JOURNAL OF ALGORITHMS **25,** 144–161 (1997)
ARTICLE NO. AL970877

An Efficient Algorithm for Controller Synthesis under Full Observation*

M. Barbeau,[†] F. Kabanza,[‡] and R. St-Denis[§]

Département de Mathématiques et d'Informatique, Université de Sherbrooke, Sherbrooke, Québec, Canada J1K 2R1

Received December 8, 1995; revised May 22, 1997

This paper presents a simple and flexible on-line synthesis algorithm that derives the optimal controller for a given environment. It consists of finding the greatest possible number of admissible event sequences of a discrete-event system with respect to a requirements specification. It generates and explores the state space on-the-fly and uses a control-directed backtracking technique. Compared to a previous algorithm of Wonham and Ramadge, our algorithm does not require explicit storage of the entire work space and backtracks on paths of arbitrary length to prune the search space more efficiently. This paper also discusses an implementation of our algorithm and includes an evaluation of its performance on a variety of problems. © 1997 Academic Press

1. INTRODUCTION

Control problems involve finding an efficient machine that senses and controls an environment through connections and guarantees satisfaction of requirements [9]. These machines have wide applications in areas ranging from robotics and manufacturing to telecommunications. The supervisory control theory developed by Ramadge and Wonhan [15] constitutes an appropriate framework for tackling this kind of problem when the environment is modeled as a discrete-event system (DES). It allows the automatic generation of abstract controllers that are close to real ma-

* The research described in this paper was supported in part by the Natural Sciences and Engineering Research Council of Canada and the Fonds pour la Formation de Chercheurs et l'Aide à la Recherche.

[†] E-mail: barbeau@dmi.usherb.ca.

[‡] E-mail: kabanza@dmi.usherb.ca.

[§] E-mail: stdenis@dmi.usherb.ca.

chines. Furthermore, it ensures that each controller derived by a synthesis procedure is correct and optimal. This is fundamental in the design of safety critical systems.

This paper is concerned with the synthesis problem under full observation, which is informally stated as follows. Given a model of an environment, a set of controllable events, and a requirements specification (also called legal behavior), find a controller such that the behavior of the environment connected with the controller is as large as possible but legal with respect to the requirements specification. Wonham and Ramadge [17] proposed an algorithm using an automaton-based approach to solve this problem. It synthesizes a controller mainly through comparison of the transition structures of automata modeling the environment and its legal behavior. The computational complexity of this algorithm is polynomial in the cardinalities of state spaces of automata modeling the unrestrained and legal behaviors of the environment. Note, however, that the number of states of the automaton modeling the environment grows exponentially with the number of its components [18]. This is due to the fact that the environment's transition structure is obtained by taking the shuffle or synchronous product of the transition structures of automata modeling the components. Thus the Wonham and Ramadge synthesis procedure requires first the computation of a global automaton from those of components, then the intersection with the legal behavior, and finally the extraction of the optimal controller from the result of these two operations using a fixed point calculation.

In this paper, we propose a single-phase synthesis algorithm that computes the product elements, while simultaneously synthesizing a controller, *on-the-fly* from the transition structures of components and considers only the part of the product state space necessary for generating the optimal controller. Therefore, it does not require explicit storage of the entire state space of the environment. Furthermore, our algorithm detects early unsatisfactory states using a *control-directed backtracking* technique that goes back over uncontrollable paths of arbitrary length. Although our algorithm has the same worst-case complexity as the Wonham and Ramadge algorithm, our experiments suggest a better average complexity. This is essentially due to the fact that an algorithm that enumerates states on-the-fly performs better in scenarios where relevant control information is local so much so that most of the known algorithms for computing the optimal controller are not suitable for practical implementations because they do not tackle the complexity of the problem well. The present paper constitutes a first step in this direction.

The rest of the paper is organized as follows. Section 2 gives the notation and background results of the supervisory control theory for DES. Section 3 presents the basic ideas of our synthesis approach and introduces related concepts. Section 4 details the synthesis algorithm. Section 5 provides the main results of a comparative experiment, discussion on related work, and concluding remarks.

2. PRELIMINARIES

In the context of the supervisory control theory, an environment is called a *plant* and its interconnection with a controller forms a *closed-loop system*. A plant G is a DES represented by a *generator* $(\Sigma, X, \delta, x_0, X_m)$, which is a deterministic finite automaton, where Σ is a finite set of events, X a finite set of states, $\delta: \Sigma \times X \to X$ a transition function, $x_0 \in X$ an initial state, and $X_m \subseteq X$ a set of marker states. It is convenient to introduce the extended transition function $\delta: \Sigma^* \times X \to X$ in the usual way [7]. The *active set* of a state x, denoted $\Sigma(x)$, is defined by $\{\sigma \mid \delta(\sigma, x)!\}$.[1] The *closed behavior* and *marked behavior* of G are defined, respectively, as

$$L(G) = \{w \in \Sigma^* \mid \delta(w, x_0)!\},$$
$$L_m(G) = \{w \in \Sigma^* \mid \delta(w, x_0) \in X_m\}.$$

The set X_m represents the *complete tasks* of the plant. Usually, G is *nonblocking*, that is, $\overline{L_m(G)} = L(G)$.[2] Intuitively, this means that all subtasks in G can be completed.

The alphabet Σ is divided into two disjoint subsets Σ_c and Σ_u; that is, *controllable* events and *uncontrollable* events, respectively. Let $\Lambda = \wp(\Sigma_c)$, the power set of Σ_c, be the set of all *control inputs*, and $\lambda \in \Lambda$. If $\sigma \in \lambda$, then σ is *disabled*; otherwise, it is *enabled*. An uncontrollable event is always enabled. Therefore, in a controlled DES, the transition from a state x on the event σ is enabled if $\delta(\sigma, x)!$ and $\sigma \notin \lambda$. The role of control inputs is to govern the behavior of G with respect to a *legal language* L over the alphabet Σ. Let H be a finite deterministic automaton $(\Sigma, Y, \xi, y_0, Y_m)$ such that $L = L_m(H)$.

Given a plant G and a legal language L, a *nonblocking controller* C must be constructed such that the behavior of G under the supervision of C is restrained to the greatest possible number of admissible event sequences. The language defined by these sequences is called the *supremal controllable sublanguage* of L and the synthesized controller is termed *maximally permissive*. The plant and controller are usually brought together in a

[1] $\delta(\sigma, x)!$ is an abbreviation for the expression "$\delta(\sigma, x)$ is defined."
[2] $\overline{L}$ denotes the prefix closure of L.

closed-loop system denoted by C/G. They evolve concurrently so that they interact with each other. Their interactions may be regarded as events that require their simultaneous participation; the events are generated by G with respect to control inputs established by C. Whereas the plant and controller share the same set of events, each event that actually occurs must be possible in the independent behavior of each process separately.

In the supervisory control theory for DES, one may distinguish three basic problems: the *supervisory control* (SC) problem, the *marking nonblocking supervisory control* (MNSC) problem, and the *nonblocking supervisory control* (NSC) problem. In the SC problem, the fact that states are marked is irrelevant. In the MNSC problem, some controller states are marked, so the controller must recognize complete tasks. Finally, in the NSC problem, the plant detects complete tasks. The algorithm presented in this paper can be used to solve any of these problems. For the sake of simplicity, however, we focus on the MNSC problem.

The fundamental property of *controllability* [14] must be introduced to deal with a control problem. A language $K \subseteq \Sigma^*$ is *controllable* with respect to G if $\overline{K}\Sigma_u \cap L(G) \subseteq \overline{K}$. Intuitively, a language K is controllable if any subtask of K followed by an uncontrollable event that is possible in G is also a subtask of K.

Let $\mathcal{C}(L \cap L_m(G))$ be the set of controllable sublanguages of $L \cap L_m(G)$ with respect to G; that is

$$\mathcal{C}\big(L \cap L_m(G)\big) = \big\{K \subseteq L \cap L_m(G) \mid \overline{K}\Sigma_u \cap L(G) \subseteq \overline{K}\big\}.$$

This set is nonempty and closed under unions [14]. Therefore, it contains a unique supremal controllable sublanguage denoted by $K^{\uparrow}$. If $K^{\uparrow} \neq \emptyset$, there exists a controller C for G such that $L_m(C/G) = K^{\uparrow}$, which solves the MNSC problem. From a practical point of view, the controller C is a pair (S, φ), where S is a finite deterministic automaton $(\Sigma, Q, \zeta, q_0, Q_m)$ and $\varphi: Q \to \Lambda$ is a *feedback function*. The closed-loop system C/G is represented by the product structure $(\Sigma, Q \times X, \zeta \times \delta, \langle q_0, x_0\rangle, Q_m \times X_m)$, where a transition from $\langle q, x\rangle$ to $\langle \zeta(\sigma, q), \delta(\sigma, x)\rangle$ on the event σ is defined if $\zeta(\sigma, q)!$, $\delta(\sigma, x)!$, and $\sigma \notin \varphi(q)$. From the previous definitions, the following properties hold:

$$L_m(C/G) = L_m(G) \cap L_m(S), \tag{1}$$

$$L(C/G) = L(G) \cap L(S), \tag{2}$$

$$\overline{L_m(C/G)} = L(C/G). \tag{3}$$

The most well-known computional method to obtain $K^{\uparrow}$ comes from Wonham and Ramadge [17]. It proceeds by calculating the fixpoint of the operator

$$\Omega(K) = L \cap L_m(G) \cap \sup\left\{T \subseteq \Sigma^* \mid T = \overline{T}, T\Sigma_u \cap L(G) \subseteq \overline{K}\right\}.$$

In this equation, the expression $L \cap L_m(G)$ guarantees that the closed-loop system includes only the legal complete tasks possible in G. The equality $T = \overline{T}$ and inclusion $T\Sigma_u \cap L(G) \subseteq \overline{K}$ are related to the concepts of nonblocking and controllability, respectively. The synthesis algorithm works on paths of length 1 of H, assuming that $L(H) \subseteq L(G)$ and there exists a *correspondence function* f between the states of H and states of G such that $(f \circ \xi)(w, y_0) = \delta(w, x_0)$ for all $w \in L(H)$.[3] At each step, the algorithm removes the states of H that violate the controllability constraint $\Sigma(f(y)) \cap \Sigma_u \subseteq \Sigma(y)$. At the end of each step, the algorithm eliminates the states of H that are not coreachable (i.e., not leading to a marker state). This process stops when no states can be removed. The result is then a controller automaton.

3. THE SYNTHESIS APPROACH

Our synthesis approach is described hereafter. Let us consider the intersection of $L(G)$ and $L(H)$, the closed behaviors of G and H, respectively. Let $R = (\Sigma, X \times Y, \gamma, \langle x_0, y_0 \rangle, X_m \times Y_m)$ such that for all $\sigma \in \Sigma$, $x \in X$, and $y \in Y$,

$$\gamma(\sigma, \langle x, y \rangle) = \begin{cases} \langle \delta(\sigma, x), \xi(\sigma, y) \rangle, & \text{if } \delta(\sigma, x)! \text{ and } \xi(\sigma, y)!, \\ \text{undefined}, & \text{otherwise}. \end{cases}$$

It is easily shown that $L(R) = L(H) \cap L(G)$ and $L_m(R) = L \cap L_m(G)$. A state $\langle x, y \rangle$ of R records a finite sequence of events $t = \langle \sigma_0, \sigma_1, \dots, \sigma_n \rangle$ (called a trace) in which G and H have engaged up to some moment in time. It should, however, be noted that $\delta(\sigma, x)$ may be defined for some $\sigma \in \Sigma$ while $\xi(\sigma, y)$ is undefined, as shown in Fig. 1. In this particular case, the goal of the controller is to disable the event σ, if σ is controllable, or prevent the plant from reaching the state x by disabling the latest controllable event σ_i from the end of the trace t.

Let $s, s' \in X \times Y$. Let $s.lct$ denote the set of *latest controllable transitions* (LCT) on paths leading to s with respect to some controllable events;

[3] In general, the function f is defined from $X \times Y$ (the states of $G \times H$) to X (the states of G). Specifically, for $\langle x, y \rangle \in X \times Y$, $f(\langle x, y \rangle) = x$.

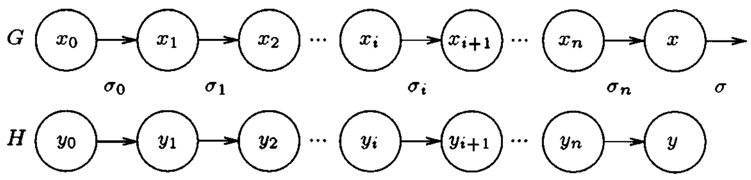

FIG. 1. Latest controllable transition of $\langle x, y\rangle$.

that is, s can be prevented by disabling σ' at s' for each $[\sigma', s'] \in s.lct$ [13]. One can see the latest controllable transitions as a function lct: $X \times Y \to \wp(\Sigma_c \times (X \times Y))$. Referring to Fig. 1, the LCT is interpreted as follows. The transition from x_i to x_{i+1} (labeled σ_i in the plant) is controllable, whereas the transition from x (labeled σ) is uncontrollable as are the transitions from x_{i+1} to x. Therefore, the only way to avoid the state x in the plant is by disabling the controllable event σ_i contained in the LCT of $\langle x, y\rangle$. The disablement of a transition from $\langle x, y\rangle$ on σ according to whether σ is controllable or not is formalized as

$$\varphi(\langle x, y\rangle) = \varphi(\langle x, y\rangle) \cup \{\sigma\}, \quad \text{if } \sigma \in \Sigma_c, \tag{4}$$

$$\varphi(\langle x', y'\rangle) = \varphi(\langle x', y'\rangle) \cup \{\sigma'\}$$
$$\text{for all } [\sigma', \langle x', y'\rangle] \in \langle x, y\rangle.\text{lct if, } \sigma \in \Sigma_u. \tag{5}$$

Two new expressions are introduced to implement the function lct. Let $s.pre \subseteq \wp(X \times Y)$ denote the set of states from which the state s is directly reachable on an uncontrollable event. Let $s.pct \subseteq \wp(\Sigma_c \times (X \times Y))$ denote the set $\{[\sigma, t] \mid \gamma(\sigma, t) = s$ and $\sigma \in \Sigma_c\}$; that is, the state s is directly reachable from the state t on the controllable event σ. If $[\sigma', s'] \in s.lct$, then there exists a sequence of states $s_1, \dots, s_n$, with $n \geq 2$, $s_1 = s'$, and $s_n = s$, such that $s_{i-1} \in s_i.pre$, $3 \leq i \leq n$, and $[\sigma', s'] \in s_2.pct$.

Let $s = \langle x, y\rangle \in X \times Y$, then $s.x$ and $s.y$ denote the states $x \in X$ and $y \in Y$, respectively. Our synthesis approach is also based on the predicates

$$Marked(s) = \begin{cases} 1, & \text{if } s.x \in X_m \text{ and } s.y \in Y_m, \\ 0, & \text{otherwise,} \end{cases} \tag{6}$$

$$Coreachable(s)$$
$$= \begin{cases} 1, & \text{if } Marked(s) \text{ or} & (7) \\ & \text{if } (\exists w \in \Sigma^*)\gamma(w, s)! \text{ and } Marked(\gamma(w, s)), & (8) \\ 0, & \text{otherwise,} \end{cases}$$

$$Sink(s) = \begin{cases} 1, & \text{if } (\exists \sigma \in \Sigma_u)\, \delta(\sigma, s.x)! \text{ and not } \xi(\sigma, s.y)! \text{ or} & (9) \\ & \text{if } (\exists \sigma \in \Sigma_u)\, \delta(\sigma, s.x)!, \xi(\sigma, s.y)! & (10) \\ & \text{and } Sink(\langle \delta(\sigma, s.x), \xi(\sigma, s.y) \rangle) = 1, & \\ 0, & \text{otherwise.} & \end{cases}$$

The first predicate is based on the definition of the intersection of two languages. It is used to record information about complete tasks in the controller. The second predicate formalizes the notion of *coreachable* state. The third predicate determines if a state is a *sink* and expresses recursively the converse of controllability. For instance, in Fig. 1, if $[\sigma_i, \langle x_i, y_i \rangle] \in \langle x, y \rangle.lct$, then the state $\langle x, y \rangle$ satisfies Eq. (9), whereas the states $\langle x_{i+1}, y_{i+1} \rangle, \ldots, \langle x_n, y_n \rangle$ fulfill Eq. (10). During the synthesis process, a status is associated with each visited state $s \in X \times Y$, indicating whether s is *sink*, *coreachable*, or *undefined*.

Generally, a synthesis procedure computes an automaton S that accepts $K^{\uparrow}$; that is, $L_m(S) = K^{\uparrow}$. Based on the fact that $L_m(S) \subseteq L \cap L_m(G)$, this can be done in at least two ways. One strategy consists of obtaining S from R by pruning undesirable states; more specifically, sink states and noncoreachable states. This strategy, however, has the disadvantage of wasting memory space because it is usually unnecessary to consider the whole state space of R and indirectly the whole state space of G. An alternative is to construct S incrementally and generate only the states of R that are needed to solve the MNSC problem. Our synthesis procedure follows this strategy. More precisely, it performs the following operations simultaneously and on-the-fly: expansion of the state space of R from the initial state, identification of sink states, identification of noncoreachable states, and control-directed backtracking to prune the state space.

4. THE SYNTHESIS ALGORITHM

Before describing the algorithm, we define the following auxiliary procedures:

- `Mark_State`(s) establishes if a state s represents a complete task.
- `Initialize_New_State`(s) initializes the attributes of a new state s.
- `Sink`(s) fixes the status of a state s to *sink* and determines those that become sinks among its predecessors.
- `Coreachable`(s) sets the status of a state s to *coreachable* and identifies those that become coreachable among its predecessors.

```
procedure Mark_State($s$)
    $s.marked \leftarrow s.x$ in $X_m$ and $s.y$ in $Y_m$;
end.

procedure Initialize_New_State($s$)
    Mark_State($s$);    $s.pct \leftarrow \{\}$;    $s.pre \leftarrow \{\}$;
    $s.status \leftarrow undefined$;    $OPEN \leftarrow OPEN \cup \{s\}$;
end.
```

FIG. 2. Procedures for marking and initializing a new state.

• Undefined(s) verifies if the status of a state s can be set to *undefined* and then determines those that can potentially become undefined among its predecessors (the status *undefined* will be made clearer later).

The pseudo-code of the first two procedures is shown in Fig. 2. The code of these procedures is straightforward. The procedure Mark_State reflects Eq. (6). Note that it can be adapted to solve the SC or NSC problem.

Figure 3 contains the pseudo-code of the procedure Sink. It is always called with a parameter s such that $s.status \neq sink$. This prevents the procedure Sink from looping infinitely on a sequence of states that forms a cycle in the working transition graph. When a state becomes a sink, all its incoming and outgoing transitions are removed by the procedure Delete_Bound_Transitions(line 2). Furthermore, some of its predecessors may become sinks or noncoreachable. Line 3 implements Eq. (10) of the predicate *Sink*. Lines 4–6 check if every unmarked and coreachable predecessor state remains coreachable after deletions, which is the specific role of the procedure Undefined.

The definition of the predicate *Coreachable* [Eqs. (7) and (8)] is given with respect to the actual transition structure of R. The fact that sink states are identified and their bound transitions are removed as the

```
1 procedure Sink($s$)
2     $s.status \leftarrow sink$;    Delete_Bound_Transitions($s$);
3     for each $s'$ in $s.pre$ such that $s'.status \neq sink$ do Sink($s'$);

4     for each $[\sigma, s']$ in $s.pct$ such that $s'.marked$ = false
5                                 and $s'.status = coreachable$ do
6         Undefined($s'$);
7 end.
```

FIG. 3. Procedure for determining sinks.

working transition structure is expanded impacts on the coreachability of some states. Indeed, when a marker state is identified as a sink, some coreachable states may become noncoreachable. Besides, when a new marker state is added to the working transition structure, some noncoreachable states may become coreachable. The role of the procedures Coreachable and Undefined is to update the status of such states. Figure 4 shows the pseudo-code of these procedures; they are analogous and complementary. The former is always invoked with a parameter s such that $s.status = undefined$ and the latter is always called with a parameter s such that $s.marked =$ false $\wedge$ $s.status = coreachable$. When a state becomes coreachable (line 2 of the procedure Coreachable), every state leading to it also becomes coreachable [in accordance with Eq. (8) of the predicate *Coreachable*]. Lines 3–6 of the procedure Coreachable search backward for these states. On the contrary, when a state becomes undefined (that is, neither *coreachable* nor *sink*), the status of its predecessors may be set to *undefined*. As in the procedure Coreachable, lines 4–7 of the procedure Undefined search backward for these states.

The synthesis algorithm is shown in Fig. 5. It accepts as input a list of finite automata $C_1, \dots, C_n$ modeling the components of a plant, a finite automaton H accepting the legal language, and a set of uncontrollable events Σ_u. It produces as output a nonblocking and maximally permissive controller (S, φ) satisfying the control requirements and consisting of an automaton S and a feedback function φ. The algorithm incrementally computes a product transition structure of $C_1, \dots, C_n$, and H from which

```
1 procedure Coreachable($s$)
2     $s.status \leftarrow coreachable$;
3     for each $s'$ in $s.pre$ such that $s'.status = undefined$ do
4         Coreachable($s'$);
5     for each $[\sigma, s']$ in $s.pct$ such that $s'.status = undefined$ do
6         Coreachable($s'$);
7 end.

1 procedure Undefined($s$)
2     if not Marker_State_Reachable_From($s$) then
3        $s.status \leftarrow undefined$;
4        for each $s'$ in $s.pre$ such that $s'.marked$ = false and
5             $s'.status = coreachable$ do Undefined($s'$);
6        for each $[\sigma, s']$ in $s.pct$ such that $s'.marked$ = false and
7             $s'.status = coreachable$ do Undefined($s'$);
8 end.
```

FIG. 4. Procedures for determining coreachable and undefined states.

```
procedure Derive_Controller(in $C_1, \dots, C_n$, $H$, $\Sigma_u$, out $(S, \varphi)$)
  $CLOSED \leftarrow \{\}$;   $OPEN \leftarrow \{\}$;   Initialize_New_State($s_0$);

  repeat
     select $s$ in $OPEN$;
     $OPEN \leftarrow OPEN - \{s\}$;
     for each $\sigma$ in $\Sigma$ such that $\delta(\sigma, s.x)!$ do
         $x' \leftarrow \delta(\sigma, s.x)$;
         if not $\xi(\sigma, s.y)!$ then
           if $\sigma$ in $\Sigma_u$ then {Sink($s$);   break}
         else
           $s' \leftarrow \langle x', \xi(\sigma, s.y)\rangle$;
           if $s'$ in $CLOSED$ and $s'.status = sink$ then
             if $\sigma$ in $\Sigma_u$ then {Sink($s$);   break}
           else
             $\gamma(\sigma, s) = s'$;
             if $s'$ not in $CLOSED$ and $s'$ not in $OPEN$ then
               Initialize_New_State($s'$);
             if $\sigma$ in $\Sigma_u$ then
               $s'.pre \leftarrow s'.pre \cup \{s\}$;
             else
               $s'.pct \leftarrow s'.pct \cup \{[\sigma, s]\}$;
             if $s.marked =$ false and $s'$ in $CLOSED$ and
               $s'.status = coreachable$ then Coreachable($s$);
     if $s.marked =$ true and $s.status = undefined$ then
       Coreachable($s$);
     $CLOSED \leftarrow CLOSED \cup \{s\}$;
  until $OPEN = \{\}$;

  while exists $s$ in $CLOSED$ such that $s.status = undefined$ then
        Sink($s$);
  if $s_0.status = coreachable$ then
     $S \leftarrow (\Sigma, CLOSED_{reachable}, \gamma, s_0, (CLOSED_m)_{reachable})$;
  else
     Error("The solution is empty.");
  for each $s$ in $CLOSED_{reachable}$ do
      for each $\sigma$ in $\Sigma - \Sigma_u$ do
          if not $\gamma(\sigma, s)!$ then $\varphi(s) \leftarrow \varphi(s) \cup \{\sigma\}$;
end.
```

FIG. 5. Synthesis algorithm.

S is extracted. The pseudo-code is logically divided into three parts: the prologue (line 2), the search process (lines 3–27), and the epilogue (lines 28–37).

The prologue creates the initial state $s_0 = \langle x_0, y_0 \rangle$ and assigns default values to its attributes. It should be noted that $x_0 = \langle x_0^1, \ldots, x_0^n \rangle$, where x_0^i is the initial state of C_i $(1 \leq i \leq n)$. The set of states $X \times Y$ is partitioned into the subsets of expanded and unexpanded states. Initially, the set of expanded states, *CLOSED*, is empty and the set of unexpanded states, *OPEN*, contains the initial state s_0. The status of a state that belongs to *OPEN* is *undefined.*

The search process has the following framework. The main loop (lines 3–27) repeatedly selects a state $s = \langle x, y \rangle$ from *OPEN*. Given a current unexpanded state s, the algorithm determines its successors for all events σ such that $\delta(\sigma, x)$ is defined. Note that $\delta(\sigma, x)$ is computed on-the-fly from the transition functions $\delta_1, \ldots, \delta_n$ of automata $C_1, \ldots, C_n$. Figure 6 shows a situation in which some successors have been calculated after a number of iterations of the **for** loop (lines 6–23). At this particular step, the algorithm proceeds with the event σ by determining if it is possible in s and, in this case, leads to state s'. The loop proceeds until all possible events have been exhausted or the status of s has become *sink*. In the last case, successors of s not yet uncovered are just ignored (see the **break** statement at lines 9 and 13). The **for** loop considers three cases, as illustrated in Fig. 7.

In the first case, s' cannot be computed because $\xi(\sigma, s.y)$ is undefined; that is, the event σ does not occur on this path in H, as illustrated in Fig. 7a. Therefore, the state x' terminates an illegal path in G. If σ is controllable, then the controller must prevent the occurrence of σ. Other-

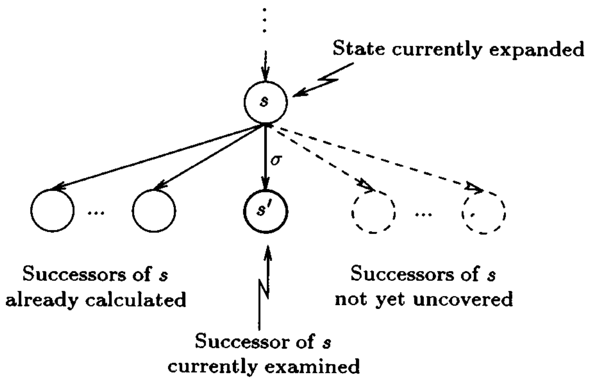
State currently expanded
s
σ
s'
Successors of s already calculated
Successors of s not yet uncovered
Successor of s currently examined

FIG. 6. The expansion of the state s.

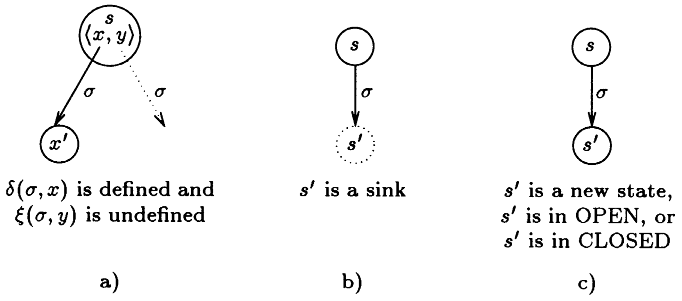

FIG. 7. The three basic cases.

wise, σ is uncontrollable and s is a sink. This particular subcase is represented by the conjunction of conditions at lines 6, 8, and 9 and corresponds to Eq. (9) of the predicate *Sink*.

The second case is shown Fig. 7b. The state s' is reachable from s on σ, but s' has already been expanded (i.e., $s' \in$ *CLOSED*) and is a sink. The operations to be performed are exactly the same as in the previous case. This subcase is represented by the conjunction of conditions at lines 6, 10, 12, and 13 and corresponds to Eq. (10) of the predicate *Sink*.

In the third case, s' is reachable from s on σ and s' is not a sink. The transition function γ is updated to keep track of the fact that this transition occurs simultaneously in the process and legal behavior on σ, from states x and y, respectively (line 15). If s' is a new state, its attributes are initialized and it is inserted into *OPEN* (line 17). Then, the set of predecessors (s'*.pre*) or previous controllable transitions (s'*.pct*) of s' is updated (line 19 or line 21) depending on whether or not σ is uncontrollable. Finally, if s is not a marker state and s' is a coreachable state in *CLOSED*, then s becomes a coreachable state (lines 22 and 23). In fact, if s' is coreachable, there exists a path labeled w from s' to a marker state s''. Hence, there exists a path labeled σw from s to s'', that is, s is coreachable [Eq. (8) of the predicate *Coreachable*].

When a state has been expanded, its status is set to *coreachable* if it is marked and not a sink (lines 24 and 25). This case represents Eq. (7) of the predicate *Coreachable*. Finally, s is added to *CLOSED* (line 26).

The epilogue builds the controller from the intermediate automaton. First, lines 28 and 29 change the status of undefined states to *sink* because they represent incomplete tasks. This prevents the closed-loop system from blocking. Because the procedure `Sink` deletes transitions, the exit states of uncontrollable transitions, thus removed, need also to be avoided. This is done by the recursive call in `Sink`. Second, lines 30 and 31 remove the sink states. The expression $CLOSED_{\text{reachable}}$ denotes the set of states in *CLOSED* that are reachable from the initial state if it is not a sink;

otherwise, it denotes the empty set (line 33). At the end of this step, the intermediate automaton contains only reachable and coreachable states; that is, the automaton S is obtained. Finally, lines 34–36 compute the feedback function.

The efficiency of this algorithm may be further improved by addressing implementation details. First, every state selected from *OPEN* at line 4 must be reachable to prevent the expansion of useless states. Two strategies can be exploited. One strategy consists of labeling nonreachable states; the other consists of removing them from *OPEN*. Deleted states would, however, be re-expanded when reached again by further expansions. Therefore, the overhead due to the update of the *nonreachable* labels is replaced by the overhead incurred by re-expanding states. This problem is comparable to the management of *backpointers* in the search algorithm A* [16]. These strategies are incomparable in terms of performance. Second, when a state is expanded, it is preferable to start with the uncontrollable events (see line 6) to prune the state space as soon as possible. Third, when sequences of states are processed backward in the procedure `Sink`, *pre* must be considered before *pct*. In this way, if $\gamma(\sigma_c, s') = s$ $(\sigma_c \in \Sigma_c)$ and $\gamma(\sigma_u, s') = s$ $(\sigma_u \in \Sigma_u)$, then $s' \in s.pct|_{X\times Y} \cap s.pre$. Therefore, if line 3 is swapped with lines 4 and 5 in the procedure `Sink`, then the status of s' can be set to *undefined* and immediately after to *sink*. This cannot happen when the operator *pre* has a higher priority than *pct*, because when the status of a state becomes *sink*, it cannot be changed afterward. Finally, it should be noted that the feedback function could be computed *on-line* during the expansion of the transition structure by using Eqs. (4) and (5) and changing lines 9 and 13 as follows:

```
**if** $\sigma$ **in** $\Sigma_u$ **then**
  Sink($s$);
  **for each** $[\sigma', s']$ **in** $s.lct$ **do**
    $\varphi(s') \leftarrow \varphi(s') \cup \{\sigma'\}$;
    **break**;
**else**
  $\varphi(s) \leftarrow \varphi(s) \cup \{\sigma\}$;
```

Nevertheless, the on-line approach is more expensive than the off-line approach because the feedback function is calculated for all expanded states, including those that will become *sink*. Furthermore, it requires computation of the function *lct*.

The correctness of the algorithm is based on the following theorem.

THEOREM 1. *The procedure* `Derive_Controller` *always terminates and, if* $K^{\uparrow} \neq \emptyset$, *then it returns a controller* $C = (S, \varphi)$ *satisfying* $L_m(C/G) = K^{\uparrow}$.

The proof of this theorem relies on four lemmas and one corollary. We do not give a detailed proof because the synthesis algorithm is a refinement of the known optimal supervisory computation mainly based on the notion of the synchronous composition of the plant and controller [12]. Such an approach simplifies the analysis of synthesis problems considered in this paper and proofs of synthesis algorithms. The reader can find a detailed proof of correctness in a companion technical report [2].

LEMMA 1 *(Invariant of the repeat-until loop, lines 3–27). Let predicate Sink be restricted to the elements of CLOSED. For every state $s \in CLOSED$, $s.status = sink$ iff $Sink(s)$ is true.*

LEMMA 2 *(Invariant of the repeat-until loop, lines 3–27). Let predicates Coreachable and Sink be restricted to the elements of CLOSED. For every state $s \in CLOSED$, $s.status = coreachable$ iff $Coreachable(s)$ is true and $Sink(s)$ is false.*

LEMMA 3 *(Invariant of the while loop, lines 28 and 29). For every state $s \in CLOSED$, $s.status = coreachable$ or $s.status = undefined$ iff $Controllable(s)$ is true, where $Controllable(s)$ is true if for all $\sigma \in \Sigma_u$, $\delta(\sigma, s.x)!$ implies $\gamma(\sigma, s)!$.*

LEMMA 4 *(Invariant of the while loop, lines 28 and 29). For every state $s \in CLOSED$, $s.status = coreachable$ iff both $Coreachable(s)$ and $Controllable(s)$ are true.*

COROLLARY 1. *If $K^{\uparrow} \neq \emptyset$ and $C = (S, \varphi)$ is calculated by* `Derive_Controller`, *for every $w \in \Sigma^*$, $\gamma(w, s_0)!$, and $\gamma(w, s_0) \in (CLOSED_m)_{\text{reachable}}$ iff $Marked(\langle \delta(w, x_0), \xi(w, y_0)\rangle)$ is true and for all $s \in \{\langle \delta(w', x_0), \xi(w', y_0)\rangle \mid w' \in \overline{w}\}$, $Controllable(s)$ is true.*

Proof of Theorem 1. Note that the procedure `Derive_Controller` always terminates. It takes as input the finite transition structures $C_1, \ldots, C_n$, and H. In output, it yields a transition structure in which every state is an $(n+1)$-tuple in which the component at position i is a state picked in C_i $(i = 1, \ldots, n)$ or H $(i = n+1)$. There is a finite number of such combinations. The number of times the **repeat-until** loop (lines 3–27) is executed is determined by the number of elements inserted in *OPEN* and is bounded, because every $(n+1)$-tuple is inserted once in *OPEN* (lines 16 and 17, Fig. 5) and selected once from *OPEN* (line 5). The number of iterations of the **while** loop (lines 28 and 29) is bounded as well, because every execution of it changes the status of a state from *undefined* to *sink*.

Now, let us assume that $K^{\uparrow} \neq \emptyset$ and let $C = (S, \varphi)$ be the controller calculated by `Derive_Controller`. We show the following. For every

$w \in \Sigma^*$, $w \in L_m(C/G)$ iff $w \in L$, $w \in L_m(G)$, and for all $w' \in \overline{w}$, $\sigma \in \Sigma_u$, if $w'\sigma \in L(G)$, then $w'\sigma \in \overline{L}$.

String $w \in L_m(C/G)$ iff $w \in L_m(S)$ [by Eq. (1) and $L_m(S) \subseteq L_m(R) = L \cap L_m(G)$]. By construction of S, $w \in L_m(S)$ iff $\gamma(w, s_0)!$ and $\gamma(w, s_0) \in (CLOSED_m)_{\text{reachable}}$. From Corollary 1, $\gamma(w, s_0)!$ and $\gamma(w, s_0) \in (CLOSED_m)_{\text{reachable}}$ iff $Marked(\langle \delta(w, x_0), \xi(w, y_0)\rangle)$ is true and for all $s \in \{\langle \delta(w', x_0), \xi(w', y_0)\rangle \mid w' \in \overline{w}\}$, $Controllable(s)$ is true. The preceding statement is true, however, iff $w \in L$, $w \in L_m(G)$, and for all $w' \in \overline{w}$ and $\sigma \in \Sigma_u$, if $w'\sigma \in L(G)$, then $w'\sigma \in \overline{L}$. ▮

5. DISCUSSION

Table 1 summarizes statistics obtained from a comparative experiment conducted with the algorithm in Fig. 5 and the Wonham and Ramadge algorithm (W & R algorithm). These two algorithms have been implemented in C++ by using an object-oriented approach. They use the same base classes and code results from a straightforward translation of their description while considering implementation details mentioned in the previous section. Four different problems consisting of tens to several thousands of states were used. The first three problems are extracted from a paper by Ramadge and Wonham [15]: the maze problem in which a cat and a mouse move, the problem of sharing a single resource by two users, and the factory problem in which two machines feed a buffer and another takes parts from the buffer. The fourth problem was initially suggested by Jensen [10], then reconsidered by Makungu et al. [13] in the framework of control theory, in which the plant is represented by a colored Petri net. It consists of synthesizing a controller that supervises m $(2 \leq m \leq 4)$ trains on a circular railway composed of 10 sections. Table 1 indicates the state space complexity of these problems. The W & R algorithm requires the

TABLE 1
Summary of Results

							Our algorithm	
Problem	C_i	G	H	$G \times H$	S	W & R	Gen.	Exp.
Maze	5	25	18	18	6	25	8	8
Two users	3	9	9	9	9	9	9	9
Factory	3	27	32	120	96	120	111	111
Two trains	10	100	70	70	60	100	70	70
Three trains	10	1,000	150	150	90	1,000	135	135
Four trains	10	10,000	100	100	40	10,000	75	60

computation of G from all C_i $(1 \leq i \leq n)$, then the calculation of $G \times H$. Our algorithm derives the automaton S directly from H and all C_i $(1 \leq i \leq n)$. In the W & R algorithm, the state space is characterized by the number of states generated, which is the maximum of $|X|$ (column labeled G) and $|X \times Y|$ (column labeled $G \times H$). With our algorithm, the state space is characterized by the number of states in $G \times H$ that are generated and the number of states in $G \times H$ that are expanded. Accordingly, our algorithm is better in terms of the number of states generated. The computation time savings come from different sources: the product of components on-the-fly, forward-chaining search, and control-directed backtracking technique. The first two sources focus on local information, whereas the third source prunes the state space as soon as possible. Finally, it should be noted that the measures are all identical for the problem with *two users*, because the legal behavior, represented as a regular language, is controllable and as large (with respect to the number of states) as the unrestrained behavior. This is solely a case of verification and is representative of the worst case.

Our algorithm can also be compared with other similar algorithms, in particular with the algorithm of Kumar, Garg, and Marcus [12] and the algorithm of Kumar and Garg [11] that solve the SC problem and MNSC problem, respectively. These algorithms, however, obtain S from R by pruning *bad states* (see the strategies presented at the end of Section 3). As in the Wonham and Ramadge algorithm, they require explicit storage of the entire state space. Several algorithms have also been proposed in the context of supervisory control of DES using limited lookahead [6]. The VLP-S (variable lookahead policy with state information) algorithm computes a state cost function that is used to define a control function on-line [4]. An off-line version that derives the supremal controllable sublanguage of a legal language has also been developed by using VLP-S. As in our approach, VLP-S is based on a forward search technique. However, with VLP-S it is assumed that the automaton H is a submachine of G. This property can be easily satisfied, but it implies a product of transition structures. Finally, VLP-S has been extended to the case of partially observed DES. This new on-line algorithm is called VLP-PO (variable lookahead policies under partial observation) [5] and works for legal languages that are prefix-closed. In conclusion, implementations of all these algorithms require some strong assumptions regarding the state space. Furthermore, to the best of our knowledge, no comparative experiment has been reported in the literature.

Another advantage of our algorithm is that it has been adapted to situations where the automaton H accepting the legal behavior is replaced by a temporal logic formula [1]. In this particular case, the states of H are replaced by formulas and the transition function ξ is replaced by a mechanism of formula progression. This allows compact representation of the legal behavior and the uniform handling of safety, liveness, and real-time constraints [3]. Our algorithm could also be used as the underlying algorithm to compute controllers on-line under the assumption of partial observation [8].

Finally, our approach could benefit by adapting various search space techniques developed in areas that model discrete-event systems, such as heuristic search in artificial intelligence and partial-order exploration in verification. These methods work on graphs and are based on forward-chaining search using state expansion. As such, our approach seems promising for integrating such techniques to further improve the exploration of search spaces.

REFERENCES

1. M. Barbeau, F. Kabanza, and R. St-Denis, Synthesizing plant controllers using real-time goals, *in* ‘‘Proceedings of the International Joint Conference on Artificial Intelligence,’’ Montreal, 1995, pp. 791–798.
2. M. Barbeau, F. Kabanza, and R. St-Denis, ‘‘An Efficient Algorithm for Controller Synthesis under Full Observation,’’ Rapport Technique 166, Département de Mathématiques et d’Informatique, Université de Sherbrooke, 1995.
3. M. Barbeau, F. Kabanza, and R. St-Denis, Supervisory control synthesis from metric temporal logic specifications, *in* ‘‘Proceedings of the 33rd Annual Allerton Conference on Communication, Control, and Computing,’’ Monticello, 1995, pp. 96–105.
4. N. Ben Hadj-Alouane, S. Lafortune, and F. Lin, Variable lookahead supervisory control with state information, *IEEE Trans. Automat. Control* **39** (1994), 2398–2410.
5. N. Ben Hadj-Alouane, S. Lafortune, and F. Lin, Centralized and distributed algorithms for on-line synthesis of maximal control policies under partial observation, *J. Discrete Event Dynamic Systems: Theory Appl.* **6** (1996), 379–427.
6. S. L. Chung, S. Lafortune, and F. Lin, Limited lookahead policies in supervisory control of discrete-event systems, *IEEE Trans. Automat. Control* **37** (1992), 1921–1935.
7. J. E. Hopcroft and J. D. Ullman, ‘‘Introduction to Automata Theory Languages and Computation,’’ Addison-Wesley, Reading, MA, 1979.
8. M. Heymann and F. Lin, On-line control of partially observed discrete event systems, *J. Discrete Event Dynamic Systems* **4** (1994), 221–236.
9. M. Jackson, Problems, methods and specialization, *IEEE Software* **11** (1994), 57–62.
10. K. Jensen, ‘‘Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use,’’ Vol. 1, Springer-Verlag, Berlin, 1992.
11. R. Kumar and V. K. Garg, ‘‘Modeling and Control of Logical Discrete Event Systems,’’ Kluwer Academic Publishers, Boston, 1995.
12. R. Kumar, V. Garg, and S. I. Marcus, On controllability and normality of discrete event dynamical systems, *Systems Control Lett.* **17** (1991), 157–168.
13. M. Makungu, M. Barbeau, and R. St-Denis, Synthesis of controllers with colored Petri nets, *in* ‘‘Proceedings of the 32nd Annual Allerton Conference on Communication, Control, and Computing,’’ Monticello, 1994, pp. 709–718.
14. P. J. Ramadge and W. M. Wonham, Supervisory control of a class of discrete event processes, *SIAM J. Control Optimization* **25** (1987), 206–230.

15. P. J. Ramadge and W. M. Wonham, The control of discrete event systems, *Proc*. *IEEE* **77** (1989), 81–98.
16. E. Rich and K. Knight, ''Artifical Intelligence,'' McGraw-Hill, New York, 1991.
17. W. M. Wonham and P. J. Ramadge, On the supremal controllable sublanguage of a given language, *SIAM J*. *Control Optimization* **25** (1987), 637–659.
18. J. N. Tsitsiklis, On the control of discrete-event dynamical system, *Math*. *Control Signals Systems* **2** (1989), 95–107.