinite), node n is a *zeno cycle*.

Because time constraints are only added as the formula is progressed down the tableaux tree, the temporal constraint set of a node always subsumes those of its successors. Subsumption for state constraints, however, must still be checked.

A cycle node n represents an infinite development, consisting of the states along the path from the root to the ancestor node n', followed by an infinite number of repetitions of the states from n' to, but not including, n. If the maximal starting time of n (which is also the ending time of the cycle) is bounded, the time in this development can not diverge beyond that bound, *i.e.* it will pass through an infinite number of states in finite time: hence the name "zeno cycle".

Evaluation in Cycles That a node is a cycle does not mean that it does not satisfy the formula labeling the node. For example, $\mathcal{T}_T(\Box_{[0,\infty]}\Diamond_{[0,10]}p)$ contains a cycle node at depth 2, labeled by $\Box_{[X_0:0,\infty]}\Diamond_{[0,10]}p$ and, among others, the constraints $\{X_2 - X_1 \leqslant 10, p_2 = \mathrm{T}\}$. This represents a development consisting of an infinite sequence of states, each with duration at most 10, in all of which p is true, clearly a model for the formula.

For any node n that is a cycle but not a zeno cycle, if the formula labeling the node at the start of the cycle (node n' in definition 10) holds in the corresponding infinite development, according to the standard MITL semantics, n is satisfying. Even though the development is infinite, the number of distinct states it visits is finite, which makes evaluation possible, although not without complications: the distance, in time, between different states in the cycle may be "stretched", though not arbitrarily, by inserting repetitions of the cycle.

Improved Cycle Detection The condition for cycle detection defined above is too weak to ensure that the search for a satisfying node in the tableaux tree terminates. Two examples illustrate the problem:

Example 3 $\mathcal{T}_T(\Box_{[0,\infty]}(\Diamond_{[0,\infty]}p \wedge \Diamond_{[0,\infty]}\neg p))$ contains an infinite branch beginning with

$$
\begin{aligned}
n_0 &: \Box_{[0,\infty]}(\Diamond_{[0,\infty]}\neg p \wedge \Diamond_{[0,\infty]}p)\\
n_1 &: \quad \Diamond_{[X_0:0,\infty]}\neg p \wedge \Box_{[X_0:0,\infty]}(\Diamond_{[0,\infty]}\neg p \wedge \Diamond_{[0,\infty]}p)\\
n_2 &: \qquad \Diamond_{[X_1:0,\infty]}p \wedge \Box_{[X_0:0,\infty]}(\Diamond_{[0,\infty]}\neg p \wedge \Diamond_{[0,\infty]}p)\\
n_3 &: \qquad\quad \Diamond_{[X_2:0,\infty]}\neg p \wedge \Box_{[X_0:0,\infty]}(\Diamond_{[0,\infty]}\neg p \wedge \Diamond_{[0,\infty]}p)\\
&\qquad\quad \vdots
\end{aligned}
$$

corresponding to a sequence of states in which p alternates between T and F.

Example 4 $\mathcal{T}_T(\Box_{[0,\infty]}\Diamond_{[0,\infty]}(p \wedge \neg p))$ contains only infinite branches of the form

$$
\begin{aligned}
n_0 &: \Box_{[0,\infty]}\Diamond_{[0,\infty]}(p \wedge \neg p)\\
n_1 &: \quad \Diamond_{[X_0:0,\infty]}(p \wedge \neg p) \wedge \Box_{[X_0:0,\infty]}\Diamond_{[0,\infty]}(p \wedge \neg p)\\
n_2 &: \qquad \Diamond_{[X_0:0,\infty]}(p \wedge \neg p) \wedge \Diamond_{[X_1:0,\infty]}(p \wedge \neg p) \wedge\\
&\qquad \Box_{[X_0:0,\infty]}\Diamond_{[0,\infty]}(p \wedge \neg p)\\
&\qquad\quad \vdots
\end{aligned}
$$

since the state formula $p \wedge \neg p$ will never be true.

In example 3, the formulas labelling nodes n_3 and n_1 are identical, except for the reference time point of the relative interval adjoined to the $\Diamond$ operator. If, however, the constraints on the time variable X_0 at n_1 are not more restrictive than those on X_2 at n_3, the set of successors, *i.e.* progressions, of n_3 will also be identical to those of n_1, except for this difference: in particular, if there exists a satisfying node among the successors of n_3, the "same" satisfying node must be found by the same sequence of progressions from n_1. To state this condition precisely requires some extra definitions:

Definition 11 (Formula Shift)
The (backwards) *shift by* k of a formula ϕ is obtained by replacing every occurrence of every time variable X_i with $i \geqslant k$ by X_{i-k} in ϕ.

Definition 12 (Subsumes with Shift)
Let C and C' be constraint sets, where $C \subseteq C'$. Then C *subsumes* C' *with shift by* k iff

- $X_i - X_j$ in C subsumes $X_i - X_j$ in C', for all $i \leqslant j < k$,
- $X_i - X_j$ in C subsumes $X_i - X_{j+k}$ in C', for all $i < k \leqslant j$,
- $X_i - X_j$ in C subsumes $X_{i+k} - X_{j+k}$ in C', for all $k < i \leqslant j$.

The meaning of "$X_i - X_j$ in C subsumes $X_u - X_v$ in C'" is that the bounds on $X_i - X_j$ set by C are less restrictive than those placed on $X_u - X_v$ by C'.

Definition 13 (Extended Cycle)
A node n starting at X_j in $\mathcal{T}_T(\varphi)$ for which there exists an ancestor node n' starting at X_i such that the formula labelling n' equals the formula labelling n backwards shifted by $j - i$ and such that the constraint set at n' subsumes that at n with shift $j - i$, is also a cycle.

By the extended definition, node n_3 in example 3 is a cycle so the branch is closed. The formula labelling node n_1 at the start of the cycle is true in the corresponding infinite development.

This, however, is not enough to close the branch in example 4, because the formula resulting from progression is redundant. To fix this, the progression result is rewritten into an equivalent, simpler, form, based on the notion of "weak implication":

Definition 14 (Weakly Implies)
Formula α *weakly implies* β, *w.r.t.* constraint set C, in case

(*i*) α and β are identical (*i.e.* a formula always weakly implies itself), or

(*ii*) $\alpha = \square_{[X:s,t]}\phi$, $\beta = \square_{[X':s',t']}\phi'$, $[X : s, t]$ necessarily contains $[X' : s', t']$ given C and ϕ weakly implies ϕ', or

(*iii*) $\alpha = \diamond_{[X:s,t]}\phi$, $\beta = \diamond_{[X':s',t']}\phi'$, $[X : s, t]$ is necessarily contained in $[X' : s', t']$ given C and ϕ weakly implies ϕ'.

The rewrite rules applied to the formula resulting from progression are:

R1. Eliminate from a conjunction every conjunct that is weakly implied by another conjunct.

R2. Eliminate from a disjunction every disjunct that weakly implies another disjunct.

Weak implication is determined *w.r.t.* the constraint set input to progression and the set returned along with the formula. Since weak implication entails ordinary implication, the rules preserve equivalence.

Because of the constraint $X_1 - X_0 \geqslant 0$, $[X_1 : 0, \infty]$ must be contained in $[X_0 : 0, \infty]$, so $\diamond_{[X_1:0,\infty]}(p \wedge \neg p)$ weakly implies $\diamond_{[X_0:0,\infty]}(p \wedge \neg p)$, and the formula labelling node n_2 in example 4 can be simplified to

$$\diamond_{[X_1:0,\infty]}(p \wedge \neg p) \wedge \square_{[X_0:0,\infty]}\diamond_{[0,\infty]}(p \wedge \neg p)$$

using rewrite rule *R1*, which makes the node a cycle (by the extended condition).

Correctness and Complexity

The tableaux method is clearly sound, in the sense that whenever a satisfying node is found, the path leading to that node (with infinite repetition if it is a cycle node) is a model for the formula. Likewise, if only the basic cycle definition is applied and all branches are closed and not satisfying, the formula is inconsistent: the argument is that the progression algorithm is exhaustive and that the development corresponding to a closed node can not be a model for the formula. The extended cycle definition appears, intuitively, to be correct, but since it is so complicated, intuition is not quite reliable and it should be proved formally.

Even the extended cycle detection, however, is not strong enough to ensure termination, *e.g.* the tableaux tree for the formula $\square_{[0,\infty]}(\diamond_{[0,10]}\neg p \wedge \diamond_{[0,10]}p)$ contains infinite branches. The reason, in this case, is that the time limit on the $\diamond$ operator causes weak implication to fail: $\diamond_{[X_1:0,10]}p$ does not weakly imply $\diamond_{[X_0:0,10]}p$ unless constraints entail $X_1 = X_0$, since only then is $[X_1 : 0, 10]$ necessarily contained in $[X_0 : 0, 10]$.

Deciding MITL consistency is EXPSPACE-complete (Alur, Feder, & Henzinger 1996). The algorithm by Alur *et al.* requires time exponential in the number of connectives and the largest integer constant appearing in the formula.

The tableaux method may not be able to do too much better. Consider the number of consistent progressions of a formula φ, in the worst case: The number of temporal branch nodes in the progression tree is bounded by the number of temporal operators. At each node, the difference $X_k - X_i$, where k is the index of the state through which the formula is progressed (equal to the tableaux depth), and $i < k$, is compared to a constant interval (the restriction interval of a temporal operator). With n branch nodes, the comparisons involve at most $2n$ different constants, which when ordered yield $2n + 1$ different intervals that each $X_k - X_i$ may fall into. Denoting the number of distinct variable differences occuring in constraints in the tree by $d \leqslant \max(k, n)$, the number of consistent paths through the temporal branch nodes is at most

$$\frac{(2n+d)!}{d!(2n)!} \leqslant (2n+1)^d,$$

since $X_{i+1} - X_i \geqslant 0$ for all i, and therefore $X_k - X_j \leqslant X_k - X_i$ whenever $j > i$. The number of consistent truth value assignments is of course 2^p, where p is the number of distinct propositions occuring in the formula.

Thus, the worst case branching factor in the tableaux tree is polynomial in the number of temporal operators in the formula, but exponential in the number of distinct propositions and in the tableaux depth. As for how deeply the tree may have to be searched to find a satisfying node or close all branches (assuming the search terminates at all), I have currently no idea.

6. Concluding Remarks

MITL is a powerful language for expressing properties over time: it has been used to express requirements in formal verification, goals and control rules in planning, and knowledge in predictive models. Likewise, progression is a powerful tool for working with MITL. The extended algorithm enables it to be used also with a compact representation of sets of developments, which in turn enables enumeration of the finite development prefixes generated by a timed automaton.

The tableaux algorithm for deciding MITL consistency shows some promise: with extended cycle detection and simplification rewriting, it manages to prove the formula $\Box_{[0,\infty]}\Diamond_{[0,\infty]}p \wedge \Diamond_{[0,\infty]}\Box_{[0,\infty]}\neg p$ unsatisfiable. In difference to the method by Alur *et al.*, it does not depend on time constants being integral, or even rational, except as far as constraint management does. Also, it appears simpler, which is not an unimportant property.

Still, it is very much work in progress: besides making it complete, the extended cycle detection, possibly strengthened, needs to be proved correct, and a more thorough analysis of the algorithms complexity to be done. Although it certainly requires both exponential time and space, it may be exponential in different variables than the existing algorithm: for example, it is hard to see that the size of the constants in the intervals adjoining the temporal operators should play a part, and this may make a difference in practice.

That aside, rewriting based on weak implication is inelegant and *ad hoc*. It may be seen as imposing a "weak normal form" on formulas, but exactly what this form is, and what set of rewrite rules is sufficient to obtain it, needs to be clarified.

References

Alur, R., and Dill, D. 1994. A theory of timed automata. *Theoretical Computer Science* 126(2):183 – 236.

Alur, R., and Henzinger, T. 1994. A really temporal logic. *Journal of the ACM* 41(1):181 – 204.

Alur, R.; Feder, T.; and Henzinger, T. 1996. The benefits of relaxing punctuality. *Journal of the ACM* 43(1):116 – 146.

Alur, R. 1999. Timed automata. In *NATO-ASI Summer School on Verification of Digital and Hybrid Systems*. Available at http://www.cis.upenn.edu/∼alur/Nato97.ps.gz.

Bacchus, F., and Kabanza, F. 1996. Planning for temporally extended goals. In *Proc. 13th National Conference on Artificial Intelligence (AAAI'96)*.

Brusoni, V.; Console, L.; and Terenziani, P. 1995. On the computational complexity of querying bounds on differences constraints. *Artificial Intelligence* 74:367 – 379.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61 – 95.

Doherty, P.; Granlund, G.; Kuchcinski, K.; Sandewall, E.; Nordberg, K.; Skarman, E.; and Wiklund, J. 2000. The WITAS unmanned aerial vehicle project. In *Proc. European Conference on Artificial Intelligence*.

Emerson, E. 1990. Temporal and modal logic. In *Handbook of Theoretical Computer Science*. Elsevier. 997 – 1072.

Gerth, R.; Peled, D.; Vardi, M.; and Wolper, P. 1995. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the 15th Workshop on Protocol Specification, Testing and Verification*. North-Holland.

Haslum, P. 2001. Models for prediction. In *Proc. IJCAI 2001 workshop on Planning under Uncertainty*.

Schwendimann, S. 1998. A new one-pass tableau calculus for PLTL. In de Swart, H., ed., *Proc. International Conference on Theorem Proving with Analytic Tableaux and Related Methods*, volume 1397 of *Lecture Notes in Artificial Intelligence*, 277 – 291. Springer Verlag.

Wolper, P. 1989. On the relation of programs and computations to models of temporal logic. In Banieqbal, B.; Barringer, H.; and Pnueli, A., eds., *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, 75 – 123. Springer Verlag.

An Efficient BDD-Based A* Algorithm

Rune M. Jensen, Randal E. Bryant, and Manuela M. Veloso
Computer Science Department,Carnegie Mellon University,
5000 Forbes Ave, Pittsburgh,PA 15213-3891, USA
{runej,bryant,mmv}@cs.cmu.edu

Abstract

In this paper we combine the goal directed search of A* with the ability of BDDs to traverse an exponential number of states in polynomial time. We introduce a new algorithm, SetA*, that generalizes A* to expand sets of states in each iteration. SetA* has substantial advantages over BDDA*, the only previous BDD-based A* implementation we are aware of. Our experimental evaluation proves SetA* to be a powerful search paradigm. For some of the studied problems it outperforms BDDA*, A*, and BDD-based breadth-first search by several orders of magnitude. We believe exploring sets of states to be essential when the heuristic function is weak. For problems with strong heuristics, SetA* efficiently specializes to single-state search and consequently challenges single-state heuristic search in general.

Introduction

During the last decade, powerful search techniques using an implicit state representation based on the *reduced ordered binary decision diagram* (BDD, Bryant 1986) have been developed in the area of *symbolic model checking* (McMillan 1993). Using blind exploration strategies these techniques have been successfully applied to verify systems with very large state spaces. Similar results have been obtained in well-structured AI search domains (Cimatti *et al.* 1997). However for hard combinatorial problems the search fringe often grows exponentially with the search depth.

A classical AI approach for avoiding the state explosion problem is to use heuristics to guide the search toward the goal states. The question is whether heuristics can be applied to BDD-based search such that their ability to efficiently expand a large set of states in each iteration is preserved. The answer is non-trivial since heuristic search algorithms require values to be associated with each state and manipulated during search. A task for which BDDs often have proven less efficient.

In this paper, we present a new search algorithm called SetA*. The main idea is to avoid the above problem by generalizing A* (Hart, Nilsson, & Raphael 1968) from single states to sets of states in the search queue. Recall that A* associates two values g and h to each state in the search queue. g is the cost of reaching the state and h is an estimate of the remaining cost of reaching the goal given by a *heuristic function*. In SetA* states with similar g and h values are merged such that we can represent them implicitly by a BDD without having to store any numerical information. In each iteration, SetA*: 1) pops the set with highest priority, 2) computes its next states, and 3) partitions the next states into child sets with unique g and h values, which are reinserted into the queue. A straightforward implementation of the three phases has disappointing performance (see PreSetA*, Table 2). A key idea of our work is therefore to combine phase 2 and 3. The technique fits nicely with the so called disjunctive partitioning of BDD-based search (Clarke, Grumberg, & Peled 1999). In addition it can be applied to any heuristic function. Our experimental evaluation of SetA* proves it a powerful search paradigm. For some problems it dominates both A* and BDD-based breadth-first search (see Table 2). In addition, it outperforms the only previous BDD-based implementation of A* (BDDA*, Edelkamp & Reffel 1998), we are aware of, with up to two orders of magnitude (see Table 4).

The remainder of the paper is organized as follows. First we briefly describe BDDs and BDD-based search. We then define the SetA* algorithm and evaluate it experimentally in a range of search and planning domains. Finally we discuss related work and draw conclusions.

BDD-based Search

A BDD is a canonical representation of a Boolean function with n linear ordered arguments $x_1, x_2, ..., x_n$. It is a rooted, directed acyclic graph with one or two terminal nodes labeled 1 or 0, and a set of variable nodes u of out-degree two. The two outgoing edges are given by the functions $high(u)$ and $low(u)$ (drawn as solid and dotted arrows). Each variable node is associated with a propositional variable in the Boolean function the BDD represents. The graph is ordered in the sense that all paths in the graph respect the ordering of the variables. A BDD representing the function $f(x_1, x_2) = x_1 \wedge x_2$ is shown in Figure 1 (left). Given an assignment of the arguments x_1 and x_2, the value of f is determined by a path starting at the root node and iteratively following the high edge, if the associated variable is true, and the low edge, if the associated variable is false. The value of f is $True$ if the label of the reached terminal node is 1; otherwise it is $False$. A BDD is reduced so that no two distinct nodes u and v have the same variable name

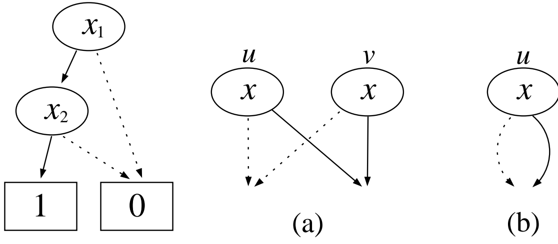


Figure 1: A BDD representing the function $f(x_1, x_2) = x_1 \wedge x_2$. True and false edges are drawn solid and dotted, respectively. (a) and (b) Reductions of BDDs.

and low and high successors (Figure 1(a)), and no variable node u has identical low and high successors (Figure 1(b)). The BDD representation has two major advantages: first, many functions encountered in practice have a polynomial size. Second, any operation on two BDDs, corresponding to a Boolean operation on the functions they represent, has a low complexity bounded by the product of their node counts.

A search problem is a 4-tuple (S, T, i, G). S is a set of states. $T : S \times S$ is a *transition relation* defining the search graph. $(s, s') \in T$ iff there exists a transition leading from s to s'. i is the initial state of the search while G is the set of goal states. A solution to a search problem is a path $\pi = s_0, \cdots, s_n$ where $s_0 = i$ and $s_n \in G$ and $\bigwedge_{j=0}^{n-1}(s_j, s_{j+1}) \in T$. Assuming that states can be encoded as bit vectors, BDDs can be used to represent the *characteristic function* of a set of states and the transition relation. To make this clear, consider the simple search problem shown in Figure 2. A state s is represented by a bit vector with two elements $\vec{s} = (s_0, s_1)$. Thus the initial state is represented by a BDD for the expression $\neg s_0 \wedge \neg s_1$. Similarly we have $G = s_0 \wedge s_1$. To encode the transition relation, we need to refer to current state variables and next state variables. We adopt the usual notation in BDD literature of primed variables for the next state

$$\begin{aligned} T(s_0, s_1, s'_0, s'_1) &= \neg s_0 \wedge \neg s_1 \ \wedge \ s'_0 \wedge \neg s'_1 \\ &\vee \ \neg s_0 \wedge \neg s_1 \ \wedge \ \neg s'_0 \wedge s'_1 \\ &\vee \ \neg s_0 \wedge s_1 \ \wedge \ s'_0 \wedge s'_1 \\ &\vee \ s_0 \wedge s_1 \ \wedge \ s'_0 \wedge \neg s'_1. \end{aligned}$$

The main idea in BDD-based search is to stay at the BDD level when finding the next states of a set of states. This can be done by computing the image of a set of states V encoded in current state variables

$$\text{Img} = \big(\exists \vec{s}.\, V(\vec{s}) \wedge T(\vec{s}, \vec{s}')\big)[\vec{s}/\vec{s}'].$$

Consider the first step of the search from i in the example domain. We have $V(s_0, s_1) = \neg s_0 \wedge \neg s_1$. Thus,

$$\begin{aligned} \text{Img} &= \big(\exists \vec{s}.\, \neg s_0 \wedge \neg s_1 \wedge T(s_0, s_1, s'_0, s'_1)\big)[\vec{s}/\vec{s}'] \\ &= \big(s'_0 \wedge \neg s'_1 \ \vee \ \neg s'_0 \wedge s'_1\big)[\vec{s}/\vec{s}'] \\ &= s_0 \wedge \neg s_1 \ \vee \ \neg s_0 \wedge s_1. \end{aligned}$$

The image computation is applied for searching in forward

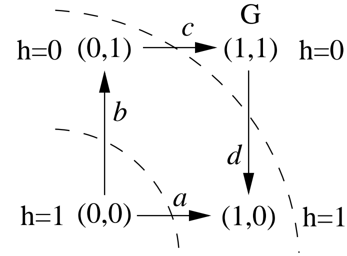


Figure 2: An example search problem consisting of four states and four transitions a,b,c, and d. The dashed lines indicate the two search fringes of a BDD-based breadth-first search from the initial state $i = (0, 0)$ to the goal states $G = \{(1, 1)\}$. The h-values is a heuristic function equal to the vertical goal distance.

direction. For searching backward an analogous computation called the *preimage* is applied. In this section, we focus on techniques for performing the image computation efficiently, but similar techniques exist for the preimage computation.

A common problem in BDD-based search is that intermediate BDDs in the image computation tend to be large compared to the BDD representing the result. In symbolic model checking, a range of techniques has been proposed to avoid this problem. Among the most successful of these are *transition relation partitioning*. For search problems, where each transition normally only modifies a small subset of the state variables, the suitable partitioning technique is *disjunctive partitioning* (Clarke, Grumberg, & Peled 1999). To make a disjunctive partitioning, the part of the individual transition expressions keeping the unmodified variables unchanged is removed. The transition expressions are then partitioned according to what variables they modify. For our example we get two partitions

$$\begin{aligned} P_1 &= \neg s_0 \wedge \neg s_1 \wedge s'_0 \ \vee \ \neg s_0 \wedge s_1 \wedge s'_0 \\ m_1 &= (s_0) \\ P_2 &= \neg s_0 \wedge \neg s_1 \wedge s'_1 \ \vee \ s_0 \wedge s_1 \wedge \neg s'_1 \\ m_2 &= (s_1). \end{aligned}$$

In addition to large space savings, disjunctive partitioning often lowers the complexity of the image computation which now can skip the quantification of unchanged variables and operate on smaller expressions

$$\text{Img} = \bigvee_{j=1}^{|\mathbf{P}|} \big(\exists m_j \,.\, V(\vec{s}) \wedge P_j(\vec{s}, m'_j)\big)[m_j/m'_j].$$

The complexity of the image computation depends on the number of partitions. Notice that for each new partition, a new conjunction with $V(\vec{s})$ is introduced. For this reason the best performance is often obtained by clustering some of the partitions according to an upper bound on the size of the BDD representing a partition (Burch, Clarke, & Long 1991; Ranjan *et al.* 1995).

SetA*

SetA* is a generalization of weighted A* where the definition of f is changed from $f = g + h$ to $f = (1 - w)g + wh,\ w \in [0,1]$ (Pohl 1970). Similar to BDDA*, SetA* assumes a finite search domain and unit-cost transitions. SetA* expands a set of states instead of just a single state. The main input is what we will call, an *improvement partitioning*. That is, a disjunctive partitioning where the transitions of a partition reduce the h-value by the same amount. The improvement partitioning is non-trivial to compute. The reason is that it may be intractable to calculate each transition expression in turn. Fortunately large sets of transitions are often described in more abstract terms (e.g. by *actions* or *guarded commands*) that can be directly translated into BDDs. This allows for an implicit way to partition a set of transitions according to their improvement. Assume that a set of transitions are represented by a BDD $T(\vec{s}, \vec{s}')$. Given a BDD $h(\vec{s}, \vec{v})$ encoding the heuristic function, such that $\vec{v}$ is a bit vector representation of the h-value associated with state s, the set of transitions with improvement equal to k is

$$T(\vec{s}, \vec{s}') \wedge h(\vec{s}, \vec{v}) \wedge h(\vec{s}', \vec{v}') \wedge \vec{v} - \vec{v}' = \vec{k}.$$

The improvement partitioning is computed only once prior to the search, and in practice it turns out that it often can be produced directly from the description of transitions or by splitting the disjunctive partitioning. In fact, for the heuristics we have studied so far, no BDD encoding of the heuristic function has been necessary. The improvement partitioning may containing several partitions with similar improvement. This may be an advantage if the partitions otherwise grow too large.

SetA* uses two main data structures: a prioritized queue Q and a reach structure R. Each node in Q contains a BDD representing a set of states with particular g and h values. The node with lowest f-value has highest priority. Ties are solved by giving highest priority to the entry with lowest h-value. An important parameter of Q is an upper bound u on the BDD sizes. When inserting a new node it is unioned with an existing node in Q with the same g and h value if the sum of the size of their two BDDs is less than u. Otherwise a new entry is created for the node. The reach structure is for loop detection. R keeps track of the lowest g-value of every reached state and is used to prune states from a set of next states already reached with a lower g-value. The algorithm is shown in Figure 3. All sets and set operations are carried out with BDDs. SetA* takes five arguments. **IP** is the improvement partitioning described above. $init$ and $goal$ are the initial and goal states of the search. u is the upper bound parameter of Q and w is the usual weight parameter of weighted A*. Initially the algorithm inserts the initial state in Q. Observe that the h-value of the initial state has to be found. However since $init$ is a single state this is trivial. Similar to the regular A* algorithm, SetA* continues popping the top node of the queue until the queue is either empty or the states of the top node overlaps with the goal. The top node is expanded by finding the image of it for each improvement partition in turn (l.9). Before being inserted in Q, the new nodes are pruned for states seen at a lower

function $SetA^*(\mathbf{IP}, init, goal, u, w)$
1 $\quad Q.initialize(u, w, goal)$
2 $\quad g \leftarrow 0$
3 $\quad h \leftarrow h(init)$
4 $\quad Q.insert(init, g, h)$
5 $\quad R.update(init, g)$
6 $\quad$ **while** $\neg Q.empty()$ **and** $\neg Q.topAtGoal()$
7 $\quad\quad top \leftarrow Q.pop()$
8 $\quad\quad$ **for** $j = 0$ **to** $|\mathbf{IP}|$
9 $\quad\quad\quad next \leftarrow image(top, IP_j)$
10 $\quad\quad\quad R.prune(next)$
11 $\quad\quad\quad g \leftarrow top.g + 1$
12 $\quad\quad\quad h \leftarrow top.h - impr(IP_j)$
13 $\quad\quad\quad Q.insert(next, g, h)$
14 $\quad\quad\quad R.update(next, g)$
15 $\quad$ **if** $Q.empty()$ **then** NoPathExists
16 $\quad\quad$ **else** $R.extractPath()$

Figure 3: The SetA* algorithm.

search depth, and the reach structure is updated (l.10-14). If the loop was aborted due to Q being empty no solution path exists. Otherwise the path is extracted by applying transitions backwards on the states in R from one of the reached goal states.

SetA* is *sound* due to the soundness of the image computation. Since no states reached by the search are pruned, SetA* is also *complete*. Given an *admissible heuristic* and $w = 0.5$, SetA* further finds *optimal length paths*. As for A*, the reason is that a state on the optimal path eventually will reach the top of Q because states on finalized but sub-optimal paths have higher f-value (Pearl 1984).

The upper bound u can be used to adjust how many states SetA* expands. If each partition in **IP** contains a single transition and $u = 0$ then SetA* specializes to A*. Interestingly it is even a highly efficient implementation of A*. The memory sharing of BDDs robustly scales to tens of thousands of BDDs, and loop detection is still handled implicitly via BDDs in the R structure. For problems with many shortest length solution paths like the DVM and Logistics described in the next section, it may be an advantage to focus on a subset of them by choosing a low u-value. A similar approach is used by A^*_ϵ described in (Pearl 1984)

The weight w has the usual effect. For $w = 0.5$ Set A* behaves like A*. For $w = 1.0$ it performs best-first search, and for $w = 0.0$ it carries out a regular breadth-first search. The fact that w can take any value in the range $[0, 1]$ is important in practice, since it can be used to strengthen a conservative heuristic and vice versa.

We end this section by demonstrating SetA* on our example problem. For this demonstration we assume $w = 0.5$ and $u = \infty$. The heuristic function is the vertical distance to the goal state. In Figure 2 the states have been labeled with h-values. We see that **IP** must contain at least three partitions: one containing transition d that improves by minus one, one containing a and c that improve by zero, and

one containing b that improves by one. Initially we have

$$Q_0 = < (f = 0.5, g = 0, h = 1, \{(0,0)\}) >$$
$$R_0 = < (g = 0, \{(0,0)\}) > .$$

In the first iteration, state $(0,0)$ is expanded to one child containing state $(1,0)$ and one child containing $(0,1)$. According to the improvements of the partitions, we get

$$\begin{aligned} Q_1 &= < (f = 0.5, g = 1, h = 0, \{(0,1)\}), \\ &\quad (f = 1.0, g = 1, h = 1, \{(1,0)\}) > \\ R_1 &= < (g = 0, \{(0,0)\}), (g = 1, \{(0,1),(1,0)\}) > . \end{aligned}$$

In the second iteration, only the c transition can fire resulting in

$$\begin{aligned} Q_2 &= < (f = 1.0, g = 2, h = 0, \{(0,1)\}), \\ &\quad (f = 1.0, g = 1, h = 1, \{(1,0)\}) > \\ R_2 &= < (g = 0, \{(0,0)\}), (g = 1, \{(0,1),(1,0)\}), \\ &\quad (g = 2, \{(1,1)\}) > . \end{aligned}$$

The tie breaking rule causes the goal state to be at the top of Q at the beginning of the third iteration. Thus the while loop is aborted and the solution path $(0,0),(0,1),(1,1)$ is extracted from R_2.

Experimental Evaluation

SetA* has been implemented in the UMOP multi-agent planning framework (Jensen & Veloso 2000) to study its performance characteristics relative to blind bidirectional BDD-based breadth-first search (also implemented in UMOP) and an A* implementation with explicit state representation and cycle detection. In a second evaluation round we developed a domain independent STRIPS planning system called DOP. The state encoding and heuristic function used by the MIPS planner (Edelkamp & Helmert 2001) was reproduced in order to conduct a fair comparison with BDDA* implemented in MIPS. In addition to SetA*, two blind BDD-based breadth-first search algorithms were implemented in DOP, one searching forward and one searching backward. MIPS also includes an algorithm called Pure BDDA*. Pure BDDA* performs best-first search.

All experiments were carried out on a Linux 5.2 PC with a 500 MHz Pentium 3 CPU, 512 KB L2 cache and 512 MB RAM. The time limit (TIME) was 600 seconds and the memory limit (MEM) was 450 MB. For UMOP and DOP the number allocated BDD nodes and the cache size used by the BDD-package were hand-tuned for best performance. A disjunctive partitioning with a minimum number of partitions was applied unless otherwise noted.

Artificial Problems

Two problems IG^k and $D^x V^y M^z$ were defined and studied using the minimum *Hamming distance* to the goal states as heuristic function (the mimimum number of different bits between the bit vector representing the state and a goal state). In these experiments the improvement partitioning was computed by splitting a disjunctive partitioning using a specialized BDD-function. Given an improvement k, this function traverses the BDD of an action and picks transitions of the action improving k. The complexity of the function is linear in the size of the action BDD when the goal is a conjunction and the variable ordering interleaves current and next state variables.

k	SetA* #it	SetA* T (sec)	A* #it	A* T (sec)
1	16	0.2	16	0.13
2	16	0.2	145	0.39
3	16	0.2	514	1.26
4	16	0.2	2861	7.46
5	16	0.2	9955	29.02
6	16	0.2	24931	80.10
7	16	0.2	51098	181.77
8	16	0.2	90080	344.00
9	16	0.2	140756	579.22
10	16	0.2	-	TIME
11	16	0.2	-	-
12	16	0.2	-	-
13	16	0.2	-	-
14	16	0.2	-	-
15	16	0.2	-	-

Table 1: Results for the IG^k problem. #it is the number of iterations, and T is the total CPU time.

IG^k This problem is simplest to define using the STRIPS language (Fikes & Nilsson 1971). Thus a state is a set of facts and an action is a triple of sets of facts. In a given state S, an action (pre, add, del) is applicable if $pre \subseteq S$, and the resulting state is $S' = (S \cup add) \setminus del$. The actions are

$$\begin{array}{lll} \mathbf{A}_1^1 & \mathbf{A}_j^1 \ j = 2, \cdots, n & \mathbf{A}_j^2 \ j = 1, \cdots, n \\ \ pre : \{I^*\} & \ pre : \{I^*, G_{j-1}\} & \ pre : \{\} \\ \ add : \{G_1\} & \ add : \{G_j\} & \ add : \{I_j\} \\ \ del : \{\} & \ del : \{\} & \ del : \{I^*\}. \end{array}$$

The initial state is $\{I^*\}$ and the goal state is $\{G_j | k < j \leq n\}$. Only $\mathbf{A}_j^1$ actions should be applied to reach the goal. Applying an $\mathbf{A}_j^2$ action in any state leads to a wild path since I^* is deleted. The states on wild paths contain I_j facts. Since any subset of I_j facts is possible, the number of states on wild paths grows exponentially with n. The only solution is $\mathbf{A}_1^1, \cdots, \mathbf{A}_n^1$ which is non-trivial to find, since the heuristic gives no information to guide the search on the first k steps. The purpose of the experiment is to investigate how well SetA* copes with this situation compared to A*. For SetA* $w = 0.5$ and $u = \infty$. For the IG^k problems considered, n equals 16. This corresponds to a state space size of 2^{33}. The results are shown in Table 1.

The experiment shows a fast degradation of A*'s performance with the number of unguided steps. A* gets lost expanding an exponentially growing set of states on wild paths. SetA* is hardly affected by the lack of guidance. The reason is that all transitions on the unguided part improve by zero. Thus on this part, SetA* performs a regular BDD-based breadth-first search, which due to the structure

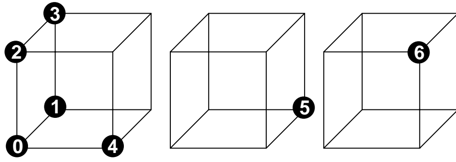


Figure 4: The initial state of $D^5V^3M^7$.

of the problem scales well.

$\mathbf{D^xV^yM^z}$ In this domain a set of sliders are moved between the corner positions of hypercubes. In any state, a corner position can be occupied by at most one slider. The dimension of the hypercubes is y. There are z sliders of which x are moving on the same cube. The remaining $z - x$ sliders are moving on individual cubes. Figure 4 shows the initial state of $D^5V^3M^7$. When $x = z$ all sliders are moving on the same cube. If further $x = 2^y - 1$ all corners of the cube except one will be occupied. In this form, DVM is a *permutation problem* similar to the 15puzzle and Rubik's Cube. We choose to investigate DVM instead of these well-known problems because it has a direct Boolean encoding. In this way, the complexity of the problem is solely caused by the interaction between sliders, allowing us to adjust the dependency of sliders linearly with the x parameter. For the 15puzzle and Rubik's Cube there would be two sources of complexity. One due to the interaction between objects and one due to the physical constraints of the puzzles.

The purpose of the first experiment is to investigate how SetA* degrades when the dependency of the domain is increased and to compare its performance to A* and BDD-based breadth-first search. In this experiment we study the $D^xV^4M^{15}$ problem. For all experiments the size of the state space is 2^{60}. We also show the results of PreSetA*, a premature version of SetA* finding the next states and splitting them in two separate phases. Both versions of SetA* were run with $w = 0.5$ and $u = 200$. In this experiment an upper bound of 1000 on the size of the disjunctive partition BDDs was chosen. The results are shown in Table 2.

The upper bound on the partitions is crucial for large values of x. Despite applying this technique, BDD-based bidirectional search does not scale due to a blow-up of the search fringe in both directions. A* works well when x is small since f is a perfect or near perfect discriminator. However when the quality of the heuristic degrades A* gets lost tracking equally promissing paths. The good performance of SetA* is due to the low upper bound of the size of BDDs in the search queue. It focuses the search on a reasonable subset of the paths. Interestingly the search time is very low even for the hardest problems. Time and memory are spent on building and splitting the transition relation. Separating the next state computation and the splitting as done by the earlier version of SetA*, seems to come with a large performance penalty.

In the second experiment we measure the performance of SetA* for increasing upper bounds (u) of the size of BDDs in the search queue. The results are shown in Table 3 and were obtained for the $D^9V^4M^{15}$ problem using the same disjunctive partitioning as in the previous experiment.

u	#it	T (sec)	T_t (sec)	T_s (sec)
100	73	7.4	3.3	1.3
200	34	6.8	3.3	0.7
400	34	7.3	3.3	1.1
800	52	8.3	3.3	2.2
1600	51	10.1	3.3	4.0
3200	49	14.1	3.3	8.0
6400	49	24.4	3.3	17.7
12800	45	47.5	3.3	39.6
25600	42	110.4	3.3	102.8
51200	34	474.4	3.3	466.8

Table 3: Upper bound results for SetA* on the $D^9V^4M^{15}$ problem. u is the upper bound, #it is the number of iterations, T is the total CPU time, T_t is the time used to generate the improvement partitioning, and T_s is time used on search.

As depicted the performance degrades substantially for large values of u. The problem is that the sets of most promising states is large and have no compact BDD representation. By choosing a low u-value we focus on a subset of the most promising states in each iteration. As long as the problem has many solution paths this approach may work well. For $D^9V^4M^{15}$ this is reasonable to assume, since the sliders still are fairly independent. For highly dependent problems, however, a low u-value may lead to SetA* getting lost on wild paths.

Planning Problems

Like MIPS, the DOP planning system uses an approximation to the HSPr heuristic (Bonet & Geffner 1999) for STRIPS domains. In addition, it performs similar analysis to minimize the state encoding length. HSPr is an efficient but non-admissible heuristic. We approximate it by summing the depth $d(f)$ of each fact in a state given by a relaxed forward breadth-first search. The heuristic is applied in a backward search from the goal states to the initial state. For any action (pre, add, del) leading from S to S' (when applied in forward direction), we assume

$$del \subseteq pre \text{ and } add \not\subseteq pre.$$

Since the search is backward the improvement of the action is

$$\begin{aligned} impr &= h(S') - h(S) \\ &= h(S' \cap (pre \cup add)) - h(S \cap (pre \cup add)) \\ &= \sum_{f \in add \setminus S} d(f) - \sum_{f \in del} d(f). \end{aligned}$$

Thus the improvement of an action can be computed without any BDD-based encoding of the heuristic function. Each action is partitioned in up to $2^{|add|}$ sets of transitions with different improvement.

x	SetA*		PreSetA*		A*		BiDir	
	#it	T (sec)	#it	T (sec)	#it	T (sec)	#it	T (sec)
1	34	0.6	34	0.8	34	1.1	34	0.7
2	34	0.7	34	0.9	34	1.1	34	0.7
3	34	0.6	34	1.4	34	1.1	34	1.6
4	34	0.6	34	1.5	34	1.1	34	8.1
5	34	0.6	34	3.5	34	1.0	34	334.0
6	34	0.8	34	14.4	-	TIME	-	TIME
7	34	1.3	34	39.8	-	TIME	-	TIME
8	34	2.1	34	50.7	-	TIME	-	TIME
9	94	6.8	34	202.6	-	TIME	-	TIME
10	58	16.3	34	297.2	-	TIME	-	TIME
11	34	39.3	-	TIME	-	TIME	-	TIME
12	-	MEM	-	TIME	-	TIME	-	TIME

Table 2: Results for the $D^x V^4 M^{15}$ problem. #it is the number of iterations, and T is the total CPU time.

The problems we consider, are *Gripper* from the STRIPS track of the AIPS-98 planning competition (Long 2000) and *Logistics* from the first round of the STRIPS track of the AIPS-00 planning competition (Bacchus 2001). The purpose of these experiments is to compare the performance of SetA* and BDDA*, not to solve the problems particularly fast. In that case, a more informative heuristic like the FF heuristic (Hoffmann 2001) should be applied.

Gripper This domain considers a robot with two grippers moving an increasing number of balls between two connected rooms. The first experiment compares forward BDD-based breadth-first search, SetA* with $w = 1.0$ and $u = \infty$, backward BDD-based breadth-first search, pure BDDA*, and BDDA*. Recall that Pure BDDA* performs best-first search like SetA* with $w = 1.0$. The results are shown in Table 4.

This domain is efficiently solved using blind BDD-based breadth-first search. The reason is that the search fringe grows only polynomially with the search depth. As is often observed both for planning and model checking problems, the best performance is obtained when searching forward. The performance of SetA* is almost as good as forward search even though, this algorithm relies on the slower backward expansion. BDDA* spends considerable time prior to search computing BDD formulas for arithmetic operations. During search the fringe expansion of BDDA* seems to degrade fast with the size of the fringe. Pure BDDA* on the other hand successfully completes a large number of iterations due to a lower growth rate of the fringe. All algorithms find shortest plans.

The second experiment shows the impact of the weight setting in problem 20. The results are shown in Table 5. Since the problem can be solved efficiently by blind BDD-based breadth-first search, it is not surprising that the weight setting turns out to be less important for the performance of SetA*.

Logistics This domain considers moving packages with trucks between sub-cities and with airplanes between cities. In the first experiment SetA* was run with $w = 1.0$ and

w	#it	$\|p\|$	T (sec)	T_s (sec)
0.0	360	125	7.6	4.6
0.1	358	125	7.7	4.5
0.2	354	125	7.9	4.7
0.3	347	125	8.0	4.7
0.4	338	125	8.0	4.8
0.5	373	125	9.2	5.9
0.6	204	125	6.1	2.9
0.7	204	125	6.1	3.0
0.8	204	125	6.2	3.0
0.9	204	125	6.2	3.2
1.0	204	125	5.9	2.9

Table 5: Results of the second Gripper experiment. w is the weight, $|p|$ is the solution length, #it is the number of iterations, T is the total CPU time, and T_s is time used on search.

$u = 200$. The upper bound of the size of partitions in the disjunctive partitioning was 400. The results are shown in Table 6.

Due to the fact that there are no resource constraints in the Logistics domain, and thus no conflicts between subgoals, the HSPr heuristic is quite efficient. Both SetA* and Pure BDDA* search fast in this domain. However, Pure BDDA* and BDDA* have a significant overhead due to their precomputation of arithmetic formulas. For SetA* the upper bound on the size of the partitions in the disjunctive partitioning is crucial for the larger Logistics problems. In addition the upper bound on the size of BDDs in the search queue speeds up SetA* on the last five problems. The search fringe for blind BDD-based search blows up in both directions. The plans of SetA* are slightly longer than Pure BDDA*. The plans of BDDA* are shorter than both SetA* and Pure BDDA*, but only BDD-based breadth-first search finds optimal length plans.

The second experiment was carried out on problem 7 of the Logistics domain. In this experiment SetA* was run with

#p	$\|S\|$	Forward	SetA*			Backward	Pure BDDA*			BDDA*		
		T (sec)	#it	T (sec)	T_s (sec)	T (sec)	#it	T (sec)	T_s (sec)	#it	T (sec)	T_s (sec)
1	2^{11}	0.1	14	0.1	0.0	0.1	22	3.0	0.0	14	2.8	0.0
2	2^{15}	0.1	24	0.1	0.0	0.1	62	4.0	0.1	22	3.9	0.1
3	2^{19}	0.2	34	0.2	0.1	0.3	138	5.5	0.5	30	5.3	0.4
4	2^{23}	0.2	44	0.3	0.1	0.6	250	8.0	1.8	38	7.1	1.2
5	2^{27}	0.3	54	0.5	0.1	1.0	398	12.9	5.5	46	10.3	3.2
6	2^{31}	0.4	64	0.6	0.2	1.4	582	22.4	13.6	54	15.4	7.0
7	2^{35}	0.6	74	0.7	0.2	2.1	802	40.4	29.5	62	25.2	15.5
8	2^{39}	0.9	84	1.0	0.4	2.9	1058	72.5	59.4	70	47.1	36.2
9	2^{43}	1.0	94	1.2	0.4	4.1	1350	137.4	120.7	78	123.5	111.8
10	2^{47}	1.2	104	1.4	0.5	5.3	1678	317.2	295.8	-	TIME	-
11	2^{51}	1.4	114	1.7	0.7	7.1	-	TIME	-	-	TIME	-
12	2^{55}	1.7	124	2.0	0.8	9.1	-	TIME	-	-	TIME	-
13	2^{59}	2.0	134	2.3	1.0	13.0	-	TIME	-	-	TIME	-
14	2^{63}	2.2	144	2.7	1.2	17.2	-	TIME	-	-	TIME	-
15	2^{67}	2.7	154	3.1	1.4	16.4	-	TIME	-	-	TIME	-
16	2^{71}	3.5	164	3.5	1.6	19.7	-	TIME	-	-	TIME	-
17	2^{75}	3.4	174	4.0	1.9	23.1	-	TIME	-	-	TIME	-
18	2^{79}	3.9	184	4.9	2.3	27.5	-	TIME	-	-	TIME	-
19	2^{83}	4.5	194	5.1	2.6	32.4	-	TIME	-	-	TIME	-
20	2^{87}	5.0	204	5.8	3.0	37.2	-	TIME	-	-	TIME	-

Table 4: Results of the first Gripper experiment. #p is the problem number, $|S|$ is the size of the state space, #it is the number of iterations, T is the total CPU time, and T_s is time used on search.

#p	$\|S\|$	SetA*				Pure BDDA*				Forward		BDDA*			
		#it	$\|P\|$	T (sec)	T_s (sec)	#it	$\|P\|$	T (sec)	T_s (sec)	$\|P\|$	T (sec)	#it	$\|P\|$	T (sec)	T_s (sec)
4	2^{21}	21	21	0.2	0.1	22	22	6.5	0.0	20	0.3	54	22	7.7	1.2
5	2^{21}	33	33	0.3	0.1	30	30	6.7	0.1	27	0.5	65	28	9.5	2.7
6	2^{21}	31	31	0.3	0.1	30	30	6.7	0.1	25	0.4	64	26	8.4	1.6
7	2^{41}	46	44	0.9	0.3	44	42	13.9	0.3	36	99.0	-	-	TIME	-
8	2^{41}	41	40	1.0	0.3	36	36	14.1	0.2	31	59.5	94	32	138.5	118.5
9	2^{41}	48	46	0.9	0.2	46	45	14.0	0.3	36	100.0	102	38	132.6	115.8
10	2^{54}	66	56	2.5	1.1	54	51	25.1	1.1	-	MEM	-	-	TIME	-
11	2^{54}	71	61	2.2	0.7	62	60	25.2	1.2	-	-	-	-	TIME	-
12	2^{54}	60	54	2.0	0.6	52	49	24.9	0.8	-	-	-	-	TIME	-
13	2^{86}	154	94	8.5	5.0	96	94	57.5	6.5	-	-	-	-	TIME	-
14	2^{86}	127	78	7.7	3.9	70	65	56.7	8.3	-	-	-	-	TIME	-
15	2^{86}	140	96	7.3	3.2	92	91	53.9	5.5	-	-	-	-	TIME	-

Table 6: Results of the first Logistics experiment. #p is the problem number, $|S|$ is the size of the state space, #it is the number of iterations, $|P|$ is the plan length, T is the total CPU time, and T_s is time used on search.

w	#it	$\|p\|$	T (sec)	T_s (sec)
0.0	279	25	8.6	7.9
0.1	248	25	9.0	8.3
0.2	203	25	8.9	8.1
0.3	154	25	7.9	7.1
0.4	102	25	4.7	4.0
0.5	180	27	2.3	1.6
0.6	49	29	0.9	0.1
0.7	31	31	0.8	0.1
0.8	31	31	0.8	0.1
0.9	31	31	0.8	0.1
1.0	31	31	0.9	0.1

Table 7: Results of the second Logistics experiment. w is the weight, $|p|$ is the solution length, #it is the number of iterations, T is the total CPU time, and T_s is time used on search.

$u = \infty$. The results are shown in Table 7. As depicted HSPr is a good heuristic for this domain increasing the speed significantly while preserving a relative high solution quality. Notice that the relaxation of the upper bound does not affect the performance of SetA for this problem.

Related Work

Directed BDD-based search has received little attention in symbolic model checking. The reason is that the main application of BDDs in this field is verification where all reachable states must be explored. For Computation Tree Logic (CTL) checking, guiding techniques have been proposed to avoid a blow-up of intermediate BDDs (Bloem, Ravi, & Somenzi 2000). However these techniques are not applicable to search since they are based on defining lower and upper bounds on the fixed-point. Directed search techniques are relevant for *falsification* where the goal is to find a state not satisfying an invariant. The first work on BDD-based directed search, we are aware of, was for this application (Yang & Dill 1998). The proposed algorithm is a simple best-first search where the search fringe is partitioned with a specialized BDD-operator according to the Hamming distance to the goal state. Even though this operation is fairly efficient for the Hamming distance, it is not obvious how to define it in general.

As far as we know, the only previous BDD-based implementation of A* is BDDA*. BDDA* can use a general heuristic function and has been applied to planning as well as model checking. Similar to SetA*, it assumes unit-cost transitions and Boolean encoding of states. In contrast to SetA*, however, BDDA* requires arithmetic operations at the BDD level during search and includes no tools to control the growth of the search fringe or for cycle detection. In addition BDDA* is non-trivial to generalize to weighted A*.

The search queue in BDDA* is represented by a BDD $Open(s, f)$ that associates each state s in the queue with its f-value. Given a BDD encoding of the heuristic function $h(s, v)$ and a BDD encoding of the set of states with minimum f-value $Min(s)$, BDDA* expands all states with minimum f-value by computing

$$\begin{aligned} Open'(s,f) \;\leftarrow\; & \exists s' .\, Min(s) \wedge T(s,s') \wedge \\ & \exists e' .\, h(s',e') \wedge \exists e .\, h(s,e) \wedge \\ & (f = f_{min} + e - e' + 1). \end{aligned}$$

Since the BDDs representing the heuristic function and the transition relation often are large, a naive implementation of this computation would be very slow. The MIPS implementation of BDDA* seems to use another strategy where the largest possible subset of the computations are carried out prior to the search. However this strategy has not been described in the literature and as indicated by our experiments, it still leads to substantial performance degradation.

Conclusion and Outlook

In this paper, we have combined BDD-based search and heuristic search into a new search paradigm. The experimental evaluation of SetA* proves it a powerful algorithm often several orders of magnitude faster than BDD-based breadth-first search and A*. Today planning problems are efficiently solved by heuristic single-state search algorithms. However as recently noticed, the success may be due to an inherent simplicity of the benchmark domains when using the right heuristics (Hoffmann 2001). For less domain-tuned heuristics, we believe that the ability of SetA* to explore an exponential growing set of paths in polynomial time is essential. Our ongoing research includes identifying such problems and comparing the performance of SetA* and single-state search algorithms.

Acknowledgments

This research is sponsored in part by the United States Air Force under Grants Nos F30602-00-2-0549 and F30602-98-2-0135. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force, or the US Government.

References

Bacchus, F. 2001. AIPS'00 planning competition : The fifth international conference on artificial intelligence planning and scheduling systems. *AI Magazine* 22(3):47–56.

Bloem, R.; Ravi, K.; and Somenzi, F. 2000. Symbolic guided search for CTL model checking. In *Proceedings of the 37th Design Automation Conference (DAC'00)*, 29–34. ACM.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *Proceedings of the European Conference on Planning (ECP-99)*. Springer.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 8:677–691.

Burch, J.; Clarke, E.; and Long, D. 1991. Symbolic model checking with partitioned transition relations. In *International Conference on Very Large Scale Integration*, 49–58. North-Holland.

Cimatti, A.; Giunchiglia, E.; Giunchiglia, F.; and Traverso, P. 1997. Planning via model checking: A decision procedure for $\mathcal{AR}$. In *Proceedings of the 4th European Conference on Planning (ECP'97)*, 130–142. Springer.

Clarke, E.; Grumberg, O.; and Peled, D. 1999. *Model Checking*. MIT Press.

Edelkamp, S., and Helmert, M. 2001. MIPS the model-checking integrated planning system. *AI Magazine* 22(3):67–71.

Edelkamp, S., and Reffel, F. 1998. OBDDs in heuristic search. In *Proceedings of the 22nd Annual German Conference on Advances in Artificial Intelligence (KI-98)*, 81–92. Springer.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on SSC* 100(4).

Hoffmann, J. 2001. Local search topology in planning benchmarks: An empirical analysis. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, 453–458. Morgan Kaufmann.

Jensen, R., and Veloso, M. M. 2000. OBDD-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research* 13:189–226.

Long, D. 2000. The AIPS-98 planning competition. *AI Magazine* 21(2):13–34.

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publ.

Pearl, J. 1984. *Heuristics : Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Pohl, I. 1970. First results on the effect of error in heuristic search. *Machine Intelligence* 5:127–140.

Ranjan, R. K.; Aziz, A.; Brayton, R. K.; Plessier, B.; and Pixley, C. 1995. Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM Proceedings of the International Workshop on Logic Synthesis*.

Yang, C. H., and Dill, D. L. 1998. Validation with guided search of the state space. In *Proceedings of the 35th Design Automation Conference (DAC'98)*, 599–604. ACM.

Probabilistic Plan Verification through Acceptance Sampling

Håkan L. S. Younes*
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.
lorens@cs.cmu.edu

David J. Musliner
Automated Reasoning Group
Honeywell Laboratories
3660 Technology Drive
Minneapolis, MN 55418, U.S.A.
musliner@htc.honeywell.com

Abstract

CIRCA is an architecture for real-time intelligent control. The CIRCA planner can generate plans that are guaranteed to maintain system safety, given certain timing constraints. To prove that its plans guarantee safety, CIRCA relies on formal verification methods. However, in many domains it is impossible to build 100% guaranteed safe plans, either because it requires more resources than available, or because the possibility of failure simply cannot be eliminated. By extending the CIRCA world model to allow for uncertainty in the form of probability distribution functions, we can instead generate plans that maintain system safety with high probability. This paper presents a procedure for probabilistic plan verification to ensure that heuristically-generated plans achieve the desired level of safety. Drawing from the theory of quality control, this approach aims to minimize verification effort while guaranteeing that at most a specified proportion of good plans are rejected and bad plans accepted.

Introduction

Realistic domains for autonomous agents present a broad spectrum of uncertainty, including uncertainty in external events and in the outcome of internally-selected actions. Planning to achieve system goals and maintain safety in the face of this uncertainty can be highly challenging. We can attempt to generate a universal plan (cf. (Schoppers 1987)), taking every contingency into account, but with limited resources this may be futile. A more advantageous approach may be to quantify the uncertainty and incorporate this information into the reasoning process. This enables us to set an arbitrary failure threshold for plans, and we can reject plans with failure probability above the threshold. Furthermore, the additional information can be used to focus our planning efforts on situations we are more likely to encounter (cf. (Atkins *et al.* 1996)).

In this paper we introduce a probabilistic extension to CIRCA—an architecture for real-time control—(Musliner *et al.* 1993). The original planner in CIRCA builds reactive control plans that achieve system goals and maintain system safety, subject to strict time bounds and models of the dynamic external world (the environment). While the original CIRCA model includes nondeterminism in the outcome of actions and uncertainty about the timing and occurrence of externally-caused transitions, it does not have quantified uncertainty information. The extended model includes quantified uncertainty in the form of probability distributions on the timing of different transitions. This allows CIRCA to build plans that are not completely guaranteed to prevent failure—plans may allow for the possibility of failure, as long as the failure probability is below some threshold. The world model of the probabilistic extension corresponds to a generalized semi-Markov process.

Standard model checking techniques cannot handle the full generality of our model. We have therefore developed a novel verification algorithm that can be used to efficiently verify whether a potential plan satisfies given safety constraints. Our verifier uses Monte Carlo simulation, or more precisely discrete event simulation, to generate sample execution paths given a plan and a world model. Requiring relatively few sample paths, the verifier can guarantee that at most a specified proportion of good plans are rejected and bad plans accepted.

The interaction between planner and verifier is depicted in Figure 1. The planner generates a plan to achieve given objectives in a given dynamic real-time environment. The plan is passed to the verifier that then tests whether the plan satisfies given safety constraints. The result of the verification is passed back to the planner, which based on that information decides whether the generated plan needs to be revised. If the plan passed the verification, then no revision is needed, and otherwise the verification result is used to guide plan revision. We are not concerned with how to generate and revise plans in this paper, but only how to verify plans.

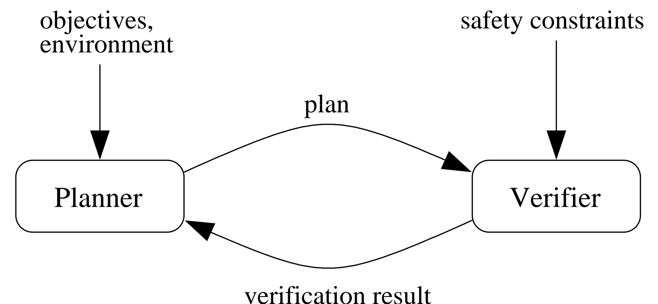


Figure 1: Interaction between planner and verifier.

*The work reported in this paper was performed during a summer internship at Honeywell Laboratories.

The World Model

Musliner, Durfee, & Shin (1993) introduced a formalization of the CIRCA world model, and later extended it (Musliner *et al.* 1995). We will deviate slightly from their formalization here. The purpose is to enable us to view the CIRCA world model as a *generalized semi-Markov process* (GSMP), which is a formalism for discrete event systems introduced by Matthes (1962) (see also (Glynn 1989; Shedler 1993)). The formal non-probabilistic world model has seven elements $(S, S_F, S_0, T, E, min\Delta, max\Delta)$:

1. A finite set of states S, where each state represents a description of relevant features.
2. A set of failure states $S_F \subset S$, which consists of all states in S that violate domain constraints or control-level goals.
3. A set of possible initial states $S_0 \subset S$.
4. A finite set of transitions $T = T_E \cup T_A \cup T_T$, where T_E is a set of event transitions representing world occurrences as instantaneous state changes, T_A is a set of action transitions representing actions performed by the run-time system, and T_T is a set of temporal transitions representing the progression of time.
5. A function $E : S \to 2^T$ mapping a state s to a set of transitions enabled in s.
6. A function $min\Delta : T \to \mathbb{R}$ mapping transitions to minimum trigger times.
7. A function $max\Delta : T \to \mathbb{R}$ mapping transitions to maximum trigger times.

Each transition $\tau \in T$ is a mapping between states; $\tau : S \to S$.

For a given planning problem, the environment is represented by a world model M_{env} without any action transitions. Given M_{env}, the planner generates a plan (or *policy*) π, which is a mapping from states to action transitions. The composition of M_{env} and π is a stochastic process representing the execution of π in the given environment. When verifying that a plan π is safe, we are really verifying that certain properties hold for the stochastic process representing the composition of π and the environment model. Figure 2(a) shows an environment for an unmanned aerial vehicle. In Figure 2(b), actions constituting a plan have been added to the environment.

Model Dynamics

At any particular point in time, the world is considered to occupy a single state in the model. The initial world state can be any state $s \in S_0$. The world state changes when a transition is triggered. If the current state is s and transition τ is triggered, the next state is given by $\tau(s)$. Not all transitions are necessarily enabled in all states. For each state $s \in S$, $E(s)$ is the subset of T denoting the set of transitions that can be triggered in state s. Only one transition can be triggered in each state at any given time, so transitions in $E(s)$ compete to trigger a state change.

We can associate a clock $r_{s,\tau}$ with each enabled transition τ in a state s, showing the time remaining until τ is scheduled to occur in s. The clock value $r_{s,\tau}$ is called the *residual lifetime* of τ in s (Glynn 1989). When a transition τ^* is triggered in state s, causing a transition to state $s' = \tau'(s)$, then the lifetimes of the transitions enabled in s' are initialized as follows:

1. If $\tau \in E(s) \setminus \{\tau^*\}$, then let $r_{s',\tau} = r_{s,\tau} - r_{s,\tau^*}$.
2. If $\tau \notin E(s) \setminus \{\tau^*\}$, then $r_{s',\tau}$ is set to some value in the interval $[min\Delta(\tau), max\Delta(\tau)]$.

The type of a transition determines the general form of the interval $[min\Delta(\tau), max\Delta(\tau)]$. Event transitions can occur at any time, and thus have a lower limit of zero and an upper limit of infinity. Temporal transitions are similar to events, but can have a non-zero lower limit. An action transition represents an action taken by the run-time system, and has a finite upper limit representing the worst-case execution time for that action.

Note that with the formalism given here, the possible trigger time of a transition in a given state at a given time can depend on the history of state transitions, making the world model non-Markovian.

Probabilistic Extension

The world model, as presented so far, has limited expressive power. We can say that an event may occur in a state s by representing the event with a transition τ which is enabled in s, and we can bound the time that the world must stay in s before the event may (or must) occur. We can, however, say nothing about the *expected* time that the world must stay in state s before the event occurs. There is no way to distinguish more frequently occurring events, such as rain delaying a tennis match at Wimbledon, from far less frequent events, such as a player being struck by lightning.

A natural extension is to associate a probability distribution function $F(t; \tau)$ with each transition τ, giving the probability that τ will be triggered t time units after it was last enabled. We can easily define the previously used interval limits in terms of F:

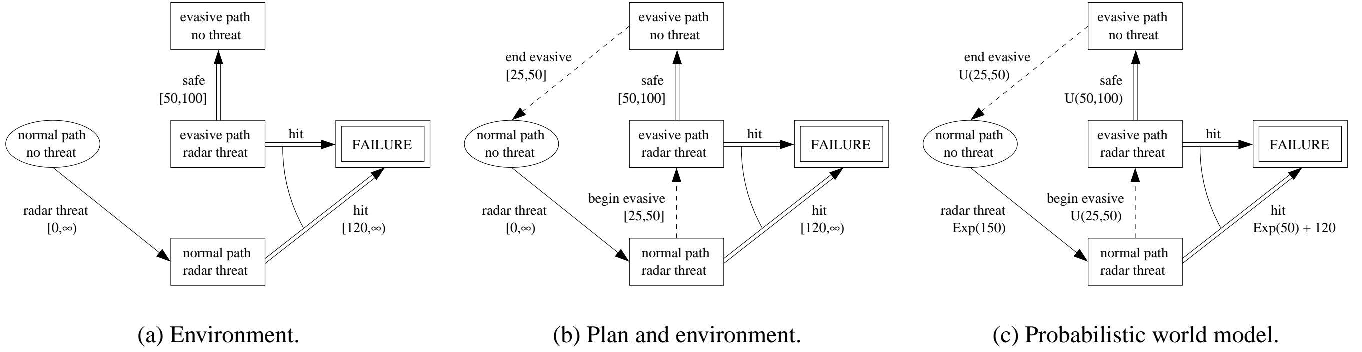
$$min\Delta(\tau) = \sup\{t \mid F(t;\tau) = 0\}$$
$$max\Delta(\tau) = \inf\{t \mid F(t;\tau) = 1\}$$

We require that $F(0; \tau) = 0$ (i.e. the distribution function corresponds to a positive random variable), because no transition can be triggered before it has been enabled. A typical choice of distribution for an event transition would be an exponential distribution, and for a temporal transition a shifted exponential distribution. For an action transition one could, for example, use a uniform distribution or a truncated normal distribution.

In addition we can replace S_0 with a probability distribution p_0 over S, where $p_0(s)$ is the probability that the world starts in state s. The set of possible initial states is then simply

$$S_0 = \{s \mid p_0(s) > 0\}.$$

This way we obtain a probabilistic world model consisting of six elements (S, S_F, p_0, T, E, F). To make this a GSMP we need to define transition probabilities $p(s'; s, \tau)$ expressing the probability of the next state being s' given



(a) Environment.

(b) Plan and environment.

(c) Probabilistic world model.

Figure 2: Three different world models. To the left is a world model representing the environment. The initial state is drawn as an oval, and there is only one failure state. The solid arrow represents an event transition, and the double arrows represent temporal transitions. The arc between the two "hit" transitions indicate that they are in fact the same transition enabled in two different states. In the middle, actions (dashed arrows) constituting a plan have been added to the environment. To the right is the same plan/environment model, but with probability distributions associated with each transition instead of just intervals.

that τ is triggered in state s. This is straightforward, however, because the next state is determined by $\tau(s)$. We thus get the following:

$$p(s'; s, \tau) = \begin{cases} 1 & \text{if } \tau(s) = s' \\ 0 & \text{otherwise} \end{cases}$$

In addition, we set all clock speeds $r(s, \tau)$ to 1. The elements (S, p_0, p, T, E, F, r) constitute a GSMP. In fact, the form of the state transition probabilities and the probability distribution functions makes this a *time-homogeneous* GSMP (Glynn 1989).

Figure 2(c) shows the probabilistic version of the model in Figure 2(b). Instead of an interval, a probability distribution function is given for each transition.

Plan Verification

In the non-probabilistic world model, a safe plan is one where no state $s \in S_F$ is reachable from the set of possible initial states S_0. Action transitions are planned to "preempt" temporal transitions to failure states. Transition τ_1 preempts τ_2 in state s if it can be proven that τ_1 will always be triggered before τ_2, independent of the history of state transitions. This immediately rules out plans that have event transitions to failure states. Because event transitions can trigger instantaneously, no other transition can preempt them. Safety of a plan can be verified by applying certain correctness-preserving model transformations, pruning out unreachable states (Musliner *et al.* 1995). As was shown above, S_0, $min\Delta$, and $max\Delta$ can be extracted from the probabilistic world model, enabling us to verify probabilistic plans using the same technique. This would, however, be a waste of all the extra information available to us regarding the stochastic behavior of transitions.

In probabilistic terms, the above technique can only distinguish between zero and non-zero probability of reaching a set of states. Plans with non-zero probability of reaching a failure state are considered unsafe. With probability distribution functions available for the transitions, we can set an arbitrary threshold θ representing the highest acceptable failure probability of a plan. Setting $\theta = 0$ we revert to the old model. With $\theta > 0$, though, we can accept plans that would have otherwise been discarded. For example, it now becomes possible to have a plan with event transitions to failure states provided that the events represented by these transitions are sufficiently infrequent, or the probability of entering a states in which such events are enabled is sufficiently low.

As an example, consider the plan in Figure 2(b). The "begin evasive" action preempts the "hit" transition in the bottom state, but the "hit" transition can still be triggered in the center state. If, for example, the lifetime of "begin evasive" is 50 time units, the lifetime of "safe" is 100 time units, and the lifetime of "hit" is 120 time units, then "hit" will trigger before "safe" in the center state, causing a transition to the failure state. The plan in Figure 2(c) is the same as in Figure 2(b), but the intervals have been substituted for probability distribution functions. We will see later that this plan is acceptable with a failure threshold of 0.05, and an upper limit on the execution time set to 200 time units.

We use an acceptance sampling algorithm to probabilistically determine if a plan should be accepted. The samples used by the algorithm are sample execution paths generated through discrete event simulation. A plan is acceptable if the probability of reaching a failure state within some time limit $t_{\max}$ is below the failure threshold θ. The failure probability of a plan will typically depend on $t_{\max}$. For example, the plan in Figure 2(c) has a zero failure probability if $t_{\max}$ is less than 120, but the failure probability approaches 1 as $t_{\max}$ approaches infinity. We assume that a natural time limit is given by the application.

Generating Sample Execution Paths

A plan π, when executed, represents a stochastic process $\{X_\pi(t) \mid t \geq 0\}$, where $X_\pi(t)$ is the state of the world t time units after the plan is set in action. We are interested

in determining whether the probability of visiting a failure state $s \in S_F$ within a specified time limit $t_{\max}$ from the start of the execution of π is below a given threshold θ.

For any one state s, the probability of visiting s before time $t_{\max}$ is $\Pr[\inf\{t \mid X_\pi(t) = s\} \leq t_{\max}]$. For a set of absorbing states, such as the set of failure states S_F, at most one state in the set can be visited during execution. Furthermore, if $X_\pi(t) = s$ for an absorbing state s at time t, then $X_\pi(t') = s$ for all $t' \geq t$. The failure probability of a plan after $t_{\max}$ time units is therefore

$$\Pr[\inf\{t \mid X_\pi(t) \in S_F\} \leq t_{\max}] = \sum_{s \in S_F} \Pr[X_\pi(t_{\max}) = s].$$

Depending on the stochastic characteristics of the process, an analytical calculation of the failure probability may be impossible, and the only feasible approach may be to use sampling techniques that make use of simulation to generate sample paths (Heidelberger 1995).

Let us define the random variable $Y_\pi(s)$ as follows:

$$Y_\pi(s) = \begin{cases} 1 & \text{if } \inf\{t \mid X_\pi(t) = s\} \leq t_{\max} \\ 0 & \text{otherwise} \end{cases}$$

Clearly, $Y_\pi(s)$ is a binomial variate with parameters $1, p_s$; p_s being the probability that state s is visited within $t_{\max}$ time units. Given n samples $y_1(s)$ through $y_n(s)$ of $Y_\pi(s)$, let $x_n(s) = |\{y_i(s) \mid y_i(s) = 1\}|$. This is the number of samples in which state s is visited within $t_{\max}$ time units. A point estimate of p_s is $x_n(s)/n$. This estimate can be used as a heuristic to guide the effort of the planner towards the most likely states (cf. (Atkins *et al.* 1996)). Note that $x_n(s)$ can be computed as $\sum_{i=1}^{n} y_i(s)$. Let f_n denote the number of failures observed in n samples. Clearly

$$f_n = \sum_{s \in F} \sum_{i=1}^{n} y_i(s). \tag{1}$$

Each sample $y_i(s)$ is generated using discrete event simulation. Algorithm 1 outlines a procedure for generating $y_i(s)$ for each state $s \in S' \subset S$ simultaneously (cf. Algorithm 4.17 of (Shedler 1993)).

The sample generation algorithm does not consider the case of two transitions being triggered simultaneously. If all distribution functions F are continuous, the probability of this happening is in fact zero. Yet when implementing the sampling algorithm on a computer, where real numbers are represented by finite precision floating-point numbers, the occurrence of simultaneous transitions becomes an issue we have to deal with. We can address this by modifying step 4, so that instead of simply letting $s' = \tau^*(s)$, we select s' with uniform probability from the set of transitions with the shortest residual lifetime in s.

Algorithm 1 Procedure for generating $y_i(s)$ for all $s \in S' \subset S$ simultaneously.

1. Let $t = 0$ and $y_i(s) = 0$ for all $s \in S'$. Generate an initial state s in accordance with the probability distribution p_0.
2. For each transition $\tau \in E(s)$, sample a residual lifetime $r_{s,\tau}$ (cf. (Glynn 1989)) according to the distribution function $F(\cdot;\tau)$. This is the amount of time remaining until τ triggers a transition out of state s.
3. Set $y_i(s)$ to 1. Terminate the simulation if $s \in S_F$ or $E(s) \cap T_A = \emptyset$.
4. Let τ^* be the transition with the shortest residual lifetime in s. Terminate the simulation if $t + r_{s,\tau^*} > t_{\max}$. Otherwise, let $s' = \tau^*(s)$.
5. Generate new residual lifetimes for transitions $\tau \in E(s')$ as follows:
   - If $\tau = \tau^*$ or $\tau \notin E(s)$, then sample $r_{s',\tau}$ according to $F(\cdot;\tau)$.
   - Otherwise, let $r_{s',\tau} = r_{s,\tau} - r_{s,\tau^*}$.
6. Set s to s' and t to $t + r_{s,\tau^*}$, and go to step 3.

Acceptance Sampling

A plan π, when executed, can either fail or succeed. We denote the probability of failure by p_F. As alluded to earlier, we can specify a positive threshold θ representing the maximum acceptable failure probability. If p_F does not exceed θ we are willing to accept π, but would reject it otherwise. Deciding whether to accept or reject a plan can be cast as the problem of testing the hypothesis $p_F \leq \theta$ against the alternative hypothesis $p_F > \theta$. This is an important problem in manufacturing industry and engineering, and has been studied thoroughly in the field of statistical quality control (cf. (Chorafas 1960; Montgomery 1991)). The problem also arises in the area of software certification (Poore *et al.* 1993).

Risk Tolerance and Hypothesis Testing. We would ideally like to accept only those plans with a failure probability no larger than θ and reject all other plans. In general, however, we cannot calculate the failure probability of a plan analytically, nor can we determine it with absolute certainty if falling back on sampling techniques. In the latter case the reason is the potentially infinite sample space. Therefore we must tolerate a certain risk of rejecting a plan with true failure probability at most θ, or accepting a plan with failure probability above θ. In statistical quality control the former kind of error is referred to as a type I error (reject when acceptable), and the latter a type II error (accept when rejectable). We associate a risk level with each type of error. The risk levels are denoted by α and β respectively, and these represent the acceptable probability of making an error of respective type.

Let H_0 be the hypothesis that $p_F \leq \theta$ (*null hypothesis*), and let H_1 be the alternative hypothesis that $p_F > \theta$. We would like to test the hypothesis H_0 against H_1 so that the probability of accepting H_1 when H_0 holds is at most α, and the probability of accepting H_0 when H_1 holds is at most β.

In order to be able to choose α and β freely, however, we need to relax the hypotheses somewhat.[1] For this pur-

[1]We would have to choose $\alpha = 1 - \beta$ without the suggested

pose we introduce an indifference region of non-zero width δ. Let $p_F \leq \theta - \delta$ be H_0 and let $p_F \geq \theta + \delta$ be H_1. We use acceptance sampling to test hypothesis H_0 against H_1. The motivation for the indifference region, other than that it allows us to choose the two risk levels independently, is that if the true failure probability is sufficiently close to the threshold, then we are indifferent to whether the plan is accepted or rejected.

Sequential Sampling. A sequential test is one where the number of observations is not predetermined but is dependent on the outcome of the observations (Wald 1945). Wald (*loc. cit.*) develops the theory of *sequential analysis*, and defines the sequential probability ratio test (see also (Wald 1947)), which is optimal for testing a simple hypothesis against a simple alternative in the sense that it minimizes the expected number of samples needed to reach a decision.

Let X be a binary random variable with unknown parameter p such that $\Pr[X = 1] = p$. The sequential probability ratio test is carried out as follows to test the hypothesis H_0 that $p \leq \theta - \delta$ against the hypothesis H_1 that $p \geq \theta + \delta$. At each stage of the test, calculate the ratio

$$\frac{p_{1n}}{p_{0n}} = \frac{\prod_{i=1}^{n} \Pr[X = x_i \mid p = \theta + \delta]}{\prod_{i=1}^{n} \Pr[X = x_i \mid p = \theta - \delta]},$$

where x_i is the sample of X generated at stage i. Accept H_1 if

$$\frac{p_{1n}}{p_{0n}} \geq \frac{1-\beta}{\alpha}.$$

Accept H_0 if

$$\frac{p_{1n}}{p_{0n}} \leq \frac{\beta}{1-\alpha}.$$

Otherwise, generate an additional sample and repeat the termination test. This test procedure respects the risk levels α and β.[2]

Let $\theta_0 = \theta - \delta$ and $\theta_1 = \theta + \delta$. Applied to the problem of validating a plan, if at stage n we have observed f_n failures, the ratio to compute is

$$\frac{p_{1n}}{p_{0n}} = \frac{\theta_1^{f_n}(1-\theta_1)^{n-f_n}}{\theta_0^{f_n}(1-\theta_0)^{n-f_n}}.$$

For purposes of practical computation we work with logarithms, and carry out the test as follows. At the inspection of the nth sample, compute

$$\log \frac{p_{1n}}{p_{0n}} = f_n \log \frac{\theta_1}{\theta_0} + (n - f_n) \log \frac{1-\theta_1}{1-\theta_0}.$$

Continue sampling if

$$\log \frac{\beta}{1-\alpha} < \log \frac{p_{1n}}{p_{0n}} < \log \frac{1-\beta}{\alpha}.$$

Terminate by accepting hypothesis H_1 if

$$\log \frac{p_{1n}}{p_{0n}} \geq \log \frac{1-\beta}{\alpha}.$$

relaxation, which means if one of the risk levels was low, then the other would have to be high.

[2]There is a slight approximation involved in the stopping criteria of the test. See (Wald 1945) for details.

Terminate by accepting H_0 if

$$\log \frac{p_{1n}}{p_{0n}} \leq \log \frac{\beta}{1-\alpha}.$$

Alternatively, we can compute an acceptance number a_n and a rejection number r_n. We accept H_0 if $f_n \leq a_n$, reject H_0 (accept H_1) if $f_n \geq r_n$, and continue sampling otherwise. Let

$$u = \log \frac{\theta_1}{\theta_0} \quad \text{and} \quad v = \log \frac{1-\theta_0}{1-\theta_1}.$$

The acceptance number at stage n is

$$a_n = \frac{\log \frac{\beta}{1-\alpha} + nv}{u+v}, \tag{2}$$

and the rejection number is

$$r_n = \frac{\log \frac{1-\beta}{\alpha} + nv}{u+v}. \tag{3}$$

Verification Algorithm

We now have all the pieces needed to specify a plan verification algorithm. Algorithm 2 describes the steps of the procedure. The algorithm uses Wald's sequential probability ratio test, and so has input parameters θ, δ, α, and β. In addition, the parameter $t_{\max}$ needs to be specified for the sample generation algorithm used by the verification procedure.

Algorithm 2 Procedure for verifying plan π.

1. Let $n = 0$.
2. Increment n by one and generate samples $y_n(s)$ of $Y_\pi(s)$ for all $s \in S_F$ (Algorithm 1).
3. Compute f_n (equation (1)).
4. Compute a_n (equation (2)) and r_n (equation (3)).
5. Accept π if $f_n \leq a_n$, and reject π if $f_n \geq r_n$. Otherwise go to step 2.

Figure 3 graphically represents the execution of the verification algorithm on the plan in Figure 2(c) using parameters $\theta = 0.05$, $\delta = 0.01$, $\alpha = \beta = 0.05$, and with $t_{\max}$ set to 200 time units. The acceptance and rejection lines correspond to equations 2 and 3 respectively. The curve starting out between the two lines represents the number of observed failures. After generating 201 sample execution paths (of which 3 ended in a failure state), the curve crosses the acceptance line, which means we accept the plan. Had the curve crossed the rejection line instead, we would have rejected the plan. With the given parameters, we are 95% confident that the true failure probability of the plan is less than 0.06.

The number of samples n that the algorithm needs to inspect before a decision is reached does not have a definite upper bound. Wald (1947) proves that the sequential probability ratio test terminates with probability 1. Although the

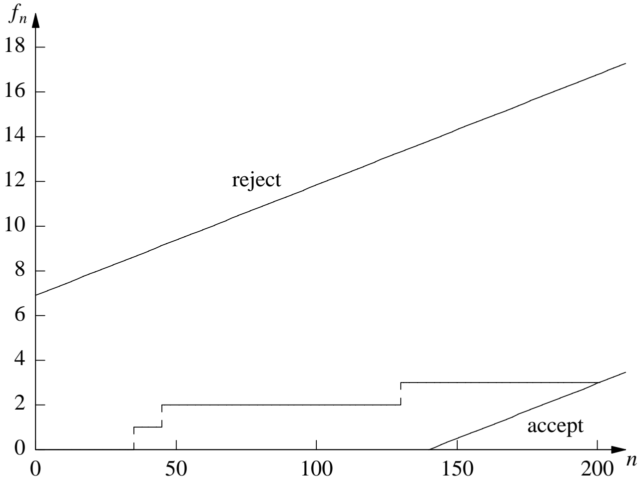


Figure 3: Result of executing the verification algorithm on the plan in Figure 2(c).

probability is small that the required sample size will exceed twice or three times the expected number of required samples, it may be desirable to set an upper bound $n_{\max}$ in some cases. If an upper bound is provided and the test does not terminate for $n \leq n_{\max}$, Wald (*loc. cit.*) suggests that the null hypothesis be rejected if $f_{n_{\max}} \geq (a_{n_{\max}} + r_{n_{\max}})/2$ and accepted otherwise. If the upper bound is set sufficiently high (e.g. three times the expected value of n), then truncating the process has negligible effect on the strength of the test.

Performance

The performance of our verification algorithm depends on several factors. We can separate these factors into two groups—domain dependent and domain independent.

The domain dependent factors affect the performance of Algorithm 1, used for generating the samples $y_i(s)$. The main factors of this kind are the time period considered ($t_{\max}$) and the mean values of the distribution functions F. If the mean values are small relative to $t_{\max}$, the number of transitions triggering before the simulation terminates will be high, hence increasing the time needed to generate each set of samples. Note, however, that the size of the state space plays a minimal role. Only once, in the initialization step, do we need to perform work at most linear in $|S|$. This work amounts to clearing all the $y_i(s)$'s, which can be done quite efficiently using a bit-vector to represent each set of samples.

As a domain independent factor we view the number of samples, n, needed to be generated before the verification algorithm terminates. We will show below how this number depends on the true failure probability p_F. Although arguably dependent on the domain and the current plan π, because this ultimately determines p_F, we can estimate the sample size needed independently of any particular domain or plan, hence motivating the label "domain independent".

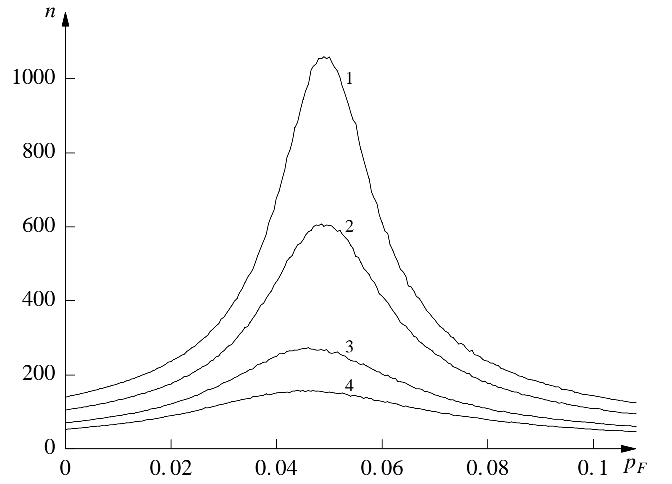


Figure 4: Average number of samples for $\delta = 0.01$ and (1) $\alpha = \beta = 0.05$ and (2) $\alpha = \beta = 0.10$, and for $\delta = 0.02$ with (3) $\alpha = \beta = 0.05$ and (4) $\alpha = \beta = 0.10$ ($\theta = 0.05$ in all cases).

Sample Size

We can expect to need fewer samples the further the true failure probability is from the indifference region. If p_F is significantly less than θ_0, we can expect to satisfy the acceptance criterion at an early stage. Conversely, if p_F is much above θ_1, the number of failures observed will tend to quickly exceed the rejection number. The number of samples, n, required by the verification algorithm is a random variable since it depends on the outcome of the observations. The expected value of n, often called the *average sample number*, depends on p_F.

Wald (1945) provides an approximation formula for the expected sample size. Let $\mathrm{E}_p[n]$ denote the expected number of samples required given $p_F = p$. An approximate value for this expectation is

$$\tilde{\mathrm{E}}_p[n] = \frac{L(p)\log\dfrac{\beta}{1-\alpha} + (1-L(p))\log\dfrac{1-\beta}{\alpha}}{p\log\dfrac{\theta_1}{\theta_0} + (1-p)\log\dfrac{1-\theta_1}{1-\theta_0}}, \qquad (4)$$

where $L(p)$ is the probability that the sequential test terminates with acceptance if $p_F = p$.

Figure 4 plots the average sample number in a region close to θ (for $\theta = 0.05$) and with different choices of δ, α, and β. We can see that by raising the risk levels or widening the indifference region, we will need fewer runs on average to reach a decision. This gives us an opportunity to trade quality for performance.

Truncated Test. As mentioned earlier, we may want to set an upper limit $n_{\max}$ on the number of samples generated. Equation (4) can help us choose this upper bound. The average sample number is at a maximum at, or close to, the

common slope s of the acceptance and rejection lines:

$$s = \frac{\log \dfrac{1-\theta_0}{1-\theta_1}}{\log \dfrac{\theta_1}{\theta_0} - \log \dfrac{1-\theta_1}{1-\theta_0}}$$

The average sample number at this point is approximately

$$\tilde{\mathrm{E}}_s(n) = \frac{\log \dfrac{\beta}{1-\alpha} \log \dfrac{1-\beta}{\alpha}}{\log \dfrac{\theta_1}{\theta_0} \log \dfrac{1-\theta_1}{1-\theta_0}}.$$

With $n_{\max} = 3\tilde{\mathrm{E}}_s(n)$, the probability that the sequential test has terminated before n reaches $n_{\max}$ is nearly 1, and the truncation has a negligible effect on the strength of the test.

Related Work

BURIDAN uses a notion of plan failure similar to ours, where a threshold is given representing the maximum acceptable failure probability (Kushmerick *et al.* 1995). BURIDAN implements several methods for plan assessment, computing a guaranteed upper bound on the failure probability. The cost of obtaining a guaranteed bound is that the efficiency of these methods vary significantly between domains. In our approach, we can trade efficiency for accuracy by adjusting the risk levels. A further difference is that our world model allows for external events, while in BURIDAN only actions can be represented, and there is only a limited notion of time, where each performed action represents a discrete time step. The same holds for planners adopting a model based on Markov decision processes.

The work by Dean & Kanazawa (1989) is more closely related to ours. They use what they call probabilistic projection to reason about persistence of propositions, but their model is Markovian. They construct a belief network in order to compute probabilistic predictions. Blythe (1994) uses a similar approach for computing the failure probability of a plan subject to external events. Probabilistic inference in belief networks is known to be NP-hard (Cooper 1990), however, and current exact algorithms have worst-case exponential behavior. Both Blythe and Dean & Kanazawa consider approximate algorithms, but they do not provide any guaranteed error bounds, and the convergence is often slow.

Atkins, Durfee, & Shin (1996) consider a probabilistic extension of CIRCA similar to ours. Their approach is analytical, and they present an iterative algorithm for state probability estimation. Their state probability calculations are based on heuristic approximations of transition times, but no quantitative error bounds are provided by the algorithm. Furthermore, they do not propagate probabilities around cycles in the state space, which can lead to serious underestimation (although this problem is addressed in later work (Li *et al.* 2000)).

Alur, Courcoubetis, & Dill (1991) describe an algorithm for verifying formulas specified in a language called TCTL, with an underlying GSMP world model. TCTL is a branching-time temporal logic for expressing real-time properties, but lacks support for expressing quantitative bounds on probabilities. Aziz *et al.* (1996) present CSL (continuous stochastic logic), which is a formalism in which quantitative probability bounds can be expressed, and they show that the problem of verifying CSL formulas is decidable. Baier, Katoen, & Hermanns (1999) describe an implementation of a model checker using an analogous formalism. The underlying model for this work is continuous-time Markov chains, however, and not GSMPs. The difference is that in the former model only exponential probability distribution functions are permitted.

Conclusions

We have presented a probabilistic extension to CIRCA. The extended world model can be viewed as a generalized semi-Markov process. We use discrete event simulation to generate sample paths in a world model, and use acceptance sampling theory to minimize the expected number of samples needed to determine if the failure probability is sufficiently low. Our work differs from other probabilistic planners in that our world model is more expressive. Yet, we are able to bound the fraction of erroneous classifications that our plan verifier makes. Using sequential acceptance sampling, we often need very few samples to reach a decision with sufficient confidence, but the user (or a higher-level deliberation scheduling module) can easily trade efficiency for accuracy by varying the parameters δ, α, and β.

For future work, we would like to combine importance sampling (Heidelberger 1995) with acceptance sampling. Doing so could improve performance of our verification algorithm when the failure threshold is close to zero, or when the indifference region is narrow. We are also interested in developing techniques for ananlyzing how sensitive the failure probability of a plan is to variations in the time limit $t_{\max}$. In addition, more work on how to use probabilistic information in guiding plan generation is needed. We have mentioned how to obtain point estimates of state probabilities from sample execution paths, and that these could be used to guide the planning effort towards more likely states, effectively pruning the least likely states from the search space if limited resources are given to the planner. We would like to investigate the effectiveness of such pruning techniques in the future.

Acknowledgments. This material is based upon work supported by DARPA/ATO and the Space and Naval Warfare Systems Center, San Diego, under contract no. N66001–00–C–8039. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, the U.S. Government, or the Space and Naval Warfare Systems Center, San Diego. Approved for public release; distribution is unlimited.

References

Alur, R.; Courcoubetis, C.; and Dill, D. 1991. Model-checking for probabilistic real-time systems. In Albert, J. L.; Monien, B.; and Artalejo, M. R., eds., *Proceedings*

of the 18th International Colloquium on Automata, Languages and Programming, volume 510 of *Lecture Notes in Computer Science*, 115–126. Madrid, Spain: Springer.

Atkins, E. M.; Durfee, E. H.; and Shin, K. G. 1996. Plan development using local probabilistic models. In Horvitz, E., and Jensen, F. V., eds., *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, 49–56. Portland, OR: Morgan Kaufmann Publishers.

Aziz, A.; Sanwal, K.; Singhal, V.; and Brayton, R. 1996. Verifying continuous time markov chains. In Alur, R., and Henzinger, T. A., eds., *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, 269–276. New Brunswick, NJ: Springer.

Baier, C.; Katoen, J.-P.; and Hermanns, H. 1999. Approximate symbolic model checking of continuous-time Markov chains. In Baeten, J. C. M., and Mauw, S., eds., *Proceedings of the 10th International Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, 146–161. Eindhoven, the Netherlands: Springer.

Blythe, J. 1994. Planning with external events. In de Mantaras, R. L., and Poole, D., eds., *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, 94–101. Seattle, WA: Morgan Kaufmann Publishers.

Chorafas, D. N. 1960. *Statistical Processes and Reliability Engineering*. Princeton, NJ: D. Van Nostrand Company.

Cooper, G. F. 1990. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence* 42(2–3):393–405.

Dean, T., and Kanazawa, K. 1989. A model for reasoning about persistence and causation. *Computational Intelligence* 5(3):142–150.

Glynn, P. W. 1989. A GSMP formalism for discrete event systems. *Proceedings of the IEEE* 77(1):14–23.

Heidelberger, P. 1995. Fast simulation of rare events in queueing and reliability models. *ACM Transactions on Modeling and Computer Simulation* 5(1):43–85.

Kushmerick, N.; Hanks, S.; and Weld, D. S. 1995. An algorithm for probabilistic planning. *Artificial Intelligence* 76(1–2):239–286.

Li, H.; Atkins, E. M.; Durfee, E. H.; and Shin, K. G. 2000. Resource allocation for a limited real-time agent using a temporal probabilistic world model. In *Working Notes of the AAAI Spring Symposium on Real-Time Autonomous Systems*.

Matthes, K. 1962. Zur Theorie der Bedienungsprozesse. In Kožešník, J., ed., *Transactions of the Third Prague Conference on Information Theory, Statistical Decision Functions, Random Processes*, 513–528. Liblice, Czechoslovakia: Publishing House of the Czechoslovak Academy of Sciences.

Montgomery, D. C. 1991. *Introduction to Statistical Quality Control*. New York, NY: John Wiley & Sons, second edition.

Musliner, D. J.; Durfee, E. H.; and Shin, K. G. 1993. CIRCA: A cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics* 23(6):1561–1574.

Musliner, D. J.; Durfee, E. H.; and Shin, K. G. 1995. World modeling for the dynamic construction of real-time control plans. *Artificial Intelligence* 74(1):83–127.

Poore, J. H.; Mills, H. D.; and Mutchler, D. 1993. Planning and certifying software system reliability. *IEEE Software* 10(1):88–99.

Schoppers, M. J. 1987. Universal plans for reactive robots in unpredictable environments. In McDermott, J., ed., *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1039–1046. Milan, Italy: Morgan Kaufmann Publishers.

Shedler, G. S. 1993. *Regenerative Stochastic Simulation*. Boston, MA: Academic Press.

Wald, A. 1945. Sequential tests of statistical hypotheses. *Annals of Mathematical Statistics* 16(2):117–186.

Wald, A. 1947. *Sequential Analysis*. New York, NY: John Wiley & Sons.