Synthesis and Implementation of Procedural Controllers for Event-Driven Operations

A. Sanchez, G. Rotstein, N. Alsop, and S. Macchietto
Centre for Process Systems Engineering, Imperial College, London SW7 2BY, U.K.

A framework is presented for synthesizing logic feedback controllers for event-driven operations, which are used typically for startup and shutdown operations, emergency procedures, and alarm handling. The framework encompasses techniques for discrete-event modeling of open-loop process behavior and operational specifications, as well as the synthesis of feedback control mechanisms called procedural controllers. A procedural controller, if it exists, is mathematically guaranteed to satisfy its operational specifications. This is of particular importance for control systems in which high integrity and correctness are required by design (such as systems in which human life is at risk). The notions introduced and the framework presented are illustrated with a small example. The applicability of the framework to cases of industrial complexity is demonstrated by synthesizing a procedural controller and implementing it as a control code for a section of the operation of an automated multipurpose-multiproduct batch pilot plant.

Introduction

The operation of chemical plants involves a large number of sequential and event-driven activities during plant startup and shutdown, alarm handling, execution of emergency procedures, and equipment interlocking. In automated plants, these activities are normally executed by control devices with logic and numerical processing capabilities, such as programmable logic controllers (PLCs) and distributed control systems (DCSs). The generation of the control logic and its associated implementation are key stages in the development of new processes or the retrofitting of existing ones. Safety considerations mandate that this logic and its implementation must guarantee correct normal operation of the plant and the safe handling of abnormal situations.

Despite the extent to which discrete-event control systems are used in practice, limited theoretical frameworks exist to support their formal analysis and design. The need for such a framework, as well as formal methods and tools for synthesis, verification and validation, has been stressed by academics and industrialists (Schuler et al., 1991). The methods should help to guarantee that the control logic and its implementation perform according to specification and are free from errors that compromise the safe operation of the system. This is particularly relevant in flexible production environments, characterized by frequent changes in the product recipes, production modes, and equipment configuration (and, therefore, in the required controllers).

Previous research on development of such control systems for chemical processes has concentrated on two main topics, including synthesis of operating procedures and verification. The first topic deals with the design of a sequence of steps that take a process from a given initial state to a desired final state, while taking into account equipment and operational constraints. From the pioneering work by Rivas and Rudd (1974), several articles have been published employing artificial intelligence (Fusillo and Powers, 1987; Lakshmanan and Stephanopoulos, 1988) or a combination of AI and mathematical programming methods (Crooks and Macchietto, 1992; Rotstein et al., 1994). From the perspective of control systems, the synthesis of operating procedures can be understood as the search for open-loop optimal control trajectories. The use of feedback to implement emergency pro-

Correspondence concerning this article should be addressed to S. Macchietto.
Current address of A. Sanchez: Dept. of Electrical Engineering and Computer Science, CINVESTAV, Apdo. Postal 31-438, Guadalajara 44550, Jalisco, Mexico.
Current address of G. Rotstein: i2 Technologies, The Priory, Stomp Road, Burnham, Buckinghamshire, SL1 7LL, U.K.
Current address of N. Alsop: The Broken Hill Proprietary Limited, Newcastle Research Laboratories, P.O. Box 188, Wallsend NSW 2287, Australia.

cedures and to deal with abnormal behavior has not been studied. Yamalidou and Kantor (1991) identified the need for feedback control to guarantee the execution of a given event trajectory. However, they restricted their work to the specific case of pipe/valve networks modeled as high level Petri Nets. Control policies were obtained as the result of an MILP problem minimizing the number (cost) of control actions required to reach a final state. The firing in the net was carried out in a feedback fashion following the predefined policy.

The second related research topic, verification, is concerned with the problem of determining whether a proposed control law satisfies a set of specifications when used to control a given process. Several articles have been recently presented to verify discrete systems (Moon et al., 1992; Park and Barton, 1997) or a combination of continuous and discrete systems (Dimitriadis et al., 1997). The problem of how to design a control law guaranteed to satisfy certain specifications is not considered in these articles. Verification, in fact, is complementary to design. One can first design the controller based on simplified process models and then verify the results using a suitable approach.

In this article, a formal framework is proposed for the synthesis of procedural control logic based on a control engineering approach. That is, the problem of finding the sequence of control commands satisfying given operation goals and constraints is posed as the synthesis of a feedback control law for a discrete-event model of the process. The control law is designed to satisfy well-defined structural properties while considering both normal and abnormal operation of the process. The approach builds on techniques developed for specification in software engineering (Ostroff, 1989a) and control of event-driven processes (Wonham, 1988). Of particular relevance to this work is the supervisory control theory (SCT) (Ramadge and Wonham, 1987a,b), which gives general foundations for the study of control issues for this class of systems. The controlling mechanism proposed in SCT acts upon the process by disabling external actions (such as predefined interlocks) in such a way that it guarantees the process behavior is restricted to a well-defined set of trajectories satisfying operational specifications. The synthesis techniques presented in this article build upon SCT, specializing the notion of the control action to deal with forcible actions in an explicit manner. In our experience, a control mechanism which operates by sending commands to process actuators rather than only by disabling controllable actions is closer to those mechanisms used in the process industries.

A first attempt to introduce forcible actions within a process control setting was done by relaxing the controllability property of SCT, allowing the existence of controllable actions simultaneously with process-generated events (Sanchez, 1996). However, a decision mechanism was not explicitly defined leaving the problem of which control command to execute to be resolved at the implementation stage. The approach presented here proposes a formal control mechanism which clearly establishes the sequence of execution of control commands in a deterministic fashion. In other words, the procedural controller does not require an external decision mechanism to drive the process in a safe fashion.

A motivating example discusses the conceptual requirements for the synthesis of a procedural control system for

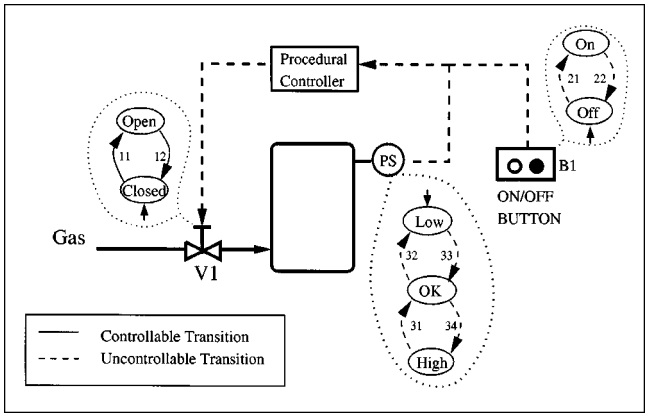

Figure 1. Simple pressurized gas feed system.

event-driven operations. Then, the methodology and tools employed to satisfy each of these requirements are presented. The notions of process and procedural controller models are introduced. Tools for model building are also given, while the capture of behavior specifications is discussed. The control paradigm and the mathematical background to perform the synthesis of the control law are presented. Once in possession of the mathematical machinery to represent the process model, its desired behavior and tools for synthesizing a feedback control law, a synthesis method is presented. The application of the method is then illustrated step by step using the motivating example presented earlier. The applicability of the proposed paradigm for the synthesis of a procedural controller for an industrial pilot plant (including its implementation and experimental testing) is demonstrated, followed by a summary of conclusions and directions for future work.

Motivating Example

The overall purpose of this work is best demonstrated using an example. The process equipment is shown in Figure 1. A gas line is connected to a pressurized storage tank. The tank is charged using on/off valve $v1$ until reaching a predefined pressure. In order to open valve $v1$, an operator must press the button provided. For safety reasons, if the operator releases the button at any time during the operation, the valve must be shut immediately even when the target pressure has not been reached. It is also desirable to introduce a safety system to supervise the normal operation and to override the operator actions if unsafe conditions arise (that is, if the pressure goes beyond the target value) by closing valve $v1$ regardless of the button position.

The objective of the example is to synthesize the control logic capturing the above requirements. This logic serves as the high level specification for a feedback sequential control system which can be either hardwired to the process or implemented as software. Following a standard control engineering paradigm, synthesizing the sequence logic involves the following aspects:

(1) *Defining a Control Strategy.* Issues such as adopting a feedback or feedforward arrangement, or the capability of issuing control commands are to be addressed.

(2) *Process Modeling.* A framework and tools must be provided for constructing mathematical models for the process, the controller, and the closed-loop behavior generated by their interaction.

(3) *Specification Modeling.* The modeling of control objectives must also be formalized. These are expressed in terms of desired properties that the closed-loop system must fulfill, as well as undesired situations that are to be avoided. In the gas system example, a possible specification is that, whenever the operator releases the on/off button, the gas charge must be stopped (that is, valve $\upsilon 1$ must close).

(4) *Analysis.* Establishing the realizability of the specifications is a key issue before embarking into the synthesis of a control law. For example, in the gas system, it is desirable to know whether it is possible to avoid all the situations where the pressure is high and the valve is still open.

(5) *Control Law Synthesis.* This activity generates a control law that guarantees the system will follow the desired behavior in spite of disturbances. This implies the ability to prove theoretical properties on how the synthesized control law will perform when coupled to the process. For example, it is desirable to guarantee that the control law can cope with all the foreseeable abnormal behavior, such as an unexpected release of the on/off button by the operator, and that it can force the system to stay within the predefined acceptable behavior.

(6) *Verification.* The closed-loop system is analyzed via simulation or verification techniques using more detailed process models (Moon et al., 1992; Park and Barton, 1997; Sanchez et al., 1998).

(7) *Implementation.* The control law is refined and implemented as control software.

The first five issues mentioned above are addressed in the following three sections of the article. The objective is to build a formal framework for synthesizing controllers for the class of dynamic systems under consideration. The last issue, implementation, is discussed as part of the case study presented later in this article.

Modeling

This article is concerned with processes characterized by the occurrence of discrete events (discrete-event systems) in which time issues are treated in a qualitative manner. Thus, the process dynamics are characterized as feasible sequences of events, rather than continuous evolution of variables with the progress of time. Finite state machines (FSMs) and their associated languages (Hopcroft and Ullman, 1977) have been found to provide a suitable modeling framework for these processes. An FSM is a state-transition structure where the relationship between states and transitions is given by a partial function. Here, a class of FSMs is used in which each state is labeled. Its mathematical definition is

$$P = \{Q, V^{n_\upsilon}, \Sigma, \delta, \zeta, q_0, Q_m\}$$

where Q is the set of states: $q \in Q$, V^{n_υ} is the set of state-variables: $\{(\upsilon_j)_q;\ j = 1 \ldots n_\upsilon$ (number of state-variables defining state $q)\}$, Σ is the set of transitions: $\sigma \in \Sigma$, δ is the state-transition function, defined as a partial function $\delta: \Sigma \times Q \rightarrow Q$, ζ is the state-variable transition function, defined as a partial function $\zeta: \Sigma \times V^{n_\upsilon} \rightarrow V^{n_\upsilon}$, q_0 is the initial state, and Q_m is the set of marked states.

A *state-variable* $(\upsilon_j)_q$, $j = 1 \ldots n_\upsilon$ describes the state of an elementary component of the process (such as the position of an on/off valve and the status of a pump) and is characterized by a domain of possible values (such as {open, closed}). A system is assumed to be fully described by a set of n_υ state-variables at any given moment. A *state* $q \in Q$ is defined by an assigned set of n_υ state-variables. For instance, consider a system composed of one elementary component, an on/off valve. The state-variable describing this component is ''valve position'' and its domain is {open, closed}. Therefore, a model of this system has two states in total. It follows that a model of two on/off valves considering all possible combinations is composed of four states with two state-variables in each state.

In addition to the domain values, two more symbols may be assigned to a state-variable: ∞_j, which symbolizes all the possible values that state-variable j can take and $\infty_j^{E_j}$, which represents all the possible values for state-variable j, except those in the set E_j. These symbols are introduced to handle incomplete information in the modeling of either the process or the specifications during the specification stage.

Transitions are instantaneous events leading from one source state to a destination state and change only one state-variable value. Σ is the set of process transitions, and Σ^* is the set of all process trajectories composed of transitions in Σ. The partial function $\delta: \Sigma \times Q \rightarrow Q$ defines the relation between states and transitions, while the partial function $\zeta: \Sigma \times V^{n_\upsilon} \rightarrow V^{n_\upsilon}$ does the same for state-variables and transitions. The partial function δ is extended inductively for strings. Note that if ζ and V^{n_υ} are excluded from P, a standard FSM is obtained. The set of transitions is partitioned into two disjoint subsets $\Sigma = \Sigma_c \cup \Sigma_u$ (Wonham, 1988), in which Σ_c is the subset of controllable transitions and Σ_u is the subset of uncontrollable transitions. Controllable transitions are control inputs to the process (such as sending a command to open a valve), while uncontrollable transitions are associated with the process responses (such as a signal indicating the position of the valve) or input disturbances (such as an operator pressing an emergency stop button).

The initial state q_0 is a unique state, while the set of marked states Q_m distinguishes states of special significance for the system, for instance, states where operation can be held or in which a task is completed may be considered as marked.

Elementary models are generated individually for each single part of the process (such as valves, measuring devices, and switches) as FSMs. Only one state-variable is associated with each elementary model (such as ''valve_position'' or ''level_status''). A transition is always associated with a change in the state-variable value (such as if the current value of state-variable ''valve_position'' is ''open'' and transition ''valve is closing'' occurs, in the next state of the model, the value of state-variable ''valve_position'' will become ''closed''). Thus, in a more complex model generated from several elementary models, the partial function $\zeta: \Sigma \times V^{n_\upsilon} \rightarrow V^{n_\upsilon}$ defines which state-variable is changing under the execution of a given transition. Also, an ordered set of state-variables $(\upsilon_j)_q$, $j = 1 \ldots n_\upsilon$ is associated with each state q. Therefore, a homomorphism $\beta: Q \rightarrow V^{n_\upsilon}$ is defined such that

(Sanchez, 1996)

$$\beta[\delta(\sigma, q)] = \zeta[\sigma, \beta(q)]$$

Homomorphism β is particularly useful when building specifications, as will be shown in the next section.

During the closed-loop operation, it is important to guarantee that certain states are visited. Reachability properties are formalized using the following definitions (Wonham, 1988):

The *reachable state subset* Q_r is the set of states which can be reached from the initial state

$$Q_r:\{q \in Q | \exists s \in \Sigma^*,\ \delta(\sigma, q_0) = q\} \tag{1}$$

The *coreachable state subset* Q_{cr}, is the set of states from which a marked state can be reached.

$$Q_{cr}:\{q \in Q | \exists s \in \Sigma^*,\ \delta(\sigma, q) \in Q_m\} \tag{2}$$

An FSM M is *reachable* if all states $q \in Q$ can be reached from q_0. M is *coreachable* if all marked states $q \in Q_m$ can be reached. A machine which is reachable and coreachable is said to be *trim*.

A set of finite strings of transitions generated by an FSM is called a *regular language* L (Hopcroft and Ullman, 1977), that is

$$L:\{s \in \Sigma^* | \delta(s, q_0)!\}$$

The symbol ! is used here to denote ''is defined.'' In other words, if the model of a system is given by an FSM P, then its feasible behavior (that is, the trajectories from the initial state) is described by the language generated by P. Languages are thus a compact representation for process behavior, and, together with their operators, provide a useful framework complementary to FSMs. In the following paragraphs some other standard properties and operators from Automata theory (Ramadge and Wonham, 1987b) used in this work are presented.

The *prefix closure* $\overline{L}$ of a language $L \subseteq \Sigma^*$ is the language which possesses all the prefixes of the strings of L

$$\overline{L}:\{s \in \Sigma^* / \exists t \in \Sigma^*,\ st \in L\}$$

The prefix closure of a language L realized by an FSM P is the set of partial strings generated from the initial state of P, which can be extended to a complete string belonging to L. In this work P is described as an FSM; thus, is always the case that $L(P)$ is closed, that is, $L(P) = \overline{L}(P)$.

The language intersection of two languages $L(M_1)$ and $L(M_2)$, $L(R) = L(M_1) \cap L(M_2)$ is given by (Hopcroft and Ullman, 1977)

$$L(R):\{s \in \Sigma^* | s \in L(M_1) \wedge s \in L(M_2)\}$$

The *marked language* of an FSM is defined by the set of strings from the initial state which terminate at a member of the set of marked states Q_m

$$L_m(M):\{s \in L(M) | \delta(s, q_0) \in Q_m\}$$

L_m represents all the trajectories finishing at states considered relevant to the process, such as the completion of normal or emergency operation steps or conditions in which the process can be safely halted. If the language $L(M)$ reaches all the marked states of a given FSM M, then it is said that M is *nonblocking*. That is

$$L(M) = \overline{L}_m(M)$$

Thus, if an FSM M is reachable and coreachable (that is, trim), M is nonblocking (Wonham, 1988).

Model building

The construction of a process model takes place in three main stages:

(1) The elementary components of the process are identified and suitable models are proposed. Here an ''input-output'' view of the process is adopted in which all the components of the process are identified and then modeled ''as seen'' by the control system. These components are typically instruments, actuators, and software devices built into the process.

(2) The overall process model is generated from the interaction among the elementary components identified in stage 1. The events modeled in each component can, in principle, take place in parallel. This is represented in this work by interleaved modeling.

(3) Often, extra causal behavior must be considered (such as certain states or sequences must be excluded from the model, because they are physically infeasible). For instance, in the motivating example, if the pressure line valve is closed and failures are not considered, then the pressure should not increase.

The implementation of the second and third stages requires the introduction of two FSM operators, the *asynchronous* and the *synchronous* products (Heymann, 1990) adapted to incorporate state-variables.

Asynchronous Product. The FSM interleaved model is constructed by applying the asynchronous product operator to all the elementary process models. The following definition takes into consideration the existence of state-variables in each state by constructing the state-variable vector explicitly:

Definition 1: Asynchronous Product of FSMs ($\|$). Given two FSMs $M_1:\{\sigma \in \Sigma_1;\ q_1,\ q_1' \in Q_1\}$ and $M_2:\{\mu \in \Sigma_2;\ q_2,\ q_2' \in Q_2\}$, in which $\Sigma_1 \cap \Sigma_2 = \emptyset$, the asynchronous product $M_R = M_1 \| M_2$ is given by the interleaving of states of each machine

$$M_R:\delta(\sigma, q_R) = q_R'$$

$$\delta(\mu, q_R) = q_R''$$

and

$$\Sigma_R = \Sigma_1 \cup \Sigma_2$$

where the state variables $v_{q_R}, v_{q'_R}, v_{q''_R} \in V_R : V_1 \times V_2$ of the new machine R are given by the cartesian product of state-variables for each corresponding state

$$v_{q_R} = (v_{q_1}, v_{q_2})$$

$$v_{q'_R} = (v_{q'_1}, v_{q_2})$$

$$v_{q''_R} = (v_{q_1}, v_{q'_2})$$

$$v_{q_1}, v_{q'_1} \in V_1 \quad \text{and} \quad v_{q_2}, v_{q'_2} \in V_2$$

Synchronous Product. During the controller synthesis and the model construction, it is necessary to calculate the intersection of trajectories among different FSMs (that is, the language intersection of the machines involved). For regular languages, this is equivalent to a product in which, from a given state of each machine, only identical transitions are considered. This is named the *synchronous product.* For specification modeling, the ambiguities introduced by an FSM when handling explicitly state-variables due to the symbols ∞_j and $\infty_j^{E_j}$ are resolved using suitable information from any of the FSMs being intersected by substituting refined state-variable values in the resultant state from the intersection. The above is captured in the following definition.

Definition 2: Synchronous Product (§). Given two FSMs M_1 and M_2, the synchronous product $M_R = M_1 \S M_2$ is given by the language intersection of both FSMs

$$M_R : \delta(\sigma, q_R) = q'_R$$

and

$$\Sigma_R = \Sigma_1 \cap \Sigma_2, \ \sigma \in \Sigma_R$$

where $L(R) = L(M_1) \cap L(M_2)$ and $\zeta(\sigma, (v_j)_{q_R})$ takes the refined state-variable value from $\zeta(\sigma, (v_j)_{q_1})$ and $\zeta(\sigma, (v_j)_{q_2})$.

In other words the synchronous product is given by the FSM that realizes the language intersection in which

(1) Every trajectory is in $L(M_1)$ and $L(M_2)$.

(2) The refined state-variable value from states in either FSMs M_1 or M_2 is assigned to the resultant state-variable of the product. If different refined values are given by each machine, then the product is not defined.

When state-variables are not required (such as during the controller synthesis), the language intersection operator (Hopcroft and Ullman, 1977) is used. This is equivalent to the standard synchronous product (Heymann, 1990). It is important to stress that a FSM resulting from the synchronous product of two trim FSMs is not necessarily trim (Wonham, 1988).

Control law representation: procedural controller

A *procedural controller* is a feedback device driving the execution of controllable transitions in a given process. In this work, procedural controllers are modeled using the FSM formalism. A restriction is imposed on the FSM structure of a procedural controller, in which either controllable (at most one, that is, the one to be enforced) or uncontrollable transitions may occur at each state.

Definition 3: Procedural Controller. A procedural controller is an FSM:

$$C = \{X, \Sigma, \gamma, x_0, X_m\},$$

in which for each $x \in X$, $\sigma \in \Sigma$ such that $\gamma(\sigma, x)!$, one of the following is true:

(1) $\sigma \in \Sigma_u \wedge (\forall \sigma_c \in \Sigma_c, \gamma(\sigma_c, x)$ is undefined$)$.

(2) $\sigma \in \Sigma_c \wedge (\forall \sigma' | (\sigma' \neq \sigma) \in \Sigma, \gamma(\sigma', x)$ is undefined$)$.

If an FSM represents a model of a process P, then the controller C executes synchronously with the process and in this way "controls" it. The closed-loop system C/P is given by the *synchronization* of the process and the procedural controller (Kumar et al., 1991). The closed-loop behavior is given by the language intersection of the process P and the controller C, $L(C/P) = L(P) \cap L(C)$ (Ramadge and Wonham, 1987b).

A procedural controller can either be in: (1) a state in which one of a set of uncontrollable transitions occurs; or (2) a state in which it forces the execution of the only controllable transition defined. In this way, a controller acting synchronously with a process preempts the occurrence of any of the uncontrollable transitions that can occur from the current process state. The possible exclusion of behavior generated by uncontrollable transitions from the closed-loop behavior relies on the assumption that controllable transitions can preempt uncontrollable transitions. Even though this assumption may appear to be strong, it is largely a modeling issue. That is, if a controllable transition has slow dynamics, it can always be partitioned into two transitions: a fast controllable one, representing the execution decision, and an uncontrollable transition, representing the slow system response to that decision. The assumption allows the possibility of disabling some uncontrollable events by preempting them with the execution of controllable transitions.

The procedural controller FSM is defined in such a way that no decision branching points exist (that is, there is no need for an external mechanism to decide which controllable transition to execute at a given state). This facilitates the refinement and translation of the procedural controller into control code or hardwired logic.

Specification of Closed-Loop Behavior

The correct and complete specification of the desired closed-loop behavior is a key step in the synthesis of any event-driven control system. Desired and undesired process behavior must be prescribed for both normal and abnormal operations. However, this specification task has proved to be cumbersome even for simple cases (Brooks et al., 1990). It is difficult to provide specification methods enabling a compact and unambiguous representation of the operational objectives and constraints. In practice it is common to find underspecified systems which behave unexpectedly in situations not considered by the system designer. This is of particular concern in safety critical systems, which have received great attention by the software engineering community (Abrial et al., 1996; Fitzgerald et al., 1997). Sanchez (1996) adapted for their

use in process applications logic-based specification techniques from software engineering that have proven to be effective in handling complexity and size of realistic engineering specifications while maintaining compatibility with the synthesis framework for the controller mechanisms proposed here. In the following paragraphs a summary of these techniques is presented. Predicate logic is used to prescribe states that the system must avoid during operation. These states are termed forbidden states, while temporal logic does its part for dynamic behavior including safety and liveness properties.

Specification of forbidden states using predicate logic

The use of predicate statements as static specifications has been studied in the past by Ramadge and Wonham (1987a) and Kumar et al. (1991). Predicates are associated with states in Q, and it is shown that there exists an isomorphism h between the family of all predicates $p \in \mathcal{P}$ associated with Q and the Boolean lattice of the power set of Q under the correspondence

$$h:Q \leftrightarrow \mathcal{P}\{q:q \in Q \quad \text{and} \quad p(q)=1\}$$

Therefore, the terms ''state'' or ''predicate statement'' can be considered essentially the same in this context. Given the existence of the isomorphism $h:Q \leftrightarrow \mathcal{P}$ and homomorphism $\beta:Q \rightarrow V^{n_v}$ defined earlier, it is possible to represent in terms of predicates the state-variables corresponding to its isomorphic state such that

$$H(p)=\beta\left(h^{-1}(p)\right)=(v_j)_{h^{-1}(p)}, \quad p \in \mathcal{P}, \quad j=1 \ldots n_v$$

This suggests that a state in the predicate logic domain is given by the conjunction of atomic formulas representing the different state-variables of the given state

$$h(q)=a_1 \wedge a_2 \wedge \ldots \wedge a_{n_v}$$

where $a_1, a_2, \ldots, a_{n_v} \in \Phi$ are atomic propositions representing the state-variables and Φ is the set of atomic propositions.

Specification of dynamic behavior using temporal logic

Quantitative time is not treated by the modeling structures presented in the previous section. This relies on the assumption that the system dynamics are governed by events. The passage of time is then associated with the function δ that imposes an order of execution on the FSM transitions. Therefore, dynamic behavioral specifications deal only with this type of qualitative-time characteristic, such as the sequencing of events or the eventual execution of a determined action. Temporal logic (TL) is a modal logic in which operators are interpreted in a qualitative time domain. In the present work, a restricted syntax has been chosen from the framework devised by Ostroff (1989b) and Thistle and Wonham (1986) based on Manna-Pnueli's linear temporal logic (LTL) system (Manna and Pnueli, 1982).

LTL Syntax. The formulas used in this work are constructed using the following scheme, expressed in BNF notation

$$[\,p|\,p \wedge \tau\,] \rightarrow \left[\bigcirc p|\left(\bar{\bigvee}\ \tau\right)\right] \tag{3}$$

The first element of a formula must be either a predicate p representing a state or a predicate p and a ''next transition'' symbol τ. This is followed by the implication symbol ($\rightarrow$). The righthand side of the formula is either a LTL operator ''next'' ($\bigcirc$) operating over a predicate p describing a state, or an exclusive disjunction ($\bar{\bigvee}$) of ''next transition'' symbols describing the following actions to be executed.

LTL Semantics. Standard semantics are used for Scheme 3 (Goldblatt, 1992). A frame is a pair $F=(S, R)$, where S is a nonempty set and $R:S \times S$ is a binary relation. A model is a triple $M=(S, R, V)$, with $V:\Phi \rightarrow 2^S$. The function V assigns to each atomic formula $a \in \Phi$, a subset $V(a)$ of S. In other words, $V(a)$ defines the elements of S in which a is true.

A formula A is true in model M, denoted $M \vDash A$, if it is true at all elements in M, that is, if $M \vDash_s A, \forall s \in S$. A formula A is valid in frame $F=(S, R)$, denoted $F \vDash A$, if $F \vDash_M A, \forall M=(S, R, V)$ based on F. In other words, a valid formula is a formula which is true in all elements of every model. A scheme is said to be valid in a frame (respectively, true in a model) if all instances of the scheme are valid (respectively, true). Then, it is said that the frame validates the scheme.

Translation of LTL Formulas into FSMs. The translation of LTL formulas into the structures used for the controller synthesis is based on a homomorphism for the scheme mentioned above. A detailed discussion of the translation process is presented in Sanchez (1996). This homomorphism defines sufficient conditions to guarantee that the resultant FSM is a LTL model in which the LTL formula holds and is a subset of a LTL frame in which the LTL formula is valid. A LTL frame for a given process is provided by the asynchronous product of all elementary process components (that is, the interleaved behavior that the elementary components can generate). Having developed this frame, a LTL model (that is, an FSM) is constructed for each dynamic specification (that is, LTL formula) modeled by the scheme described above and employing the same elementary process components utilized in the construction of the LTL frame. Consequently, if the behavior prescribed in the LTL formula cannot be represented by the combination of the behavior of elementary process components in an interleaved fashion, then the formula is not true and, therefore, not valid. In other words, the required dynamic specification cannot be achieved.

The first step in the translation of a given LTL formula is to identify or create the appropriate state in the FSM domain from where to start the translation. Such a state must contain the same state-variable values as the predicate p modeling the first state of the LTL formula under consideration. This is accomplished by constructing a lattice (Davey and Priestley, 1990) in which the greatest lower bound (*glb*) is a state containing the initial state-variable values given by the LTL formula and the least upper bound (*lub*) is its complement. Once the initial state of the LTL formula under translation has been created, the rest of the LTL formula is constructed using the appropriate morphism. At each step of the translation procedure, the validity of the LTL formula is

established by checking that each transition leads to the appropriate state-variable value using the elementary FSM process models. The result is an FSM modeling all the allowable process sequences of state-transitions which satisfy the dynamic specification. Examples are presented in the section Solution to the Motivating Example.

Controllability Analysis and Controller Synthesis

Using the modeling formalisms proposed in the previous section, the specifications prescribing the desired behavior are captured. However, assuming that the specifications describe feasible behavior, it is not always possible to guarantee that a procedural controller satisfying the proposed specifications exists because either there may not exist sufficient degrees of freedom (that is, control actions) in the system or more information may be needed for resolving indeterminacies regarding the process behavior. Thus, there is the need to introduce the notion of a controlling mechanism and, together with it, a framework to characterize the system's closed-loop behavior.

Completeness and controllability analysis

Within the proposed framework, two concepts are of particular relevance:

- completeness
- controllability.

The notion of completeness declares a controller capable of tracing all trajectories that can arise during the closed-loop operation while controllability defines the set of trajectories in which the process can safely exist.

Definition 4: Completeness. An FSM $M=\{X, \Sigma, \gamma, x_0, X_m\}$ is complete with respect to a given process $P=\{Q, \Sigma, \delta, q_0, Q_m\}$ if for $s\in\Sigma^*$ and $\sigma_u\in\Sigma_u$, the conditions $\gamma(s, x_0)!$ and $\delta(s\sigma_u, q_0)!$ imply that one of the following assertions is true:

(1) $\gamma(s\sigma_u, x_0)!$; and

(2) $\exists\sigma_c\in\Sigma_c$ s.t. $\gamma(s\sigma_c, x_0)!\wedge\gamma(s\Sigma_u, x_0)$ is undefined.

That is, any extension of s with one of the uncontrollable transitions realizable by the process must be also generated by the controller (Ramadge and Wonham, 1987b) or there exists a controllable transition preempting the occurrence of uncontrollable transitions.

Definition 5: Controllability. A language $K\subseteq L$ is said to be controllable with respect to a language L if $\forall s\in K$ either:

(1) $s\Sigma_u\in L\wedge s\Sigma_u\in\overline{K}$; or

(2) $\exists\sigma_c\in\Sigma_c$ s.t. $(s\sigma_c\in\overline{K})\wedge(s\Sigma_u\cap\overline{K}=\emptyset)$.

That is, for every state in an FSM generating a controllable language K either:

(1) All the uncontrollable transitions end in one of the states of the FSM [and this is the controllability concept as proposed in SCT (Ramadge and Wonham, 1987b)]; or

(2) There is at least one controllable transition leaving the state and ending in another state of the FSM. This means that even if there are uncontrollable transitions leading outside the FSM, it is possible to synthesize a controller that can preempt them by proper control actions and keep the process in the desired state space. Clearly, the aim is to synthesize procedural controllers that are complete, and, when run in synchronization with the process to be controlled, generate controllable languages. The following proposition relates both properties. The proof is included in Appendix A.

Proposition 1: Given a process P and a procedural controller C, then $L(C/P)$ is controllable with respect to $L(P)$ if C is complete.

At each state, there may exist several controllable actions to execute and, therefore, several complete procedural controllers with corresponding controllable languages. The union of two procedural controllers is not, in general, a procedural controller and no ''minimal restrictive'' solution, in the SCT sense, exists (Golaszewski and Ramadge, 1987). However, as in SCT, it is possible to show that the union of two controllable languages is controllable:

Theorem 1: Given an arbitrary set of languages $K^i\subseteq L$ and $N=\cup_i K^i$, if $\forall i$, K^i is controllable with respect to L then N is controllable with respect to L.

The proof of this theorem is given in Appendix B. From Theorem 1, it follows that given a language $K\subseteq L$, there is a unique and well-defined supremal controllable sub-language of K with respect to L.

Definition 6: Supremal Controllable Sub-Language. The supremal controllable sub-language $K^{\uparrow}$ of a given language $K\subseteq L$ is given by

$$K^{\uparrow}=\cup\{K':K'\subseteq K \text{ and } K' \text{ is controllable with respect to } L\}$$

Superstructure of procedural controllers

The language representing the closed-loop behavior of a system controlled by a complete procedural controller is indeed controllable with respect to the system language (see Proposition 1). However, not every controllable sub-language can be generated by a procedural controller. For instance, consider the FSMs shown in Figure 2 generating languages K and L. Continuous line arrows represent controllable transitions, while uncontrollable transitions are shown as dashed

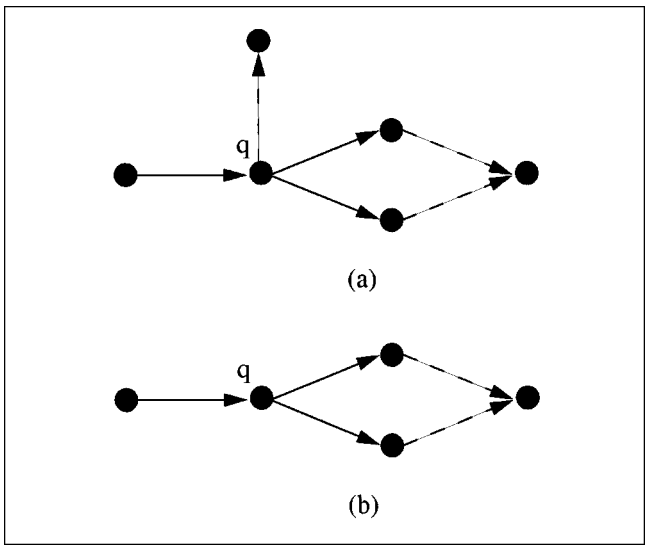

Figure 2. Language generators for: (a) the system (L) and (b) the specification (K).

– – – uncontrollable transitions; —— controllable transitions.

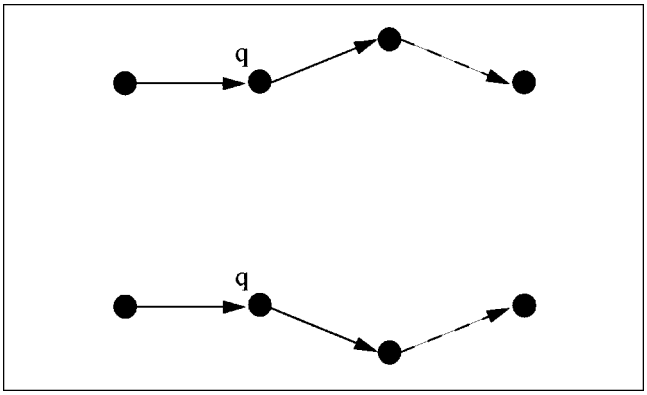

Figure 3. Two alternative procedural control structures.

lined arrows. The language K is controllable with respect to L, but no procedural controller capable of generating K exists. In order to obtain a complete procedural controller, one must force only one of the two transitions enabled at state q. Thus, for the example in Figure 2, there are two procedural controllers generating controllable behavior within K as shown in Figure 3.

Although there is no minimally restrictive procedural controller, it is useful to have a compact description of the closed-loop behavior which the process can safely undertake and from which different operation policies (that is, procedural controllers) can be derived. This leads to the following definition:

Definition 7: Superstructure of Procedural Controllers. An FSM R is a superstructure of procedural controllers if $L(R) \cap L(P)$ is controllable with respect to $L(P)$.

This superstructure encompasses a set of candidate structures for the procedural control of a process. Moreover, using Proposition 1, it is possible to characterize a superstructure that contains *all* the possible complete procedural controllers.

Definition 8: Maximal Superstructure of Procedural Controllers. A superstructure of complete procedural controllers R is maximal for a specification language K s.t. $L(R) \subseteq K$, if $L(R) \cap L(P) = K\uparrow$.

Given a process model $P = (Q, \Sigma, \delta, q_0, Q_m)$ and a specification model $M = (X, \Sigma, \gamma, x_0, X_m)$, the following algorithm is proposed for the synthesis of the maximal superstructure.

Algorithm 1: Maximal Superstructure Synthesis

(1) Let $U = X$, $f: X \to Q$, $f(x) = q$ if $\exists s \in \Sigma^* | q = \delta(s, q_0)$, $x = \gamma(s, x_0)$

(2) For each $u \in U$, if $\exists \sigma_u \in \Sigma_u$ s.t. $\delta(\sigma_u, f(u))! \wedge \gamma(\sigma_u, u)$ undefined, then:

(a) If $\exists \sigma_c \in \Sigma_c$ s.t. $\gamma(\sigma_c, u)!$ then set $\gamma(\Sigma_u, u)$ to undefined.

(b) Otherwise, let $U = U - u$ and $\forall x \in X$, $\sigma \in \Sigma$ if $u = \gamma(\sigma, x)$ then set $\gamma(\sigma, x)$ to undefined.

(3) Next u.

(4) If $U \neq X$ then let $X = U$ and go to Step 1.

(5) Stop with $R = (X, \Sigma, \gamma, x_0, X_m)$, $K\uparrow = L(R)$

Algorithm 1 terminates generating the maximal superstructure (see proof in Appendix C).

Synthesis of a procedural controller

Finally, the following theorem gives conditions for the existence of a nonblocking procedural controller satisfying a specification language K.

Theorem 2: Given a specification language $K(M) \subseteq L(P)$ and $K_m = L_m(P) \cap K$ $(K_m \neq \emptyset)$, if M is trim and K_m is closed and controllable then $\exists C = (X, \Sigma, \gamma, q_0, X_m)$ nonblocking and complete, with $L(C/P) \subseteq \overline{K_m}$.

The proof is carried out by construction and serves, as well, as the algorithm for calculating the procedural controller.

Proof and Algorithm 2: The following procedure generates a complete nonblocking controller C from $M = (X, \Sigma, \lambda, x_0, Q_m)$, $L(M) = \overline{K_m}$.

Procedure:

(1) Let $C = M$ and $X_u = \{x | x \in X \wedge \gamma(\Sigma_u, x) \neq \emptyset \wedge \gamma(\Sigma_c, x) \neq \emptyset\}$.

(2) For each $x \in X_u$:

(a) $\forall x' \in X$, $\sigma \in \Sigma$ s.t. $\gamma(\sigma, x')!$, define $\psi(\sigma, x') = \gamma(\sigma, x')$ if $(\sigma \in \Sigma_u \vee x' \neq x)$.

(b) If $\forall \sigma_u \in \Sigma_u$ s.t. $\gamma(\sigma_u, x)!$, $\exists s \in \Sigma^*$ s.t. $\psi(\sigma_u s, x) \in X_m$ then set $\gamma(\Sigma_c, x)$ to undefined.

(c) Otherwise set $\gamma(\Sigma_u, x)$ to undefined.

(d) Next x.

(3) Let $X_c = \{x | x \in X \wedge \exists \sigma_c, \sigma_c' \in \Sigma_c$ s.t. $(\gamma(\sigma_c, x)! \wedge \gamma(\sigma_c', x)!)\}$.

(4) For each $x \in X_c$:

(a) Let $\Sigma_x = \{\sigma_c | \gamma(\sigma_c, x)!\}$.

(b) Select $\sigma_c \in \Sigma_x$. Let $\Sigma_x = \Sigma_x - \sigma_c$.

(c) $\forall x' \in X$ define $\psi(\sigma, x') = \gamma(\sigma, x')$ if $(\sigma = \sigma_c \vee x' \neq x)$.

(d) If $\exists s \in \Sigma^*$ s.t. $\psi(\sigma_c s, x) \in X_m$ then $\forall \sigma \in \Sigma$, $\sigma \neq \sigma_c$ set $\gamma(\sigma, x)$ to undefined. Otherwise go to Step 4b.

(e) Next x.

(5) Trim C.

(6) Stop with $C = (X, \Sigma, \gamma, q_0, X_m)$ a complete nonblocking controller.

This procedure will stop in at the most $|2X|$ iterations. Note that the controller is not necessarily trim. Appendix D presents an inductive demonstration that the C obtained is nonblocking and complete.

Synthesis Method

The modeling tools proposed in the Modeling and Specification of Closed-Loop Behavior sections, and the control paradigm presented in the previous section, are assembled together in the synthesis method shown in Figure 4. Rounded rectangles represent input information, while sharp edged rectangles depict information created during the synthesis method. The ovals correspond to the four main steps of the method. Steps 1 and 2 are associated with the modeling of the process and the desired behavior specifications. Steps 3 and 4 are concerned with the actual design of the superstructure and the procedural controller. The method is iterative. Decision points are presented as diamonds.

Step 1: process modeling

The process is modeled using the tools presented earlier. First, the elementary process components are identified from a suitable description of the system (such as a P&I diagram)

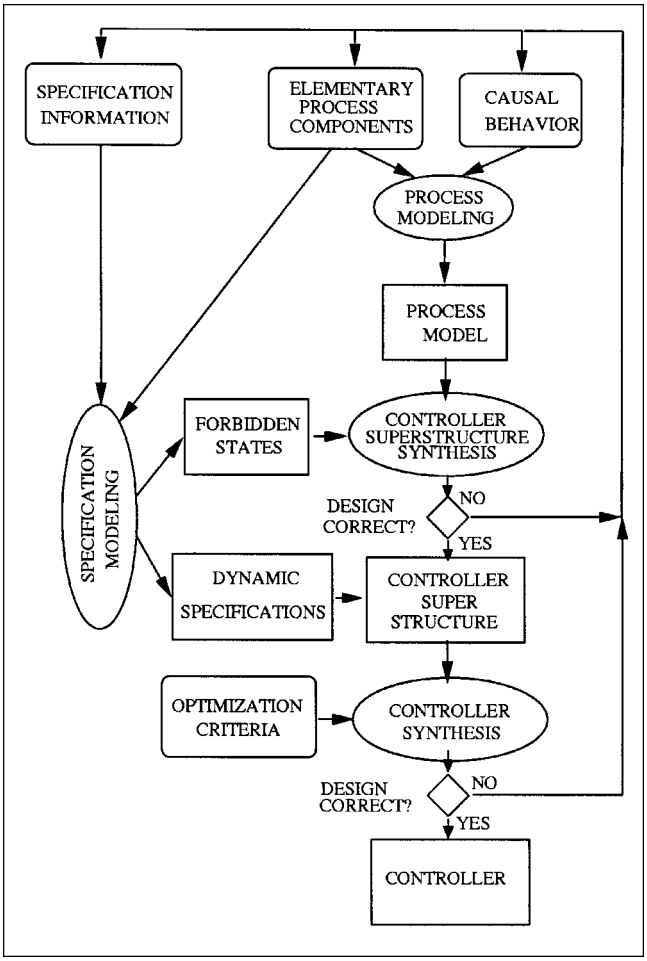

Figure 4. Procedural controller synthesis procedure.

and modeled as FSMs. Then, nonphysical behavior is specified using predicate and temporal logic and the open-loop model is constructed using the relevant FSM operations. Finally, the states of particular relevance to the operation are *marked* and/or proposed as attractors. These states represent initial and final conditions of startup and shutdown operations, conditions in which the process can be halted, normal operating conditions, or safe states. The final FSM for the overall process is trimmed, so that all states (marked and nonmarked) can be reached from the initial state.

Step 2: specification modeling

Forbidden states specifications are captured using predicate logic while dynamic specifications, including liveness properties as well as desired and undesired partial sequences, are formalized using temporal logic. The specifications are then translated into the FSM domain according to the approach described in the Closed-Loop Behavior section.

Step 3: derivation of maximal superstructure

The model resulting from eliminating forbidden states represents the ''legal'' behavior of the process. From this, the maximal superstructure (Definition 8) is obtained using Algorithm 1. If its accepted language is empty or does not include part of the required behavior, then design changes need to take place.

Step 4: controller synthesis

The controller synthesis is based on the successive application of the synchronous product of the maximal superstructure with each dynamic specification FSM. If the resulting structure does not meet the requirements for a procedural controller (Definition 3), then either the controller is underspecified and further dynamic specifications are necessary; or the choice of a final control structure is carried out using Algorithm 2. Rotstein et al. (1999) proposed another approach for the controller synthesis based on the notion of *attraction* for discrete event systems (Brave and Heymann, 1993) and a notion of weighted transitions to assess optimal trajectory performance. Obviously, these two options can be combined. From the point of view of optimization, the direct imposition of specifications is equivalent to the addition of constraints to the system. On the other hand, the optimization procedures select the best procedural control structure that fulfills the constraints specified by the user. For instance, if there are two alternative trajectories from a state to an attractor, one can either choose directly one of the options, by incremental specification, or select the ''optimal'' according to a defined objective function (such as the smallest probability of failure) of control actions. The chosen alternative depends on the application and the user.

The final result is an FSM, termed procedural controller, that ''accepts'' those sequences satisfying the given specification and the definitions of controllability and completeness. In other words, a procedural controller is an abstract representation of the logic driving the process according to the specification. Thus, it can be used as a provable correct specification for a given logic control system.

Solution to the Motivating Example

The proposed synthesis method is first demonstrated by generating a provable correct procedural controller for the pressure system example presented earlier. It is also shown how the method can be used to aid in the analysis of operation requirements. The information available is given by the description of the example in the second section. According to the method presented in the previous section, the first step is building the open-loop behavior model. The next step is eliciting the specifications both as forbidden states and dynamic behavior. In a first attempt, two forbidden states are declared: (1) never allowing the valve being open when the button is off, and (2) conditions when the pressure sensor indicates high. Forbidden state 1 will result, when carrying out Step 3, in an operating condition which is not possible to satisfy, thus generating an empty controllable language. This will lead to reconsider the operation requirements and recognize the impossibility of avoiding forbidden state 1, thus improving the understanding of the process behavior and therefore the design of the controller. A second attempt considers only forbidden state 2 in the generation of the controller superstructure. This leads to a successful calculation of a procedural controller applying Step 4.

Table 1. Transition List for Elementary Models of the Pressurized Gas System

Elementary Component	Transitions			
	Label	Description	From State	To State
Gas Valve	11	open_valve	closed	open
	12	close_valve	open	closed
On/Off Button	21*	switch_on	off	on
	22*	switch_off	on	off
Pressure Switch	31*	from_high_to_ok	high	ok
	32*	from_ok_to_low	ok	low
	33*	from_low_to_ok	low	ok
	34*	from_ok_to_high	ok	high

*Uncontrollable transition.

Step 1: process modeling

Three elementary process models are identified: (1) control valve (V1); (2) button for operator intervention (B1); and (3) pressure sensor (PS). The FSM models for each of the sensors and control items in the P&I diagram are shown in Figure 1. Initial states are indicated with an entering arrow. Controllable and uncontrollable transitions are labeled according to Table 1. Note that the only control actions that the controller can enforce is the opening and closing of valve V1. The operator acts independently from the control system and, therefore, his/her actions are represented by uncontrollable transitions (that is, the controller cannot prevent the operator from pressing the button at any moment during the operation). The pressure sensor indicates three landmark values: (1) low (below target); (2) OK (at target); (3) high (above target). The global state of the process is represented by a vector of 3 elementary state-variables ordered as follows

$$(\text{V1, B1, PS})$$

The fully interleaved process model, obtained from the asynchronous product of the elementary FSMs, contains $2\times2\times3=12$ states. This model is simplified by pruning transitions lacking physical meaning, such as a PS indicating an increase in pressure when the valve is closed. The final model is shown in Figure 5. For clarity, transitions are labeled with number tags. Its tabular equivalent is presented in Table 1. The initial state of the system is indicated with an entering arrow. The only marked state (the desired target state) is denoted with an exiting arrow.

Step 2: specification modeling

From the description of the desired operation given in the second section, forbidden states are expressed in predicate logic statements as follows:

- If the button B1 is off, then the valve V1 must always be closed

$$(\text{open}, \text{off}, \infty) = \text{false} \quad (4)$$

- The pressure should never be high

$$(\infty, \infty, \text{high}) = \text{false} \quad (5)$$

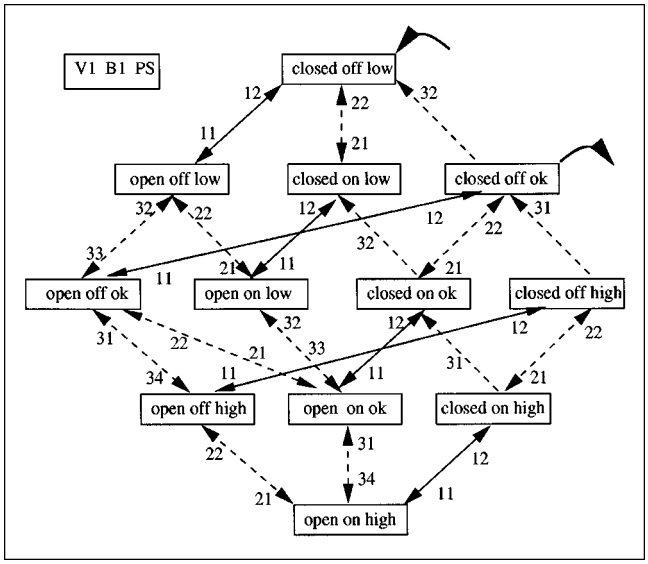

Figure 5. FSM model for the open-loop behavior of the pressurized gas system.

Statement 4 represents 3 states, one for each value that variable PS can take: (open, off, low), (open, off, ok), (open, off, high). Statement 5 can be unfolded into 4 states: (closed, off, high), (open, off, high), (closed, on, high), (open, on, high).

Next, the desired normal dynamic behavior of the system is specified as follows:

- From the initial state, when the pressure is low and the valve is closed, the operator must press the button for the controller to issue a command to open V1. The formal representation of this requirement is given by the following temporal logic formula modeling a sequencing of events

$$(\text{closed}, \text{off}, \text{low}) \rightarrow \bigcirc[\tau = \text{switch\_on}] \rightarrow \bigcirc[\tau = \text{open\_valve}] \quad (6)$$

- If the operator changes the button B1 to off, a command must be issued to shut valve V1

$$(\text{open}, \text{on}, \infty) \wedge (\tau = \text{switch\_off}) \rightarrow \bigcirc[\tau = \text{close\_valve}] \quad (7)$$

- If the signal ''pressure OK'' is detected from PS then a command must be issued to close V1

$$(\text{open}, \infty, \text{low}) \wedge (\tau = \text{from\_low\_to\_ok}) \rightarrow \bigcirc[\tau = \text{close\_valve}] \quad (8)$$

These three specifications are translated into the FSM domain for their use in controller synthesis. The homomorphism mentioned in the fourth section is applied to obtain the FSMs shown in Figures 6, 7, and 8. For example, in the case of formula 6, a lattice is generated in which the *glb* is state {closed, off, low} and the *lub* is its complement $\{\infty^{\text{closed}}, \infty^{\text{off}}, \infty^{\text{low}}\}$. The next step takes the transition executing from

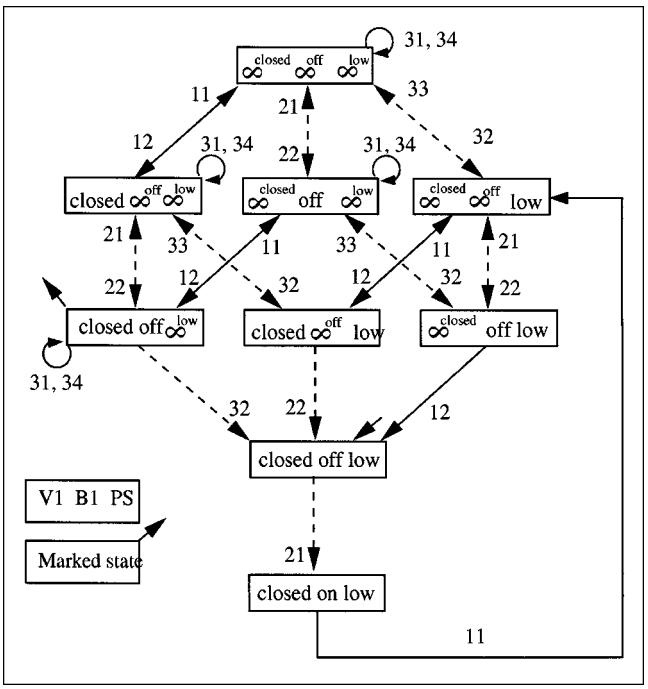

Figure 6. FSM model for specification (6).

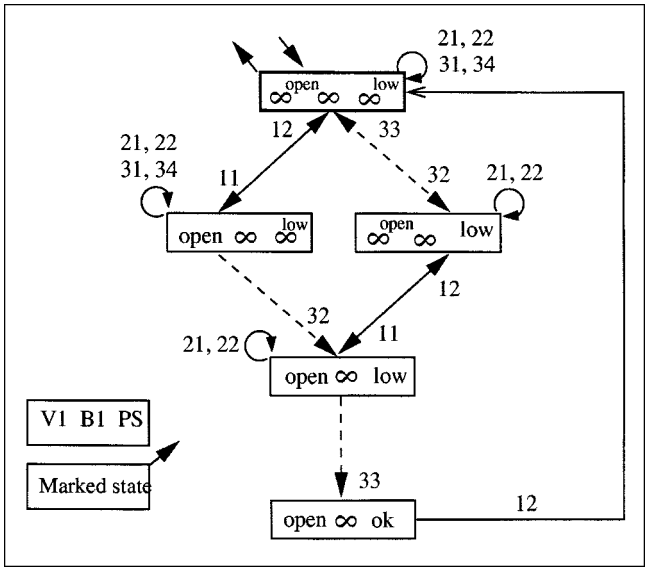

Figure 8. FSM model for specification (8).

state {closed, off, low} and constructs the rest of the FSM introducing the necessary states. In this case, the state {closed, on, low} follows the detection of the signal switching on the button as the only alternative prescribed in the TL formula. Then the control command to open the valve is issued. This last transition is connected to the most similar node corresponding to state {open, on, low}.

Step 3: maximal controller superstructure synthesis

First Attempt Using all Forbidden States. The maximal controller superstructure is obtained first by eliminating the forbidden states. The resulting FSM is shown in Figure 9. Note that all the states where the pressure is high, as well as those where B1 = off and V1 = open simultaneously, have been deleted. The maximal superstructure of controllers (that is, that generates the supremal controllable sublanguage of the legal model) is then obtained applying algorithm 1. The result is shown in Figure 10. Unfortunately, there are only two reachable nodes remaining in the superstructure (that is, {closed on low} and {open on low}). This means that the only possible behavior described by this machine is an infinite loop manipulating the start button and the valve. Thus, no procedural controller will exist satisfying the proposed specifications in a useful manner. The reason can be found in the specification of forbidden states. Statement (4), which pre-

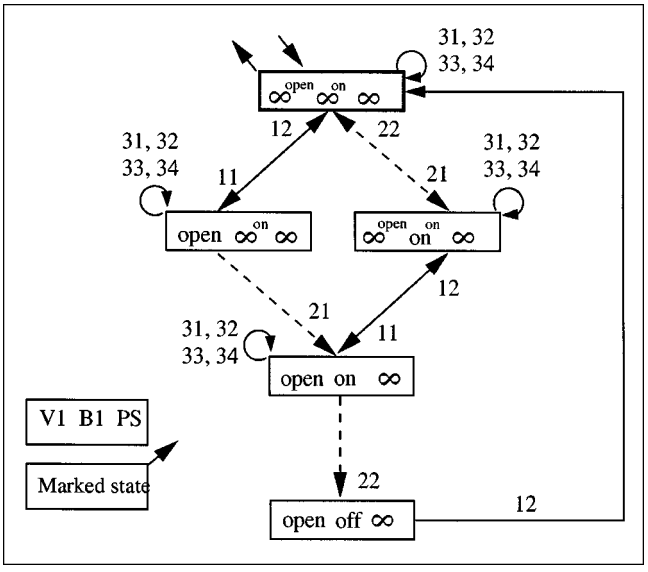

Figure 7. FSM model for specification (7).

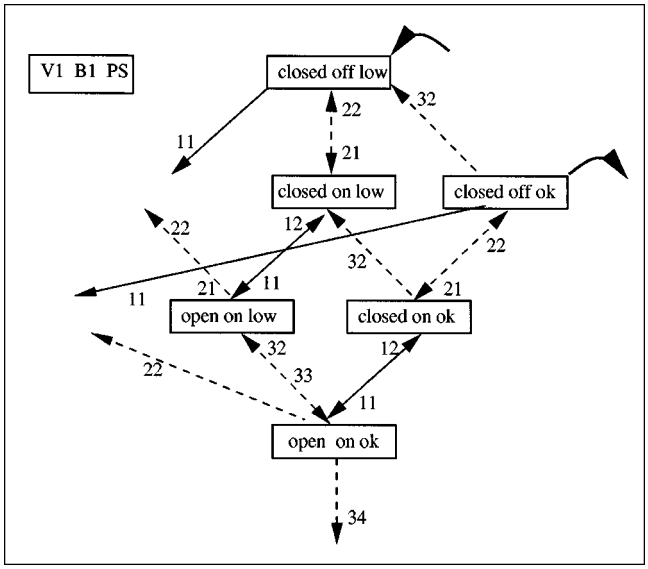

Figure 9. FSM after eliminating forbidden states (4) and (5).

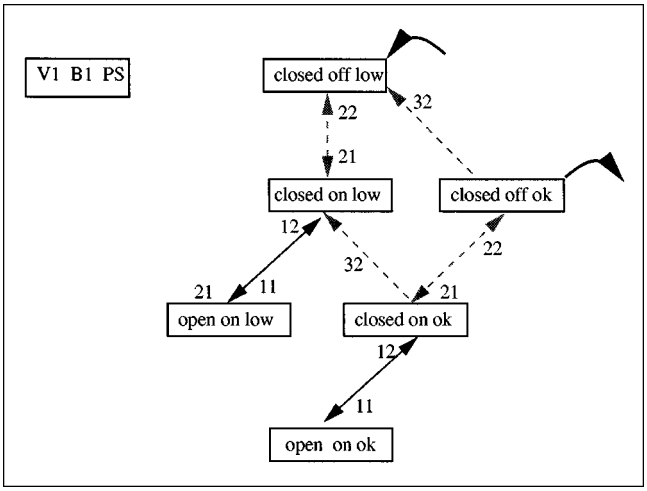

Figure 10. Maximal superstructure for forbidden states (4) and (5).

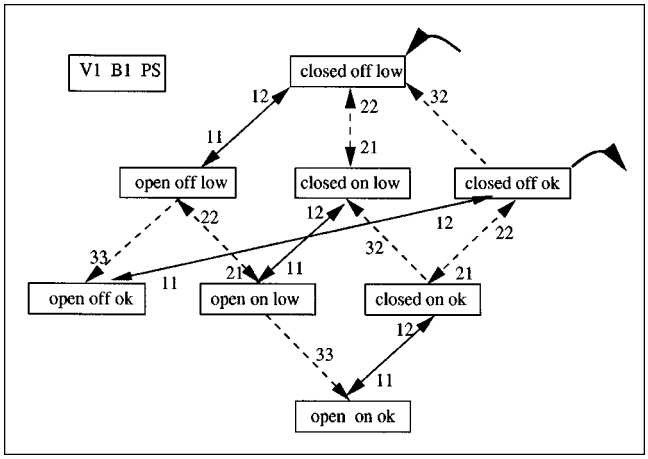

Figure 11. Maximal superstructure for forbidden state (5).

scribes that valve V1 must always be closed when the button B1 is off, is invalid when the operator changes the position of the button to off if the valve is open, thereby driving the system to the forbidden state. The control system cannot prevent the operator from doing that. Thus, the specifications must be modified. Rather than forbidding a behavior over which the controller has no control, this behavior must be included as a part of the legal behavior and a suitable control action defined for it by means of a dynamic specification. Note that the required behavior is already included in dynamic Specifications (6) and (7). In Formula (6), from the initial state {closed, off, low}, the only way of opening the valve is after receiving the signal that the button has been pressed. Formula (7) prescribes that if the valve is open and the button is switched off, the next action must be to close the valve.

If this approach is adopted, then the only forbidden state specified is Statement (5) prescribing that the pressure must not increase beyond OK.

Second Attempt Using only Forbidden State (5). Eliminating forbidden state (5) from the open-loop model (Figure 5) and applying Algorithm 1, the maximal superstructure in Figure 11 is obtained. Controllability with respect to the open-loop model is checked applying the definition state by state and corroborating that for each state in the superstructure either

- all uncontrollable transitions existing in the equivalent state in the open-loop model are also present; or
- there exists at least one controllable transition.

An example of the first condition is state {open, off, low} where uncontrollable Transition 21 (button can be pressed) or Transition 33 (pressure can rise) can occur. An example of the second condition is state {open, off, ok} where uncontrollable transitions (that is, Transition 34, increasing pressure to high) in the open-loop model lead to erased states in the superstructure. Thus, only controllable Transition 12 (command to close the valve) was kept. The next step is to determine whether a procedural controller exists satisfying the given dynamic specifications.

Step 4: procedural controller synthesis

A procedural controller implementing the three dynamic specifications (Figures 6, 7, and 8) is then obtained by executing first the synchronous product of the FSMs representing these specifications and the maximal superstructure derived above (Figure 11). As an illustration, the result of the synchronous product between Specification (6) and the maximal superstructure is shown in Figure 12. From the initial state {closed, off, low}, the only surviving transition is 21 (*switch_on* the button), as stated by Specification (6). Executing this transition leads to a state where the valve is closed, the button is on, the pressure sensor indicates low and the command to open the valve must be issued [Transition (11)]. However, note that, according to the open-loop model, this condition can be part of other trajectories. For instance, if the valve is open, the button is on, the pressure is low, and a command is issued to close the valve, it should be possible to switch the button off or even to open valve V1 again, since there are no specifications impeding it. Thus, the situation is

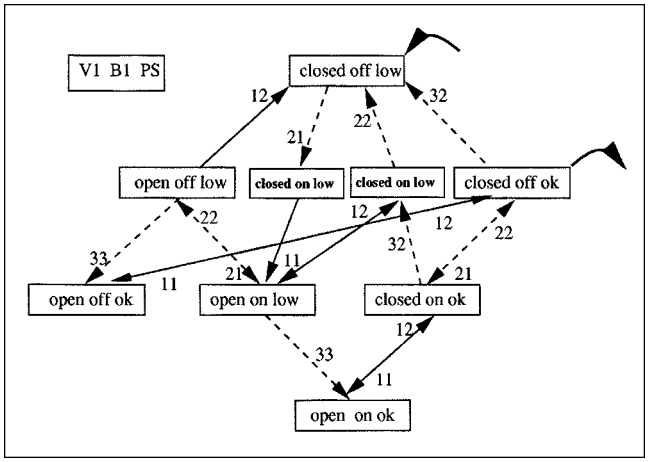

Figure 12. Resultant FSM from the synchronous product of dynamic specification 8 and controller superstructure.

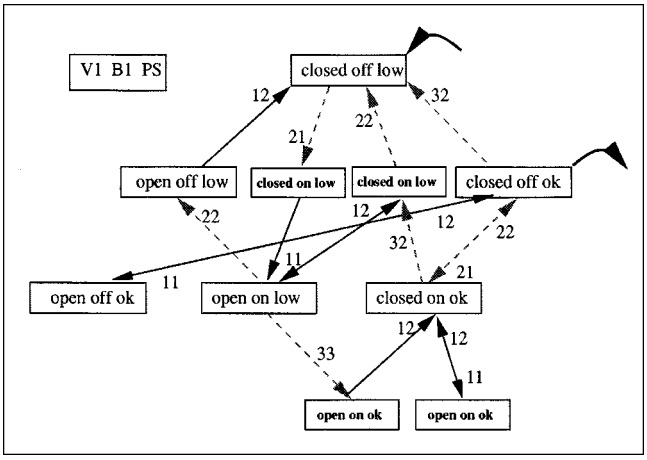

Figure 13. Resultant FSM from the synchronous product of dynamic specifications and controller superstructure.

resolved during the execution of the synchronous product by duplicating this state to allow the execution of other legal trajectories not declared in Specification (6) as shown in Figure 12. The resulting machine forms the synchronous product of the superstructure with the three specifications is shown in Figure 13. This FSM is not yet a procedural controller. For instance, from state {open on low}, uncontrollable Transitions 22 (switch button off) and 33 (pressure rising to normal) are allowed to execute together with controllable Transition 12 (command to close valve). Thus, Algorithm 2 is applied to obtain a nonblocking procedural controller. The trimmed FSM is shown in Figure 14. Its behavior satisfies the three dynamic specifications as indicated by regions 1 [Specification (6)], 2 [Specification (7)] and 3 [Specification (8)] in the figure. Region 4 shows other legal behavior that the controller will be able to handle.

Alternatively, the synthesis could be carried out using the stability approach (Rotstein et al., 1999). The first step is to identify transitions in the system which indicate a *failure* and, therefore, should not be included when the stabilizing controller is generated. In the example these transitions are the unexpected release of the button by the operator when the pressure is not OK and a change in the pressure when the valve is closed. Note that the controller obtained will still be able to drive the system to the attractor even if these transitions take place. Now, the main option existing in this process is whether to wait or not for an indication from the operator to start the operation. Using the successive specification approach previously illustrated, the first specification was used to express the desire to wait for the operator signal. Here one can either use the same specification, and generate a controller forcing stability in the remaining behavior, or introduce a large weight in the transition corresponding to the opening of the valve from the initial state. Note that a stabilizing controller will automatically satisfy the second and third specifications and, therefore, these are not required.

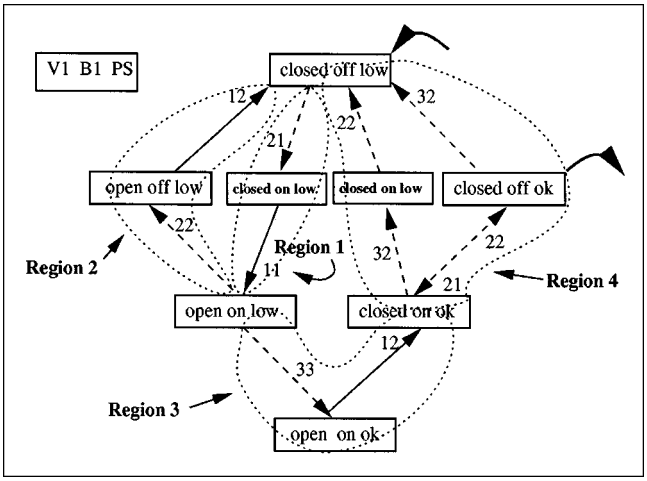

Figure 14. Procedural controller for pressurized gas system.

Case Study: CIP Operation

The specification and synthesis methods described above were applied to a Cleaning-In-Place (CIP) operation of a computer-controlled batch pilot plant at Imperial College. The process and instrumentation diagram of the plant is shown in Figure 15 [reproduced from Liu (1995)] and described fully in Macchietto (1992). This highly instrumented and flexible plant is representative of small-scale multipurpose food, fine chemicals, or pharmaceutical plants. The pilot plant is equipped with a 100L multipurpose batch reactor (tank T3) with two 100L feed preparation vessels (tanks T1 and T2), two 100L product storage vessels (tanks T4 and T5), and three plate heat exchangers. Highly flexible connectivity between plant units is achieved via a complex network of pipes, pumps, and single and double-seat valves. Transfers may be carried out simultaneously except where they share common pipework. Most of the 45 automated on/off valves have two feedback position sensors. T3 is equipped with a jacket for heating or cooling, a stirrer, load cell, viscometer and facilities for sparging and dosing. In addition to the main process equipment, a cleaning-in-place (CIP) system enables sections of the plant to be individually cleaned with a hot caustic detergent solution from the detergent station (tank T7).

This environment is seen as a challenging area for the testing of methods for the design of procedural controllers for the following reasons:

- The processes are typically run in batch mode and require a CIP to maintain hygiene standards and product consistency. Batching and cleaning operations are inherently discrete, the control of which fits well within the procedural control paradigm.
- The production environment is highly flexible. Products, recipes, and equipment configurations change frequently. The procedural control imposed on the plant must change accordingly.
- Process operations are often complex and can only be handled using a hierarchical control structure [such as that proposed in the ISA-S88 standard (1995)]. Many communicating procedural controllers arranged in a suitable hierarchical structure are required to perform processing operations.

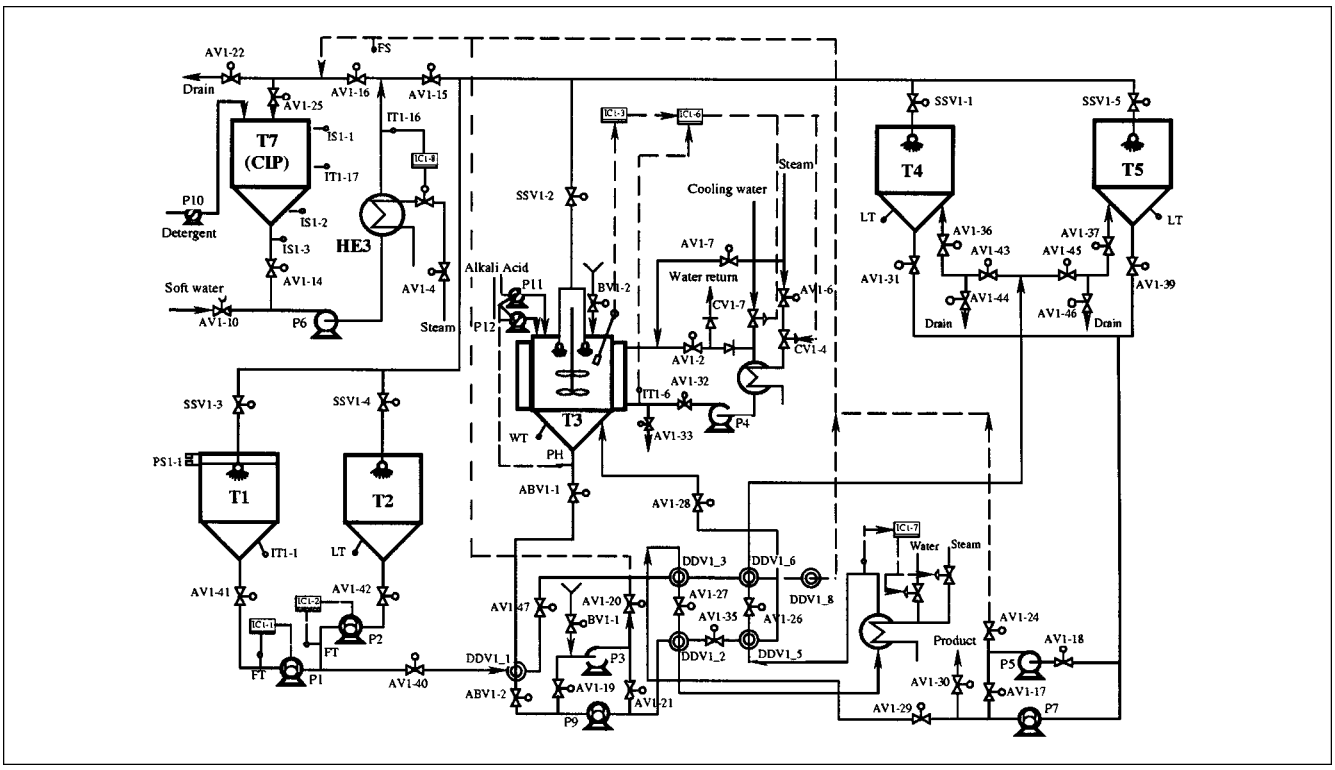

Figure 15. Batch pilot-plant process and instrumentation diagram.

The batch pilot plant comprises approximately 75 output channels (such as valves, pumps, and control loops) and 45 inputs channels (such as sensors and switches). If each channel has two discrete states, the total number of states of the batch pilot plant is of the order of 10^{36}. Obviously, decomposition techniques are required for handling the full batch pilot-plant model. Using ISA-S88 standards, an appropriate decomposition was achieved (Alsop et al., 1996). In this section, the generation of the procedural controller and its implementation as software for one phase (pre-rinse) of the CIP operation for tank T1 and its associated pipework, as described in Liu and Macchietto (1993), is used as an example. The objective is to perform the cleaning automatically and *in situ*. The overall CIP operation is a complex sequence of discrete event-driven activities which may be broadly divided into four distinct phases including:

(1) *Pre-rinse.* In which residual solids are removed from the interior of T1 and associated pipe work by bursts of high-pressure water.

(2) *Detergent service.* The preparation of an inventory of hot concentrated caustic solution at the CIP station (T7).

(3) *Detergent rinse.* Interior surface cleansing of T1 by high-pressure bursts of hot concentrated caustic solution.

(4) *Post-rinse.* Final rinsing of T1 with water to remove residual detergent and to yield the equipment suitable for food contact.

The phase structure needed to perform the CIP operation is implemented in a proprietary sequential control language, running in an industrial DCS. At the highest level (phase control), the pre-rinse, detergent service, detergent clean, and post-rinse phases are started and stopped, respectively. Phases call lower level sequences, which perform progressively more detailed steps.

The rest of this section is devoted to the synthesis of the procedural controller for the pre-rinse phase using the methods proposed here. Finally, it is shown how the procedural controller is implemented as control code.

Controller synthesis

A simplified P&I diagram of the equipment employed for the pre-rinse phase only is shown in Figure 16. Mains water is fed via a complex series of valves (collectively labeled ''feed route''), pump P6, and valve SSV1-3 to tank T1. T1 is fitted with a proximity switch PS1-1, which detects the position of the lid. For safety reasons, the rinsing procedure must cease should the lid be opened during operation. T1 is also fitted with a continuous level sensor IT1-1. Spent water from tank T1 is drained via valve AV1-41, pump P1, and a complex series of single and double-seat valves (collectively labeled ''return route'') to drain.

Pre-rinse is effected by successive drain and fill cycles. High-pressure water from P6 is sprayed into T1 until a volume of 20 L is achieved. Pump P1 drains the residue to a volume of 6 L after which the cycle repeats. Since the capacity of P6 is much greater than P1, P1 need not be switched off during the fill cycle. However to avoid cavitation, P1 is deenergized if the volume in tank T1 falls below 3 L.

Step 1: Process Modeling. Eight elementary process models are identified from the flowsheet for the purpose of build-

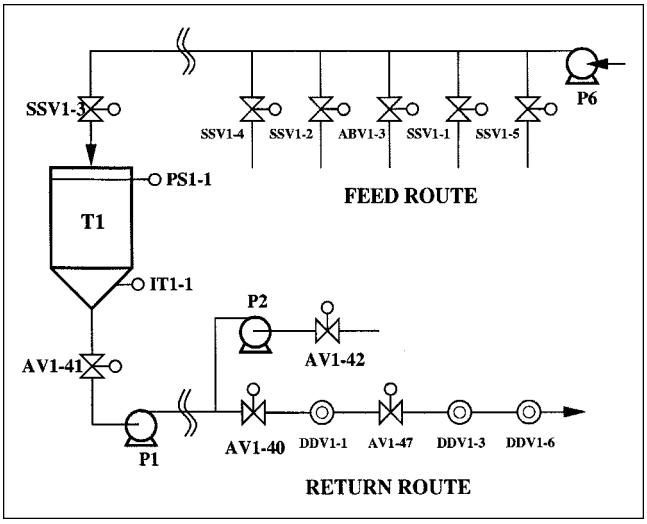

Figure 16. Simplified process diagram for pre-rinse phase.

ing the process models and specifications. These include the feed route, pump P6, valve SSV1-3, valve AV1-41, pump P1, the return route, the proximity switch PS1-1, and the level indicator IT1-1. The valves, pumps, and routes are all modeled by similar two-state on/off FSMs, as shown in Figure 17 in which the two states are separated by controllable transitions. Table 2 assigns to each controllable transition a unique transition number.

The feed and return route FSMs model a series of valves along a path. Routes are defined as ''set'' when every valve along the path is open. To set or close a route requires a sequence of primitive valve operations. These sequences are subtasks of the pre-rinse phase, which are synthesized separately. The tank lid may at any time be opened by an operator. Thus, the two states of the FSM modeling the tank lid proximity switch are separated by uncontrollable transitions

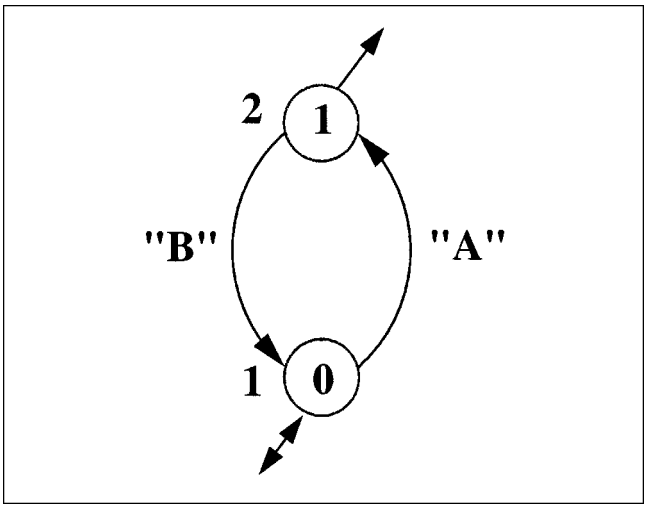

Figure 17. Elementary two-state on/off FSM for pre-rinse case study.

Table 2. Controllable Transitions of FSM Model in Pre-Rinse Phase Case Study

Item	Transition	Description	Tag
Feed route	A	Set route	43
	B	Close route	44
P6	A	Energize pump	17
	B	Deenergize pump	18
SSV1-3	A	Open valve	1
	B	Close valve	2
AV1-41	A	Open valve	33
	B	Close valve	34
P1	A	Energize pump	7
	B	Deenergize pump	8
Return route	A	Set route	29
	B	Close route	30

as shown in Figure 18. IT1-1 continuously measures level. A transition is defined when this measurement crosses one of the thresholds as described earlier. The threshold levels are defined at 3, 6, and 20 L. Thus, the four-state FSM shown in Figure 19 models the level measurement from IT1-1.

The global state of the process is represented by the vector of 8 elementary state-variables ordered as follows

(feed route, P6, SSV1-3, PS1-1, IT1-1, AV1-41, P1,

return route)

The fully interleaved process model, obtained from the asynchronous product of the elementary FSMs contains $2^7 \times 4 = 512$ states.

Step 2: Specification Modeling. The specifications for the system emerge from a consideration of the equipment constraints and the desired startup, continuous, and emergency operation. Forbidden states and dynamic specifications are listed in Appendices E and F, respectively. There are two types of dynamic specifications, the first specifies startup and normal behavior (Appendix F) and the second denotes abnormal behavior (Appendix F). Specifications are presented first in natural language and then translated into predicate or temporal logic formulas. For example, consider the first state specification from Appendix E.

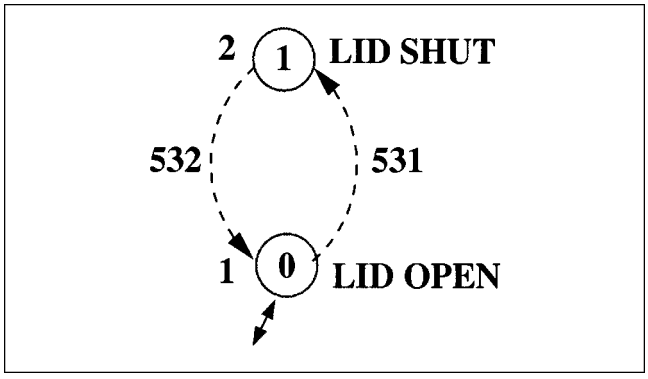

Figure 18. Proximity switch FSM for pre-rinse case study.

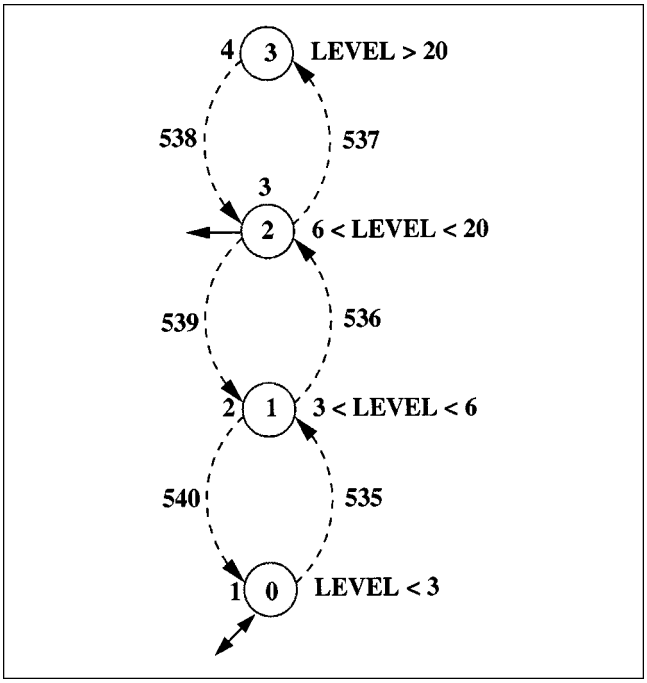

Figure 19. Level sensor FSM for pre-rinse case study.

If the feed route is not set then P6 cannot be energized.

This is imposed to protect pump P6 from dry pumping. It translates into a predicate logic statement as follows

$$(0, 1, \infty, \infty, \infty, \infty, \infty, \infty) = \text{FALSE} \tag{9}$$

The covering symbol ∞ is employed in Statement 9 to indicate that the states of SSV1-3, PS1-1, IT1-1, AV1-41, P1 and the return route are irrelevant to this specification. Expression (9) says that the state in which the feed route is closed (that is, has the value 0) and pump P6 is energized (that is, has the value 1) is a forbidden state (that is, cannot occur in the controlled system).

As an example of dynamic specification, consider the following statement taken from Appendix F handling abnormal behavior

If the lid opens while P6 is energized, then immediately

de-energize P6 and close SSV1-3

This specification protects the operator from exposure to the potentially dangerous cleaning fluid. It translates into the following temporal logic formula

$$(1, 1, 1, 1, \infty, \infty, \infty, 1) \wedge (\tau = 532) \rightarrow \bigcirc[\tau = 18] \rightarrow \bigcirc[\tau = 2] \tag{10}$$

Again, the covering symbol ∞ is employed to indicate that the states of IT1-1, AV1-41, and P1 are irrelevant to this specification. Formula 10 says that when both the feed and product routes are set (have the value 1), SSV1-3 is open (that is, has the value 1) and pump P6 is energized (that is, has the value 1), if the lid opens (that is, transition 532 occurs), then the next transition must be to de-energize pump P6 (that is, execute Transition 18) followed immediately by closing valve SSV1-3 (that is, execute Transition 2). This TL formula is translated into a 65-state FSM (64 states for the refined state-variable lattice plus an extra state reached by Transition 18).

Step 3: Synthesis of Maximal Controller Superstructure. The 512 state FSM obtained from the asynchronous product of the elementary FSMs was reduced to a 128-state FSM following the deletion of the forbidden states given in Appendix E and the generation of the trim FSM accepting a controllable language using Algorithm 1. In other words, all states outside the controllable region are automatically avoided, and the 128 state structure represents the maximal superstructure of procedural controllers. Should the language not be controllable, the design engineer must change either the P&I diagram or the specifications.

Note that the design engineer could initially attempt to include a forbidden state specifying that the lid should never be open when the pump is functioning. Since the transition representing the opening of the lid is uncontrollable (that is, the control system cannot prevent the operator from performing this action) an empty maximal superstructure would result. This indicates that one cannot prevent this state but must rather specify actions for it, as done in formula 10.

Step 4: Controller Synthesis. On performing the synchronous product of the maximal superstructure and the temporal logic formulas (Appendix F) mapped to the FSM domain and obtaining a controllable language, a 37 state FSM, shown in Figure 20, was generated. Again, this FSM is trim and its language is controllable. Thus, it can be interpreted as a procedural controller. Normal operation sequencing is indicated in the figure, comprising 15 nodes, while the rest characterizes the abnormal operation.

Implementation in automated systems

Automated procedural control of chemical processes is usually implemented by PLCs or DCSs. Control is exerted upon a process as the program chains through a set of executable instructions (controlled transitions), branches, and conditionals on plant feedbacks. Control algorithms are preprogrammed into these devices using a variety of proprietary languages. A long-term goal for the control systems industry is to automatically generate provably correct low-cost modules of control code that can be implemented in one of these devices. The feasibility of this goal is partially demonstrated here by translating the procedural controller for the pre-rinse phase, including normal and abnormal operation, into an industrial proprietary sequential language, PARACODE (APV-Baker, 1994) and implementing it in a DCS. PARACODE is a high level, sequential, real-time control language which supports a modular structure. The language and its associated tools and control system has been in industrial use for over 15 years mainly in the food industry. The procedural controller synthesized here represents the operational part of the phase while PARACODE supplies the computational and monitoring support needed for real-time operation.

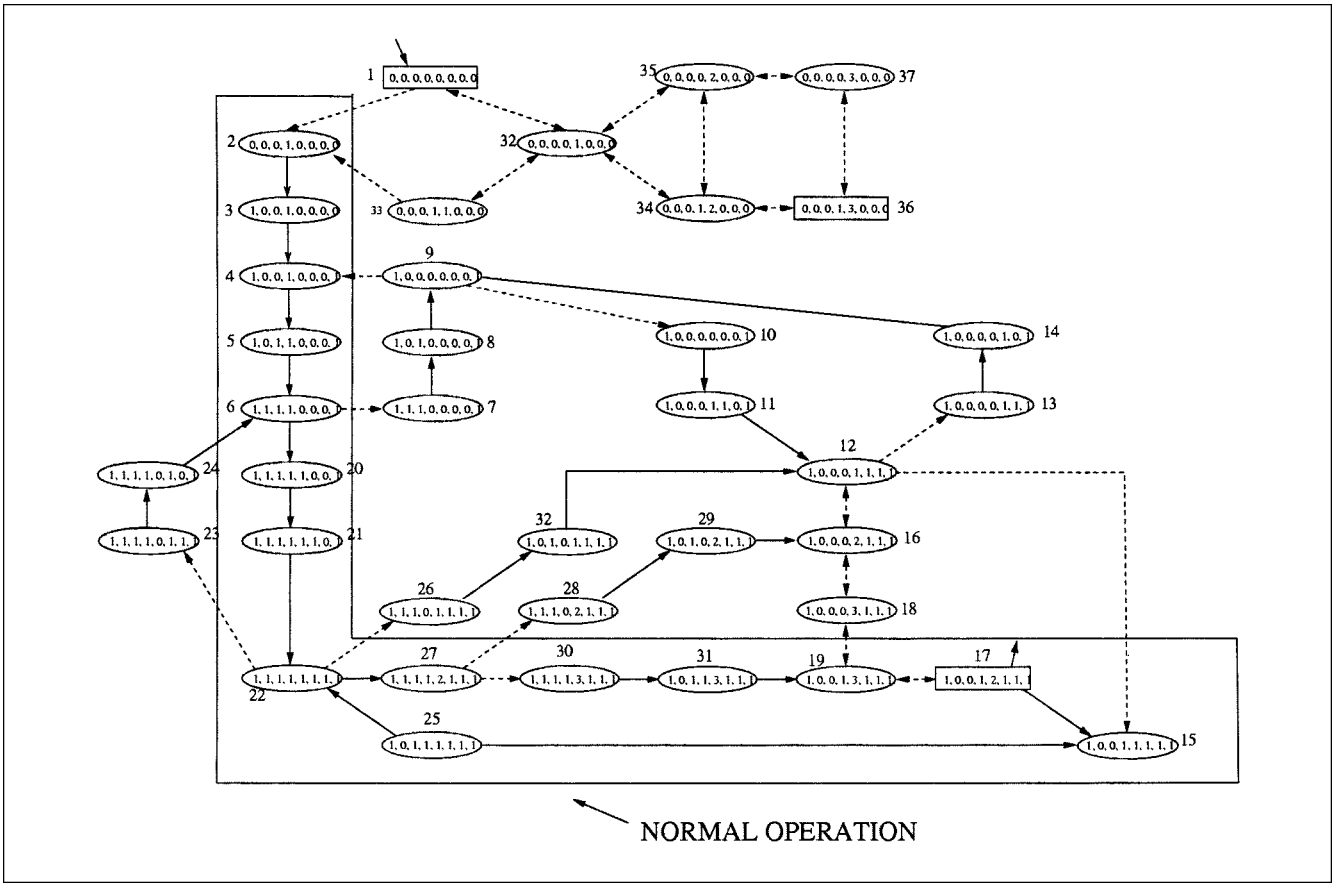

Figure 20. Procedural controller for pre-rinse phase.

Requirements for Implementation of a Procedural Controller. In order to map a procedural controller $C=\{X, \Sigma, \xi, x_0, X_m\}$ into PARACODE, a mapping is required such that, for every controllable transition, $\sigma \in \Sigma_c$ a corresponding executable instruction (such as, open valve) exists and for every uncontrollable transition $\sigma \in \Sigma_u$ a conditional statement (such as, if flag set then) exists. This requirement can be met by appropriate modeling of the primitive components from which the FSM procedural controller is synthesized. Executables (that is, controllable transitions) and conditionals (that is, uncontrollable transitions) refer to various plant items including pumps, valves, switches, counters, flags, or other devices. In addition, the implementation of the procedural controller C must have the ability to: (a) track the state transition function γ within the program, and (b) initialize the program in a state corresponding to x_0.

FSM Implementation into a Sequential Real-Time Control Language. The topology of C may be represented in a sequential program by associating a line (or a number of sequential lines) in the code with every state $x \in X$ plus an integer variable to uniquely identify x. A unique *line_number* is attached to the first line of code corresponding to state x in C. This line is always an assignment of the current *state_number* to the *diagnostic_variable* (a register used to store the corresponding state-variable value), and in PARACODE reads

line_number MOVN *state_number*, *diagnostic_variable*

The requirement for the program to start at the initial state x_0 is met by the following command before any other executable

GOTO *initial_state_line_number*

At any time, if C is in x, the program pointer is at or just below this line. Each state generates one of the following code structures:

(1) If $x \in X_m$, a command is inserted to freeze the program at its current position. Here marked states are those in which the operation of the controller is to be temporarily stopped. For example, a marked state is achieved once a controller has established and interlocked a path through a valve network by switching a series of valves. A freeze of the controller is implemented via the command

HLSQ *sequence_number*

(2) If $\gamma(\sigma, x)! \wedge \sigma \in \Sigma_c$ (that is, if a controllable transition occurs from x), the subsequent code is defined by the mapping to an executable instruction. The executable instruction set from the PARACODE language includes commands such as

- ENGE (To open a solenoid valve or to energize a pump and so on).

- STSQ (Starts a subroutine).
- RLTM (Releases a timer).
- SEFL (Sets a control flag).

Following a transition in C, the current state x changes to a new state in X as defined by γ. Correspondingly, the program relocates to the line corresponding to the new state of C. In PARACODE relocation is effected via the command

GOTO *new_line_number*

(3) For each σ such that $\gamma(\sigma, x)! \wedge \sigma \in \Sigma_u$ (that is, for each uncontrollable transition from x), the next line of code is defined by the mapping to a conditional followed by the line number to which the program branches should the conditional evaluate to true. This line number corresponds to the new state defined by the transition function γ.

The conditional set in PARACODE includes commands such as:

- IFIE (Tests on the status of valves or pumps, and so on).
- IFSNAC (Tests the status of a subroutine).
- IFTZ (Tests a timer status).
- IFFS (Tests a control flag status).
- IF *argument*, relational operator, *value* (that is, a relational expression).

For example, if an uncontrollable transition corresponds to a change in tank level from low to high level, the corresponding PARACODE reads

IF *level_variable*, GT, *value*, *new line_number*

Following the set of conditionals, an additional GOTO command is required in order that the program remains in the current state. Thus, the program is forced to await the uncontrollable transition(s) defined at that state.

(4) If $\gamma(\sigma, x)$ is undefined (that is, there exist no transitions from x), then a termination state has been achieved. At this point, the controller irretrievably ceases all operations (c.f., the frozen state). This is implemented in PARACODE by the STOP command.

The FSM procedural controller of Figure 20 was automatically translated into a program in PARACODE, a small sample of which is shown in Figure 21. The total number of program lines for this module is 425. An additional 25 lines of code were added by hand to inhibit existing sequences on the DCS which would disrupt the CIP phase, should they be inadvertently started while the CIP phase is in operation.

Finally, the PARACODE program was successfully compiled and executed on the DCS as part of the overall CIP phase. Figure 22 shows the profile of the T1 level and the lid position during an experimental pilot plant run of approximately 19 min. Normal and abnormal operations (that is, upsets in tank level and disturbance inputs from the lid switch) were all handled safely. At time 0, the pre-rinse module is invoked, however, the pre-rinse is prevented from starting as the tank lid is initially in the open position. At time 44s, the lid is closed and the pre-rinse phase begins. Water is sprayed into tank T1 by pump P6 filling it to a volume of 20 L. Measurement and control lags, however, result in a level overshoot of 3 L. Pump P1 commenced operation once the level first exceeded 3 L. The level is observed to cycle between

```
S301S13 MOVN    13, R1.301              / SET STATE VARIABLE
        ENGE    SSV1-3                  / CONTRL TRANS. TO OPEN SSV1-3
        WAIT    5                       / GRACE TIME
        GOTO    S301S14                 / GO TO STATE 14
/
S301S14 MOVN    14, R1.301              / SET STATE VARIABLE
        ENGE    P6                      / CONTRL TRANS. TO START P6
        WAIT    5                       / GRACE TIME
        GOTO    S301S15                 / GO TO STATE 15
/
S301S15 MOVN    15, R1.301              / SET STATE VARIABLE
        IFINE   PS1-1, S301S16          / UNCONTR TRANS TO STATE 16
        ADCI    IT1-1, F1.301           / READ IT-1 INTO REGISTER
        IF      F1.301, GE, 6, S301S18  / UNCONTR TRANS TO STATE 18
        ADCI    IT1-1, F1.301           / READ IT-1 INTO REGISTER
        IF      F1.301, LT, 3, S301S27  / UNCONTR TRANS TO STATE 27
        GOTO    S301S15
/
S301S16 MOVN    16, R1.301              / SET STATE VARIABLE
        DENG    P6                      / CONTRL TRANS. TO STOP P6
        WAIT    5                       / GRACE TIME
        GOTO    S301S17                 / GO TO STATE 15
```

Figure 21. PARACODE sample.

nominal levels of 6 and 20 L until an event giving rise to an abnormal situation occurs at time 373 s. At this time, an operator has inadvertently opened the lid while the pre-rinse phase is in operation. The controller responds by disabling the filling cycle. The level in tank T1 continues to fall below 6 L until a value of 3 L is achieved at which time pump P1 is deactivated to avoid dry pumping or cavitation. Once the lid is again closed at time 530 s, the system returns to normal operation.

In conclusion, the pre-rinse phase was found to work exactly as specified under normal and abnormal circumstances.

Conclusions

Historically, the control of event-driven process operations has received a different treatment from its continuous counterpart. However, this article demonstrated that the synthesis of feedback controllers for both types of operations can follow a similar design philosophy. Here, a framework has been proposed for the synthesis of feedback controllers for sequence behavior, termed procedural controllers, including the theoretical background and a method for synthesis. The proposed formal representation for the control law can be refined into implementable control code, as has been shown in this article with the generation of sequential control code for a real process. The case study showed that the method is promising for automatically generating control code relatively free of errors. However, there is a need for further work in

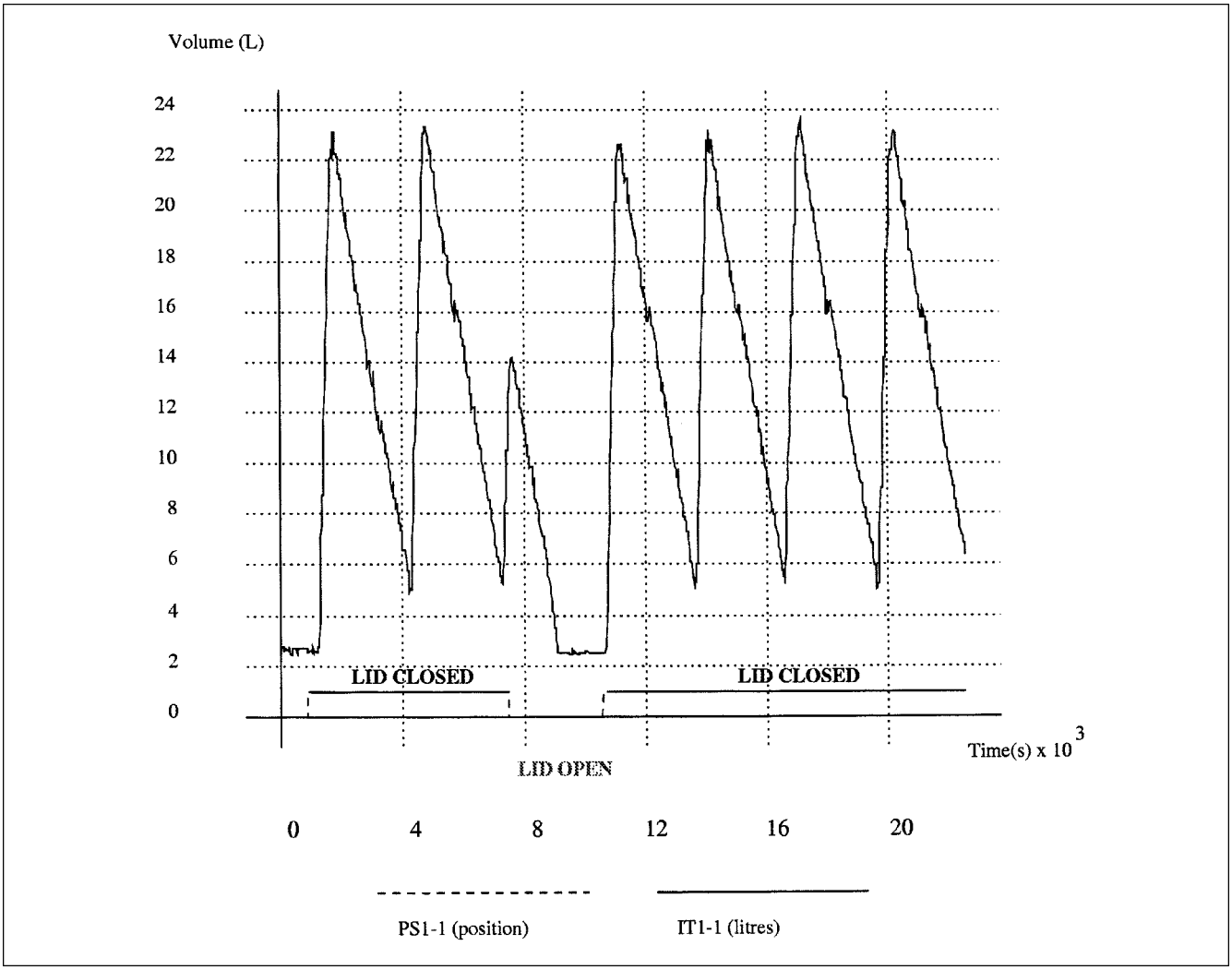

Figure 22. Experimental trace of a batch pilot-plant run.

several areas, in particular:

- *Complex Control Structures.* Issues related to hierarchical and modular control structures were identified in the case study, for instance, in the division of the operation ''phases'' into ''steps,'' or in the aggregation of valves and pipelines into ''routes.'' Means for proper modeling, analysis, and synthesis of these aspects are required.

- *Time.* All models are qualitative in time. This is practical for the generation of sequential control code used in ''pure reactive'' systems, but real-time software implementations require the explicit quantitative consideration of time.

- *Code Implementation.* The procedural controller can be considered as a formal specification of the logic for a discrete-event control system. This specification must be used in conjunction with methods and tools that guarantee a formally correct implementation.

- *Verification and Validation.* The validation of the specification and the verification of the final results must be considered. Being the synthesis based on a process model, efficient verification-validation techniques are still a need. Recent attempts have been made in chemical engineering using techniques taken from software engineering (Moon et al., 1992; Rotstein and Macchietto, 1995).

Acknowledgments

This work was supported by EPSRC. G. E. R. would like to acknowledge the financial aid provided by a Foreign and Commonwealth Office Clore Foundation Scholarship. N. A. would like to acknowledge the British Council for their funding under a Commonwealth Scholarship.

Literature Cited

Abrial, J. R., E. Borger, and H. Langmaack, ''Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control,'' *Lecture Notes in Computer Sciences*, Vol. 1165, Springer, Berlin (1996).

Alsop, N., L. Camillocci, A. Sanchez, and S. Macchietto, ''Synthesis of Procedural Controllers—Application to a Batch Plant,'' *Comput. and Chem. Eng.*, **20**(Suppl.), S1481 (1996).

APV-Baker, ''ACCOS 30 Training Manual,'' Technical Report Issue 1, Crawley, U.K. (1994).

Brave, Y., and M. Heymann, ''On Optimal Attraction in Discrete Event Processes,'' *Int. J. of Inf. Sci.*, **67**, 245 (1993).

Brooks, A., R. Cieslak, and P. Varaiya, "A Method for Specifying, Implementing and Verifying Media Access Protocols," *IEEE Control Systems Magazine*, 87 (June 1990).
Crooks, C. A., and S. Macchietto, "A Combined MILP and Logic-Based Approach for the Synthesis of Operating Procedures for Batch Plants," *Chem. Eng. Comm.*, **114**, 117 (1992).
Davey, B. A., and H. A. Priestley, *Introduction to Lattices and Order*, Cambridge University Press, Cambridge, U.K. (1990).
Dimitriadis, V. D., N. Shah, and C. C. Pantelides, "Modeling and Safety Verification of Discrete/Continuous Processing Systems," *AIChE J.*, **43**, 1041 (April 1997).
Fitzgerald, J., C. B. Jones, and P. Lucas, "FME 97. Industrial Applications and Strengthened Foundations of Formal Methods," *Lecture Notes in Computer Sciences*, Vol. 1313, Springer (1997).
Fusillo, R. H., and G. J. Powers, "A Synthesis Method for Chemical Plant Operating Procedures," *Comput. Chem. Eng.*, **11**, 369 (1987).
Golaszewski, C. H., and P. J. Ramadge, "Control of Discrete-Event Processes with Forced Events," *Proc. of 28th Conf. on Decision and Control*, Tampa, FL, p. 247 (Dec. 1987).
Goldblatt, R., *Logics of Time and Computation*, 2nd ed., Center for The Study of Language and Information (1992).
Heymann, M., "Concurrency and Discrete Event Control," *IEEE Contr. Sys. Mag.*, **10**, 103 (June 1990).
Hopcroft, J. E., and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA (1977).
ISA-S88.01, "Batch Control Systems. Models and Terminology. Standards," Technical Report, ISA , Research Triangle Park, NC (1995).
Kumar, R., V. Garg, and S. Marcus, "Using Predicate Transformers for Supervisory Control," *Proc. of 30th Conf. on Decision and Control*, Brighton, U.K., p. 98 (Dec. 1991).
Lakshmanan, R., and G. Stephanopoulos, "Synthesis of Operating Procedures for Complete Chemical Plants. I. Hierarchical Structured Modeling for Nonlinear Planning," *Comp. Chem. Eng.*, **12**, 1003 (1988).
Liu, Z. H., and S. Macchietto, "Cleaning In Place Policies for a Food Processing Batch Pilot Plant," *Food and Bioproducts Processing*, **71**(C3) (Sept. 1993).
Liu, Z. H., "An Advanced Process Manufacturing System—Design and Application to a Food Processing Pilot Plant," PhD Thesis, Univ. of London (1995).
Macchietto, S., "Automation Research on a Food Processing Pilot Plant," *IChemE Symp. Series*, No. 126, 179 (1992).
Manna, Z., and A. Pnueli, "Verification of Concurrent Programs: The Temporal Framework," *The Correctness Problem in Computer Science, International Lecture Series in Computer Science*, R. S. Boyer and J. S. Moore, eds., Academic Press, London, p. 215 (1982).
Moon, I., G. Powers, J. R. Burch, and E. M. Clarke, "Automatic Verification of Sequential Control Systems Using Temporal Logic," *AIChE J.*, **38**, 67 (Jan. 1992).
Ostroff, J. S., "Synthesis of Controllers for Real-Time Discrete Event Systems," *Proc. Conf. on Decision and Control*, Tampa, FL, p. 138 (Dec. 1989).
Ostroff, J. S., *Temporal Logic for Real-Time Systems*, Research Studies Press/Wiley (1989).
Park, T., and P. Barton, "Implicit Model Checking of Logic-Based Control Systems," *AIChE J.*, **43**, 2246 (Sept. 1997).
Ramadge, P. J., and W. M. Wonham, "Modular Feedback Logic for Discrete Event Systems," *SIAM J. of Contr. and Optimiz.*, **25**, 1202 (1987).
Ramadge, P. J., and W. M. Wonham, "Supervisory Control of a Class of Discrete−Event Processes," *SIAM J. of Contr. and Optimiz.*, **25**, 206 (1987).
Rivas, J. R., and D. F. Rudd, "Synthesis of Failure-Safe Operations," *AIChE J.*, **20**, 320 (1974).
Rotstein, G. E., and S. Macchietto, "Verification of Operating Procedures for Chemical Processes," *Proc. of IChemE Res. Event*, Edinburgh, U.K., p. 158 (1995).
Rotstein, G. E., R. Lavie, and D. R. Lewin, "Automatic Synthesis of Batch Plant Procedures. A Process-Oriented Approach," *AIChE J.*, **40**, 1650 (1994).
Rotstein, G., P. Mouille, S. Macchietto, and A. Sanchez, "Design of Stable and Optimal Procedural Controllers for Chemical Processes," *Chem. Eng. Sci.*, in press (1999).
Sanchez, A., C. Gollain, and S. Macchietto, "Rapid Development of Structured Simulation Models for the Verification of Discrete-Event Control Systems," *IFAC DYCOPS-5*, Corfu, Greece, p. 662 (June 8−10, 1998).
Sanchez, A., "Formal Specification and Synthesis of Procedural Controllers for Process Systems," *Lecture Notes on Control and Information Sciences*, Vol. 212, Springer-Verlag (1996).
Schuler, H., F. Allgower, and E. D. Gilles, "Chemical Process Control: Present Status and Future Needs. The View from the European Industry," *CPC IV*, Y. Arkun and W. H. Ray, eds., Elsevier, Amsterdam, p. 29 (1991).
Thistle, J. G., and W. M. Wonham, "Control Problems in a Temporal Logic Framework," *Int. J. Contr.*, **44**, 943 (1986).
Wonham, W. M., "A Control Theory for Discrete-Event Systems," *Advanced Computing Concepts and Techniques in Control Engineering*, M. J. Denham and A. J. Laub, eds., Springer-Verlag, Berlin, p. 129 (1988).
Yamalidou, E. C., and J. C. Kantor, "Modeling and Optimal Control of Discrete-Event Chemical Processes using Petri Nets," *Comput. and Chem. Eng.*, **15**, 503 (1991).

Appendix A: Proof of Proposition 1

Proof: Let $K = L(C/P)$ and $K = K_1 \cup K_2$ where K_1 and K_2 meet conditions 1 and 2, respectively, of Definition 5. Thus, $\forall s \in K,\ s \in K_1 \vee s \in K_2$. $\Rightarrow$ If $L(C/P)$ is controllable with respect to $L(P)$ then C is complete.

By definition of controllability, $\forall s \in K,\ s \in K_1 \vee s \in K_2$. Two alternatives arise:

(1) $s \in K_1$. Then, from Definition 5,

$$\forall \sigma_u \in \Sigma_u, \quad s\sigma_u \cap L(P) \subseteq \overline{K} = \overline{L}(C/P) = L(C/P) \quad \text{(A1)}$$

and since $L(C/P) = L(C) \cap L(P)$ one can conclude that

$$\forall \sigma_u \in \Sigma_u, \quad \text{If } \delta(s\sigma_u, q_0)! \text{ then } \gamma(s\sigma_u, x_0)! \quad \text{(A2)}$$

(2) $s \in K_2$. Again, from Definition 5,

$$\exists \sigma_c \in \Sigma_c,\ s\sigma_c \in \overline{K} = L(C/P) = L(C) \cap L(P),\ s\Sigma_u \cap \overline{K} = \emptyset$$

and then one can conclude that

$$\exists \sigma_c \in \Sigma_c, \text{ s.t. } \gamma(s\sigma_c, x_0)! \wedge \gamma(s\Sigma_u, x_0) \text{ undefined} \quad \text{(A3)}$$

Now, $\forall s \in \Sigma^*,\ \sigma_u \in \Sigma_u$ such that $\gamma(s, x_0)!$ and $\delta(s\sigma_u, q_0)!$ if $s \in K_1$ then, from Expression A2, Assertion 1 in Definition 4 is true. On the other hand, if $s \in K_2$, then from Expression A3, Assertion 2 in Definition 4 is true. One concludes that $K = L(C/P)$ is complete with respect to P.

$\Leftarrow$ If C is complete with respect to P then $K = L(C/P)$ is controllable with respect to $L(P)$.

C is complete, then $\forall s \in \Sigma^*,\ \sigma_u \in \Sigma_u$

$$\gamma(s, x_0)! \wedge \delta(s\sigma_u, q_0)! \Rightarrow \gamma(s\sigma_u, x_0)! \vee \big(\exists \sigma_c \in \Sigma_c \text{ s.t. } \gamma(s\sigma_c, x_0)! \wedge \gamma(s\Sigma_u, x_0) \text{ undefined}\big) \quad \text{(A4)}$$

Now by definition of $L(P)$ and $L(C/P)$, $\gamma(s, x_0)!$ iff $s \in L(C/P) = K$ and $\delta(s\sigma_u, q_0)!$ iff $s\sigma_u \in L(P)$. It is then possi-

ble to rewrite Expression A4 as

$$s \in K \wedge s\sigma_u \in L(P) \Rightarrow s\sigma_u \in K \vee \left(\exists \sigma_c \in \Sigma_c \text{ s.t. } s\sigma_c \in K \wedge s\Sigma_u \cap \overline{K} = \emptyset\right) \quad \text{(A5)}$$

Define now

$$K_1 = \left\{s | s \in \Sigma^* \wedge (\forall \sigma_u \in \Sigma_u,\ s\sigma_u \in K)\right\} \quad \text{(A6)}$$

$$K_2 = \left\{s | s \in \Sigma^* \wedge \left(\exists \sigma_c \in \Sigma_c \text{ s.t. } \gamma(s\sigma_c, x_0)! s\Sigma_u \cap \overline{K} = \emptyset\right)\right\} \quad \text{(A7)}$$

From Expression A5, $K = K_1 \cup K_2$. Moreover, from Expression A6, $K_1\Sigma_u \cap L(P) \subseteq K = L(C/P)$ and, therefore, K_1 satisfies Assertion 1 in Definition 5. Finally, from Expression 17, K_2 satisfies Assertion 2 in Definition 5. It is concluded then that $L(C/P)$ is controllable with respect to P.

Appendix B: Proof of Theorem 1

Proof: From Definition 5, partition each language K^i as follows

$$K^i = K_1^i \cup K_2^i \quad \text{(B1)}$$

$$K_1^i\Sigma_u \cap L \subseteq \overline{K^i} \quad \text{(B2)}$$

$$\forall s \in K_2^i,\ \exists \sigma_c \in \Sigma_c \text{ s.t. } s\sigma_c \in \overline{K^i} \wedge s\Sigma_u \cap \overline{K^i} = \emptyset \quad \text{(B3)}$$

Define $N_1 = \cup_i K_1^i$ and $N_2 = \cup_i K_2^i$. Using Expression B1 for each language K^i, $N = \cup_i K^i = \cup_i (K_1^i \cup K_2^i) = (\cup_i K_1^i) \cup (\cup_i K_2^i)$ and then

$$N = N_1 \cup N_2 \quad \text{(B4)}$$

From Expression B2 for each K^i, $\cup_i (K_1^i\Sigma_u \cap L) \subseteq \cup_i \overline{K^i}$. Distributing the intersection and knowing that N is a regular language

$$\left(\cup_i K_1^i\Sigma_u\right) \cap L \subseteq \overline{N}$$

This expression can be rewritten as follows

$$\left(\cup_i K_1^i\right)\Sigma_u \cap L = N_1\Sigma_u \cap L \subseteq \overline{N} \quad \text{(B5)}$$

Next, by definition of N_2, if $s \in N_2$ then $\exists K^i$ s.t. $s \in K_2^i$ and using expression B3

$$\forall s \in N_2, \quad \exists \sigma_c \in \Sigma_c \text{ s.t. } s\sigma_c \in \overline{N} \wedge s\Sigma_u \cap \overline{N} = \emptyset \quad \text{(B6)}$$

From Expressions B4, B5, and B6, N is controllable with respect to L.

Appendix C: Generation of Maximal Superstructure

Proposition 2: Algorithm 1 generates the maximal superstructure of procedural controllers.

Proof: The algorithm stops after at most $|X|$ iterations (that is, when all the states in M are deleted). Next, it is shown that $K = L(R)$ is controllable. Assume that K is not controllable. Then, from Definition 5, there exists $s \in K$, $\sigma_u \in \Sigma_u$ s.t.

$$s\sigma_u \in L(P) \wedge s\sigma_u \notin K \wedge \left(s\Sigma_c \cap \overline{K} = \emptyset\right) \quad \text{(C1)}$$

In other words, $s \notin (K_1 \cup K_2)$. From Expression C1, $\exists x \in X$, $s \in K$, $\sigma_u \in \Sigma_u$ such that $x = \gamma(s, x_0)$, $\delta(s\sigma_u, q_0)! \wedge \gamma(\sigma_u, x)$ is undefined. Moreover, $\nexists\, \sigma_c \in \Sigma_c$ s.t. $\gamma(\sigma_c, x)!$. The conditions to apply Step 2(b) are then satisfied (that is, state x must be deleted), which contradicts the fact that the algorithm stopped when no state to be deleted was found in the last iteration.

Next, it is shown by induction that $L(R) = L^{\uparrow}(M)$. Name R_i the FSM obtained after i iterations. Obviously, $L^{\uparrow}(R_0) = L^{\uparrow}(R) = L(R)$ (R is controllable and $R_0 = R$). Now, assume $L^{\uparrow}(R_i) = L(R)$ and prove that $L^{\uparrow}(R_{i-1}) = L(R)$. Two alternatives arise, the first is that R_i was obtained applying Step 2(a) of Algorithm 1

$$L(R_{i-1}) = L(R_i) \cup_j s\sigma_u^j | \left\{s \in L(R_i),\ \sigma_u^j \in \Sigma_u,\ s\sigma_u^j \in L(M),\ \left[\exists \sigma_u \in \Sigma_u \text{ s.t. } s\sigma_u \in L(P) \wedge s\sigma_u \notin L(R_{i-1})\right]\right\} \quad \text{(C2)}$$

Obviously, the string s as defined in $L(R_{i-1})$ is not controllable. It does not satisfy either condition 1 in Definition 5 or Condition 2 [a nonempty subset of the uncontrollable transitions that can extend s in $L(P)$ can extend s in $L(R_{i+1})$]. One concludes that $\forall \sigma_u \in \Sigma_u$, $s\sigma_u \notin L^{\uparrow}(R_{i-1})$. Since these strings are the only difference between $L(R_i)$ and $L(R_{i-1})$, $L^{\uparrow}(R_{i-1}) = L^{\uparrow}(R_i) = L(R)$ (from the induction assumption). The second alternative is that R_i was obtained applying Step 2b then

$$L(R_{i-1}) = L(R_i) \cup_{j,k} s_j\sigma_u^k | \left\{s_j \in L(R_i),\ \sigma_u^k \in \Sigma_u,\ s_j\sigma_u^k \in L(M),\ \left[\exists \sigma_u \in \Sigma_u \text{ s.t. } s_j\sigma_u \in L(P),\ s_j\sigma_u \notin L(R_{i-1})\right] \wedge \left[\nexists\, \sigma_c \in \Sigma_c,\ s_j\sigma_c \in L(R_{i-1})\right]\right\} \quad \text{(C3)}$$

and the same line of reasoning as before can be followed to show that $L(R_{i-1})^{\uparrow} = L(R)$.

Appendix D: Inductive Proof of Theorem 2

Proof: The first step is to prove by induction that the C obtained is nonblocking. Initially, $C_0 = M$ is obviously nonblocking. Assume that at iteration n, C_n is nonblocking. Then, let x be the state selected at iteration $n+1$. At this iteration, only the definition of γ at x changes, so C_{n+1} will not be nonblocking only if there is no string from x to Q_m in $L(C/P)$. There are three possibilities:

(1) γ_{n+1} at x is defined following 2(b). Then, in C_{n+1}, any $\sigma_u \in \Sigma_u$, s.t. $\gamma_{n+1}(\sigma_u, x)!$ can be extended to a string reaching Q_m.

(2) γ_{n+1} is defined following 2(c). C_n is assumed to be nonblocking and then any transition exiting x can be extended to a string reaching Q_m. Since conditions in step 2(b)

are not satisfied, deleting from C_n all the uncontrollable transitions defined at x blocks at least one string $s \in \Sigma^*$ s.t. $\gamma_n(\sigma_u, s, x) \in Q_m$. Thus, $\exists s'$, $s'' \in \Sigma^*$ s.t. $\sigma_u s = \sigma_u s' \sigma_c s''$ $[\gamma(\sigma_u s', x) = x]$ and since after 2(c) $\gamma_{n+1}(\sigma_c, x)!$, C_{n+1} is nonblocking.

(3) γ_{n+1} is defined following 4(d). $x \in X_c$ and then $\gamma_n(\Sigma_u, x)$ is undefined. Since C_n is nonblocking, $\exists \sigma_c \in \Sigma_c$, $s \in \Sigma^*$ such that $\gamma_n(\sigma_c s, x) \in Q_m$. Moreover, at least one of these strings $\sigma_c s$ must be such that if $s' \in \overline{\sigma_c s}$ then $\gamma_n(s', x) \neq x$. This σ_c will eventually be selected in 4(d) [that is, $\gamma_{n+1}(\sigma_c, x) = \gamma_n(\sigma_c, x)$] and C_{n+1} will then be nonblocking.

Now it is shown that C is a procedural controller. All the states that do not satisfy Definition 3 are in $X_u \cup X_c$. Once step 2 is finished, from 2(a)(b), $\forall x \in X_u$ (and by definition of X_u also $\forall x \in X$) $\gamma(\Sigma_c, x)$ is undefined or $\gamma(\Sigma_u, x)$ is undefined. The C obtained would not be a controller only if there is a state x such that more than one controllable transition is enabled at it (that is, $x \in X_c$). However, it has been shown above that $\forall x \in X_c$, γ will be redefined at step 4(d) enabling only one controllable transition. Therefore, C satisfies Definition 3.

It only remains to be proved that C is complete with respect to M. A state $x \in C$ can make C incomplete only if $\exists \sigma_u \in \Sigma_u$ s.t. $\delta(\sigma_u, x)!$. If $x \in X - X_u$ (and note that by definition $x \notin X_c$), then the procedure does not affect the definition of γ and since $\forall \sigma \in \Sigma$, $\gamma(\sigma, x) = \delta(\sigma, x)$, C is complete in x. Otherwise, if $x \in X_u$, then either step 2(a) is applied, and $\forall \sigma_u \in \Sigma_u$, $\gamma(\sigma_u, x) = \delta(\sigma_u, x)$ (that is, C is complete in x) or step 2(b) is applied. In the last case, two possibilities arise. The first is that $x \in X_c$ and then γ for these states will be redefined in step 4(b). The second is that $x \notin X_c$. In both cases, $\exists \sigma_c \in \Sigma_c$ s.t. $\gamma(\sigma_c, x) = \delta(\sigma_c, x)$ and again C is complete in x.

Appendix E: Static Specifications (Forbidden States) for the Case Study

(1) If the feed route is not set, then P6 cannot be energized

$$(0, 1, \infty, \infty, \infty, \infty, \infty, \infty) = \text{FALSE}$$

(2) If the feed route is set and P6 is energized, then SSV1-3 cannot be closed

$$(1, 1, 0, \infty, \infty, \infty, \infty, \infty) = \text{FALSE}$$

(3) All items should remain in their relaxed state when the feed route is closed

$$(0, 0, 1, \infty, \infty, \infty, \infty, \infty) = \text{FALSE}$$

$$(0, 0, 0, \infty, \infty, 1, \infty, \infty) = \text{FALSE}$$

$$(0, 0, 0, \infty, \infty, 0, 1, \infty) = \text{FALSE}$$

$$(0, 0, 0, \infty, \infty, 0, 0, 1) = \text{FALSE}$$

(4) If AV1-41 is closed, then P1 must remain in its deenergized state

$$(1, \infty, \infty, \infty, \infty, 0, 1, \infty) = \text{FALSE}$$

(5) The return route must never be closed when P1 is energized

$$(1, \infty, \infty, \infty, \infty, 1, 1, 0) = \text{FALSE}$$

Appendix F: Dynamic Specifications for the Case Study

Normal behavior

(1) If the lid is open, then do not leave the initial state

$$(0, \infty, \infty, 0, \infty, \infty, \infty, \infty) \rightarrow \bigcirc[\tau \neq 43]$$

(2) If IT1-1 is above 3 L, then do not leave the initial state

$$(0, \infty, \infty, 1, \infty^0, \infty, \infty, \infty) \rightarrow \bigcirc[\tau \neq 43]$$

(3) If the lid is shut and IT1-1 is less than 3 L, set the feed and return routes, open SSV1-3 and energize P6

$$(0, 0, 0, 1, 0, 0, 0, 0) \rightarrow \bigcirc[\tau = 43] \rightarrow \bigcirc[\tau = 29] \rightarrow \bigcirc[\tau = 1] \rightarrow \bigcirc[\tau = 17]$$

(4) Open AV1-41 and energize P1 once the feed and return routes are set and IT1-1 exceeds 3 L:

$$(1, \infty, \infty, \infty, 0, 0, 0, 1) \wedge (\tau = 535) \rightarrow \bigcirc[\tau = 33] \rightarrow \bigcirc[\tau = 7]$$

(5) Once the feed and return routes are set, they must always remain set while the controller is in operation

$$(1, \infty, \infty, \infty, \infty, \infty, \infty, 1) \rightarrow \Box(1, \infty, \infty, \infty, \infty, \infty, \infty, 1)$$

(6) If IT1-1 is less than 3 L, do not open AV1-41 or energize P1

$$(1, \infty, \infty, \infty, 0, \infty, \infty, 1) \rightarrow \bigcirc[(\tau \neq 33) \wedge (\tau \neq 7)]$$

(7) If IT1-1 exceeds 20 L, then deenergize P6 and close SSV1-3

$$(1, 1, 1, 1, 3, 1, 1, 1) \rightarrow \bigcirc[\tau = 18] \rightarrow \bigcirc[\tau = 2]$$

(8) If IT1-1 is below 20 L and the lid is shut, then keep SSV1-3 open and P6 energized

$$(1, \infty, \infty, 1, \infty^3, \infty, \infty, 1) \rightarrow \bigcirc[(\tau \neq 18) \wedge (\tau \neq 2)]$$

(9) If IT1-1 is above 3 L, then keep AV1-41 open and P1 energized

$$(1, \infty, \infty, \infty, \infty^0, \infty, \infty, 1) \rightarrow \bigcirc[(\tau \neq 8) \wedge (\tau \neq 34)]$$

(10) If the lid is shut, SSV1-3 is closed and P6 deenergized, do not reopen SSV1-3 if IT1-1 is above 6 L

$$(1, 0, 0, 1, \infty^{0,1}, 1, 1, 1) \rightarrow \bigcirc[\tau \neq 1]$$

(11) Reopen SSV1-3 and reenergize P6 when IT1-1 reaches a level between 3 and 6 L

$$(1, 0, 0, 1, 1, 1, 1, 1) \rightarrow \bigcirc[\tau = 1] \rightarrow \bigcirc[\tau = 17]$$

Abnormal behavior

(12) If the lid opens while P6 is energized, then immediately deenergize P6 and close SSV1-3

$$(1, 1, 1, 1, \infty, \infty, \infty, 1) \wedge (\tau = 532) \rightarrow \bigcirc[\tau = 18] \rightarrow \bigcirc[\tau = 2]$$

(13) If the lid is open, do not open SSV1-3 or energize P6

$$(1, \infty, \infty, 0, \infty, \infty, \infty, 1) \rightarrow \bigcirc[(\tau \neq 1) \wedge (\tau \neq 17)]$$

(14) If IT1-1 falls below 3 L, then immediately deenergize P1 and close AV1-41

$$(1, \infty, \infty, \infty, 0, 1, 1, 1) \rightarrow \bigcirc[\tau = 8] \rightarrow \bigcirc[\tau = 34]$$

Manuscript received Apr. 21, 1997, and revision received Apr. 28, 1999.