Appears in Working Notes of the 2nd Workshop on Real-Time Tools
August 1, 2002, Copenhagen, Denmark

Verifier Trace-Directed Backjumping for Controller Synthesis

Robert P. Goldman, Michael J. S. Pelican, David J. Musliner
{goldman,musliner,pelican}@htc.honeywell.com

Honeywell Laboratories
3660 Technology Drive
Minneapolis, MN 55418

Abstract. Model-checking verification systems can return counterexample traces when desired properties are shown to be violated. Unfortunately, it can be very difficult to determine how to repair a system design from a counterexample trace. In this paper, we describe an automatic technique for extracting repair candidates from counterexample traces, in the context of an on-the-fly algorithm for timed automaton controller synthesis (reactive planning). By mapping a counterexample trace back into a set of search stack entries (forming a *nogood*), we identify decisions that may be causing the verification failure. These nogoods allow us to use backjumping search in the controller synthesis. Backjumping search is guaranteed to visit fewer nodes than conventional chronological backtracking search, and in many problems visits far less. We present data to show that, in large controller synthesis problems, backjumping may provide substantial speedup by removing large portions of the search space, without sacrificing completeness.

1 Introduction

Ordinarily, model-checking verifiers are used as part of an iterative, manual system design process. Once an initial system design is complete, the designer asks a model-checker to verify that some property holds over all possible traces of system execution. If the model-checker finds that this property does *not* hold, it produces a counterexample trace demonstrating how the system can violate the desired property. Unfortunately, it is very difficult for people to use these traces to guide design revisions. Havelund argues that tools are needed to support human examination of error traces, and cites as one example the message sequence charts provided by SPIN [10]. Simmons also reports "...it is usually quite difficult to diagnose the error directly from the counterexample" [18]. Several correspondents agree that counterexamples, especially long counterexamples, are hard to comprehend [16, 17].

We use timed automaton verification as an integral, on-line component of a fully-automatic controller synthesis system. The synthesis system is based on heuristic search: the algorithm makes heuristic decisions about what controller actions should be taken in particular states, and uses a verifier to confirm that

these control choices will prevent certain types of failure. 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 controller synthesis system 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.*

The method described here was developed for controller synthesis in the context of the CIRCA intelligent control architecture [14]. However, the method should be generally useful in any "on-the-fly" controller synthesis method, such as that of Tripakis and Altisen [19].

We begin by briefly reviewing our controller synthesis method, the CIRCA Controller Synthesis Module. We then review backjumping, a technique for directed backtracking that is guaranteed to search fewer 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.

2 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 time-constrained transitions that modify world features. Transitions have preconditions describing when they are applicable, as well as temporal characteristics. For example, Fig. 1 shows several transitions taken from a CIRCA problem description for controlling the Cassini spacecraft during Saturn Orbital Insertion [7, 15]. 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.

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

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

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

;; If the engine is burning while the active IRU breaks,
;; we have a limited amount of time to fix the problem before
;; the spacecraft will go 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
```

Fig. 1. Example transition descriptions given to CIRCA's CSM.

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 for that state an action 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.

3 Backjumping

Although the search algorithm includes heuristic guidance (using our own heuristic, based on [12]), 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 Fig. 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

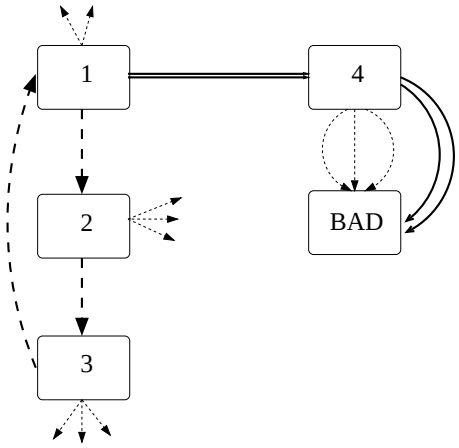

Fig. 2. An example showing the utility of backjumping.

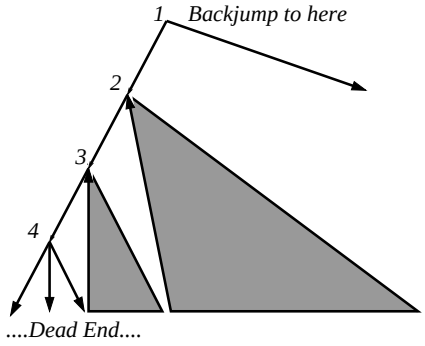

Fig. 3. A search tree corresponding to the controller synthesis problem in Fig. 2

that will take it to state 2, an action from state 2 that will take it to state 3, and an action for state 3 that will return the system to its initial state. Unexplored alternatives are shown as fainter dotted lines. There are two alternative actions for state 1 and three each for states 2 and 3. The planner has also permitted a nonvolitional transition to carry the system from the initial state to state 4.

Unfortunately, when trying to choose an action for state 4, we will reach an impasse. All three possible action choices will lead us to **BAD**. Worse, we cannot simply do `no-op`, because if we do, there is a nonvolitional transition that will carry us to **BAD**. 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 Fig. 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 simply to "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 Fig. 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 [6], but our discussion follows the presentation by Ginsberg [8], which is admirably elegant and lucid. 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 Fig. 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 $-$.

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.

4 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 [9]. 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:

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

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. $s_{\text{fail}}$is a state whose location is the distinguished failure location.

The transition $s_n \stackrel{t_n}{\rightarrow} s_{\text{fail}}$ 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, $s_i \stackrel{t_i}{\rightarrow} s_{i+1}$ into search decisions. This function is defined over the CSM states and transitions, $\sigma_i \equiv \text{CSMstate}(s_i), \sigma_{i+1} \equiv \text{CSMstate}(s_{i+1})$ and $\tau_i \equiv \text{CSMtrans}(t_i)$, as follows:

Definition 7. *Eliminating Explanations from Counterexamples*

$$f(\sigma_i, \tau_i, \sigma_{i+1}) = \begin{cases} \{\alpha(\sigma_i)\} & \textit{if } \tau_i \textit{ is a TTF and } \sigma_{i+1} = s_{fail}. \\ \{\alpha(\sigma_i), \varpi(\sigma_i, \tau_i)\} & \textit{if } \tau_i \in \pi(\sigma_i) \textit{ and } \sigma_{i+1} = s_{fail}. \\ \{\alpha(\sigma_i)\} & \textit{if } \tau_i \textit{ is an action or an event} \\ & \textit{(see Sect. 2) and } \sigma_{i+1} \neq s_{fail}. \\ \{\varpi(\sigma_i, \tau_i)\} & \textit{if } \tau_i \textit{ is a temporal transition} \\ & \textit{(and not an event) and } \sigma_{i+1} \neq s_{fail}. \end{cases}$$

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, $\varpi(l, t) = \top(\text{resp.}, -)$ 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 [8]. 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. That is, a set of variable assignments in the CSM search is consistent iff the sub-space generated by those variable assignments

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

is safe. If the timed automaton verifier we use is correct the correctness condition is met. ∴

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:

$$s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots s_m \xrightarrow{t_m} \ldots s_n \xrightarrow{t_n} s_{\text{fail}}$$

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 $s_m \xrightarrow{t_m} s_{m+1}$ will add to the nogood at least one variable in $\overline{P'} - \overline{P}$. ∴

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. ∴

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

5 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 same verifier to assess whether the current plan is acceptable, but if the verifier finds failure is reachable we do not return a culprit, instead simply backtracking one step and changing the last decision on the stack.

For example, on one of our mid-sized regression testing domains, the chronological backtracking version requires 105 backtracks and 1.5 seconds to solve the problem, while the backjumping version uses only 32 backjumps and 0.9 seconds. Larger domains show much more dramatic improvements. For example, on one of our larger testing domains the chronological backtracking version performs over 14,000 backtracks before timing out after 20 minutes and failing to solve the problem[2]. The backjumping version makes only 25 well-directed backjumps and solves the problem in 9.5 seconds.

[2] Experience shows that, if a solution has not been found by then, the search is typically hopelessly lost. These tests were performed on a 750MHz Pentium machine running Linux.

These performance improvements can also be seen in a closely-related system developed at the University of Michigan [4]. Their search algorithm was recently extended to include backjumping. Their evaluation of backjumping on a large set of randomly-generated domains also illustrates the value of this technique.

6 Related Work

Buccafurri, et. al. [2] 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 [19] 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 [13, 14] have typically been on-the-fly in this sense. We believe that a suitably-modified backjumping scheme could be profitably incorporated into their algorithm.

Backjumping provides a restricted form of guidance to backtracking, using limited record-keeping. There are other algorithms that provide more guidance, at the expense of more record-keeping. Three of the most significant are dynamic backtracking [8], dependency-directed backtracking (sometimes called Truth Maintenance Systems or TMSes) [5], and multiple-context (assumption-based) systems (ATMSes) [11, 3]. Dynamic backtracking is similar to backjumping, but additionally permits the search algorithm to salvage some of the work done between the backjump point and the point at which failure is detected. However, for many applications, this seems to be counterproductive, sometimes behaving exponentially worse than simpler approaches [1]. It is now generally agreed that for almost all applications TMSes provide a poor return for their space cost. On the other hand, ATMSes have been widely used in diagnostic applications. It is possible that they would be useful here.

7 Conclusions

We have presented a technique for extracting repair candidates from counterexample traces in a controller synthesis application. These repair candidates take the form of entries in a search stack, and allow us to use backjumping search in the controller synthesis. In difficult controller synthesis problems, backjumping provides a crucial advantage. Our technique should be directly usable in 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.

References

[1] Baker, A. B. The hazards of fancy backtracking. In *Proceedings of the Twelfth National Conference on Artificial Intelligence* (1994), AAAI Press/MIT Press, pp. 288–293.

[2] Buccafurri, F., Eiter, T., Gottlob, G., and Leone, N. Enhancing model checking in verification by AI techniques. *Artificial Intelligence 112* (1999), 57–104.

[3] deKleer, J. An assumption-based TMS. *Artificial Intelligence 28* (1986), 127–162.

[4] Dolgov, D. A., and Durfee, E. H. Satisficing strategies for resource-limited policy search in dynamic environments. In *Proc. First Int'l Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS)* (July 2002).

[5] Doyle, J. A truth maintenance system. *Artificial Intelligence 12*, 3 (1979), 231–272.

[6] Gaschnig, J. Performance measurement and analysis of certain search algorithms. Tech. Rep. CMU-CS-79-124, Carnegie-Mellon University, 1979.

[7] Gat, E. News from the trenches: An overview of unmanned spacecraft for AI. In *AAAI Technical Report SSS-96-04: Planning with Incomplete Information for Robot Problems* (Mar. 1996), I. Nourbakhsh, Ed., American Association for Artificial Intelligence. Available at `http://www-aig.jpl.nasa.gov/home/gat/gp.html`.

[8] Ginsberg, M. L. Dynamic backtracking. *Journal of Artificial Intelligence Research 1* (1993), 25–46.

[9] Goldman, R. P., Musliner, D. J., and Pelican, M. S. Exploiting implicit representations in timed automaton verification for controller synthesis. In *Hybrid Systems: Computation and Control (HSCC 2002)*, C. J. Tomlin and M. R. Greenstreet, Eds., no. 2289 in LNCS. Springer Verlag, Mar. 2002, pp. 225–238.

[10] Havelund, K., May 2002. personal communication.

[11] McDermott, D. V. Contexts and data dependencies: A synthesis. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-5*, 3 (1983), 237–246.

[12] McDermott, D. V. Using regression-match graphs to control search in planning. *Artificial Intelligence 109*, 1 – 2 (Apr. 1999), 111–159.

[13] Musliner, D. J., Durfee, E. H., and Shin, K. G. CIRCA: a cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man and Cybernetics 23*, 6 (1993), 1561–1574.

[14] Musliner, D. J., Durfee, E. H., and Shin, K. G. World modeling for the dynamic construction of real-time control plans. *Artificial Intelligence 74*, 1 (Mar. 1995), 83–127.

[15] Musliner, D. J., and Goldman, R. P. CIRCA and the Cassini Saturn orbit insertion: Solving a prepositioning problem. In *Working Notes of the NASA Workshop on Planning and Scheduling for Space* (Oct. 1997).

[16] Pecheur, C., May 2002. personal communication.

[17] Rangarajan, M., May 2002. personal communication.

[18] Simmons, R., Pecheur, C., and Srinivasan, G. Towards automatic verification of autonomous systems. In *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2000), IEEE.

[19] Tripakis, S., and Altisen, K. On-the-fly controller synthesis for discrete and dense-time systems. In *Formal Methods 1999*, J. Wing, J. Woodcock, and J. Davies, Eds., no. 1708 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1999, pp. 233–252.