Modular Control-Loop Detection

Petra Malik Robi Malik
Department of Computer Science, University of Waikato
Hamilton, New Zealand
{petra,robi}@cs.waikato.ac.nz

***Abstract*— This paper presents an efficient algorithm to detect control-loops in large finite-state systems. The proposed algorithm exploits the modular structure present in many models of practical relevance, and often successfully avoids the explicit synchronous composition of subsystems and thereby the state explosion problem. Experimental results show that the method can be used to verify industrial applications of considerable complexity.**

I. Introduction

Supervisory control theory is centred on the setting of a *supervisor* or *controller* interacting with a *plant*, which typically represents a technical system to be controlled. The theory provides fundamental results about the existence of supervisors, and algorithms that automatically synthesise supervisors satisfying certain properties [1], [2].

Ever more controllers are implemented in software, either running on dedicated programmable logic circuits (PLC) or on multi-purpose computers, so it is highly desirable to construct control software from models that have been obtained using supervisory control theory. However, traditional supervisory control is situated on a higher level of abstraction and does not provide all the details needed to actually implement a controller for physical devices. It has been recognised by several researchers [3]–[5] that, in order to implement a controller, its behaviour needs to satisfy additional properties that are not covered by traditional supervisory control theory.

One of these properties is the absence of *control-loops* [6], [7]. In many technical applications, a set of *control actions* can be identified. These correspond to commands generated by the controller and sent to the physical device or plant to achieve the desired behaviour. A system is *control-loop free*, if the controller never tries to generate an infinite sequence of control actions in response to any finite input from the plant. Clearly, this is a very desirable property of most controllers implemented in software.

This paper proposes an efficient algorithm to check whether a complex system is control-loop free. The proposed algorithm, which is based on results from [7], is an adaptation of a very efficient algorithm [8] to check safety properties of discrete event systems. The method exploits the modular structure of discrete event system models to identify and check subsystems, such that the results of checking the subsystems can give conclusions about the properties of the entire system. Counterexamples are used for guidance, to augment these subsystems if needed.

Existing model checking techniques for CTL [9] can also be used to check whether a system is control-loop free, and, using symbolic representations [10], have been used successfully to verify models of considerable complexity. Abstractions [11] can enhance the performance of these algorithms. This paper proposes an alternative approach that exploits the modular structure present in most discrete event models, which can be combined with and further enhance the existing methods.

In the following, section II presents a motivating example to clarify the need for control-loop detection. Section III gives the notations and definitions used throughout the rest of the paper. The problem of checking for control-loops is defined in section IV, where a basic algorithm is given. Then, section V presents the results underlying the modular control-loop detection algorithm that is the centre of this paper. Section VI describes the algorithm, and section VII lists experimental results. Finally, section VIII adds some concluding remarks.

II. Dosing Tank Example

This section illustrates the idea of checking for control-loops using the example of a dosing unit in a chemical batch plant, used to supply a defined amount of liquid material to a subsequent process. It consists of a tank, an inlet valve, an outlet valve, and two sensors to check the filling level of the tank. The following is a simplified version of a system introduced in [12].

The two sensors, **S1** at the bottom and **S2** at the top of the tank, can either be on or off, indicating whether the tank has reached the corresponding filling level. Their state changes are modelled as events *s1_on*, *s1_off*, *s2_on*, and *s2_off*. The user can request the process of filling and emptying the tank to be started (*req_start*), or suspended (*req_stop*). To meet these requests, the controller has to open or close the inlet and the outlet valves appropriately, using control actions open_in, close_in, open_out, and close_out.

The possible behaviour of this plant is modelled by the automata **sensors**, **requests**, **in**, and **out** in fig. 1. A controller has been designed using additional automata to constrain the behaviour of the plant and satisfy certain requirements. Automaton **no_flow** ensures that liquid never flows through the tank, i.e., that the two valves are never open at the same time. Automaton **req_spec** guarantees that discharging or filling of the tank can only start when a request is present, i.e., after *req_start* has occurred. Finally, to provide

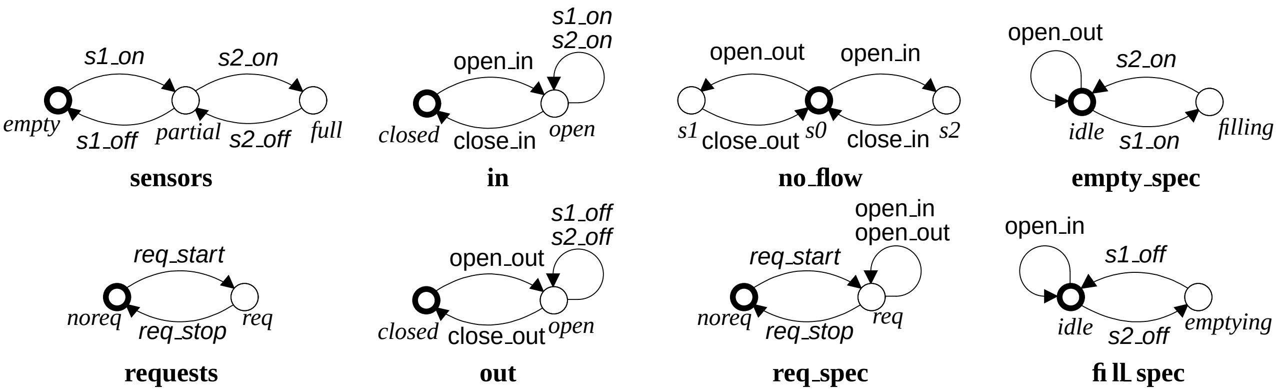

Fig. 1. Automata for dosing tank example.

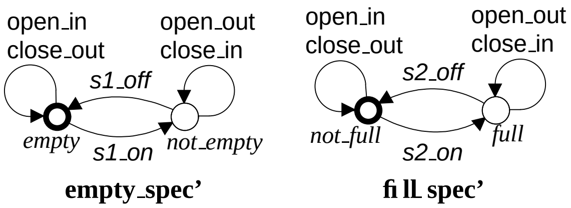

Fig. 2. Additional specifications for dosing tank example.

the correct amount of liquid to the subsequent process, the tank must always be completely filled, i.e., until sensor **S2** goes on, before discharging can start (**empty_spec**), and completely emptied before filling can start (**fill_spec**).

The question is whether a controller can be constructed that opens and closes the valves appropriately to yield the behaviour of the dosing unit as obtained when synchronising all the automata in fig. 1. A check for control-loops shows that this is not feasible. The automata can execute the trace

$$\textit{req\_start}\ \text{open\_in close\_in open\_in close\_in} \ldots \tag{1}$$

where the inlet valve is opened and closed indefinitely. While such behaviour does not violate the requirements given so far, it is very undesirable when executed by an actual controller.

To avoid this problem, additional specifications as shown in fig. 2 can be used to constrain the model further: the inlet valve can only be opened if the tank is empty and closed if the tank is full, and the outlet valve can only be opened if the tank is full and closed if the tank is empty. With the additional two automata, the model can be proven to be control-loop free.

III. Notation and Preliminaries

Event sequences and languages are a simple means to describe discrete system behaviours. Their basic building blocks are *events*, which are taken from a finite *alphabet* Σ. Then, Σ^+ denotes the set of all finite *strings* of the form $\sigma_1\sigma_2\cdots\sigma_k$ of events from Σ, not including the *empty string* ε. To include it, $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$ is used. The *catenation* of two strings $s, t \in \Sigma^*$ is written as st. A string $t \in \Sigma^*$ is called a *prefix* of $s \in \Sigma^*$, written $t \sqsubseteq s$, if $s = tu$ for some $u \in \Sigma^*$.

A *language* over Σ is any subset $\mathcal{L} \subseteq \Sigma^*$. The *prefix-closure* $\overline{\mathcal{L}}$ of $\mathcal{L} \subseteq \Sigma^*$ is the set of all prefixes of strings in $\mathcal{L}$,

$$\overline{\mathcal{L}} \;=\; \{\, t \in \Sigma^* \mid t \sqsubseteq s \text{ for some } s \in \mathcal{L} \,\} \;. \tag{2}$$

If $\mathcal{L} = \overline{\mathcal{L}}$, then $\mathcal{L}$ is called *prefix-closed*.

Automata are used as a simple means of language representation. A *(finite-state) automaton* is a 4-tuple $A = (\Sigma, Q, \Delta, J)$ where Σ is an alphabet of *events*, Q is a finite set of *states*, $\Delta \subseteq Q \times \Sigma \times Q$ is the *state transition relation*, and $J \subseteq Q$ is the set of *initial states*.

A *path* in A is an alternating sequence of states and events

$$\pi \;=\; q_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} q_n \;\;, \tag{3}$$

where $(q_i, \sigma_{i+1}, q_{i+1}) \in \Delta$ for all $0 \le i < n$. State q_0 is called its *origin* and state q_n its *end*. $\mathcal{L}(\pi) = \sigma_1 \cdots \sigma_n$ is the *label* of path π. A state $q \in Q$ is *reachable* in A if there exists a path with origin $p \in J$ and end q. A path $q \xrightarrow{\sigma} q$ is called a *selfloop*, and if it exists, event σ is said to be *selflooped* in q. To simplify the graphical representation of automata (fig. 1), events selflooped in all states of an automaton are not shown.

The language accepted by the automaton A is

$$\mathcal{L}(A) \;=\; \{\, \mathcal{L}(\pi) \mid \pi \text{ is a path in } A \text{ with origin in } J \,\} \;. \tag{4}$$

Two automata over the same alphabet can be combined to a new automaton by means of the *synchronous product* operation [13]. The synchronous product of automata $A = (\Sigma, Q_A, \Delta_A, J_A)$ and $B = (\Sigma, Q_B, \Delta_B, J_B)$ is the automaton

$$A \parallel B \;=\; (\Sigma, Q_A \times Q_B, \Delta, J_A \times J_B) \;, \tag{5}$$

where

$$\begin{aligned} \Delta \;=\; \{\, & ((p_1, q_1), \sigma, (p_2, q_2)) \mid \\ & (p_1, \sigma, p_2) \in \Delta_A,\; (q_1, \sigma, q_2) \in \Delta_B \,\} \;. \end{aligned} \tag{6}$$

Note that $\mathcal{L}(A \parallel B) = \mathcal{L}(A) \cap \mathcal{L}(B)$ [1].

IV. Termination and Loops

The concept of control-loops relies on a set of *control actions*, represented by a subset $\Xi \subseteq \Sigma$ of the event alphabet. Control actions are understood to be commands caused by a controller, and it is usually desired that the sequence of such commands terminates after a finite number of steps.

This requirement is now introduced under the name of Ξ-*termination*, and it is shown that it corresponds to the absence of Ξ-*loops* in an automaton. The latter problem can be dealt with using standard graph-theoretic algorithms.

Definition 1: Let $\mathcal{L} \subseteq \Sigma^*$ be a prefix-closed language, and let $\Xi \subseteq \Sigma$. Then $\mathcal{L}$ is Ξ-*terminating* if for all $s \in \mathcal{L}$ there exists $n \in \mathbb{N}$ such that $st \in \mathcal{L}$ with $|t| \geq n$ implies $t \notin \Xi^*$.

For a Ξ-terminating language, there is no infinite sequence consisting of events contained in Ξ only. Then, assuming that only events contained in Ξ occur, the system will eventually stabilise, i.e., it will reach a state in which only events not contained in Ξ are possible.

Definition 2: Let $A = (\Sigma, Q, \Delta, J)$ be an automaton, and let $\Xi \subseteq \Sigma$. A path π in A is called a Ξ-*path* if $\mathcal{L}(\pi) \in \Xi^*$. A Ξ-*loop* is a Ξ-path of length at least one with the same origin and end. A Ξ-loop is *reachable* in A if its origin is reachable in A. If A does not contain any reachable Ξ-loop, A is called Ξ-*loop free*.

Proposition 1: Let $A = (\Sigma, Q, \Delta, J)$ be an automaton, and let $\Xi \subseteq \Sigma$. A is Ξ-loop free if and only if $\mathcal{L}(A)$ is Ξ-terminating.

Proof: *(Only If)* Assume $\mathcal{L}(A)$ is not Ξ-terminating. Then there exists $s \in \mathcal{L}(A)$ such that, for each $n \in \mathbb{N}$, there exists $t \in \Xi^*$ such that $|t| \geq n$ and $st \in \mathcal{L}(A)$. In particular, there exists $t \in \Xi^*$ such that $|t| > |Q|$ and $st \in \mathcal{L}(A)$. Since $st \in \mathcal{L}(A)$, there exists a path $q_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_n} q_n$ with $q_0 \in J$ and $\sigma_1 \cdots \sigma_n = st$. Since $|t| > |Q|$ there exists $|s| \leq i < k \leq n$ such that $q_i = q_k$. Therefore the subpath $q_i \xrightarrow{\sigma_{i+1}} \cdots \xrightarrow{\sigma_k} q_k$ is a reachable Ξ-loop in A.

(If) Assume A is not Ξ-loop free. Let π be a reachable Ξ-loop in A with origin $q \in Q$. Since π is reachable, there exists a path π' in A with end q. Let $s = \mathcal{L}(\pi')$ and $t = \mathcal{L}(\pi)$. Note that $|t| \geq 1$. Then for each $k \in \mathbb{N}$ it holds that $st^k \in \mathcal{L}(A)$, $t^k \in \Xi^*$, and $|t^k| \geq k$. This shows that $\mathcal{L}(A)$ is not Ξ-terminating. ■

Ξ-loops can be identified by finding *strongly* Ξ-*connected components* and selfloops labelled with events contained in Ξ. In order to define the concept of strongly Ξ-connected components, a relation over states is defined: two states are strongly Ξ-connected if they can mutually be reached by means of events contained in Ξ.

Definition 3: Let $A = (\Sigma, Q, \Delta, J)$ be an automaton, and let $\Xi \subseteq \Sigma$. Two states $p, q \in Q$ are *strongly* Ξ-*connected* in A, written $p \leftrightarrow_\Xi q$, if there exist a Ξ-path with origin p and end q and another Ξ-path with origin q and end p.

It can easily be shown that this relation is an equivalence relation. Its equivalence classes are called strongly Ξ-connected components. Such an equivalence class contains states where each state can be reached by means of Ξ-paths from all other states in the equivalence class.

Definition 4: Let $A = (\Sigma, Q, \Delta, J)$ be an automaton, and let $\Xi \subseteq \Sigma$. The equivalence classes of $\leftrightarrow_\Xi$ are called the *strongly* Ξ-*connected components* of A. A strongly Ξ-connected component is said to be *reachable in* A, if it contains a state that is reachable in A.

The next result states that, if an automaton has a strongly Ξ-connected component with more than one state, then there exists a Ξ-loop in the automaton.

Proposition 2: Let $A = (\Sigma, Q, \Delta, J)$ be an automaton, and let $\Xi \subseteq \Sigma$. If there exist $p, q \in Q$ such that $p \leftrightarrow_\Xi q$ and $p \neq q$, then there exists a Ξ-loop in A.

Proof: Since $p \leftrightarrow_\Xi q$, there exist two Ξ-paths, one with origin p and end q and another with origin q and end p. By concatenating these two paths, a Ξ-loop in A is obtained. ■

If a strongly Ξ-connected component is reachable, then each of its states is reachable. Therefore, the loop constructed in proposition 2 is also reachable. Thus, if there exists a reachable strongly Ξ-connected component in an automaton, then the automaton cannot be Ξ-loop free.

Iterative Ξ-*pairs* are counterexamples, which can be provided when an automaton is not Ξ-loop free. An iterative Ξ-pair consists of two strings, where the second string can be iterated while staying within the behaviour of the system.

Definition 5: Let $A = (\Sigma, Q, \Delta, J)$ be an automaton, and let $\Xi \subseteq \Sigma$. The pair $(s, t) \in \Sigma^* \times \Xi^+$ is an *iterative* Ξ-*pair* in A if $st^n \in \mathcal{L}(A)$ for all $n \in \mathbb{N}$.

If there exists a reachable Ξ-loop π in an automaton, then there exists an iterative Ξ-pair in this automaton. This can be seen as follows. Since π is reachable, there exists a path π' to the origin of π. Then $(\mathcal{L}(\pi'), \mathcal{L}(\pi))$ is an iterative Ξ-pair in the automaton. The first string shows how a loop can be reached. The second string describes the loop. This provides illustrative information why a system is not Ξ-loop free.

The results given so far suggest an algorithm to check an automaton for Ξ-loops. Given an automaton $A = (\Sigma, Q, \Delta, J)$ and a set of control actions $\Xi \subseteq \Sigma$, there are only two possible ways how a Ξ-loop can exist.

(i) There is a reachable state $q \in Q$ and a control action $\xi \in \Xi$ such that $(q, \xi, q) \in \Delta$. Then $q \xrightarrow{\xi} q$ is a reachable Ξ-loop in A.

(ii) There exists a reachable strongly Ξ-connected component of A consisting of at least two states $p, q \in Q$. Then there exist Ξ-paths from p to q and from q to p, i.e., A contains a reachable Ξ-loop.

If neither of these two conditions is satisfied, then A is Ξ-loop free. This can be shown as follows. If A contains a reachable Ξ-loop, it is either a selfloop, and thus condition (i) is satisfied, or the loop contains at least two different states. Then these states are in the same strongly Ξ-connected component and condition (ii) is satisfied. Therefore, A cannot have a reachable Ξ-loop.

Thus, to check an automaton for Ξ loops, it suffices to check for the existence of selfloops and for the existence of reachable strongly Ξ-connected components with more than one state. Efficient algorithms [14], [15] can be used to find strongly connected components in a directed graph in linear complexity (with respect to the number of nodes and edges).

V. Exploiting Modularity

The results of the previous section provide good algorithms to check whether a single automaton is Ξ-loop free. However, practical systems are composed of several automata, and then their synchronous composition needs to be computed first, which can quickly lead to intractably large automata. This section shows that synchronous composition can often be avoided by exploiting the modular structure of the system.

The most important result is that it suffices to find a Ξ-loop free subsystem to show that the whole system is Ξ-loop free. Therefore, if a Ξ-loop free subsystem can be found, there is no more need to compute any larger synchronous product.

Proposition 3: Let A_1 and A_2 be automata over the alphabet Σ, and let $\Xi \subseteq \Sigma$. If A_1 is Ξ-loop free then $A_1 \parallel A_2$ is also Ξ-loop free.

Proof: Assume that $A_1 \parallel A_2$ is not Ξ-loop free. Then $\mathcal{L}(A_1 \parallel A_2)$ is not Ξ-terminating, that is, there exists $s \in \mathcal{L}(A_1 \parallel A_2) \subseteq \mathcal{L}(A_1)$ such that for all $n \in \mathbb{N}$ there exists $t_n \in \Xi^*$ such that $|t_n| \geq n$ and $st_n \in \mathcal{L}(A_1 \parallel A_2) \subseteq \mathcal{L}(A_1)$. But then $\mathcal{L}(A_1)$ is not Ξ-terminating. This proves that A_1 is not Ξ-loop free if $A_1 \parallel A_2$ is not Ξ-loop free. ■

It is also possible to prove that a system is *not* Ξ-loop free without constructing its entire state space. Assume a counterexample, i.e., an iterative Ξ-pair has been found for a subsystem. If this counterexample is accepted by all automata constituting the whole system, then it is a counterexample for the entire system, i.e., the system is not Ξ-loop free.

Proposition 4: Let A_1 and A_2 be automata over the alphabet Σ, and let $\Xi \subseteq \Sigma$. Every iterative Ξ-pair in both A_1 and A_2 also is an iterative Ξ-pair in $A_1 \parallel A_2$.

Proof: Let $A = A_1 \parallel A_2$, and let (s,t) be an iterative Ξ-pair in A_1 and A_2. Then $t \in \Xi^+$, $st^n \in \mathcal{L}(A_1)$, and $st^n \in \mathcal{L}(A_2)$ for each $n \in \mathbb{N}$. This implies $st^n \in \mathcal{L}(A_1) \cap \mathcal{L}(A_2) = \mathcal{L}(A)$. Thus, (s,t) is an iterative Ξ-pair in A. ■

Propositions 3 and 4 can be combined to a strategy to construct subsystems iteratively. Assume a system

$$A \;=\; A_1 \parallel \cdots \parallel A_n \tag{7}$$

is to be checked for Ξ-loops, and some subsystem A_i of A has already been identified. If A_i is Ξ-loop free, then the entire system A is known to be Ξ-loop free by proposition 3. If A_i is not Ξ-loop free, there exists an iterative Ξ-pair demonstrating that A_i is not Ξ-loop free. If this also is an iterative Ξ-pair in all other components $A_1, \ldots, A_n$ of A, then the entire system A is not Ξ-loop free by proposition 4. Otherwise, there is an automaton A_j that does not accept the iterative Ξ-pair, and this automaton is a good candidate to augment the subsystem A_i and consider $A_i \parallel A_j$ in a next step.

Example 1: Consider the automata in fig. 1, with the set of control events

$$\Xi \;=\; \{\mathsf{open\_in}, \mathsf{close\_in}, \mathsf{open\_out}, \mathsf{close\_out}\}\ . \tag{8}$$

The subsystem **req_spec** is not Ξ-loop free as shown by the iterative Ξ-pair $(\textsf{\textit{req\_start}}, \mathsf{open\_in})$. This is not an iterative Ξ-pair in automaton **no_flow**, so proposition 4 cannot yet be applied. Therefore, the larger subsystem **req_spec** $\parallel$ **no_flow** is considered. This yields the new iterative Ξ-pair $(\textsf{\textit{req\_start}}, \mathsf{open\_in}\,\mathsf{close\_in})$, which is accepted by all other automata and therefore is a counterexample for the model.

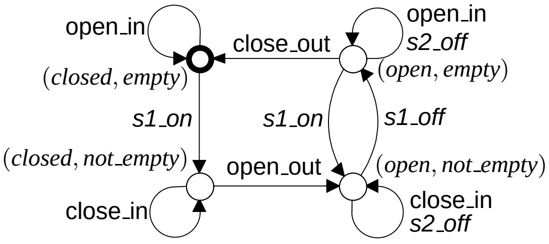

Fig. 3. The synchronous composition **out** $\parallel$ **empty_spec’**.

This method can be improved further. The set Ξ of control events can be reduced by removing events that are known not to be contained in any Ξ-loop of the system. Consider an event $\xi \in \Xi$ and a subsystem such that ξ is not contained in any Ξ-loop of this subsystem. Then the event ξ cannot occur in a Ξ-loop of the entire system since each Ξ-loop in the synchronous product must also be a Ξ-loop in each of the automata constituting the synchronous product. Therefore, it suffices to check whether the system is $(\Xi \setminus \{\xi\})$-loop free, which is simpler than checking whether it is Ξ-loop free.

Proposition 5: Let A_1 and A_2 be automata over the alphabet Σ, and let $\Xi \subseteq \Sigma$. Furthermore, let $\xi \in \Xi$ such that $\xi \notin \mathcal{L}(\pi)$ for each reachable Ξ-loop π of A_1. Then $A_1 \parallel A_2$ is Ξ-loop free if and only if $A_1 \parallel A_2$ is $(\Xi \setminus \{\xi\})$-loop free.

Proof: Let $A = A_1 \parallel A_2$ and $\Xi' = \Xi \setminus \{\xi\}$.

(Only If) Assume that A is not Ξ'-loop free. Then there exists a reachable Ξ'-loop in A. This is also a reachable Ξ-loop in A, i.e., A is not Ξ-loop free.

(If) Assume that A is not Ξ-loop free, i.e., there exists a reachable Ξ-loop

$$\pi \;=\; (p_0, q_0) \xrightarrow{\xi_1} \cdots \xrightarrow{\xi_n} (p_n, q_n) \tag{9}$$

in $A = A_1 \parallel A_2$. Then

$$\pi_1 \;=\; p_0 \xrightarrow{\xi_1} \cdots \xrightarrow{\xi_n} p_n \tag{10}$$

is a reachable Ξ-loop in A_1. By assumption, $\xi \notin \mathcal{L}(\pi_1) = \mathcal{L}(\pi)$, i.e., $\mathcal{L}(\pi) \in (\Xi')^*$. Hence, A is not Ξ'-loop free. ■

Example 2: Consider the extended model of the introductory example given by the automata in fig. 1 and fig. 2. The system can be shown to be Ξ-loop free as follows. Consider the synchronous product of automata **out** and **empty_spec’**, shown in fig. 3. This automaton contains Ξ-loops, but none of them contains the events $\mathsf{open\_out}$ or $\mathsf{close\_out}$. By proposition 5, these two events can be removed from Ξ, i.e., it is sufficient to check for $\{\mathsf{open\_in}, \mathsf{close\_in}\}$-loops. The same argument can be applied to the remaining two events when considering the synchronous product of automata **in** and **fill spec’**. Thereby Ξ is reduced to the empty set, already proving that the entire system is control-loop free.

VI. A Modular Algorithm

The results from the previous section can be combined in different ways to obtain algorithms to find control loops in modular systems. The method proposed in this paper uses ideas from [8]. Given a modular system

$$A \;=\; A_1 \parallel \cdots \parallel A_n \;, \tag{11}$$

to be checked, it tries to construct a Ξ-loop free subsystem of A incrementally, including more automata as needed.

But first, all automata A_i are checked for Ξ-loops individually. Typically, none of them is Ξ-loop free by themselves because of implicit selfloops. Yet, by proposition 5, this first pass can identify events that cannot occur in any Ξ-loop and thereby considerably reduce the set Ξ of control actions.

Then the construction of subsystems of A begins. Starting from the empty subsystem, which accepts Σ^* and therefore is never loop-free, new automata are added until a result is obtained. Assume a subsystem

$$A' \;=\; A_{i_1} \parallel \cdots \parallel A_{i_m} \tag{12}$$

has been constructed. If A' is Ξ-loop free, then the entire system A is Ξ-loop free by proposition 3. Otherwise an iterative Ξ-pair can be constructed. If this counterexample is an iterative Ξ-pair for all automata in A, then the entire system A has a Ξ-loop by proposition 4. Otherwise, some automaton does not accept the counterexample. The algorithm chooses one of the automata A_j not accepting the counterexample, and analyses the larger subsystem

$$A'' \;=\; A' \parallel A_j \tag{13}$$

in the next step. In addition, the set of control events occurring in all Ξ-loops of A' is identified to apply proposition 5 and replace Ξ by a smaller set when analysing A''.

Given a counterexample, the above algorithm looks for automata not accepting the particular counterexample to include one or more of these automata in the next analysis attempt. Performance may depend on which automaton is chosen. A number of heuristics for selecting automata not accepting a counterexample of interest have been proposed in [8] and are listed below. They have all been implemented and tested on the industrial examples of section VII.

- All. Use all automata not accepting the counterexample.
- EarlyNotAccept. Use the automaton rejecting the counterexample as early as possible, i.e., the automaton accepting as little as possible of the counterexample.
- LateNotAccept. Use the automaton rejecting the counterexample as late as possible.
- MaxCommonEvents. Use the automaton with the most events in common with the system considered so far. An automaton sharing many events with the system considered is likely to interact more closely with it, and is therefore more likely to contribute to the analysis.
- MaxCommonUncontr. Use the automaton with the most non-control events in common with the system considered so far. This is similar to MaxCommonEvents, but more tuned to the control-loop check.
- MinEvents. Use the automaton with the fewest events, in an attempt to use the simplest automata and construct the smallest possible synchronous product.
- MinNewEvents. Use the automaton adding the fewest events to the system considered. This is similar to MaxCommonEvents, which looks for an automaton interacting closely with the system considered.
- MinStates. Use the automaton with the fewest states. Similar to the MinEvents heuristic, this is an attempt to construct a small synchronous product.
- MinTransitions. Use the automaton with the fewest transitions.
- One. Use the first automaton found not to accept the counterexample.
- RelMaxCommonEvents. Use the automaton with the maximum ratio of shared events with the system considered to the number of used events. This is similar to MaxCommonEvents, but tries to avoid adding complex automata using many events, but sharing few events with the system considered.

VII. Experimental Results

The proposed algorithm has been tested with a set of industrial examples and case studies previously used in [8]. All examples considered are listed below, together with the corresponding automata models, also referred to in table I.

- Case study production cell I [16]: fzelle.
- Case study production cell II [17]: ftechnik.
- PROFIsafe field bus protocol [18]–[20]: profisafe_i4, profisafe_o4.
- AIP automated manufacturing system [21]–[23]: rhone-alps, rhone_tough.
- Train testbed [24]: tbed_uncont, tbed_ctct, tbed_valid.
- Central locking system (KorSys project): verriegel4.

These models have been checked for control-loops, using the proposed algorithm and each of the heuristics mentioned in section VI. All tests were performed on a standard PC with a 1.4 GHz processor and 256 MB of RAM. The results are shown in table I.

The first two columns list the model name and the number of automata for each model. The subsequent column pairs list for each heuristic and all examples the maximum number of automata composed and the total number of states constructed. The examples above the horizontal line in the table body are control-loop free, whereas the examples below do not satisfy this property.

In spite of the complexity of the models, they all could be shown to be control-loop free or not, requiring not more than a few seconds of CPU time for each run.

All control-loop free examples could be verified easily, never considering more than a single automaton at a time. This becomes possible by the initial step of the algorithm, which identifies the loop events of each automaton individually. Using proposition 5, it is always possible to reduce the set of possible loop events to the empty set, completely avoiding synchronous composition. The examples that contain control-loops are only slightly harder to

TABLE I

EXPERIMENTAL DATA FROM INDUSTRIAL EXAMPLES

Model		All		Early NotAccept		Late NotAccept		MaxCommon Events		MaxCommon Uncontr		Min Events		Min NewEvents		Min States		Min Transitions		One		RelMax Common	
Name	Aut	Aut	States	Aut	States	Aut	States	Aut	States	Aut	States	Aut	States	Aut	States	Aut	States	Aut	States	Aut	States	Aut	States
profisafe_i4	75	1	267	1	267	1	267	1	267	1	267	1	267	1	267	1	267	1	267	1	267	1	267
profisafe_o4	84	1	292	1	292	1	292	1	292	1	292	1	292	1	292	1	292	1	292	1	292	1	292
rhone_tough	61	1	318	1	318	1	318	1	318	1	318	1	318	1	318	1	318	1	318	1	318	1	318
tbed_ctct	84	1	507	1	507	1	507	1	507	1	507	1	507	1	507	1	507	1	507	1	507	1	507
tbed_uncont	58	1	421	1	421	1	421	1	421	1	421	1	421	1	421	1	421	1	421	1	421	1	421
tbed_valid	84	1	468	1	468	1	468	1	468	1	468	1	468	1	468	1	468	1	468	1	468	1	468
verriegel4	65	1	726	1	726	1	726	1	726	1	726	1	726	1	726	1	726	1	726	1	726	1	726
ftechnik	36	4	2927	4	3053	4	3053	4	3136	4	3136	4	3053	4	3053	4	3053	4	3053	4	3053	4	3136
fzelle	67	3	830	2	834	2	834	2	834	2	834	2	834	2	834	2	834	2	834	2	834	2	830
rhone_alps	35	2	247	2	247	2	247	2	247	2	247	2	247	2	247	2	247	2	247	2	247	2	247

check: their faults are not so conspicuous that they can be found considering only one automaton. All results are largely independent of the heuristics for the selection of automata based on counterexamples.

These examples suggest that models of practical systems are typically well-structured in a way that makes modular verification easy. Control-loops can be detected automatically very quickly, making this property an excellent candidate to be implemented as a push-button feature in development tools that support engineers in the design of reactive systems.

VIII. CONCLUSIONS

An efficient algorithm to check a finite-state system for control-loops has been presented. This algorithm can avoid explicit state exploration of the entire system state space by exploiting the modularity common in many practical applications. Experimental results show that the new method can quickly verify several examples of industrial scale.

The proposed algorithm can be extended and enhanced in many ways. In the future, the authors would like to experiment with different heuristics for the selection of automata, and to investigate more elaborate reasoning. With the possibility to reduce the set of possible loop events, it is not required to keep increasing the subsystem being checked. It may be more efficient to restart with smaller systems when the set of loop events has been reduced.

REFERENCES

[1] P. J. G. Ramadge and W. M. Wonham, “The control of discrete event systems,” *Proc. IEEE*, vol. 77, no. 1, pp. 81–98, Jan. 1989.

[2] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Kluwer, Sept. 1999.

[3] S. Balemi, G. J. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. F. Franklin, “Supervisory control of a rapid thermal multiprocessor,” *IEEE Trans. Automat. Contr.*, vol. 38, no. 7, pp. 1040–1059, July 1993.

[4] S. Balemi, “Input/output discrete event processes and communication delays,” *Discrete Event Dynamic Systems: Theory and Applications*, vol. 4, pp. 41–85, 1994.

[5] C. H. Golaszewski and P. J. Ramadge, “Control of discrete event processes with forced events,” in *Proc. 26th IEEE Conf. Decision and Control, CDC ’87*, Los Angeles, CA, USA, Dec. 1987, pp. 247–251.

[6] P. Dietrich, R. Malik, W. M. Wonham, and B. A. Brandin, “Implementation considerations in supervisory control,” in *Synthesis and Control of Discrete Event Systems*, B. Caillaud, P. Darondeau, L. Lavagno, and X. Xie, Eds. Kluwer, 2002, pp. 185–201.

[7] P. Malik, “From supervisory control to nonblocking controllers for discrete event systems,” Ph.D. dissertation, University of Kaiserslautern, Kaiserslautern, Germany, 2003.

[8] B. A. Brandin, R. Malik, and P. Malik, “Incremental verification and synthesis of discrete-event systems guided by counter-examples,” *IEEE Trans. Contr. Syst. Technol.*, vol. 12, no. 3, pp. 387–401, May 2004.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specifications,” *ACM Trans. Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, Apr. 1986.

[10] K. L. McMillan, *Symbolic Model Checking*. Kluwer, 1993.

[11] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[12] H.-M. Hanisch and S. Kowalewski, “Algebraic synthesis and verification of discrete supervisory controllers for forbidden path specifications,” in *Proc. 4th Int. Conf. Computer Integrated Manufacturing and Automation Technology*. Troy, NY, USA: IEEE Computer Society Press, Oct. 1994, pp. 157–162.

[13] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[14] R. Tarjan, “Depth first search and linear graph algorithms,” *SIAM J. Computing*, vol. 1, no. 2, pp. 146–160, June 1972.

[15] E. Nuutila and E. Soisalon-Soininen, “On finding the strongly connected components in a directed graph,” *Information Processing Letters*, vol. 49, no. 1, pp. 9–14, Jan. 1994.

[16] C. Lewerentz and T. Linder, *Case Study “Production Cell”*, ser. LNCS. Springer, 1995, vol. 891.

[17] A. Lötzbeyer and R. Mühlfeld, “Task description of a flexible production cell with real time properties,” FZI, Karlsruhe, Germany, Tech. Rep., 1996. [Online]. Available: http://www.fzi.de/divisions/prost/projects/korsys/korsys.html

[18] R. Malik and R. Mühlfeld, “A case study in verification of UML statecharts: the PROFIsafe protocol,” *J. Universal Computer Science*, vol. 9, no. 2, pp. 138–151, Feb. 2003.

[19] ——, “Testing the PROFIsafe protocol using automatically generated test cases based on a formally verified model,” Siemens AG, Corporate Technology, Software and Engineering 1, Munich, Germany, Tech. Rep., 2002.

[20] Profibus Nutzerorganisation e. V., “PROFIsafe—profile for safety technology, version 1.12,” 2002.

[21] B. Brandin and F. Charbonnier, “The supervisory control of the automated manufacturing system of the AIP,” in *Proc. Rensselaer’s 4th Int. Conf. Computer Integrated Manufacturing and Automation Technology*, Troy, NY, USA, 1994, pp. 319–324.

[22] F. Charbonnier, “Commande par supervision des systèmes à événements discrets: application à un site expérimental l’Atelier Interétablissement de Productique,” Laboratoire d’Automatique de Grenoble, Grenoble, France, Tech. Rep., 1994.

[23] R. J. Leduc, “Hierarchical interface-based supervisory control,” Ph.D. dissertation, Dept. of Electrical Engineering, University of Toronto, Ontario, Canada, 2002.

[24] ——, “PLC implementation of a DES supervisor for a manufacturing testbed: An implementation perspective,” Master’s thesis, Dept. of Electrical Engineering, University of Toronto, Ontario, Canada, 1996.