

Supervisory Control of a Rapid Thermal Multiprocessor

Silvano Balemi

Gérard J. Hoffmann

Paul Gyugyi

H. Wong-Toi

Gene F. Franklin

Supervisory Control of a Rapid Thermal Multiprocessor

S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin, *Life Fellow, IEEE*

Abstract—An application of supervisory control theory to a semiconductor manufacturing piece of equipment is presented. This approach allows the flexible design and reliable update of processing “recipes” to accommodate frequently changing manufacturing requirements. An input-output interpretation of supervisory control theory is given. This interpretation leads to a generic implementation scheme for manufacturing systems. A synthesis fixpoint algorithm implementation using binary decision diagrams enables the design of supervisors of realistic size. A sample synthesis for an oxide growth recipe is performed on a state space of the order of 10^6 states. The actual implementation of the logic sequencing control software for the application under investigation is described.

I. INTRODUCTION

THE supervisory control theory introduced by Ramadge and Wonham [1], [2] provides a formal framework for analyzing discrete event logic systems. The possible executions of a system are modeled mathematically, and a system specification describes the desirable executions. The role of a supervisory controller is to interact with the system in order that the closed-loop system meets its specification, i.e., it is to guarantee that undesirable executions do not occur and that certain termination states are reached. The theory provides algorithms for the automatic synthesis of supervisory controllers from their specifications.

Despite this theoretical appeal, there are very few control logic synthesis applications based on supervisory control theory. As of yet, the authors are unaware of any published account of applying supervisory control theory to a manufacturing application. The most closely related work includes an attempt to design protocols for communicating processes [3], and a recent preliminary theoretical study

Manuscript received November 18, 1991; August 7, 1991. Paper recommended by A. Benveniste and K. J. Astrom. This work was supported in part by DARPA under Grant F49620-90-C-0014, in part by Texas Instruments under Grant 7554900, and in part by the Department of the Navy, Office of the Chief of Naval Research under Grant N00014-91-J-1901. This manuscript is published with the understanding that the U.S. Government is authorized to reproduce and distribute reprints for Government purposes not withstanding any copyright notation hereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the Advanced Research Projects Agency or the U.S. Government.

S. Balemi is with the Automatic Control Laboratory, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland.

G. J. Hoffmann, P. Gyugyi and G. F. Franklin are with the Information Systems Laboratory, Stanford University, California.

H. Wong-Toi is with the Department of Computer Science, Stanford University, California.

IEEE Log Number 9210265.

on how to apply supervisory control to the management of manufacturing workcells [4].

Other related work includes the automatic verification of controllers, instead of their automatic synthesis. Here, the supervisor design is performed on a trial and error basis. Verification can be done using techniques originally developed for the formal verification of logic circuits, protocols, and logic programs [5]–[7]. Recently, model-checking verification approaches have been applied to the verification of supervisors for an automated highway car platoon system [8].

A. Difficulties

In the authors' opinion, the lack of automatic synthesis applications that base themselves directly on the Ramadge and Wonham framework is due to three classes of key problems. The following description gives a brief account of the difficulties encountered in the development of a control logic synthesis environment for the project described in this paper.

i) Model Interpretation: The logical plant model proposed in supervisory control theory assumes a plant that “generates” events spontaneously unless it is prevented from doing so. The control mechanism available to the supervisor is the ability to prevent the occurrence of some events, called *controllable* events. This model is not appropriate for most real systems. In fact, real systems usually react to commands as *inputs* with responses as *outputs*.

ii) Computational Complexity and State-Space Explosion: Other synthesis algorithms have complexity that is exponential in the state-space size of the problem input. However, the Ramadge and Wonham framework makes assumptions that ensure that synthesis is only of polynomial complexity in the size of the global system. However, even this computational expense is prohibitive, because the global system size grows *exponentially* with the number of modules. Thus the sheer size of the discrete state-space often renders traditional computational methods infeasible.

iii) Supervisor Implementation: The supervisory control literature provides few indications on how to implement the control software for a discrete event system. The synthesis procedure generates an abstract supervisor model, which is not directly usable in its form for the control software of a real system.

It is the goal of this paper to demonstrate the feasibility and usefulness of supervisory control theory by applying it to a device for the manufacture of semiconductor inte-

grated circuits. This paper reports the difficulties encountered in some detail and tries to provide at least partial remedies for the three classes of problems. By addressing the above-mentioned problems in Sections IV, V, and VIII respectively, the authors hope to provide some guidance for future implementations on one hand and for theoretical research on the other hand.

B. Rapid Thermal Multiprocessing

As manufacturing equipment hardware becomes more flexible in general, and as exploiting this flexibility can only be possible through a higher control software complexity, there is an increasing need for control logic synthesis tools that allow for fast control software prototyping. While software verification tools such as those described in [9] start to become available, relatively few attempts, if any, have been made to implement algorithmic synthesis procedures. Furthermore, there do not exist any environments that allow for the direct implementation in real-time of the synthesized control software. It appears essential to the authors to integrate in the same environment the modeling of the workcell to be controlled, the specification to be enforced, the synthesis of a controller and its implementation.

The specific application presented in this paper is a plant for the manufacture of semiconductor integrated circuits. Traditionally, a semiconductor plant consists of many pieces (*e.g.*, 200-300) of dedicated processing equipment optimized for the production of large batches of wafers. Each workcell in the factory can only perform a very specific operation, with high reliability. However, in the near future, rising equipment costs will result in a prohibitive capital investment to build such enormous plants.

This economic barrier, together with the trend toward smaller series of integrated circuits, necessitates a new concept in semiconductor manufacturing plants: the *microfactory*. A microfactory is composed of a few flexible pieces of equipment (typically 6 or 7) efficiently handling one wafer at a time, each piece performing *multiple* operations.

Such multiprocessing microfactories must have the ability to adapt efficiently to changes in manufacturing requirements, products and processes. Consequently, it is essential to develop a control programming environment that supports the fast development and reliable computer-aided design of real-time multitasking control systems. Wonham and Ramadge's supervisory control theory provides a formal setting for the design of such a system.

An instance of a piece of equipment in a microfactory is a *Rapid Thermal Multiprocessor* (RTM). The RTM described in this paper is under development by Saraswat *et al.* [10]–[12] at the Center for Integrated Systems (CIS) at Stanford University. The RTM consists of a processing chamber capable of performing a number of processing steps such as cleaning, annealing, oxidation and chemical vapor deposition using a multitude of attached processing machinery. A wafer need not be moved from one piece of

equipment to the next in a large factory. Instead, many processing steps can be executed in the same chamber.

A combination of key developments of recent years is expected to make RTM's ready for industrial use. In the Stanford project, dynamic control techniques are being studied to improve the wafer temperature uniformity [13]–[15]. Second, a rapid strategic control logic prototyping is being developed [16]. It must allow the flexible implementation of processing recipes that are changed and updated frequently to accommodate changing processing requirements.

We shall frequently refer to the RTM and its subsystems in the remainder of the paper. A somewhat more explicit description of the system is given in Section VII.

The organization of the paper is as follows. After a review of supervisory control theory in Section III, an input-output interpretation of the formal model is proposed in Section IV. A synthesis procedure based on the manipulation of binary decision diagrams is briefly described in Section V. In Section VI, the plant and supervisor models are refined and a generic scheme for the logic control of a manufacturing system is presented. In Section VII, an illustrative example and a sample manufacturing recipe from the RTM context are discussed. Finally, Section VIII gives an account of the on-line implementation of the control strategies for the RTM, while Section IX presents the implementation of the off-line control synthesis software.

II. PRELIMINARIES AND NOTATION

Let Σ be a finite alphabet of symbols. For any set Σ , let Σ^* denote the set of all finite sequences or strings over Σ . For a string $s = s_1, s_2, \dots, s_n \in \Sigma^*$, we say $\text{len}(s)$, the length of s , is n . We let s_i denote its component at the i th position, if $1 \leq i \leq \text{len}(s)$. The symbol ϵ denotes the empty string. A finite string $t \in \Sigma^*$ is a *prefix* of s if $\text{len}(t) \leq \text{len}(s)$ and $t_i = s_i$ for $1 \leq i \leq \text{len}(t)$. A *language* L over Σ is any subset of Σ^* . The set of all languages over Σ is denoted by \mathcal{L} . Let \bar{L} denote the set of prefixes of strings in L . We say L is *prefix-closed* if $L = \bar{L}$. We denote by $\text{del}(D)(L)$ the language obtained by removing all occurrences of symbols in $D \subseteq \Sigma$ from the strings in L . It is the projection of the language L on the alphabet $\Sigma \setminus D$. Its inverse is $\text{del}^{-1}(D)(L') = \sup\{L \mid \text{del}(D)(L) = L'\}$, i.e., the largest language whose strings with the symbols in D removed are strings in L' .

A *finite-state automaton* \mathcal{A} [17] is a tuple $(\Sigma, Q, \delta, I, F)$, where Σ is a finite alphabet of transition symbols, Q is a finite set of automaton states, $\delta : \Sigma \times Q \mapsto 2^Q$ is a partial transition function mapping a state and a transition symbol to a set of states. If q' is in $\delta(q, \sigma)$, then it is possible to move from state q to q' with the transition labeled by the symbol σ . $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states.

The automaton is *deterministic* if its transition function is deterministic, i.e., if $\delta(q, \sigma)$ is a singleton for every q and σ and there exists only one initial state, i.e., if the set I is

a singleton $\{q_0\}$ for some $q_0 \in Q$. $|\mathcal{A}|$ is used to denote the size of \mathcal{A} , i.e., the number of states in \mathcal{A} .

An *accepting run* of \mathcal{A} on the string $s \in \Sigma^*$ is a sequence q of $\text{len}(s) + 1$ states such that q_1 is in I , q_{i+1} is in $\delta(q_i, s_i)$ for $1 \leq i \leq \text{len}(s)$, and $q_{\text{len}(s)+1}$ is in F . The language generated by \mathcal{A} , denoted $L(\mathcal{A})$, is the set of all strings s with accepting runs. A language L is *regular* if and only if there is some finite state automaton \mathcal{A} such that $L(\mathcal{A}) = L$. For a regular language L , $|L|$ denotes the size of the smallest deterministic automaton \mathcal{A} such that $L(\mathcal{A}) = L$. The class of regular languages is a proper subset of the set of all languages.

III. REVIEW OF SUPERVISORY CONTROL THEORY

In this section, we review some results from supervisory control theory to provide the background for the rest of the paper. Ramadge and Wonham's theory of supervisory control [18], [2] uses formal languages to model both the uncontrolled discrete event system and the specification for the controlled behavior. In their approach, a discrete event system execution is modeled as a sequence of events, an event being a qualitative change occurring in the system. The set of all such sequences forms a *language* and represents all the possible executions of the system.

The basic problem of supervisory control is to modify the open-loop behavior of a discrete event system by eliminating sequences of events from the system behavior. The objective is to restrict the behavior of the system so that it is contained in a desired behavior, called the specification. This is achieved by constraining the discrete event generator to execute events only in strict synchronization with another system, called the supervisor.

We slightly reformulate the standard framework by refraining from introducing automata as models from the beginning. We introduce a pure language-based *process model* instead and formally define *process composition*. A similar approach was already taken in [19] and in the context of ω -regular task specifications in [20]. The reformulation does not affect the main results.

A. Process Model and Process Composition

A process is modeled as a pair $P = (L_P, M_P)$ of languages of finite strings over the alphabet Σ . L_P is a prefix-closed language representing all possible partial executions of P , while $M_P \subseteq L_P$, is a set of distinguished strings, the *marked language*. The language M_P marks the set of successfully *completed* strings.

Let us denote by P_i the process (L_{P_i}, M_{P_i}) over Σ_i . The *composition* $P_1 \parallel P_2$ of two processes P_1 and P_2 is the process

$$P_1 \parallel P_2 = (L_{P_1 \parallel P_2}, M_{P_1 \parallel P_2}) \quad (1)$$

where the alphabet of the composed process is $\Sigma = \Sigma_1 \cup \Sigma_2$. The language $L_{P_1 \parallel P_2} \subseteq \Sigma^*$ is defined as follows

$$\begin{aligned} L_{P_1 \parallel P_2} &= \left\{ s \left| \begin{array}{l} \text{del}(\Sigma - \Sigma_1)(s) \in L_{P_1} \wedge \\ \text{del}(\Sigma - \Sigma_2)(s) \in L_{P_2} \end{array} \right. \right\} \\ &= \text{del}(\Sigma - \Sigma_1)^{-1}(L_{P_1}) \cap \text{del}(\Sigma - \Sigma_2)^{-1}(L_{P_2}). \end{aligned}$$

Analogously, $M_{P_1 \parallel P_2} \subseteq \Sigma^*$ is defined as

$$M_{P_1 \parallel P_2} = \text{del}(\Sigma - \Sigma_1)^{-1}(M_{P_1}) \cap \text{del}(\Sigma - \Sigma_2)^{-1}(M_{P_2}).$$

Note that for the particular case when $\Sigma_1 = \Sigma_2 = \Sigma$, the composition of the two processes yields $P_1 \parallel P_2 = (L_{P_1} \cap L_{P_2}, M_{P_1} \cap M_{P_2})$. The composition operator is both associative and commutative. Thus, the composition of n processes can be defined; we denote it by $\parallel_{i=1}^n P_i = P_1 \parallel P_2 \parallel \dots \parallel P_n$.

B. Supervisor Synthesis Problem

In the following, we model the event generator by a process $P = (L_P, M_P)$ with event set Σ . Ramadge and Wonham postulate that the event generator spontaneously *generates* all events in Σ , and that the events are divided into two classes: *controllable* and *uncontrollable* events. The controllable events Σ_c can be prevented from occurring by synchronization with the supervisor while the uncontrollable events Σ_u are the remaining events over which a supervisor has no authority.

We model the supervisor as a process $S = (L_S, M_S)$ with languages over the alphabet Σ . Then, the generator process P and the supervisor process S jointly executing events can be described by the composition $P \parallel S = (L_{P \parallel S}, M_{P \parallel S})$. $L_{P \parallel S}$ and $M_{P \parallel S}$ are called the *supervised* languages. The language $M_{P \parallel S}$ is marked by both the supervisor and the plant, i.e., all strings in $M_{P \parallel S}$ are contained in both M_P and M_S .

In the synthesis procedure, we are free to choose the supervisor while the generator process is given and cannot be altered. However, a supervisor process cannot be chosen arbitrarily. In fact, as the supervisor cannot prevent the occurrence of uncontrollable events, we are interested only in those supervisors that can always track an uncontrollable event generated by the plant. For a formal characterization of such supervisors, we review the definition of completeness.

A supervisor S is called *complete* for P if

$$P \parallel S = P \parallel (L_S \cdot \Sigma_u^*, M_S). \quad (2)$$

Intuitively, a complete supervisor is such that it can always track the uncontrollable events produced by the generator plant without the ability to prevent them. Replacing the supervisor S by a process $(L_S \cdot \Sigma_u^*, M_S)$ which is modified to always accept any uncontrollable event, but is otherwise the same as S , must leave the behavior of the composed process $P \parallel S$ unchanged.

In addition, we further restrict our attention to the class of supervisors that also guarantee that a partial sequence of events in the closed-loop can always be completed to a marked string in both the generator plant and the supervisor. For this we use the concept of nonblockingness (see Ramadge and Wonham [1], [2]). A process $P = (L_P, M_P)$ is *nonblocking* if any string in L_P can be completed to a marked string in M_P , i.e., if for all strings $s \in L_P$ there exists a string t such that $s.t \in M_P$. In particular, a process P is nonblocking if and only if $L_P = \overline{M_P}$.

We are now ready to formally state the following supervisor synthesis problem ([1, S7]).

Supervisor Synthesis Problem

Given a plant P , find a supervisor S such that

- i) $M_{P\parallel S} \subseteq L_{\text{spec}}$,
- ii) S is complete for P ,
- iii) $P\parallel S$ is nonblocking

where L_{spec} is the specification language for the closed-loop behavior.¹ In particular we note that if $P\parallel S$ is nonblocking, then any string in $L_{P\parallel S}$ can be extended to a string that is marked both in the plant and in the supervisor.

C. Controllability

Ramadge and Wonham [1] introduced the notion of *controllability* to characterize the supervised sublanguages of the generator plant $P = (L_P, M_P)$. A language $K \subset \Sigma^*$ is *controllable* with respect to L_P if $\overline{K} \cdot \Sigma_u \cap L_P \subseteq \overline{K}$.

Supervisor existence was related to controllability in ([1, proposition 5.1 and theorem 6.1]). It was shown that $S = (\overline{K}, K)$ is a complete supervisor for P such that $M_{P\parallel S} = K$ and $P\parallel S$ is nonblocking if and only if K is controllable and $K \subseteq M_P$.

The class of controllable languages is closed under language union. The *supremal controllable sublanguage* of E is defined as $\sup \mathcal{C}(E) = \bigcup \{K : K \subseteq E \text{ and } K \text{ is controllable wrt. } L_P\}$. The following theorem adapted from ([1, theorem 7.1]) gives a necessary and sufficient condition for the existence of a solution to the supervisor synthesis problem.

Theorem 1: The supervisor synthesis problem has a solution if and only if $\sup \mathcal{C}(M_P \cap L_{\text{spec}})$ is nonempty.

The particular supervisor

$$S_{\text{sup}} = (\overline{\sup \mathcal{C}(M_P \cap L_{\text{spec}})}, \sup \mathcal{C}(M_P \cap L_{\text{spec}})) \quad (3)$$

is a solution if and only if $\sup \mathcal{C}(M_P \cap L_{\text{spec}})$ is nonempty. It is also the least restrictive supervisor in the sense that it does not prevent any event sequence that would be allowed by some other supervisor that is also a solution of the supervisor synthesis problem.

D. Iterative Solution

It is shown in [21] that $\sup \mathcal{C}(E)$ is the greatest fixpoint of the operator $\Omega : \mathcal{L} \mapsto \mathcal{L}$ defined as

$$\Omega(K) = E \cap \sup \{T : T \subseteq \Sigma^*, T = \overline{T} \text{ and } T \Sigma_u \cap L_P \subseteq \overline{K}\}.$$

Solving the supervisor synthesis problem reduces to effectively computing the greatest fixpoint of Ω for $E = M_P \cap L_{\text{spec}}$. We assume the generator plant and specification languages are regular languages. Let $\mathcal{A}_{\text{spec}}$ be a deterministic finite-state automaton for L_{spec} . Suppose the generator plant $P = (L_P, M_P)$ is represented by an automaton \mathcal{A}_P for $M_P \subseteq L_P$, where the language L_P is the language obtained by considering all states final. The complexity of computing the greatest fixpoint of Ω is $O(|\mathcal{A}_{\text{spec}}|^2 \cdot |\mathcal{A}_P|^2)$. An efficient implementation strat-

¹In this paper we do not consider a minimally required behavior, but only a maximally tolerable behavior.

egy for this recursive algorithm will be discussed in Section V. An alternative, nonrecursive algorithm of complexity $O(|\mathcal{A}_{\text{spec}}|^2 \cdot |\mathcal{A}_P|)$ for prefix-closed languages was presented in [22].

E. Modular Plant

The system to be controlled is usually composed from modular subsystems. Formally, each subsystem $i \in \{1, 2, \dots, n\}$ can be modeled by a process $P_i = (L_{P_i}, M_{P_i})$. The global uncontrolled system is obtained as $P = \parallel_{i=1}^n P_i$.

The controllable and uncontrollable events are identified at the global level. Note that the composition operator used to describe the concurrent behavior of the plant's components implies interleaving of the events. Events in different components (as long as they are distinct) are assumed to happen at strictly different time instants. The main advantage of interleaving is that the number of possible events in the global plant is bounded by the sum of the number of events in the components instead of their product.

It is easy to see that if the languages of the plant's components are represented by automata, the number of states in the global plant's automaton increases exponentially with n , the number of modular components. This fact is crippling when it comes to computation for realistic systems. This principle is known as *state-space explosion*, and is currently an area of intensive research in the formal verification of finite-state systems.

F. Modular Specification

The specification is typically given as a collection of languages, each of which represents a desirable property of the controlled system. These can be intersected to yield a global specification language. This procedure is subject to state-space explosion just as in the case of a modular plant.

Fortunately, in the case of modular specification, the principles of modular synthesis outlined in [23], [18] can be applied. The supervisor synthesis problem is solved for each modular specification and the resulting supervisors are composed to form a solution to the globally specified problem. In addition to being more easily synthesized, a modular supervisor is easier to modify, update and maintain. For example, if one subspecification is changed, then it is only necessary to redesign the corresponding subcontroller, rather than the entire supervisor.

Let $\mathcal{R}_a(P)$ be the set of complete supervisors for P . The modular synthesis is clearly aided by the following.

Proposition 1: $\mathcal{R}_a(P)$ is closed under composition.

Unfortunately, the composition of supervisors for a system does not preserve nonblockingness of the closed-loop behavior. A global complete supervisor such that the closed-loop system is nonblocking cannot always be constructed by composing modular supervisors. Only under a special condition can a global supervisor be obtained.

For this purpose, Ramadge and Wonham introduced the notion of *nonconflicting* languages. Two languages L_1 and

L_2 are nonconflicting if $\overline{L_1 \cap L_2} = \overline{L_1} \cap \overline{L_2}$. Since it is always the case that $\overline{L_1 \cap L_2} \subseteq \overline{L_1} \cap \overline{L_2}$, two languages are nonconflicting if $\overline{L_1 \cap L_2} \subseteq \overline{L_1} \cap \overline{L_2}$, i.e., whenever they share a prefix, they also share a word containing this prefix. For example, any two prefix-closed languages are nonconflicting.

Consider two specification languages L_{spec_1} and L_{spec_2} with the corresponding supervisors $S_1 = (\overline{K_1}, K_1)$ and $S_2 = (\overline{K_2}, K_2)$, where $K_i = \sup \mathcal{C}(L_{\text{spec}_i} \cap M_P)$ for $i = 1, 2$. Then, if K_1 and K_2 are nonconflicting, a global least restrictive supervisor for the specification $L_{\text{spec}} = L_{\text{spec}_1} \cap L_{\text{spec}_2}$ can be obtained by taking the composition of the modular supervisors S_1 and S_2 .

It was shown in [23] that the total complexity of modular synthesis with n modules is $O(n \times |P|^2 (\max_i |L_{\text{spec}_i}|)^2)$ as opposed to $O(|P|^2 (\max_i |L_{\text{spec}_i}|)^{2n})$ for the nonmodular synthesis. Note that the success of the nonmodular synthesis does not imply the success of the modular synthesis. The complexity of the modular synthesis procedure is thus conditional on its successful completion.

There exist few *a priori* conditions on the plant and specification languages under which two supervisor languages are nonconflicting. For example, it is insufficient to assume that the specification languages are nonconflicting. Some important classes of specification and plant languages that guarantee nonconflictingness of the languages of the supervisors are worthy of mention here.

- *Prefix-Closed Languages:* If M_P , L_{spec_1} and L_{spec_2} are prefix-closed, K_1 and K_2 are nonconflicting.

- *Local, Decentralized Languages:* This class of languages takes advantage of the modularity of the plant. Let P_1 and P_2 be two plant components with respective disjoint alphabets Σ_1 and Σ_2 . For local specification languages $L_{\text{spec}_1} \subseteq \Sigma_1^*$ and $L_{\text{spec}_2} \subseteq \Sigma_2^*$ we have that $\text{del}(\Sigma - \Sigma_1)^{-1}(\sup \mathcal{C}(M_{P_1} \cap L_{\text{spec}_1}))$ and $\text{del}(\Sigma - \Sigma_2)^{-1}(\sup \mathcal{C}(M_{P_2} \cap L_{\text{spec}_2}))$ are nonconflicting.

Another condition that guarantees nonconflictingness is the “*nesting*” property of specifications that was presented in [23]. This property is not exploited further in this paper.

Modular design can also be done for general languages that do not meet any of the *a priori* conditions above. This involves finding “noninner-blocking modular supervisors” as proposed in [24]. These supervisors can be found first by computing $K_1 = \sup \mathcal{C}(M_P \cap L_{\text{spec}_1})$ and then

$$\tilde{K}_2 = \sup \mathcal{C} \& \mathcal{N} \mathcal{C}(M_P \cap L_{\text{spec}_2}, K_1), \quad (4)$$

the supremal controllable and nonconflicting sublanguage of $M_P \cap L_{\text{spec}_2}$ (wrt. to K_1). The following inclusion holds.

$$K_1 \cap \tilde{K}_2 \subseteq \sup \mathcal{C}(M_P \cap L_{\text{spec}_1} \cap L_{\text{spec}_2}). \quad (5)$$

If the language \tilde{K}_2 is nonempty, we can construct modular supervisors for the languages K_1 and \tilde{K}_2 . Note that because inclusion (5) can be proper, the composition of two modular supervisors with language K_1 and \tilde{K}_2 does not necessarily yield the globally least restrictive supervisor. Moreover, if the language of (4) is empty, this does not imply that $\sup \mathcal{C}(M_P \cap L_{\text{spec}_1} \cap L_{\text{spec}_2}) = \emptyset$ and a non-

modular solution to the supervisor synthesis problem may still exist.

Before concluding, we point out the benefits of *incremental* modular design if it cannot be guaranteed that K_1 and K_2 are nonconflicting. The supremal controllable sublanguage $\sup \mathcal{C}(M_P \cap L_{\text{spec}_1} \cap L_{\text{spec}_2})$ can be computed directly from $M_P \cap L_{\text{spec}_1} \cap L_{\text{spec}_2}$, or alternatively it can be obtained as $\sup \mathcal{C}(K_1 \cap L_{\text{spec}_2})$. If for the problem at hand the comparison of computational cost is such that

$$|L_{\text{spec}_2}|^2 \times |K_1|^2 + |L_{\text{spec}_1}|^2 \times |P|^2 \ll |L_{\text{spec}_2}|^2 \times |L_{\text{spec}_1}|^2 \times |P|^2,$$

then incremental synthesis becomes attractive. Moreover, incremental synthesis can in general represent a saving in computation if K_1 is already known from a previous computation.

IV. AN INPUT-OUTPUT INTERPRETATION

A. Plant Model

The model interpretation proposed by Ramadge and Wonham consists of a plant event “generator” that “wildly” produces events; the only way to affect the behavior of the plant is by disabling the controllable events. In their semantics, the plant alone *schedules* the occurrence of all events. This model interpretation is graphically rendered by the left-hand side of Fig. 1.

In the authors’ opinion the model of a plant generating all events is not always accurate for real systems; an input-output perspective is required [16]. In fact, events do not usually occur spontaneously, but only as *responses* to *commands*. For instance, system actuators are activated by commands, while responses report sensor configuration changes.

The formal generator plant model of Section III consists of a process $P = (L_P, M_P)$ with the alphabet Σ partitioned into two disjoint subalphabets $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$. The partition of Σ is now interpreted differently. The elements of Σ_c model the inputs of the plant whereas the elements of Σ_u stand for the plant outputs. From now on we refer to the inputs as *commands* and to the outputs as *responses*. The input-output model $P = (L_P, M_P)$ can be interpreted as a transfer function from the input behavior $\text{del}(\Sigma_u)(L_P)$ to the output behavior $\text{del}(\Sigma_c)(L_P)$. Graphically, we illustrate this with the right-hand side of Fig. 1.

B. Supervisor Model

In the original model, the supervisor acts as a passive device, tracking events produced by the plant and restricting the behavior of the plant by dynamically disabling the controllable events (see Fig. 2, left-hand side).

In the proposed input-output perspective, the supervisor does not simply prevent controllable events from occurring (by means of the synchronization $P||S$) but actually triggers or *forces*² commands to the input of the plant. The generation of events is therefore initiated not only by the

²This notion of forcing events differs from the one presented in [25].

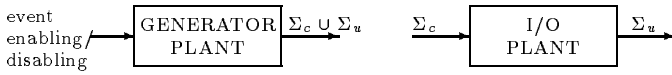


Fig. 1. Generator plant and input-output plant.

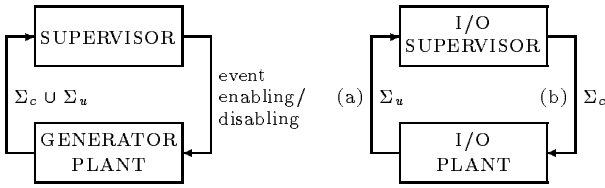


Fig. 2. Asymmetric (left) and symmetric (right) feedback loops.

plant, but by both the plant and the supervisor. Commands are produced by the supervisor, and responses by the plant (see Fig. 2, right-hand side).

C. Closed-Loop System

When connecting the plant with a supervisor, it is important to have a precise model of how the synchronization between both processes is performed. In this paper we distinguish two sorts of synchronization between processes.

a) Full Synchronization: an event that is shared between two or more processes must be agreed upon by all processes who carry the event in their alphabet.

b) Prioritized Synchronization: Certain events can be initiated by a process without consulting the other processes. These events will occur regardless of whether the other processes can execute them.

The process resulting from the connection of two processes under full synchronization is described by the composition given by (1). For a formal definition of a composition operator corresponding to prioritized synchronization and for a detailed description of synchronization modes see [26].

Ramadge and Wonham assume in their model that the set of uncontrollable events cannot be prevented from occurring. The uncontrollable events will occur regardless of whether the supervisor can “accept” them. If we choose to have only closed-loop connections of plant and supervisor in which the supervisor always accepts the uncontrollable events from the plant, the supervisor must be chosen to be complete for the plant and therefore must satisfy (2).

From an input-output perspective the closed-loop behavior of a plant with a supervisor can intuitively be thought of as follows. Let the plant and the supervisor be understood as finite-state constructs. Out of the commands and responses from its current state, the supervisor *schedules* the commands to be transmitted to the plant. The plant schedules the responses from its internal state. Both plant and supervisor processes can be thought of as *competing* for the first occurrence of one of the events they trigger. Both plant and supervisor behave like active scheduling units, while in the original model only the plant has that authority.

Both in the original Ramadge and Wonham model and from an input-output perspective, (2) can be understood as the condition under which the connection of plant and supervisor yields the same closed-loop behavior under either full synchronization or prioritized synchronization for part (a) of the feedback-loop of Fig. 2, i.e., the plant forces the unbuffered communication between the plant and the supervisor processes without consulting the supervisor.

The input-output view of the connection of the plant and the supervisor leads to the analogous constraint that the plant also cannot prevent the occurrence of commands produced by the supervisor. This choice affects part (b) of the feedback-loop in Fig. 2.

The assumption implies that the plant must be complete for the supervisor. Formally, a plant P is complete for S if

$$P||S = (L_P\Sigma_c^*, M_P)||S \quad (6)$$

Equation (6) can be seen as a *dual* to (2).

Let $\mathcal{R}_b(P)$ be the set of all supervisors for which the plant P is complete. It is easy to see the following proposition.

Proposition 2: $\mathcal{R}_b(P)$ is closed under composition. Furthermore, if either S_1 or S_2 is in $\mathcal{R}_b(P)$ but not necessarily both, $S_1||S_2$ is in $\mathcal{R}_b(P)$.

It follows from Propositions 1 and 2 that $\mathcal{R}(P) \triangleq \mathcal{R}_a(P) \cap \mathcal{R}_b(P)$ is closed under composition. We say the plant P and a supervisors from $\mathcal{R}(P)$ are *mutually complete*. In other words, a supervisor from $\mathcal{R}(P)$ satisfies both completeness conditions (2) and (6).

We now state in the following lemma a result which we need later.

Lemma 1: Let $P = (L_P, M_P)$ be a plant and $S = (L_S, M_S)$ a supervisor. If $L_S \subseteq \overline{M_P}$ then the plant is complete for the supervisor and therefore (6) is satisfied.

Proof: It is easy to see that (6) is equivalent to $L_S \cap L_P = L_S \cap L_P.\Sigma_c^*$. This is always true if $L_S \subseteq \overline{M_P} \subseteq L_P$. ■

D. Controller Synthesis Problem

We are now ready to present a slightly different synthesis problem which incorporates the additional condition (6). To distinguish the newly introduced input-output semantics from the generator disabling semantics, we call the input-output supervisor a “controller.” The following problem corresponds to the newly introduced controller semantics, which postulates that events can be generated both in the controller and the plant.

Controller Synthesis Problem

Given a plant $P = (L_P, M_P)$, find a controller $C = (L_C, M_C)$ such that

- i) $M_C||P \subseteq L_{\text{spec}}$,
- ii) C is complete for P ,
- iii) P is complete for C ,
- iv) $P||C$ is nonblocking

where L_{spec} is the specification language for the closed-loop behavior.

Conditions ii) and iii) require that the supervisor and the plant be mutually complete. The conditions are symmetric for the input and output part of the feedback loop between the controller and the plant.

Theorem 2: The controller synthesis problem has a solution if and only if the language $\text{sup } \mathcal{C}(M_P \cap L_{\text{spec}})$ is non-empty.

Proof: (only if) Follows directly from Theorem 1. (if) Existence of a solution to the supervisor synthesis problem implies that S_{sup} of (3) is a solution. But for S_{sup} , $L_S \subseteq \overline{M_P}$ and therefore, because of Lemma 1, (6) is satisfied. ■

Any supervisor S that solves the supervisor synthesis problem for P and that satisfies $L_S \subseteq \overline{M_P}$ is a solution to the controller synthesis problem. If S solves the supervisor synthesis problem, but $L_S \not\subseteq \overline{M_P}$, the controller

$$C = S \parallel P \quad (7)$$

is a solution to the controller synthesis problem. This is an important observation, as it allows the use of the so called “reduced” supervisors [27] solving the supervisor synthesis problem. These supervisors, in the case of regular plant and specification languages, have the nice property of being represented by automata with relatively small state size, but in general do not satisfy $L_S \subseteq \overline{M_P}$. Even more importantly, this observation allows us to use all the modularity-based techniques discussed in Section III-F for the controller synthesis problem.

V. SUPERVISOR SYNTHESIS

In this section, we describe the off-line synthesis of supervisors and controllers. For supervisory control to be useful in practice, the synthesis algorithms must be computationally feasible for practical problems. We demonstrate here the use of a special data structure that does indeed enable controller synthesis for real-world applications such as the RTM.

A. Difficulties

The solution of the supervisor synthesis problem involves the iterative computation of the controllability fixpoint as outlined in Section III-D. The straightforward way to do this involves a brute force enumeration technique that re-

quires all discrete states in the system to be considered individually. This approach, however, is not at all practical

because of the exponential state-space explosion occurring in modular systems as shown in Section III-E.

B. Logical Encoding of the Fixpoint Operator

To overcome these difficulties the fixpoint operator is re-expressed in a *logical* form. This allows the fixpoint computation to be performed *symbolically*, thereby avoiding the explicit enumeration of states in the global automaton.

First, we need to encode all the relevant information about the plant and its specification as logical formulas. We use formulas of boolean propositional logic. Notice that the transition functions, the initial and final states of a plant automaton and its specification automaton can each be given as sets of tuples [28]. For example, the next-state relation of an automaton can be considered to be the set of all tuples $\langle q_1, \sigma, q'_2 \rangle \in Q \times \Sigma \times Q'$, such that $q_2 \in \delta(q_1, \sigma)$. The set $Q' = \{q' \mid q \in Q\}$ represents the successor states. A set of tuples $T \subseteq D_1 \times D_2 \times \dots \times D_n$ can in turn be thought of as a boolean function, namely a mapping from $D_1 \times D_2 \times \dots \times D_n$ to $\{0, 1\}$ that returns 1 if and only if the tuple is in T . Furthermore, any nonboolean domain D can be reencoded as a vector of boolean bits. It follows then that next-state relations, predicates describing initial states, final states and uncontrollable events can all be expressed as boolean functions over boolean domains.

The controllability fixpoint operator Ω defined in Section III-D can be re-expressed as follows in terms of boolean functions describing the plant and specification automata.

Assume the uncontrollable events are expressed as the boolean formula Σ_u over the domain Σ , and the plant and specification are given as deterministic finite-state automata A_P and A_{spec} with next-state relations and final-state predicates δ_P and F_P , and δ_{spec} and F_{spec} respectively. Let $F_{P,\text{spec}}$ be the predicate $F_P \wedge F_{\text{spec}}$, i.e., the final-state predicate over the domain $Q_P \times Q_{\text{spec}}$ that returns true if a state is final in both the plant and the specification. Let $\delta_{P,\text{spec}}$ be $\delta_P \wedge \delta_{\text{spec}}$, the next-state relation of the product automaton. $\delta_{P,\text{spec}}$ is a boolean function over the domain $W = Q_P \times Q_{\text{spec}} \times \Sigma \times Q'_P \times Q'_{\text{spec}}$. The combined state from $Q'_P \times Q'_{\text{spec}}$ is the state reached from a state in $Q_P \times Q_{\text{spec}}$ for an input from Σ . If Z is a function over the same domain, the next-state relation for the least restrictive controller is the greatest fixpoint of the operator $\Omega : 2^W \mapsto 2^W$, defined as:

$$\begin{aligned} \Omega(Z) = & \delta_{P,\text{spec}}(s_1, t_1, \sigma, s_2, t_2) \wedge Z(s_1, t_1, \sigma, s_2, t_2) \wedge \forall s, t. [\exists s_3, t_3, \sigma_3. Z(s, t, \sigma_3, s_3, t_3) \vee Z(s_3, t_3, \sigma_3, s, t)] \\ \Rightarrow & \left[\underbrace{[\forall \sigma', s'. \Sigma_u(\sigma') \Rightarrow [\delta_P(s, \sigma', s') \Rightarrow \exists t'. Z(s, t, \sigma', s', t')]]}_{(I)} \wedge \underbrace{\mathcal{BR}(F_{P,\text{spec}}, Z)(s, t)}_{(II)} \right]. \end{aligned}$$

quires all discrete states in the system to be considered individually. This approach, however, is not at all practical

$\mathcal{BR}(F_{P,\text{spec}}, Z)$ is a predicate for the set of tuples that are backwards reachable from the final states under the

transitions in Z . It is itself computed as a fixpoint. Intuitively, Condition (I) asserts that no string may lead to an unpermitted uncontrollable event (guaranteeing the *safety* property of controllability). Condition (II) permits only strings that can successfully be extended to complete executions (a *liveness* property). The operator Ω iteratively removes events that cause the violation of either of these two properties. We state the main result as a theorem [28].

Theorem 3: The fixpoint iteration $Z_{i+1} := \Omega(Z_i)$ with $Z_0 = \delta_{P, \text{spec}}$ effectively computes a boolean function that symbolically represents the next-state relation of an automaton for the supremal controllable sublanguage $\text{sup } \mathcal{C}(L_{\text{spec}} \cap M_P)$ of Section III-D.

Proof: This follows from [21, proposition 5.1], which states that Ω is effectively computable if the languages are regular. The representation of Ω in terms of logical formulas as described above is a re-expression of Ω . ■

C. An Efficient Data Structure: The Binary Decision Diagram

In order to gain efficiency as a result of using a logical fixpoint operator, it must be possible to store and manipulate propositional logic expressions with significantly less computation than an explicit representation. The representation of the logical expression we choose is based on a special compact data structure known as Binary Decision Diagram (BDD) [29]. Binary decision diagrams have been shown to be an efficient way to encode boolean functions [29], [30].

They are essentially binary decision trees with the added restriction that the order of decisions respects a fixed ordering. The decision tree is represented by an acyclic directed graph. Fig. 3 shows a BDD for the function $f = (x_1 \vee x_2) \wedge (x_3 \vee x_4)$. The value of the function for a particular variable assignment can be read by following a path in the tree from the root to a leaf. Each internal node is labeled with a boolean variable. At each node the path follows the branch that corresponds to the value of that variable in the variable assignment. In the example shown above, the variable assignment $(x_1=0, x_2=1, x_3=1, x_4=1)$ leads to a leaf marked 1, indicating that f is true under this variable assignment. The operations of logical AND, OR, NOT, and existential quantification can all be performed on BDD's in polynomial time.

The main advantage of using BDD's to represent boolean functions is that they are often far smaller than an explicit truth table representation. This fact can lead in practice to greatly improved performance but does not alter the exponential worst-case complexity per se. Research in the field of formal finite-state verification has shown that BDD's can be used to dramatically extend the capability of traditional explicit algorithms [6], [30].

VI. MODELING PRINCIPLES

A. Plant Model Refinement

In this subsection we present a methodology for obtaining an input-output model for a plant component.

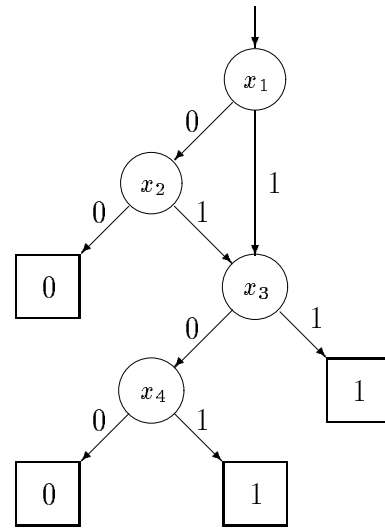


Fig. 3. BDD for the boolean function $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$.

A.1 Logical Plant versus Physical Plant

The plant is not readily available as a logical plant model. It needs to be brought from a *physical* level where we deal with voltages and bits to a *logical* level that is suitable for a behavioral language specification.

We first describe the role of the *interface* used to map the physical plant with its hardware, sensor and actuator routines into the logical plant. The *interface* extracts information from the whole behavior of the system and forwards it in the form of a response to its output channel. In the other direction, the interface receives the commands from its input channel; it interprets them, and performs on the system the operations corresponding to the information contained in them.

The interface is therefore responsible for providing the supervisor with the necessary information and enforcing the commands received from the supervisor.

Information Extraction

We remind the reader that the description of the behavior of the system at a logical level of abstraction is given by languages whose alphabet is composed of commands and responses. While the commands originate from the supervisor, the responses report qualitative changes occurring in the system or system messages.

Most responses are extracted from the system directly or indirectly by sensors.

- *Event Sensors:* Physical system configuration changes are sensed directly. This requires sensors detecting changes; those sensor changes are converted to messages which are directly forwarded to the plant output. As an example, the pushing of a button is directly sensed as an event.

- *Discrete State Sensors:* The interface has access to *discrete* configuration states. After a state change is detected, the response symbol between the previous and the current state is determined. As an example, the configuration state of a gas valve (see Fig. 5) is polled in reg-

ular intervals. Then, if the configuration state just read is `valve_open` and one previously read was `valve_closed`, the state change is converted into a response symbol corresponding to the closing of the valve.

- *Continuous State Sensors*: The interface monitors a number of continuous states and maps them to a discrete state set. This state set is chosen such as to provide enough information for supervisory specification and control. By detecting transitions between those discrete states, the response symbols are generated. As an example, consider the mapping of a temperature sensor reading to the discrete states *cold*, *processing temperature*, and *hot*.

Besides responses that correspond to physical configuration changes, also a number of responses are produced by the interface software.

- *Software Messages*: Such messages report on the status of the software. As an example, one may think of the report of a soft failure (e.g., triggered by a timeout) by a *fail* response.

Action Enforcement

The interface is also responsible for interpreting the commands and acting on the physical system, in accordance with the information contained in the commands.

A command could just require the call of a routine that performs some desirable configuration changes. However, it could also communicate to the physical plant how a continuous subsystem should evolve. It may determine a set-point or a continuous trajectory to be followed, or carry information about which continuous controller for a control module should be used. We can therefore distinguish the following cases for the actions caused by a command.

- *Discrete Actions*: Most commands activate directly a discrete state change in the system hardware. For instance, the command `c_open_valve` causes the valve to pass from the state `valve_closed` to the state `valve_open`.

- *Continuous Actions*: The commands given to the interface often carry implicitly continuous attributes and initiate a continuous control action. As an example, the command “*heat the wafer to the maximal temperature*” causes the temperature control subsystem to control the temperature of the wafer at the maximal operating temperature.

- *Changes in Software Execution*: Commands can also influence the execution of a program by pure software instructions without directly affecting the hardware. Such a command could start a timer or substitute the control algorithm for the temperature subsystem with a different algorithm.

A.2 Plant Model Construction

The task of modeling a component can be decomposed into three parts. While not all processes are easily modeled using this methodology, the steps described below have proven to be helpful for modeling the RTM, and are general enough to be of widespread practicality. The main idea is to form the hierarchical combination of a *fundamental* pro-

cess and some routines corresponding to events occurring in the *actuating* and *sensing* processes.

Step 1) Model the high-level behavior of the plant. This results in the description of a fundamental process Ξ that best matches the structure of the system to be controlled.

Step 2) Design a logical interface to the physical plant, starting from the fundamental process Ξ . This consists of choosing low-level routines Ψ_i that interact with the physical system, described by the actuating and sensing processes.

Step 3) Compose the description of the fundamental process with the above routines.

The fundamental process $\Xi = (L_\Xi, M_\Xi)$ with alphabet $\Sigma_\Xi \subseteq \Sigma$ models the qualitative changes that can occur in the system under consideration. It can be thought of as the fundamental behavior that is consistent with the physical equipment to be controlled. It represents the most abstract view of the system, because low-level information about the events not in Σ_Ξ is excluded.

At a more detailed level of modeling, each event in the fundamental process corresponds to a sequence of events taking place in the *actuating* and *sensing* processes. These underlying sequences of events are modeled as separate processes or subroutines, $\Psi_i = (L_i, M_i)$, $1 \leq i \leq p$. Note that the languages M_i are not necessarily prefix-closed, i.e., $M_i \subset L_i$.

Finally the plant process $P = (L_P, M_P)$ is constructed by taking

$$L_P = \overline{\text{del}(\Sigma - \Sigma_\Xi)^{-1}(M_\Xi) \cap [M_1 \cup M_2 \cup \dots \cup M_p]^*} \quad (8)$$

and $M_P \subseteq L_P$. The symbol * stands for the extension of the Kleene closure to languages.

The construction procedure for the gas valve is now illustrated. The complete alphabets of commands and responses that are modeled are

$$\Sigma_u = \{\text{r_valve_failed}, \text{r_valve_opened}, \text{r_valve_closed}\}$$

$$\Sigma_c = \{\text{c_open_valve}, \text{c_close_valve}, \text{c_repair_valve}\}.$$

We adopt the convention that the transition labels starting with “c_” and “r_” denote commands and responses respectively. First, the fundamental process for the gas valve is modeled by an automaton over the alphabet

$$\Sigma_\Xi = \{\text{r_valve_opened}, \text{r_valve_closed}, \text{r_valve_failed}, \text{c_repair_valve}\}.$$

The automaton is shown in Fig. 4. The only marked state is the initial state, the marked language M_Ξ is

$$M_\Xi = \{((\text{r_valve_opened.r_valve_closed}) + (\text{r_valve_failed.c_repair_valve}))^*\}$$

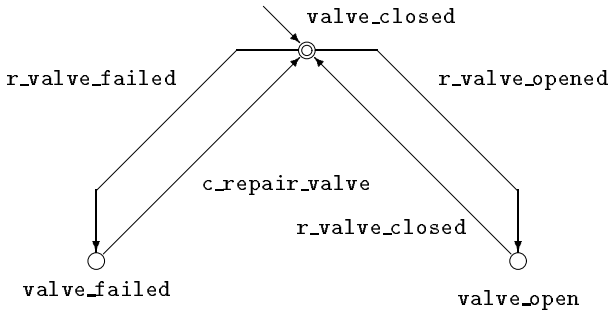


Fig. 4. Automaton model of the fundamental valve process.

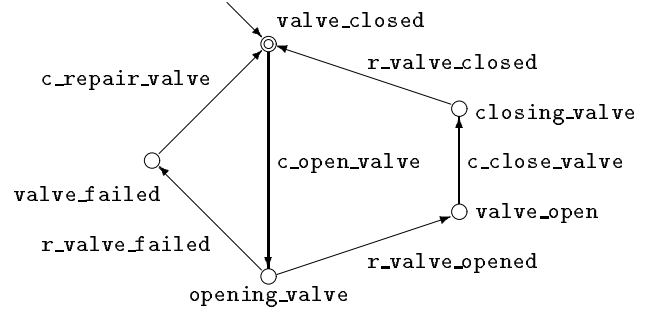


Fig. 5. Automaton model for the gas valve.

and $L_{\Xi} = \overline{M_{\Xi}}$. Second, the sensing and actuating processes Ψ_i are given by the languages

$$M_1 = \{c_open_valve.(r_valve_opened + r_valve_failed)\}$$

$$M_2 = \{c_close_valve.r_valve_closed\}$$

$$M_3 = \{c_repair_valve\}$$

and $L_i = \overline{M_i}$. The sequences in M_1 correspond to sensors that indicate whether the command to open the valve is successful or not. The strings in M_2 and M_3 are more detailed descriptions of sequences of events corresponding to closing and repairing the valve.

Third, intersecting the languages as in (8) yields the complete input-output plant model of Fig. 5. Only the initial state of the automaton is marked.

$$M_P = \{ (c_open_valve.(r_valve_opened.c_close_valve.r_valve_closed + r_valve_failed.c_repair_valve))^* \}$$

and $L_P = \overline{M_P}$.

B. Controller Model Refinement

Here we describe a methodology to obtain a reactive logic sequencing controller from its specification.

B.1 Generic Control Program Structure

We propose combining the disabling supervisor semantics as reviewed in Section III with the controller semantics introduced in Section IV. The proposed generic control structure for the input-output plant is composed of two different control entities [16]. This two-tiered control structure is illustrated in Fig. 6.

The goal is to design a global controller C' that meets a behavioral specification L_{spec} . This global controller is composed of an active scheduling controller C and a passive disabling supervisor S .

While the controller C generates commands for the plant, the supervisor S can only disable commands that emanate from the controller via a synchronization mechanism that is described below; it cannot generate events of its own. In the proposed control structure the supervisor can be regarded as a dynamic *filter* for the commands from the controller.

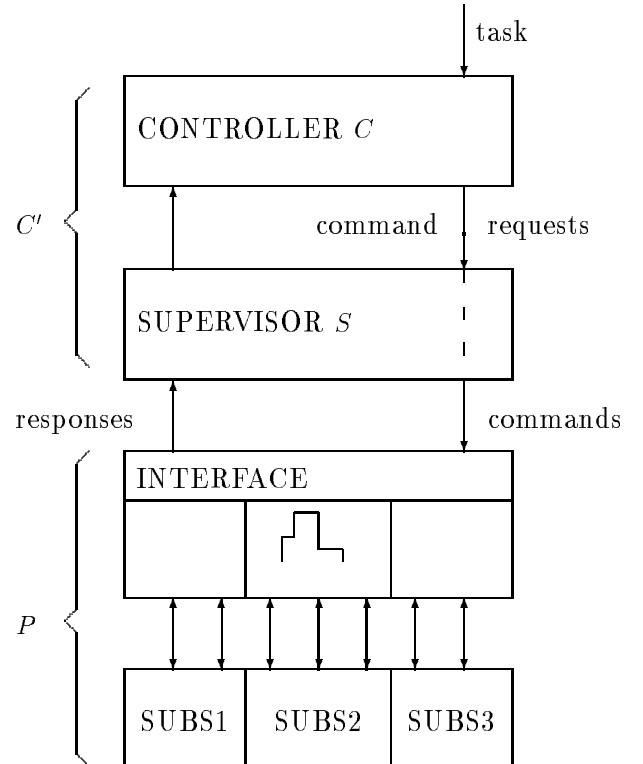


Fig. 6. Generic scheme for the control of a discrete event system.

The supervisor-plant pair $P||S$ is seen by the controller as a new reactive plant, accepting commands and generating responses. The controller has the capability of actively driving the plant $P||S$ to a desired state, while the supervisor S passively prevents the plant $P||S$ from entering undesired states. Note that without loss of generality, the controller can be a human operator sending commands according to an observed status display.

B.2 Operational Requirements

The separation of the control unit into a controller actively scheduling commands for the plant and a supervisor dynamically allowing only certain commands to be scheduled by the controller can be explained from an *on-line operation* perspective. In fact, we are interested in the distinction between *manual* and *automatic* control.

- *Manual Control*: A system must be accessible for manual control for various reasons. This requires the possibility of giving a command to the equipment and having the command checked for consistency with operational and safety constraints that must always be enforced. Clearly, this is possible only if a human operator can bypass the controller and can access directly the supervisor. In this case, a user can operate the system and be sure not to violate the operational constraints. The supervisor plays the role of an intelligent *interlock* system.

- *Automatic Control*: When the system is in automatic control mode, the controller and the supervisor together autonomously control the manufacturing process without user intervention. They form an active scheduling unit for automatic task completion.

B.3 Implicit and Explicit Specifications

The separation into two distinct control units is also justified by a separation of the behavioral specifications for the closed-loop system. We distinguish between different classes of specifications; this will be discussed in some detail as it is relevant for both synthesis and implementation. A manufacturing task specification (called *recipe* in the semiconductor manufacturing context) typically consists of two parts. Some specifications are task-specific or *explicit* while others that are equipment-specific and do not relate to a particular task are *implicit*. The implicit specifications are enforced by the supervisor, while the explicit specifications are enforced by the controller.

Implicit Specifications: These specifications are typically equipment-related and task-independent. Therefore, they must always be met throughout operation of the plant. The supervisor's enforcement of these specifications restricts the choice of commands made available to the controller. These specifications do not require the plant to be actively driven to terminate a desired event sequence. We shall restrict ourselves to the following predominant categories of implicit specifications.

- a) *Safety Specifications*: Formally, a specification language L_{spec} is a safety specification for the plant P if both M_P and L_{spec} are prefix-closed. Intuitively, such a specification only prevents the plant from doing certain sequences; it does not enforce the termination of particular sequences.

- b) *Fundamental Liveness Specifications*: A fundamental liveness specification is one that enforces a repetitive behavior, i.e., a specification L_{spec} is a fundamental liveness specification if $L_{\text{spec}}^* \subseteq L_{\text{spec}}$. Intuitively, any tolerable behavior must be repeatable an arbitrary number of times. A fundamental liveness specification can specify that the initial system configuration should always be reachable. Such a specification can be modeled by a strongly connected automaton \mathcal{A} such that its final state set $F_{\mathcal{A}}$ contains only the initial state q_0 .

Explicit Specifications: These specifications are typically task-specific. Their enforcement by the controller requires commands to be actively sent to the plant in order to accomplish the desired task, i.e., in order to complete

a marked sequence. Explicit specifications are subdivided into:

- a) *Implicit Specifications*.
- b) *General Liveness Specifications*.

General liveness specifications encompass all possible liveness specifications given for the plant; they are not necessarily prefix-closed nor fundamental liveness specifications. In particular, this category contains all those specifications that require the active termination of certain logical sequences.

The distinction between active, explicit (embodied by $L_{\text{spec}}^{\text{expl}}$) and passive, implicit ($L_{\text{spec}}^{\text{impl}}$) constraints on the closed-loop behavior naturally motivates the proposed two-tiered control structure. The global specification is $L_{\text{spec}} = L_{\text{spec}}^{\text{impl}} \cap L_{\text{spec}}^{\text{expl}}$.

B.4 Control Program Design Methodology

Based on the previous discussion, we shall now describe a general purpose methodology for a valid control structure synthesis.

Step 1) Form the global plant $P = P_1 || P_2 || \dots || P_n$ from the modular plant components.

Step 2) Solve the *supervisor synthesis problem* for the plant P and the *implicit* specification $L_{\text{spec}}^{\text{impl}}$ to obtain S' . A solution is the least restrictive supervisor

$$S' = (\overline{(\sup \mathcal{C}(M_P \cap L_{\text{spec}}^{\text{impl}})}), \sup \mathcal{C}(M_P \cap L_{\text{spec}}^{\text{impl}})}).$$

Compose S' with a copy of the plant to obtain $S = P || S'$. This makes sure that the controller-supervisor pair $S || C$ and the plant P are mutually complete for all choices of C .

Step 3) Solve the *controller synthesis problem* for the plant P and the *explicit* specification $L_{\text{spec}}^{\text{expl}}$ to obtain C . A solution is the least restrictive controller

$$C = (\overline{(\sup \mathcal{C}(M_P \cap L_{\text{spec}}^{\text{expl}})}), \sup \mathcal{C}(M_P \cap L_{\text{spec}}^{\text{expl}})}).$$

Step 4) The complete discrete event control structure is obtained as the composition $C' = S || C$ (see Fig. 6).

The control program modeled by $C' = S || C$ satisfies the specification $L_{\text{spec}} = L_{\text{spec}}^{\text{impl}} \cap L_{\text{spec}}^{\text{expl}}$. Moreover, if the marked languages of the controller and the supervisor are non-conflicting, C' solves also the *controller synthesis problem* for the plant P and the specification $L_{\text{spec}} = L_{\text{spec}}^{\text{impl}} \cap L_{\text{spec}}^{\text{expl}}$.

For the particular case where $L_{\text{spec}}^{\text{expl}} \subseteq L_{\text{spec}}^{\text{impl}}$, i.e., if the controller C enforces both implicit and explicit specifications, $C' = S || C = C$. The supervisor S is superfluous for this particular assumption. The commands generated by C are always accepted by the supervisor S .

If an arbitrary controller C (which can be modeled as a (complete) random generator of commands $C = (\Sigma^*, \Sigma^*)$) is picked, i.e., Step 3) is bypassed, the combined structure $S || C$ only satisfies the implicit specification $L_{\text{spec}}^{\text{impl}}$. This is typically the case if the controller is a human operator.

B.5 Separation Benefits for Synthesis

From a synthesis standpoint, the partition into two supervisors results in practice in a reduced computational ef-

fort. As the implicit specifications are typically more modular than the explicit specifications, it makes sense to synthesize the control structure separately, possibly exploiting fast modular techniques which may not be applicable for the controller (see Section III-F). Some of the more common specification structures that allow a modular design are now briefly mentioned.

- *Modularity of Safety Specifications:* The prefix-closedness of safety specifications is a sufficient condition for a straightforward modular supervisor synthesis.

- *Modularity of Local Specifications:* Another *a priori* condition on specification languages that allows for straightforward modular synthesis is the localness of specification languages with disjoint alphabets. Clearly this condition applies to both safety and liveness specifications.

- *Modularity of General Liveness Specifications:* Here the practical benefits of incremental synthesis apply. Alternatively, the minimally restrictive “noninner-blocking modular supervisors” can be obtained [24].

The explicit specification is typically not as modular as the implicit specification. Also, the fact that the controller is particular to the task to be performed makes modular design less attractive. In fact, one of the advantages of modular design is the easy maintenance and update of supervisors, when only a small part of the specification is changed.

The following practical reason for the separation into two clearly distinct units is valid from both an on-line operation and an off-line synthesis standpoint.

- *Possible Use of Incomplete Controllers:* A heuristic supervisor design does not necessarily yield a valid solution to the controller synthesis problem. A heuristic that does not explore the entire state space is beneficial for both the synthesis effort as well as the on-line storage needs.

Note that the choice of an incomplete controller C , if it is not a solution to the controller synthesis problem with $L_{\text{spec}}^{\text{expl}}$ [Step 3], implies that also $C||S$ is not necessarily a solution to the problem with specification language $L_{\text{spec}}^{\text{expl}} \cap L_{\text{spec}}^{\text{impl}}$. However, in such a case, we want at least to make $C||S$ complete for the plant P . Assume that the supervisor S is the solution to a supervisor synthesis problem. In order to guarantee that also $C||S$ is complete for the plant, it is sufficient, because of Proposition 1, that C be made complete for the plant such as to satisfy condition (2). In particular, we extend $C = (L_C, M_C)$ to $C_{\text{ext}} = (L_{C_{\text{ext}}}, M_{C_{\text{ext}}}) = (L_C \cdot \Sigma_u^*, M_C)$. Then, any $s \in L_{C_{\text{ext}}} \setminus L_C$ is considered to be a *failure* of the controller. The partial executions of the closed-loop system are guaranteed to be contained in $L_{\text{spec}}^{\text{impl}}$.

B.6 Synchronization Issues

To conclude the control structure discussion, we briefly describe how the synchronization between the different processes of plant, controller and supervisor is affected by the partition of the control unit into supervisor and controller.

The completeness specification (2) on the control unit $C' = C||S$ as a whole, implies that every response coming from the plant P can be accepted by both processes S and C . The full synchronization that is usually assumed between communicating processes can be replaced by a prioritized synchronization, in which P produces a response and S and C have to comply with P 's request. The dual completeness specification (6) allows the same to be done for the exchange of commands between the supervisor and the plant.

The composition of processes in the plant P but also in the control unit, and in particular the composition between C and S however cannot rely on a prioritized synchronization but requires full synchronization. The synchronization between the controller and the supervisor is performed as follows. C , the controller, initiates a command by sending a command request. The actual command can occur only if the synchronization of C with S is successful. If the command request is validated, the actual command is transmitted via an unbuffered message exchange to both P and C . If the synchronization fails, the controller C is notified by a special *synchronization failure message*. Note that this is an unmodeled event from a synthesis perspective.

Clearly, our synthesis model is based on two assumptions. First, it is assumed that the synchronizations are mutually exclusive, i.e., the exchange of messages between processes is supposed to be instantaneous. Second, there is an assumption of unbuffered message exchange between processes. If we assume delays in the communication of plant to supervisor, and of commands from supervisor to plant, the supervisory control model needs to be refined. A more detailed treatment of these issues was done in [31].

VII. SAMPLE MANUFACTURING RECIPES

First, we present a very short illustrative example for a plant constructed from the Rapid Thermal Multiprocessor (RTM) and the supervisor and controller resulting from three simple specifications. Second, we show a realistic sample specification for a typical manufacturing task.

A. Illustrative Example

The RTM consists of a small chamber with a collection of processing equipment such as gas valves, pressure sensors and halogen processing lamps attached to it. The chamber is loaded through a small door. For the small logic control example under consideration, we study the interaction of that door with a gas valve.

The two automata models for the door and valve processes P_{door} and P_{valve} considered for this example are shown in Figs. 5 and 7. Their respective alphabets are

$$\begin{aligned} \Sigma_{\text{door}} &= \{\text{c_open_door}, \text{c_close_door}, \\ &\quad \text{r_door_opened}, \text{r_door_closed}\} \\ \Sigma_{\text{valve}} &= \{\text{c_open_valve}, \text{c_close_valve}, \text{c_repair_} \\ &\quad \text{valve}, \text{r_valve_failed}, \text{r_valve_opened}\}. \end{aligned}$$

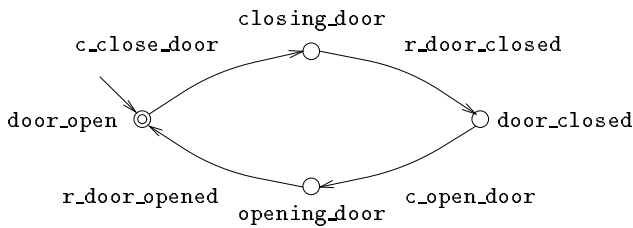


Fig. 7. Automata model for the door.

The complete alphabet of the plant $P = P_{\text{door}} || P_{\text{valve}}$ is $\Sigma = \Sigma_{\text{door}} \cup \Sigma_{\text{valve}}$. The initial states are **door_open** for the door and **valve_closed** for the valve. The only marked states are the initial states. The first basic implicit specification for this plant is thus the following.

- *Implicit Specification 1: Fundamental Liveness.* All components must be returned to their initial states.

Note that this specification is inherent to the plant model. It is part of the plant model and specified *a priori*.

For the plant P composed of the door and gas valve, we first illustrate how to enforce the following safety constraint by an appropriate implicit supervisor.

- *Implicit Specification 2: Gas and Door Mutual Exclusion.* The door and the gas valve must not be open at the same time.

A possible supervisor S' over the alphabet $\Sigma' \subset \Sigma$

$$\Sigma' = \{c_open_valve, r_valve_closed, \\ c_repair_valve, c_open_door, r_door_closed\}$$

which enforces the previous two constraints is shown in Fig. 8. If this supervisor is composed with an arbitrary controller $C = (\Sigma^*, \Sigma^*)$ the global controller $C' = C || S$ does not solve the controller synthesis problem for the safety specifications because the plant is not complete for this supervisor. However, by applying relation (7), a supervisor S such that the plant is complete for $C || S$ is obtained as $S = S' || P_{\text{door}} || P_{\text{valve}}$. We show later how the composition of the processes S' , P_{door} and P_{valve} is performed on-line in the implementation.

Intuitively, the supervisor S' allows the command **c_open_valve** only if the door is closed and the command **c_open_door** has not yet been given. If the command **c_open_door** has been sent to the plant, the command **c_open_valve** cannot be executed until the response **r_door_closed**, signaling that the door has closed, is recorded. A controller or human operator working in conjunction with the supervisor can at most realize the largest behavior that is consistent with the safety constraint stated above.

To conclude the example, we consider a liveness specification for the closed-loop behavior of the plant. Assume that a wafer is to be inserted in the chamber without being contaminated. To achieve this, the chamber is filled with an inert gas. The following recipe expresses this specification.

- *Explicit Specification 1: Short Recipe.* Enter inert gas into the chamber, then open the door.

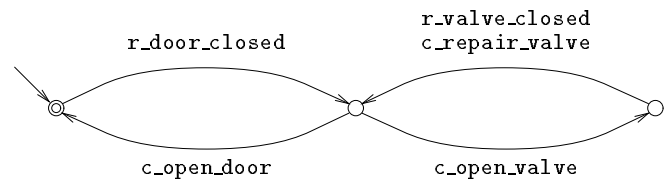


Fig. 8. Supervisor candidate.

The automaton for this specification language over the subalphabet

$$\Sigma'' = \{r_valve_opened, r_door_opened\}$$

is shown in Fig. 9. A controller C for this specification is shown in Fig. 10. The supervisor S and the controller C for this example have nonconflicting marked languages and thus their composition forms a nonblocking global controller. Moreover, we remark that $C || S = C$, and therefore C enforces all discussed specifications. This implies that the behavior of C is fully consistent with S . More supervisors for additional specifications can be designed in a modular fashion as described in Section III-F. The supervisor and controller are formed as the composition of modular supervisors that match those modular specifications.

The controller works as follows. Starting in the initial state, it sends the command **c_open_door**, waits to read the response **r_door_opened**, sends the next command and so on until it reaches the final marked state.

B. Model of Oxide Growth Recipe

We now describe in some detail a typical, larger manufacturing specification for the RTM; in the semiconductor manufacturing community, such a specification is called "recipe." The components are outlined and the implicit and explicit parts of an oxide growth recipe are discussed. In Section V, we give some results from the automatic synthesis of a controller for this example. It will be clear from a state-space analysis that a manual design may prove difficult and prone to errors.

A typical manufacturing process for the RTM consists of growing oxide on a silicon wafer. The wafer is inserted into the processing chamber. Before oxide is grown, the wafer needs to be cleaned from spurious traces of oxide that inevitably appear in the ambient environment. This is performed under a flow of hydrogen. After being cleaned the wafer is exposed to oxygen which causes oxide to grow on the surface of the wafer. Cleaning and coating the wafer both occur at well-determined temperatures. After cleaning and before oxidation, the wafer is often exposed to nitrogen, an inert gas, to stabilize the environment.

The relevant equipment components that participate in the oxide growth are three gas valves (hydrogen, nitrogen and oxygen), the chamber door, two continuous-type control elements (temperature and pressure controller) and a pump-purge mechanism.

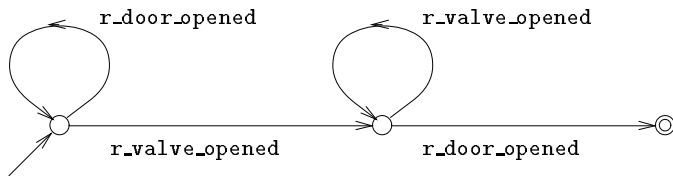


Fig. 9. Explicit liveness specification language.

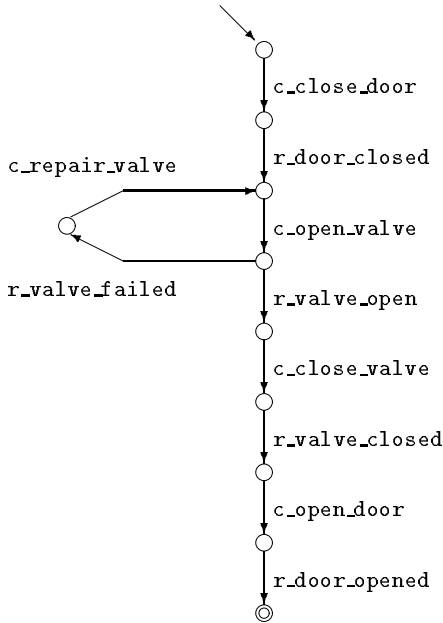


Fig. 10. Controller.

Each of the three gas valves is similar to the valve already described in Sections VI and VII. Also, the door model is identical to the one in Fig. 7.

The model of the temperature control subsystem incorporates, as states, the various discretized temperatures of the wafer, as well as a *disabled* state to indicate the inactivity of the temperature control. For the pressure controller component a similar model holds. However, this latter component is somewhat more complex as it takes into account that gas pressure cannot increase unless one of the gas valves is open. The complete models of these two components and also of the pump-purge mechanism are not presented in this paper as they are analogous in nature to those previously discussed.

For the control of the plant, we also explicitly include in the plant a model of the wafer surface. After the wafer is exposed to hydrogen at the correct temperature for a certain time, the response `wfr_clean` generated by an (unmodeled) timeout notifies the controller that the wafer is clean. Analogously, the response `wfr_coated` tells the controller that the wafer is coated with oxide. The model shares events with both the gas valves and the heating unit. However, the model is independent of the behavior of the pressure controller.

Throughout system operation, the following implicit safety constraints must be enforced.

- *Implicit Specification 1: Gas Mutual Exclusion.* No more than one gas must be in the chamber. A chamber flush must be performed between gases.

- *Implicit Specification 2: Exclusive Operation of Flush Pump.* The gas flow controller element as well as all the gases must be shut off when the chamber is flushed.

- *Implicit Specification 3: Door Closed.* If the door is open, no action other than the closing of the door is permitted.

The following implicit fundamental liveness specification must also be respected. After processing, the participating components must be returned to their initial state.

- *Implicit Specification 4: Fundamental Liveness.* Reset the components (except for the wafer surface) to their initial states.

In addition to these generic constraints, the following recipe-specific explicit liveness constraint governs the oxide-growth process.

- *Explicit Specification 1: Recipe.*

- Bring the chamber pressure to 300 mTorr.
- Clean the wafer.
- Bring the chamber pressure to 1 Torr.
- Grow oxide on the wafer.

The interpretation of the recipe is as follows. While the cleaning of the wafer can physically occur at different pressures, this recipe requires the pressure to be maintained at 300 mTorr throughout the cleaning process. This is done in steps a) and b). We remark that the wafer surface model asserts that cleaning only takes place if there is hydrogen in the chamber and the temperature is at the correct level. Such low-level details need not appear explicitly in this recipe. The only information relevant to the recipe is that the wafer is cleaned at the appropriate pressure. Steps c) and d) are interpreted in the same way. It should be clear from this example that care must be taken in accurately modeling a specification.

A schematic table showing the state size of the automata representing plant and specification is given in Fig. 14.

VIII. SUPERVISORY CONTROL SOFTWARE REALIZATION

In this section, we discuss the actual on-line implementation of the supervisory control software [16]. In essence, it bases itself on the principles presented in the previous sections. In contrast to that general discussion however, it discusses more application-specific as well as software-engineering related, practical issues.

A. Implementation Hierarchy

A complete block diagram of the control software can be seen in Fig. 11. The implementation hierarchy consists of four distinguishable levels. The lowest level incorporates the *equipment drivers*; the level above is the *command/response interpreter*. These two levels form the implementation of the interface that maps the physical plant to the logical plant whose behavior is to be specified and

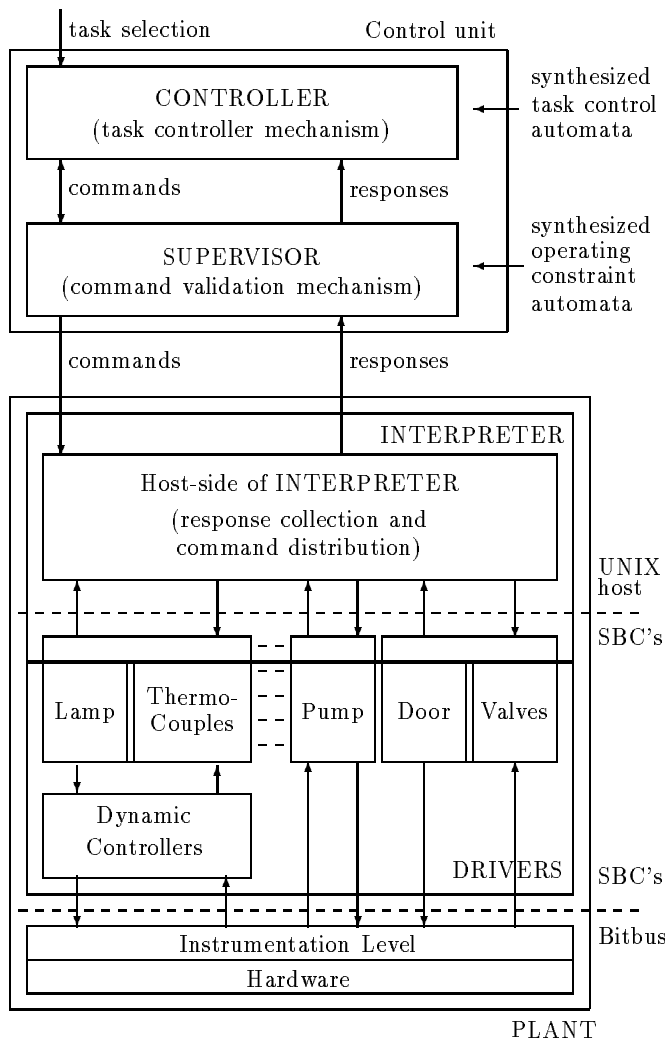


Fig. 11. Implementation scheme of the RTM control software.

controlled in the supervisory control framework. The highest and second-highest levels of the hierarchical implementation are the *controller* and *supervisor*. They form the implementation of the control unit.

B. Hardware and Software Tools

The control hardware equipment consists of a SUN Sparcstation host computer networked to two Motorola 68030 single-board-computer cards (SBC's). The host computer runs **UNIX**, while the SBC's run **VxWorks** [32], a **UNIX**-like real-time multitasking operating system from Wind River Systems, Inc. The SBC's are mounted in a **VME** chassis together with an ethernet card. High bandwidth sensors and actuators are driven by analog and digital driver boards plugged into the **VME** chassis. To reduce the wiring required, low bandwidth sensors and actuators are connected through distributed I/O networks which extend from the **VME** chassis.

All programs are written in **C**. Both the **UNIX** and the **VxWorks** software processes are compiled on the workstation. All software processes communicate via **TCP/IP** sockets over ethernet. A different compilation of the **C** code

of the **VxWorks** software processes enables the simulation and testing of the equipment drivers as **UNIX** processes running on the host.

C. Logical Plant versus Physical Plant

The implementation of the interface on the RTM control equipment is physically divided into two parts. One part runs on the host computer and the other one on the real-time SBC's.

The implementation of the interface is also functionally divided into two parts; the command/response interpreter and the equipment drivers.

Command/Response Interpreter

The main function of the command and response interpreter is to act as an interface between the equipment drivers and the supervisor. On one side, it must supply to the supervisor logical responses that reflect the information provided by the drivers. On the other side, it converts commands from the supervisor into lower-level commands for the equipment drivers.

Information that comes from the drivers falls into the four categories discussed in Section VI-A.1; *event symbols*, *discrete state changes*, *continuous state changes* and *software messages*.

Response symbols or software message generated at the SBC level are directly forwarded to the supervisor by the host side of the command/response interpreter.

For discrete or continuous state changes however, state-sensor tasks on the SBC signal that the hardware is in another state. The host side of the command/response interpreter then generates the corresponding response symbol by using a stored model of the fundamental plant component process as described in Section VI-A.2.

The command interpretation mainly consists of translating commands into the call of actuating routines. After having read a command symbol, the SBC part of the interpreter executes equipment driver routines corresponding to the received command.

The command/response interpreter roughly implements the sensing and actuating behavior described in Section VI-A.1. It is added on top of the fundamental behavior that is closely implemented by the equipment drivers discussed below.

Equipment Drivers

The main function of the equipment drivers is to provide a set of very low-level routines that directly interact with the hardware. These routines are started by the commands from the command/response interpreter, and, during their execution, they provide the command/response interpreter with messages. The drivers also monitor the physical plant and produce messages, e.g., regarding a new state, to be forwarded to the command/response interpreter.

The majority of the equipment drivers are tasks running on the SBC's which interact with the distributed I/O systems. Some drivers are straightforward, simply converting commands into the set of register manipulations and serial-

interface communications needed to turn on actuators, e.g., opening a valve. Others are more indirect, like the drivers responsible for real-time dynamic control of some continuous subsystem, e.g., temperature controllers.

D. Supervisor/Controller Implementation

As with all the previously discussed software modules, the supervisor and controller are implemented as programs written in C. However, the behavior of the two programs is not “hard-coded” in C, but easily adaptable through configuration files. These files contain automata describing the supervisor process.

Supervisor

The program implementing the supervisor is configured to behave as the synthesized supervisor by reading automata representing the modular (and possibly local) supervisors and the automata representing the plant model of the different subsystems.

The local supervisor that enforces the implicit specification for the combined behavior of the valve and the door as discussed in Section VII is shown in file form in Fig. 12.

As already explained in the example in Section VII, the modular (and possibly local) supervisors and the subplant processes need to be composed in order to solve the controller synthesis problem for the whole plant. Composing the modular components into a single automata for the entire plant would lead to a very large plant representation. Instead, the states of each component are updated *on-line* without explicitly having to compose all the modular processes into a single automaton. This procedure allows for modular storage of the plant components.

The following describes how this on-line process composition is implemented. When the supervisor is initialized, a linked list is constructed as a cross-referencing index for every event. The list contains a pointer to all the automata whose alphabet contains the event.

For a command proposed to the command input of the supervisor, the supervisor program checks if the command symbol can be accepted by all the automata whose alphabet contains the proposed command. If this is the case, these automata will undergo the transition corresponding to the command. The command is then immediately forwarded to the command/response interpreter, and because the plant is complete for the supervisor we know that the command is accepted by the plant during the next message exchange.

Note that for simplicity, we assume that the supervisor and the plant operate in a lock-step mode, i.e., that the command and response synchronization procedures are mutually exclusive and instantaneously executed. This synchronization is somewhat artificial; see [31] for a discussion of how to model communication delays and nonzero synchronization times.

The supervisor program also accepts all the responses produced by the command/response interpreter and performs an operation similar to the one just described. For any response given by the logical plant, it goes through

```
# next state relation of supervisor
# format: state event next_state
# starts with initial state, initial state is final
do_not_open_valve r_door_closed safe
safe c_open_valve do_not_open_door
safe c_open_door do_not_open_valve
do_not_open_door r_valve_closed safe
do_not_open_door c_repair_valve safe
```

Fig. 12. Safety constraint supervisor file.

the corresponding linked list of automata models for plant components and modular supervisors and triggers, from the current state of each such automaton, the transition corresponding to the response. In fact, because the supervisor is complete for the generated responses, every response sent by the interface will be accepted by all the automata containing the response symbol in their alphabet.

The supervisor program receives commands either from the controller implementation or from a manual interaction window for the user. The interaction window lists the commands that can be accepted by the supervisor at its current state. The window is constantly updated if the supervisor state changes. The update is performed by checking for every command if it can be accepted in the supervisor by all automata whose alphabet contains the command. If a command can be accepted, a button labeled with the command will appear on the window, and clicking on the button with the mouse will send the command to the plant.

Fig. 13 shows a dump of the workstation screen used in the manual control of the RTM with the interaction window of the supervisor; it shows the history of previous responses and commands, the states of the modular supervisors and plant components as well as the commands currently allowed by the supervisor.

Controller

Just like the supervisor, the controller implementation is configurable with files. The program reads a collection of automata, describing the synthesized controller for a specific task. When a task is completed, a new set of files is read for the next task.

The controller, as already described in Section VI-B, synchronizes with the supervisor for the commands it produces. This is performed by forwarding commands to the supervisor which either accepts or refuses the command. In the first case, the command is forwarded to the plant, in the latter case, the controller switches to a global failure state. The system is left in manual mode.

As the controller is complete for the responses of the plant, it accepts all responses from the plant and updates its internal state accordingly.

Ideally, the commands produced by the controller are always accepted by the supervisor. This is true when the implicit constraints are a subset of the explicit constraints as discussed in Section VI-B. If this requirement is met, the

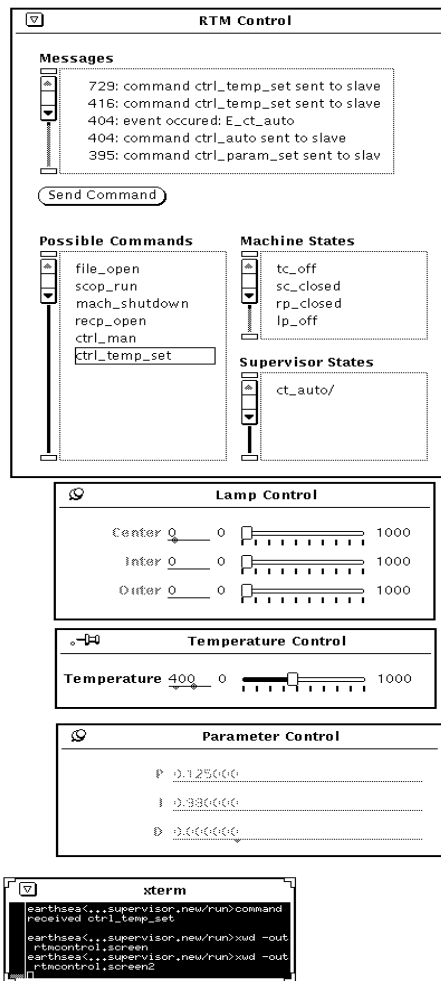


Fig. 13. Screen dump showing the supervisor interaction window.

command synchronization procedure never fails. However, the controller may fail if it does not meet this requirement, e.g., due to an incomplete design. The combined supervisor/controller, however, is always guaranteed to meet at least the implicit constraints.

IX. IMPLEMENTATION OF SYNTHESIS SOFTWARE

Consider the example of subsection VII-B. To illustrate the complexity of this typical synthesis problem, Fig. 14 gives a state-space breakdown of the considered modules.

It is in fact not necessary to consider all possible states, only the reachable ones. However, a reachability analysis shows that there are still about 1.3×10^6 reachable states. The program implemented supervisor for this problem [28].

A. Implementation and User Interface

The synthesis program consists of a toolbox collection of routines built on top of the *Ver* package designed and implemented by David Dill, Andreas Drexler, and Alan Hu, for finite-state verification using BDD's. Their implementation uses Brace, Rudell, and Bryant's package for BDD manipulation [33].

<i>Plant Components</i>	<i>Nr. States</i>
temperature controller	8
pressure controller	24
flush pump	4
H_2 valve	5
N_2 valve	5
O_2 valve	5
door	3
wafer surface	8
Maximum Total	2.3×10^6

<i>Specifications</i>	<i>Nr. States</i>
implicit 1	3
implicit 2	6
implicit 3	2
implicit 4	-
explicit 1	5
Maximum Total	180

<i>Potential State Space</i>	<i>Nr. States</i>
	4.2×10^8

Fig. 14. State-space breakdown for oxide growth example.

The *Ver* kernel provides an interactive environment for entering low-level commands for creating and manipulating BDD's. The synthesis toolbox adds process operators such as parallel composition of modules, and a routine that finds the least restrictive controller via the fixpoint computations described above. It also incorporates useful BDD inspection routines.

All variables in the system are boolean. It is undesirably tedious to have to express process descriptions and their specifications by means of their boolean encodings. Thus there is also a macro preprocessor that deals automatically with managing boolean variables, thereby allowing the user to reason at the level of an automaton's states, events, and transition relation.

Furthermore, a graphical editor for finite-state automata is being developed as a front end to the macro preprocessor. A sample screen view of the editor is shown in Fig. 15. A table describing the synthesis procedure is shown in Fig. 16.

The synthesis program is used to find a BDD for the next-state relation of the least restrictive supervisory controller. Since the program deals internally with BDD's, its most primitive form of output is the raw form of a BDD, i.e., an acyclic directed graph. However, there are more friendly means to obtain information from BDD's. The BDD inspection routines can output a list of outgoing transitions from any given automaton state, and simulate an automaton moving from one state to the next.

B. Results

The results given here relate to the Rapid Thermal Multiprocessor plant described in subsection VII-B [28]. The synthesis of a controller for the plant's implicit specifica-

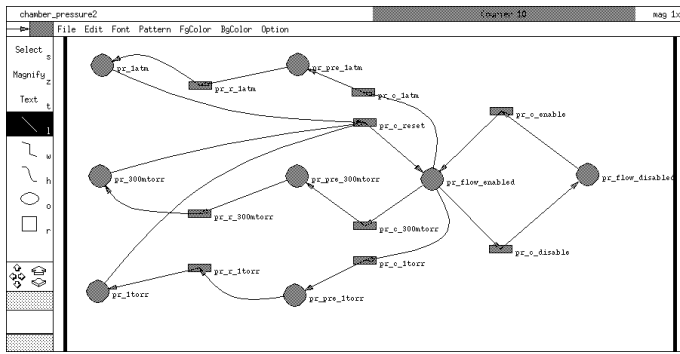


Fig. 15. View of automata editor. States are represented by circles and events by squares.

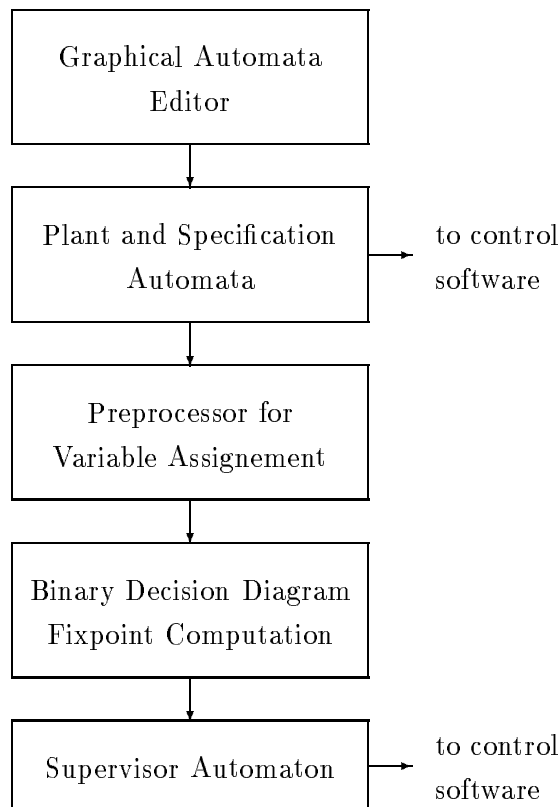


Fig. 16. Block diagram of the synthesis program.

tions takes only 160 seconds using 4.4 MB of memory on a DECstation 3100. However, when the explicit liveness specification is included finding a controller takes 1234 seconds and 7.8 MB. The increased complexity of synthesizing controllers for general liveness specifications is clearly demonstrated here.

In both instances the state-space of the explicit automata representations is greater than 10^6 . It is clear that a synthesis algorithm using directly a representation based on automata would be impractical. However, this example successfully terminates using symbolic boolean encoding. We conclude that the BDD-encoding can make the synthesis procedure of supervisory control theory feasible for realistic applications.

X. CONCLUSION

In this paper, we presented an application of supervisory control theory to the control of a Rapid Thermal Multiprocessor.

We interpreted the supervisory control theory framework from an input-output perspective. The plant is modeled as an input-output process accepting commands as inputs, and producing as outputs messages regarding changes that occurred in the system. A controller for the system has been described in a similar way, accepting the outputs of the plant, and in turn producing commands. Under these semantics both the controller and the plant form the “generating” process in the closed-loop systems. This is in contrast to the original semantics of the Ramadge and Wonham framework where the plant alone is the “generator.”

Based on these observations, a generic control scheme that is applicable to all sorts of manufacturing systems was developed. Its main feature is the separation of supervision and control into two distinct levels. At the supervision level, the implicit recipe specifications (such as safety and fundamental liveness) are enforced. At the control level, the explicit recipe requirements (such as liveness constraints that model job completion) are enforced.

Using this scheme we have implemented a control environment for a Rapid Thermal Multiprocessor (RTM) at the Center for Integrated Systems at Stanford University. The environment provides both manual and automatic control. Synthesis can be done in an interactive environment.

The controller synthesis procedures based on binary decision diagram manipulation were briefly presented. These allow the automatic synthesis of realistic size manufacturing recipes. Also a recipe example with a state space of 10^6 was discussed. Future work will include the incorporation of supervisor reduction techniques [27].

The supervisory control theory described here leaves a number of open questions about the way processes and their specifications are modeled. Of the most pressing extensions to the model, we mention only the following three. Throughout the paper we assumed that the communication between processes is not affected by *communication delays*. The effect of delays that occasionally occur in a real system can be counteracted by imposing additional constraints on the plant or the supervisor [31]. Concepts based on *hierarchical supervision* in the sense of [34] will be important in future applications. The original supervisory control framework is restricted to purely logical system models; for some applications it is critical to extend the model to systems with *real-time* constraints [35]–[38].

ACKNOWLEDGEMENTS

The authors would like to acknowledge Profs. T. Kailath and D. Dill as well as Dr. C. Schaper at Stanford University for fruitful discussions and comments as well as their support.


S. Balemi and G. Hoffmann would like to thank Profs. M. Mansour and L. Guzzella at the Swiss Federal Institute of Technology (ETH) Zürich for their support and guidance.

REFERENCES

- [1] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM J. Contr. Optimiz.*, vol. 25, pp. 206-230, Jan. 1987.
- [2] —, "The control of discrete event systems," *Proc. IEEE*, vol. 77, pp. 81-98, Jan. 1989.
- [3] K. Rudie and W. M. Wonham, "Supervisory control of communicating processes," in L. Logrippo, R. L. Probert, and H. Ural, Eds., in *Proc. 10th Int. Symp. on Protocol Specification, Testing, Verification*, pp. 243-257, Ottawa, Canada, June 1990. North-Holland: Elsevier, expanded version appears as Syst. Control Group Rep. #8907, Dept. Elect. Eng., Univ. of Toronto, 1989.
- [4] B. A. Brandin, W. M. Wonham, and B. Benhabib, "Discrete event system supervisory control applied to the management of manufacturing workcells," in V. C. Venkatesh and J. A. McGeough, Eds., in *Proc. 7th Int. Conf. on Computer-Aided Production Eng.*, pp. 527-536, Cookeville, TN, Aug. 1991. New-York: Elsevier.
- [5] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Programming Languages Syst.*, vol. 8, pp. 244 - 263, Apr. 1986.
- [6] E. M. Clarke and R. P. Kurshan, Eds., *Computer-Aided Verification '90: Proceedings of a DIMACS Workshop*, vol. 3 of *DIMACS Series in Discrete Math. and Theoretical Computer Science*. Providence, RI: American Mathematical Society, 1991.
- [7] M. V. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *Proc. 1986 IEEE Symp. Logic Comput. Sci.*, Cambridge, June 1986, pp. 322-331.
- [8] A. Hsu, F. Eskafi, S. Sachs, and P. Varaiya, "The design of platoon maneuver protocols for IVHS," Tech. Rep., PATH Res. Rep., Institute of Transportation Studies, Univ. of California at Berkeley, Apr. 1991.
- [9] Z. Har'El and R. P. Kurshan, "COSPAR user's guide," Tech. Rep., AT&T Bell Laboratories, Murray Hill, NJ, 1990.
- [10] A. Pollack, "In search of short-order chips," *The New York Times*, p. 9, Jan. 27, 1991.
- [11] K. Saraswat, M. Moslehi, D. Grossman, S. Wood, P. Wright, and L. Booth, "Single wafer rapid thermal processing," in *Mater. Res. Soc. Proc.*, vol. 146, pp. 3-13, Materials Research Society, 1989.
- [12] S. C. Wood, K. C. Saraswat, and J. M. Harrison, "The economic impact of single wafer multiprocessors," in R. Singh and M. M. Moslehi, Eds., *Rapid Thermal and Related Processing Techniques*, vol. 1393 of *Proc. SPIE*, pp. 36-48, 1991.
- [13] C. Schaper, Y. Cho, P. Park, S. Norman, P. Gyugyi, G. Hoffmann, S. Balemi, S. Boyd, G. Franklin, T. Kailath, and K. Saraswat, "Dynamics and control of a rapid thermal multiprocessor," in *SPIE Conference on Rapid Thermal and Integrated Processing*, Sept. 1991.
- [14] S. A. Norman, "Optimization of transient temperature uniformity in RTP systems," *IEEE Trans. Elec. Dev.*, vol. 39, pp. 205-207, Jan. 1992.
- [15] Y. M. Cho, C. D. Schaper, and T. Kailath, "In situ temperature estimation in rapid thermal processing systems using extended Kalman filtering," in *Mater. Res. Soc. Proc.*, vol. 224, Materials Research Society, 1991.
- [16] S. Balemi, "Discrete-event systems control of a rapid thermal multiprocessor," in *Proc. 7th IFAC/IFIP/IFORS/IMACS/ISPE Symp. on Information Control Problems in Manufacturing Technology (INCOM)*, Toronto, Canada, May 1992.
- [17] John. E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Series in Computer Science. NY: Addison-Wesley, 1979.
- [18] P. J. Ramadge and W. M. Wonham, "Modular feedback logic for discrete event systems," *SIAM J. Contr. Optimiz.*, vol. 25, pp. 1202-1218, Sept. 1987.
- [19] K. Inan, "An algebraic approach to supervisory control," *Math. Contr., Signals and Syst.*, vol. 5, pp. 151-164, 1992.
- [20] C. H. Golaszewski and R. P. Kurshan, *Task-Driven Supervisory Control of Discrete Event Systems*, pp. 251-273, vol. 3 of Clarke and Kurshan [6], 1991.
- [21] W. M. Wonham and P. J. Ramadge, "On the supremal controllable sublanguage of a given language," *SIAM J. Contr. Optimiz.*, vol. 25, pp. 637-659, May 1987.
- [22] R. D. Brandt, V. Garg, R. Kumar, F. Lin, S. I. Marcus, and W. M. Wonham, "Formulas for calculating supremal controllable and normal sublanguages," *Sys. Contr. Lett.*, vol. 15, pp. 111-117, Feb. 1990.
- [23] W. M. Wonham and P. J. Ramadge, "Modular supervisory control of discrete event systems," *Math. Control, Signals and Syst.*, vol. 1, pp. 13-30, 1988.
- [24] E. Chen and S. Lafortune, "On nonconflicting languages that arise in supervisory control of discrete event systems," Tech. Rep. #CGR-55, Control Group, College of Engineering, Univ. of Michigan, Jan. 1991.
- [25] C. H. Golaszewski and P. J. Ramadge, "Control of discrete event processes with forced events," in *Proc. 26th Conf. Decision and Control*, Los Angeles, CA, Dec. 1987, pp. 247-251.
- [26] M. Heymann, "Concurrency and discrete event control," *IEEE Control System Magazine*, vol. 10, pp. 103-112, June 1990.
- [27] A. F. Vaz and W. M. Wonham, "On supervisor reduction in discrete event systems," *Int. J. Contr.*, vol. 44, pp. 475-491, 1986.
- [28] G. Hoffmann and H. Wong-Toi, "Symbolic synthesis of supervisory controllers," in *Proc. 1992 American Control Conference*, Chicago, IL, June 1992, pp. 2789-2793.
- [29] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comp.*, vol. C-35, pp. 677-691, Aug. 1986.
- [30] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10²⁰ states and beyond," in *Proc. 1990 IEEE Symp. Logic Comput. Sci.*, Philadelphia, PA, 1990.
- [31] S. Balemi, "Communication delays in connections of input/output discrete event processes," in *Proc. 31st Conf. Decision and Control*, Tucson, AZ, Dec. 1992, pp. 3374-3379.
- [32] Wind River Systems Inc., *VxWorks, A revolution in real-time, Sample Manual*, Emeryville, CA, 1989.
- [33] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant, "Efficient implementation of a BDD package," in *Proc. 27th ACM/IEEE Design Automation Conference*, 1990, pp. 40-45.
- [34] H. Zhong and W. M. Wonham, "On the consistency of hierarchical supervision in discrete-event systems," *IEEE Trans. Automat. Contr.*, vol. 35, pp. 1125-1134, Oct. 1990.
- [35] Y. Brave and M. Heymann, "Formulation and control of real-time discrete event processes," in *Proc. 27th Conf. Decision and Control*, Austin, TX, Dec. 1988.
- [36] C. H. Golaszewski and P. J. Ramadge, "On the control of real-time discrete event systems," in *Proc. 23rd Conf. Inf. Sci. Syst.*, Princeton, NJ, Mar. 1989, pp. 98-102.
- [37] P. Kozák, "Control of elementary discrete event systems: Synthesis of controller with non-zero decision time," in D. Franke and F. Kraus, Eds., *Proc. 1st IFAC Symp. on Design Methods of Control Systems*, Zurich, Switzerland, Sept. 1991, pp. 457-462, Oxford, UK: Pergamon Press.
- [38] H. Wong-Toi and G. Hoffmann, "The control of dense real-time discrete event systems," in *Proc. 30th Conf. Decision and Control*, Brighton, UK, Dec. 1991, pp. 1527-1528.

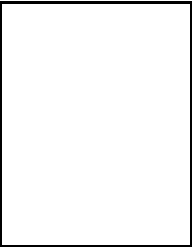
Silvano Balemi was born in Muralto, Switzerland in 1962. He received the electrical engineering degree from the Swiss Federal Institute of Technology (ETH) in Zurich, Switzerland in 1987, the M.S.E.E. from the California Institute of Technology (Caltech), Pasadena, CA in 1988, and the doctorate with honors from the Swiss Federal Institute of Technology (ETH) in Zurich, Switzerland in 1992.

From 1988 to 1991 he was with the Information Systems Laboratory of Stanford University, Stanford, CA and from 1990 to 1993 with the Automatic Control Laboratory of the Swiss Federal Institute of Technology (ETH) in Zurich, Switzerland. His current interests are in discrete event systems and in robust control with particular emphasis on control of manufacturing systems.

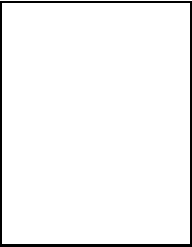


Gérard J. Hoffmann was born in Luxembourg in 1963. He received the electrical engineering degree from the Swiss Federal Institute of Technology (ETH) in Zurich, Switzerland in 1987 and the business degree from the Hochschule St. Gallen (HSG) in St. Gallen, Switzerland in 1988. He completed the Ph.D. degree in electrical engineering at Stanford University, Stanford, CA in 1992.

His research interest is the application of automatic control and signal processing techniques to manufacturing problems. He is particularly interested in real-time discrete event system models and computational issues.




Howard Wong-Toi was born in Auckland, New Zealand. He received a Bachelor of Science degree with First Class Honours in mathematics from Auckland University in 1987. He is currently working towards a Ph.D. in the Computer Science Department at Stanford University, California. His research interests include the specification, verification and automatic synthesis of real-time systems, and the control of discrete event systems.



Gene F. Franklin (S'50-M'52-SM'60-F'78) received his degrees at Georgia Tech, GA, MIT, MA and Columbia University, NY completing the doctorate in 1955.

He has been at Stanford University, Stanford, CA since 1957, where he is currently Professor of Electrical Engineering. His research and teaching interests are in the area of digital control, with current emphasis on model order reduction, adaptive control, including algorithms for implementation on microprocessors, and development of computer aided design tools for control.

Mr. Franklin is coauthor of three books on control including *Digital Control of Dynamic Systems*, Second Edition, Addison Wesley, 1990, with J. D. Powell and M. L. Workman and *Feedback Control of Dynamic Systems*, Addison Wesley, 1991, with J. D. Powell and A. Emami-Naeini. He is a Fellow of the IEEE and was Vice-President for technical affairs of the IEEE Control Society in 1986 and 1987. In 1985 he received the Education Award of the American Automatic Control Council and in 1990 he and his coauthors received the IFAC Award for the best Textbook for *Feedback Control of Dynamic Systems*.



Paul Gyugyi was born in Pittsburgh, PA, in 1966. He received the B.S. degree in electrical engineering from Pennsylvania State University, State College, PA, in 1988, and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, in 1989. He is currently pursuing the Ph.D. degree in electrical engineering at Stanford University.

His research interests include real-time embedded control systems and the application of control theory to improve and automate manufacturing equipment.