Appears in Proc. Int'l Conf. on Automated Planning and Scheduling
Whistler, British Columbia, Canada, June 2004

Guiding Planner Backjumping Using Verifier Traces

Robert P. Goldman
SIFT, LLC
rpgoldman@sift.info

Michael J. S. Pelican, David J. Musliner
Honeywell Laboratories
{pelican,musliner}@htc.honeywell.com

Abstract

In this paper, we show how a planner can use a model-checking verifier to guide state space search. In our work on hard real-time, closed-loop planning, we use a model-checker's reachability computations to determine whether plans will be successfully executed. For planning to proceed efficiently, we must be able to efficiently repair candidate plans that are not correct. Reachability verifiers can return counterexample traces when a candidate plan violates desired properties. We describe a technique that automatically extracts repair candidates from counterexample traces. We map counterexample traces onto a search algorithm's choice stack to direct backjumping. We prove that our technique will not sacrifice completeness, and present empirical results showing substantial performance improvements in difficult cases. Our results can be applied to other applications, such as automatic design, and manufacturing scheduling.

Introduction

In this paper, we show how a planning system can efficiently use a reachability-computing verification system in state space search. In our work on hard real-time, closed-loop planning, we use a model-checker's reachability computations to determine whether plans will be successfully executed. For planning to proceed efficiently, we must be able to efficiently repair candidate plans that are not correct. We show that error traces from reachability computations may be mapped onto a search algorithm's choice stack to direct backjumping, and present empirical results that show substantial performance improvements in difficult cases. Our results can be applied to other applications, such as automatic design, and manufacturing scheduling.

We use timed automaton verification as an integral, on-line component in our planning system, the CIRCA Controller Synthesis Module (CSM) (Musliner, Durfee, & Shin 1995). Unlike classical planners, CIRCA plans for an agent that operates in a dynamic environment, concurrently with uncontrolled processes. Therefore, our plans will be *closed-loop* policies: given a plan, the CIRCA agent will choose its actions by observing its environment and taking the action the plan prescribes. The CSM also differs from MDP planners and conditional planners in that its environment is assumed to evolve *asynchronously* and in *continuous time*: the CIRCA agent does not "take turns" with its environment. This means that we must concern ourselves with how long actions will take, and whether they can be completed in time to forestall (or *preempt*) uncontrolled processes that threaten the CIRCA agent. The CIRCA CSM is based on heuristic state-space search, and uses a model-checking verifier to determine whether its plans will be successful.

To limit search size and to simplify analysis and implementation of the executive, the planner reasons in a time-abstract state space: planner states are associated only with propositions, not with temporal information. Were we to include temporal information with the states, the state space would explode. If there are n processes, each taking up to k time units to evolve, even a discrete time representation would suffer up to a k^n factor state space explosion. This is why we do not plan directly with the verifier, unlike other researchers who work in untimed domains (e.g., (Giunchiglia & Traverso 1999)). Furthermore, we might generate plans that could not be realized: for example, a plan where we take action A when a given clock has *exactly* the value x, but action B when the clock is not equal to x. Since time is dense, this plan is not realizable.

While the planner reasons about states that ignore time, temporal information is critical to determining whether a given plan is correct. We use a very efficient, special-purpose model-checking verifier to check state space characteristics that depend on global temporal analysis. In particular, we use the model-checker to assess state reachability, including the reachability of failure states, which are sink states that indicate unrecoverable failures.

When the verifier finds that failure is reachable, it can return a trace illustrating a path to failure. By mapping this failure trace onto the search stack choice points, our planner is able to pinpoint the decisions that are responsible for failure, and *backjump* to revise the most recent implicated decision. This backjumping avoids revisiting more-recent but irrelevant decisions, and can considerably improve the efficiency of the search *without*

sacrificing completeness.

Our backjumping method was developed for real-time planning in the context of the CIRCA intelligent control architecture (Musliner, Durfee, & Shin 1995). However, it should be generally useful in any planning method that uses a verifier as an external correctness oracle (e.g., (Tripakis & Altisen 1999)). It should also be possible to adapt our method to the needs of AI automatic design applications that use verifiers to check correctness. There has also been work on using temporal automaton reachability computations for manufacturing scheduling applications (Hune, Larsen, & Pettersson 2001); our approach should be helpful in such scheduling applications as well.

We begin the rest of the paper by briefly reviewing our planning approach in the CIRCA Controller Synthesis Module. We then review backjumping, a technique for directed backtracking that is guaranteed to search no more nodes than chronological backtracking, without sacrificing the complete enumeration of consistent (*i.e.*, safe) solutions. The correct behavior of backjumping depends on correctly formulating eliminating explanations, or "nogoods," when an inconsistency is detected in the search process. After describing backjumping, we present the method for extracting nogoods from counterexample traces produced by a verifier (the core contribution of this paper). We present performance results from several domains illustrating the resulting reduction in search. We conclude with comparisons with related work and some summary remarks.

CIRCA Controller Synthesis

CIRCA's Controller Synthesis Module (CSM) automatically synthesizes real-time reactive discrete controllers that guarantee system safety when executed by CIRCA's Real-Time Subsystem (RTS), a reactive executive with limited memory and no internal clock. The CSM takes in a description of the processes in the system's environment, represented as a set of transitions that modify world features. Transitions have preconditions, describing when they are applicable, and bounded delays, capturing the temporal characteristics of controllable processes (i.e., actions) and uncontrollable processes (i.e., world dynamics). For example, Figure 1 shows several transitions from a CIRCA problem description for controlling the Cassini spacecraft during Saturn Orbital Insertion (Gat 1996; Musliner & Goldman 1997). Discrete states of the system are modeled as sets of feature-value assignments. Thus the transition descriptions, together with specifications of initial states, implicitly define the set of possible system states.

The CSM reasons about both controllable and uncontrollable transitions:

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

```
;; Turning on the main engine
ACTION turn_on_main_engine
   PRECONDITIONS: '((engine off))
   POSTCONDITIONS: '((engine on))
   DELAY: <= 1

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

;; If the engine is burning while the 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
```

Figure 1: Example transition descriptions.

Temporal (uncontrollable) transitions represent uncontrollable processes. 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. Transitions whose postconditions include the proposition `(failure T)` are called *temporal transitions to failure* (TTFs).

Note that each transition is an implicit description of many transitions in an automaton model. Each of these transitions is enabled in any discrete state that satisfies its preconditions, and disabled everywhere else.

If a temporal transition leads to an undesirable state, the CSM may plan an action to *preempt* the temporal:

Definition 1 (Preemption). *A temporal transition may be preempted in a (discrete) state by planning an action for that state which will* necessarily *occur before the temporal transition's delay can elapse.*

Note that successful preemption does not ensure that the threat posed by a temporal transition is handled; it may simply be postponed to a later state (in general, it may require a sequence of actions to handle a threat). A threat is handled by preempting the temporal with an action that carries the system to a state which does *not* satisfy the preconditions of the temporal.

The controller synthesis (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 ensures that the synthesized controllers are actually realizable in the RTS.

Algorithm 1 (Controller Synthesis)

1. *Choose a state from the set of reachable states (at the start of controller synthesis, only the initial states 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 single control action or* `no-op` *for this 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 backjumping.*
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 yet been chosen), terminate successfully.*
8. *If some unplanned reachable states remain, loop to step 1.*

The search algorithm maintains the decisions that have been made, along with the potential alternatives, on a search stack. The algorithm makes decisions at two points: step 2 and step 3.

The CSM uses the verifier module after each assignment of a control action (see step 4). The verifier is used to confirm both that failure is unreachable *and* that all the chosen preemptions will be enforced. 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. We describe this process in detail in the following sections.

Backjumping

Although the search algorithm includes heuristic guidance it may still need to explore a number of possible controller designs (action assignments) before finding a safe controller. If the heuristic makes a poor decision at a state, the search process will lead to dead ends, and it must back up to that state and resume searching with a different decision. The simplest approach to "backing up" is *chronological backtracking*: undoing the most recent decision in the search and trying an alternative. However, in some problems, it is possible to determine that the most recent decision was not relevant to reaching the dead end. Backjumping exploits such information by skipping over irrelevant decisions and backtracking directly to the most recent *relevant* decision.

An example taken from the CIRCA controller synthesis problem may help understand why backjumping is useful. Consider the problem shown in Figure 2. The search algorithm has assigned control actions to the states in the order indicated by the state numbering. The planned actions and the un-preempted temporal transitions are shown by the heavy lines (dashed for actions, double-solid for temporals). At this point, the system has planned an action for state 1 that will take it to state 2, an action for state 2 that will take it to state 3, and a no-op action for state 3. Unexplored alternatives are shown as fainter dotted lines. There are two alternative actions for state 1 and three for state 2. The planner has also permitted a nonvolitional transition to carry the system from the initial state to state 4.

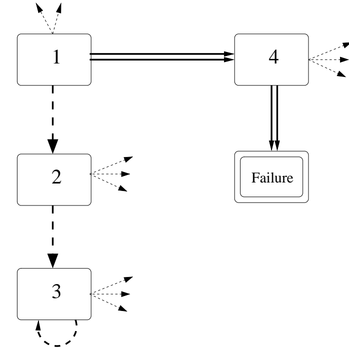


Figure 2: An example showing the utility of backjumping.

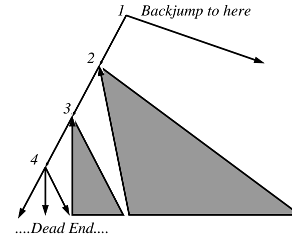


Figure 3: A search tree corresponding to the controller synthesis problem in Figure 2

Unfortunately, when trying to choose an action for state 4, we will reach an impasse. There is a nonvolitional temporal transition leading from state 4 to **Failure**, and none of the action choices applicable in state 4 are fast enough to preempt the TTF. We have reached a dead-end in our search, and must back up.

A conventional, chronological backtracking algorithm would now return to state 3 and attempt to assign a new control action to it. However, it should be clear from Figure 2 that this is a waste of time. No revision to the choices for states 2 or 3 will solve the problem, and it will take us an arbitrarily large amount of time to

find this out. It would be far better for us to simply "jump" back to state 1 and try to find a better solution to the problem posed by that state, in this case by preempting the temporal to state 4. A search tree corresponding to this problem is given as Figure 3. The wasted search is shown as shaded triangles. Note that the subtrees corresponding to these triangles may be arbitrarily deep.

Backjumping makes this kind of intelligent, guided revision possible. It can be shown that backjumping is complete and never expands more nodes than depth-first search. Backjumping was developed by Gaschnig (1979), but our discussion follows the presentation by Ginsberg (1993). We have modified Ginsberg's discussion somewhat to make it fit our search algorithm more closely.

Definition 2 (Constraint Satisfaction Problem (CSP)). *A constraint satisfaction problem is a tuple, (I, V) with I a set of variables; for each $i \in I$ there is a set $V_i = \{v_{i,0}, v_{i,1} \ldots v_{i,n_i}\}$ of possible values for the variable i. There are some constraints that limit the acceptable assignments of values to variables.*

In the case of the CIRCA controller synthesis problem, the variables in question are the preemption decisions and the action assignments. The constraints are implicitly defined by the scenario definition, and whether or not an assignment is consistent is determined by consulting the timed automaton verifier. E.g., in Figure 2 there is no assignment to the action variable for state 4 that is consistent with the assignment of *not-preempted* to the nonvolitional from state 1 to state 4.

Definition 3 (Partial Solution). *Let (I, V) be a CSP. By a partial solution to the CSP, we mean an ordered subset $J \subseteq I$, and an assignment of a value to each $i \in J$. A partial solution corresponds to a tuple of ordered pairs, where each ordered pair $\langle i, v\rangle$ assigns the value v to i. For a partial solution, P, we will write $\overline{P}$ for the set of variables assigned values by P.*

In general, we cannot assign arbitrary values to variables — some of the values are eliminated by constraints:

Definition 4 (Eliminating Explanation). *Given a partial solution P to a CSP, an eliminating explanation for a variable i is a pair $\langle v, P\rangle$ where $v \in V_i$ and $S \subseteq \overline{P}$. That is, i cannot take the value v because of the values assigned by P to the variables in S. The set of eliminating explanations for i is E_i.*

Definition 5 (Solution Checker). *A solution checker for a CSP is a function $\mathcal{C} : P, i, v \rightarrow \{\top\} \cup E_i$. That is, the solution checker, given a partial assignment P and an assignment to a variable $i \notin P$ will return either $\top$ (the assignment satisfies all complete constraints), or will return an eliminating explanation for $\langle i, v\rangle$.*

Now we can describe chronological backtracking depth-first search using the preceding definitions. We present this to provide a point of comparison that should make backjumping easier to understand. The description of depth-first search using eliminations is simply a dual of the conventional description: instead of the algorithm tracking the remaining values for the variables, this version tracks the values eliminated from the variables' domains.

Algorithm 2 (Elimination-based Depth-first Search) *Given a CSP $\mathcal{P}$ and a solution checker $\mathcal{C}$:*

1. *$P := \emptyset$ and $N_i := \emptyset$ for all $i \in I$. P will record the current partial solution, and N_i is the set of values eliminated from the domain of i at this stage of the search.*
2. *If P is a complete solution, return it.*
3. *Select a variable $i \in I - \overline{P}$.*
4. *Let $S := V_i - N_i$, the set of remaining possibilities for i.*
5. *If $S = \emptyset$, **backtrack**: if $P = \emptyset$ return failure, otherwise, let $\langle j, v_j\rangle$ be the last entry in P. Remove $\langle j, v_j\rangle$ from P, $N_i := \emptyset$, add v_j to E_j and go to step 4.*
6. *If $S \neq \emptyset$, choose a value, $v_{i,k} \in S$ to assign to i.*
7. *If $\mathcal{C}(P, i, v_{i,k}) = \top$ then $P := P \cup \{\langle i, v_{i,k}\rangle\}$ and go to step 3.*
8. *If $\mathcal{C}(P, i, v_{i,k}) \neq top$ then add $v_{i,k}$ to N_i and go to step 4.*

Note that depth first search makes only the most trivial use of the solution checker and the eliminating explanations. Indeed, for the purposes of depth-first search, the solution checker need only be a boolean function, returning either $\top$ or $\bot$.

Now we can present the definition of backjumping. For backjumping we will require the solution checker to provide more specific information.

Definition 6 (Solution Checker for Backjumping). *For an assignment that violates some constraints, we require $\mathcal{C}(P, i, v)$ to return some $E_i(v) \subset \overline{P}$, such that the set of values assigned to the variables $E_i(v)$, taken together with the assignment $\langle i, v\rangle$, violates some constraint of the problem.*

As one would expect, the smaller the explanations, the better. In the worst case, where $E_i(v) = \overline{P}$ for all i and v, backjumping degenerates to depth-first search.

Note that these eliminating explanations can be interpreted as logical implications. Given an eliminating explanation, e.g., $E_i(v) = \{j, k, l\} \subset \overline{P}$ for $\langle i, v\rangle$, we can interpret this as a clause $\neg\langle i, v\rangle \leftarrow \langle j, v_j\rangle, \langle k, v_k\rangle, \langle l, v_l\rangle$, where $\langle j, v_j\rangle, \langle k, v_k\rangle, \langle l, v_l\rangle \in P$. With some abuse of notation, we will write such clauses as: $\neg\langle i, v\rangle \leftarrow E_i(v)$. This interpretation will be helpful in understanding how the backjumping algorithm updates eliminating explanations.

Algorithm 3 (Backjumping) *Given a CSP $\mathcal{P}$ and a solution checker $\mathcal{C}$:*

1. *$P := \emptyset$ and $E_i := \emptyset$ for all $i \in I$. P will record the current partial solution. E_i will be a set of pairs of the form $\langle v_{i,k}, E_i(k)\rangle$ where $v_{i,k} \in V_i$ is a value eliminated from the domain of i and $E_i(k) \subset \overline{P}$ is an eliminating explanation for $\langle i, v_{i,k}\rangle$. The set of variables mentioned in the eliminating explanations for i, is $\overline{E}_i \equiv \bigcup_{\{k|v_{i,k}\in V_i\}} \overline{E}_i(k)$.*
2. *If P is a complete solution, return it.*
3. *Select a variable $i \in I - \overline{P}$.*
4. *Let $S := V_i - E_i$, the set of remaining possibilities for i.*
5. *If $S = \emptyset$, **backjump**: if $\overline{E}_i = \emptyset$ return failure, otherwise, let $\langle j, v_j\rangle$ be the most recent entry in P such that $j \in \overline{E}_i$. Remove $\langle j, v_j\rangle$ from P, $E_i := \emptyset$, add $\langle v_j, \overline{E}_i - j\rangle$ to E_j and go to step 4.*
6. *If $S \neq \emptyset$, choose a value, $v_{i,k} \in S$ to assign to i.*
7. *If $\mathcal{C}(P, i, v_{i,k}) = \top$ then $P := P \cup \{\langle i, v_{i,k}\rangle\}$ and go to step 3.*
8. *If $\mathcal{C}(P, i, v_{i,k}) \neq \top$ then add $\langle v_{i,k}, \mathcal{C}(P, i, v_{i,k})\rangle$ to E_i and go to step 4.*

Remarks To understand the description of an actual backjump, in step 5 of Alg. 3, recall the interpretation of eliminating explanations as implications. When backjumping we have eliminated all elements of V_i, so we have $E_i(v)$, or $\neg\langle i, v\rangle \leftarrow E_i(v)$, for all $v \in V_i$. Implicitly, we also have $\bigvee_{v\in V_i}\langle i, v\rangle$. From these, we can use resolution to conclude $\bigvee_{v\in V_i} E_i(v)$, the E_j update computation performed in step 5.

Eliminating Explanations from Verifier Traces

As we indicated earlier, CIRCA uses a timed automaton verification program as its solution checker (see Def. 5). The key to applying backjumping in our controller synthesis is to be able to translate counterexample traces into eliminating explanations, per Def. 4 and Def. 6.

The model used by the CIRCA Controller Synthesis Module is not directly interpretable by a timed automaton verifier. Since the CIRCA execution system does not use clocks, the CSM reasons only about the discrete state space. Nevertheless, one must reason about clocks in order to determine whether a CIRCA controller is safe. Accordingly, the CIRCA CSM translates its (partial) controllers into timed automata, and then submits these automata to the verifier.

We do not have space here to fully describe this translation process, which we have written about elsewhere (Goldman, Musliner, & Pelican 2002). However, there are three facts about this translation relevant to our discussion here. First, there is a function from the locations of the timed automaton model to the discrete states in the CSM model, CSMstate$(\cdot)$. Second, there is a function from the jumps in the timed automaton model to the transitions of the CSM, CSMtrans$(\cdot)$. These are both computable in constant time. Third, the timed automaton models of our controllers contain a distinguished failure location.

The counterexample traces generated as a result of checking CIRCA plans for safety have the following form:

$$s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots s_n \xrightarrow{t_n} s_{\text{fail}}$$

Each s_i is made up of a location and a clock zone (which represents an equivalence class of clock valuations for the same location). In the following, we will only be concerned with the location. Since each s_i corresponds to a single location, we will not be fussy about the notation. The state s_0 is a distinguished initial state that does not correspond to any CSM state. The state s_1 will map to some CSM initial state. s_{fail} is a state whose location is the distinguished failure location.

The transition $s_n \xrightarrow{t_n} s_{\text{fail}}$ will correspond to one of two classes of failure: either t_n is a transition to failure in the CIRCA model or t_n corresponds to a nonvolitional, *nv* the CSM has chosen to preempt.

We extract an eliminating explanation from the counterexample using a function from state-jump-state triples, $s_i \xrightarrow{t_i} s_{i+1}$ into search decisions. This function is defined over the CSM states and transitions, $\sigma_i \equiv$ CSMstate(s_i), $\sigma_{i+1} \equiv$ CSMstate(s_{i+1}) and $\tau_i \equiv$ CSMtrans(t_i), as shown in Def. 7.

Remarks The first and second pairs of cases are mutually exclusive and covering ($\sigma_{i+1} = s_{\text{fail}}$ versus $\sigma_{i+1} \neq s_{\text{fail}}$) and the cases in each pair are also mutually exclusive and covering.

In this definition, we make use of two classes of search decision: action decisions, $\alpha(\cdot)$, and preemption decisions, $\varpi(\cdot,\cdot)$. Each of these search decisions corresponds to an entry in the search stack. The domain of an action decision, $\alpha(l)$ is the set of actions enabled in state l. Preemption decisions are boolean, $\varpi(l, t) = \top$(resp., $\bot$) means that t must be (need not be) preempted in l.

Using the function of Def. 7, a timed automaton verifier can act as a solution checker for backjumping, per Def. 6. When the verifier returns a counterexample trace, we apply the mapping given in Def. 7 to the trace, remove duplicate decisions, and then remove $\alpha(s)$ from the result to get a nogood. Search using these eliminating explanations proceeds per Alg. 3. Note that the pre-checks (used to avoid unnecessary calls to the verifier) also return eliminating explanations.

There are three conditions required of the solution checker to ensure the soundness and completeness of Alg. 3 (Ginsberg 1993). Our approach meets all three:

1. **Correctness:** If $\mathcal{C}(P, i, v) = \top$ then every *complete* constraint[1] is satisfied by $P \cup \{\langle i, v\rangle\}$.

   Proof: The set of completed constraints is determined by the reachable subspace of the state space.

[1] A constraint is complete if all its participating variables have been assigned a value.

Definition 7. *Eliminating Explanations from Counterexamples*

$$f(\sigma_i, \tau_i, \sigma_{i+1}) = \begin{cases} \{\alpha(\sigma_i)\} & \text{if } \tau_i \text{ is a TTF and } \sigma_{i+1} = s_{fail}. \\ \{\alpha(\sigma_i), \varpi(\sigma_i, \tau_i)\} & \text{if } \tau_i \in \pi(\sigma_i) \text{ and } \sigma_{i+1} = s_{fail}. \\ \{\alpha(\sigma_i)\} & \text{if } \tau_i \text{ is an action or an event} \\ & \text{ and } \sigma_{i+1} \neq s_{fail}. \\ \{\varpi(\sigma_i, \tau_i)\} & \text{if } \tau_i \text{ is a temporal transition} \\ & \text{(and not an event) and } \sigma_{i+1} \neq s_{fail}. \end{cases}$$

That is, a set of variable assignments in the CSM search is consistent iff the sub-space generated by those variable assignments is safe. If the timed automaton verifier we use is correct the correctness condition is met. $\therefore$

2. **Completeness:** Whenever $P \cup \{\langle i, v\rangle\}$ is consistent, $P' \supset P$ and $P' \cup \{\langle i, v\rangle\}$ is *not* consistent, then $\mathcal{C}(P', i, v) \cap (\overline{P'} - \overline{P}) \neq \emptyset$. That is, if P can be extended by assigning v to i and P' cannot be, at least one element of $P' - P$ is identified as a possible reason for the problem.

   Proof: Corresponding to $P \cup \{\langle i, v\rangle\}$ there is a timed automaton subspace that is safe. Corresponding to $P' \cup \{\langle i, v\rangle\}$ there is a timed automaton subspace that is not safe. Ergo, there must be a trace of the following form: $s_0 \overset{t_0}{\to} s_1 \overset{t_1}{\to} \dots s_m \overset{t_m}{\to} \dots s_n \overset{t_n}{\to} s_{\text{fail}}$ where s_m corresponds to a state that was not yet planned in P (note that s_m may or may not be equal to s_n, s_1, or both), otherwise a correct verifier would have found the path when checking $P \cup \{\langle i, v\rangle\}$. Examining the mapping given in Def. 7, we can see that applying that mapping to $s_m \overset{t_m}{\to} s_{m+1}$ will add to the nogood at least one variable in $\overline{P'} - \overline{P}$.$\therefore$

3. **Concision:** Only one reason is given why a particular variable cannot have the value v in a partial solution P.

   Proof: The explanations are generated by our timed automaton verifier. For each safety verification failure, the verifier will generate a single counterexample, from which Def. 7 will generate a single nogood.$\therefore$

It follows that Alg. 1, using backjumping according to the solution checking mechanism proposed here, will be sound and complete.

Test Results

Backjumping is critical to making challenging domains feasible. We can assess the results of trace-directed backjumping by comparing against the CSM performance when backjumping is disabled, and the system only engages in chronological backtracking. Forcing the CSM to use chronological backtracking is fairly simple: we still call the verifier to assess whether the current plan is acceptable, but if the verifier finds failure is reachable we simply backtrack one step and change the last decision on the stack. Note that backjumping is guaranteed never to search *more* states (or decisions) than chronological backtracking, since it only eliminates irrelevant decisions from consideration, never adds new ones. There is some overhead computation involved with maintaining nogoods, and we must evaluate this as well, but there is no way to fool the backjumping mechanism itself into poor decisions or added search.

We evaluated the effectiveness of backjumping on three problem sets. First, we examined its performance on two scalable domains that are constructed specifically to fool the search heuristic into poor choices, which backjumping can then help overcome. Second, we assessed the benefits of backjumping on our regression testing domains of varying size, derived from real-world problems or prior investigations (available at http://www.htc.honeywell.com/projects/sa-circa/).

Deceptive Goals

The CIRCA search heuristic is based on a greedy goal regression search, guiding the system to take actions that are apparently on the path to achieving all the desired goals. However, CIRCA is designed first and foremost to build safety-preserving plans, and it can sacrifice some of its goals in service of safety. Thus one way to fool the heuristic is to build domains with goals that are not consistent with system safety; the heuristic will attempt to achieve the goals, and the planner must realize they are impossible and change decisions to achieve only a subset of its goals while guaranteeing system safety. The example shown in Figure 2 illustrates this type of domain. If we make state 3 have one or more desirable goal features, the heuristic will recognize the path to those goals and plan to achieve them. However, the only way to build a safe plan is to ignore that goal and choose a different action from state 1. We can make a scalable class of similar domains by varying the length of the chain of actions leading to the deceptive goal. The more states in that path, the more work the chronological backtracking approach will require to find the correct solution (backtracking all the way to the action decision for state 1). After generating the deceptive path, the backjumping search will have no problem skipping any number of decisions in that deceptive goal chain, recognizing at once that the state 1 action choice is the key element of the culprit path to failure, and the other decisions are irrelevant. Figure 4 summarizes the results of running the CSM on

this type of domain with goal path lengths varying up to eight steps. As expected, the chronological backtracking search grows exponentially, while the backjumping search scales linearly.

Pathological Ordering

Because the greedy CIRCA search heuristic largely ignores temporal aspects of the domain, another way to fool the heuristic into making poor choices is to make the correct choices dependent on temporal transitions. For example, we can build a scalable class of domains where a series of N actions are required to achieve the goal, and they must be executed in a specific order or else uncontrollable temporals will occur that undo the progress made towards the goal. We can then declare the final goal to be mandatory, meaning that CIRCA is not allowed to build a plan that does not achieve that goal. If we carefully construct the domain so that the heuristic always suggests the wrong ordering first, then we can force backtracking in proportion to N.

In this case, backjumping is of less utility because it cannot skip back many action decisions: as the depth-first search proceeds, when a failure is detected it always implicates the most-recent action. However, backjumping is still advantageous because it avoids backtracking to irrelevant preemption decisions (see step 2 in Alg. 1).

As shown in Figure 5, the backjumping planner still must explore the growing set of all action orderings, but the chronological backtracking planner performs even worse, because it must consider the growing set of preemption decisions.

Regression Suite

In a set of 23 domains from our regression suite, our evaluations indicate major advantages for backjumping in moderate and large domains, and no measurable overhead penalties. For example, one of our regression testing domains abstractly represents a problem the Cassini spacecraft faced in planning to pre-heat a backup system before a critical mission phase (Musliner & Goldman 1997). The chronological backtracking version requires 143 backtracks and 1.5 seconds to solve the problem, while the backjumping version uses only 58 backjumps and 0.9 seconds; both versions examine 66 states. On one of our larger regression tests, drawn from a robot arm workstation domain with complicated temporal elements, the chronological backtracking version performs 235 backtracks before timing out after 20 minutes and failing to solve the problem. The backjumping version makes only 93 well-directed backjumps and solves the problem in 4.9 seconds.

Trace-directed backjumping can help eliminate search even when no plan is possible. In a simpler variant of the robot arm domain, the chronological backtracking version explores 68 states, backtracks 90 times, and concludes that no plan is possible in about 500 milliseconds. The backjumping version comes to the same conclusion in only 85 milliseconds, after six backtracks and 52 states.

Related Work

Buccafurri, et al. (1999) also attempt to tackle the problem of automatically extracting repair information from counterexamples. They describe a technique for adding automatic repair to model checking verification. They use abductive model revision to alter a concurrent program description in the face of a counterexample. The class of systems and repairs they consider seem most appropriate for handling concurrency protocol errors, especially involving mutual exclusion and deadlocks, and rather less appropriate for control applications like the ones that interest us.

Tripakis and Altisen (1999) have independently developed an algorithm very similar to ours. They use the term "on-the-fly" for algorithms that generate their reachable state spaces at the same time as they synthesize the controller. AI planning algorithms, including the original CIRCA planning/controller synthesis algorithm (Musliner, Durfee, & Shin 1993; 1995) have typically been on-the-fly in this sense. We believe that a suitably-modified backjumping scheme could be profitably incorporated into their algorithm.

Conclusions

We have presented a technique for extracting decision repair candidates from counterexample traces produced by a model-checking plan verifier. These repair candidates take the form of entries in a search stack, and allow us to use backjumping search in the planning process. In difficult planning problems, where heuristic guidance may err, backjumping provides a crucial advantage. Our technique should be directly applicable to other on-the-fly controller synthesis and AI planning methods. We hope that it will also point the way to improvements in other uses of model-checking verification.

Acknowledgments

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

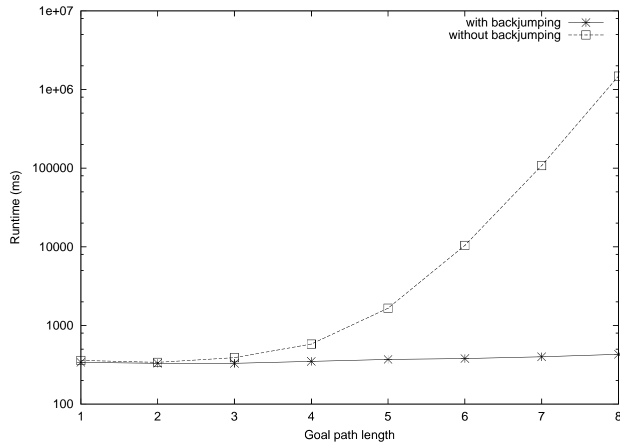
References

Buccafurri, F.; Eiter, T.; Gottlob, G.; and Leone, N. 1999. Enhancing model checking in verification by AI techniques. *Artificial Intelligence* 112:57–104.

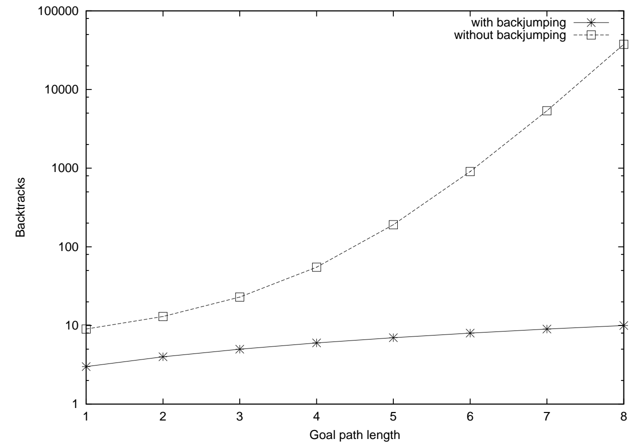
Gaschnig, J. 1979. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University.

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. Available at `http://www-aig.jpl.nasa.gov/home/gat/gp.html`.

Ginsberg, M. L. 1993. Dynamic backtracking. *Journal of Artificial Intelligence Research* 1:25–46.

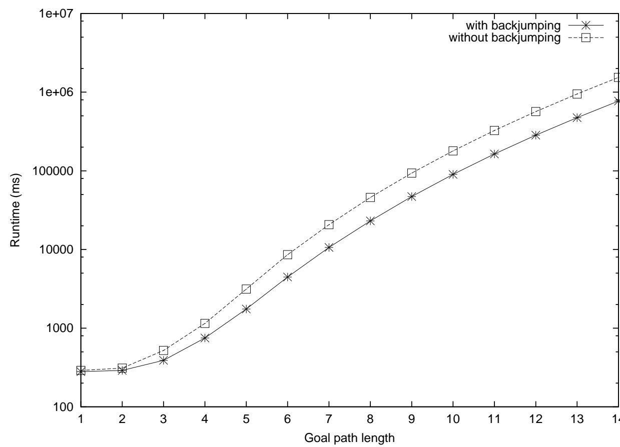


(a) Runtime [log].

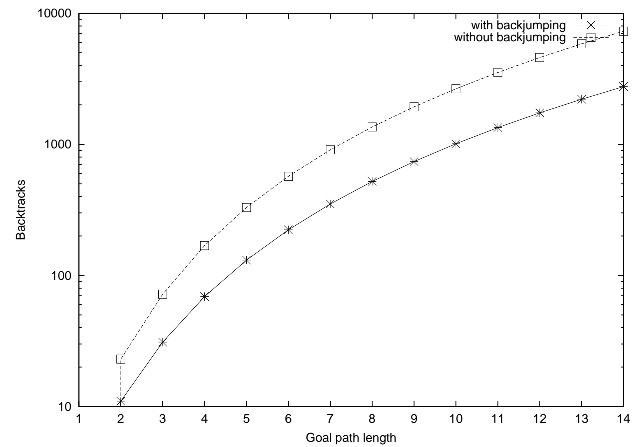


(b) Number of backtracks or backjumps [log].

Figure 4: As the path to deceptive goals grows, chronological backtracking scales badly, while backjumping search scales linearly.



(a) Runtime [log].



(b) Number of backtracks or backjumps [log].

Figure 5: Backjumping search avoids consideration of irrelevant preemption decisions.

Giunchiglia, F., and Traverso, P. 1999. Planning as model-checking. In *Proceedings of ECP-99*. Springer Verlag. Paper accompanying invited talk to be presented at the 1999 European Conference on Planning (ECP-99). Available through http://afrodite.itc.it:1024/~leaf/.

Goldman, R. P.; Musliner, D. J.; and Pelican, M. S. 2002. Exploiting implicit representations in timed automaton verification for controller synthesis. In Tomlin, C. J., and Greenstreet, M. R., eds., *Hybrid Systems: Computation and Control (HSCC 2002)*, number 2289 in LNCS. Springer Verlag. 225–238.

Hune, T.; Larsen, K. G.; and Pettersson, P. 2001. Guided Synthesis of Control Programs using UPPAAL. *Nordic Journal of Computing* 8(1):43–64.

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.; 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*, number 1708 in Lecture Notes in Computer Science. Berlin: Springer Verlag. 233–252.