Incremental Verification and Synthesis of Discrete-Event Systems Guided by Counter-Examples

Bertil A. Brandin, Robi Malik, and Petra Malik

***Abstract*— This article presents new approaches to system verification and synthesis based on subsystem verification and the novel combined use of counter-examples and heuristics to identify suitable subsystems incrementally. The scope of safety properties considered is limited to behavioural inclusion and controllability. The verification examples considered provide a comparison of the approaches presented with straightforward state exploration, and an understanding of their applicability in an industrial context.**

***Index Terms*— Software verification and validation, software requirements and specifications, control systems, formal languages, controllability, search methods.**

I. Introduction

FOR certain behavioural properties, and with obvious computational advantages, it is possible to verify that a system satisfies given behavioural requirements, by verifying that one or more subsystems satisfy the same requirements. The identification of suitable subsystems, possibly in an automated fashion, remains one of the key challenges of such verification approaches.

The identification methods presented herein are based on the use of *counter-examples*, providing information on which system components may contribute to a proof, and on corresponding selection heuristics.

The scope of safety properties considered is limited in this article to *behavioural inclusion* and *controllability*. Behavioural inclusion can be described as follows: a system behaviour is verified to satisfy given behavioural requirements when its behaviour can be shown to always remain within the behavioural requirements. Controllability can be seen as an extension under control of behavioural inclusion: a system behaviour is verified to be controllable with respect to given requirements when its behaviour can be shown to always remain through control within the behavioural requirements. In case behavioural inclusion or controllability are not satisfied, the synthesis of new subsystems is proposed, which when composed with the original system, guarantee that the composed system satisfies behavioural inclusion and controllability requirements.

The examples considered provide a comparison of the approaches presented with straightforward state exploration, and an understanding of their applicability in an industrial context.

Manuscript received XXX; revised YYY.

Incremental verification approaches are not new and have been used to improve performance of model checking procedures. A number of abstraction techniques have been proposed to simplify the verification task, for example in [1], [2], also [3] suggests the use of counter-examples for CTL model checking. The work presented in this article is based on the discrete-event system framework of [4]–[7], and proposes new approaches to system verification and synthesis based on subsystem verification and the novel combined use of counter-example and heuristics to identify suitable subsystems incrementally. Earlier related work by the authors is found in [8] and has been extended with respect to synthesis in [9].

II. Notation and Preliminaries

The following summary of notions and terms from supervisory control theory has been composed from several sources, but principally from [4], [5]. The concepts of synchronous communication used in this framework are adaptations of earlier work in the field of process algebras [10], [11].

A. Languages

Event sequences and languages are simple means of describing discrete system behaviours. In this context, we consider an *alphabet* Σ as a finite set of distinct *events*. By Σ^+ we denote the set of all finite *strings* of the form $\sigma_1\sigma_2\cdots\sigma_k$, where $k \geq 1$ and $\sigma_i \in \Sigma$. Furthermore it is convenient to introduce the *empty string* ε, where $\varepsilon \notin \Sigma$. Then we write $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. A *language* over Σ is any subset $L \subseteq \Sigma^*$.

The *length* $|s|$ of a string $s \in \Sigma^*$ is the number of symbols in s. The *catenation* of two strings $s, t \in \Sigma^*$ is written as st. Languages and alphabets can also be catenated: we write $L\Sigma = \{\, s\sigma \in \Sigma^* \mid s \in L, \sigma \in \Sigma \,\}$.

For a string $s \in \Sigma^*$ we say that $t \in \Sigma^*$ is a *prefix* of s, and write $t \leq s$, if $s = tu$ for some $u \in \Sigma^*$. The *prefix-closure* $\overline{L}$ of a language $L \subseteq \Sigma^*$ is the set of all prefixes of strings in L, i.e. $\overline{L} = \{\, t \in \Sigma^* \mid t \leq s \text{ for some } s \in L \,\}$. A language L is called *prefix-closed* if $L = \overline{L}$. For the purpose of this paper, which is restricted to safety properties, it is sufficient to consider only prefix-closed languages.

B. Automata

Languages can be represented naturally by means of automata. An *automaton* is a 4-tuple $G = (\Sigma, Q, \delta, q_0)$, where

Σ is an alphabet of *events*, Q is the *state set* (assumed to be finite and nonempty), $\delta\colon\ Q \times \Sigma \to Q$ is the *transition function*, and $q_0 \in Q$ is the *initial state*. The transition function $\delta\colon\ Q \times \Sigma \to Q$ is defined at each state $q \in Q$ only for some of the events $\sigma \in \Sigma$, i.e. δ is a partial function. The transition function δ is extended to a partial function $\delta\colon\ Q \times \Sigma^* \to Q$ by letting $\delta(q, \varepsilon) = q$, and $\delta(q, s\sigma) = \delta(\delta(q, s), \sigma)$ provided that $q' = \delta(q, s)$ and $\delta(q', \sigma)$ are defined. The *possible behaviour* of an automaton $G = (\Sigma, Q, \delta, q_0)$ is described by the language $L(G) = \{\, s \in \Sigma^* \mid \delta(q_0, s) \text{ is defined} \,\}$.

Automata are represented visually by means of *state transition graphs*. An example is shown in Fig. 1: states are represented as nodes, with the initial state highlighted, the transition function is represented by labelled edges; if $\delta(q_1, \sigma) = q_2$, then the graph contains an edge from node q_1 to q_2 labelled σ.

C. Synchronous Product

Several automata can be combined into a single, more complex automaton by means of the *synchronous product* operation. Let $G_1 = (\Sigma, Q_1, \delta_1, q_{1,0})$ and $G_2 = (\Sigma, Q_2, \delta_2, q_{2,0})$ be two automata, both using the alphabet Σ. Then their synchronous product $G_1 \parallel G_2$ is defined as

$$G_1 \parallel G_2 \;=\; (\Sigma, Q_1 \times Q_2, \delta, (q_{0,1}, q_{0,2}))$$

where

$$\delta((q_1, q_2), \sigma) \;=\; (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$$

provided that $\delta_1(q_1, \sigma)$ and $\delta_2(q_2, \sigma)$ are both defined.

Thus the synchronous product of two automata defines how two automata are composed by synchronising them on their events. Its state set consists of the Cartesian product of the state sets of the composed automata. Its language is seen to be the intersection of the languages of the composed automata, i.e.

$$L(G_1 \parallel G_2) \;=\; L(G_1) \cap L(G_2). \tag{1}$$

We define for future reference the neutral element G_{Σ^*} with respect to synchronous composition, such that $L(G_{\Sigma^*}) = \Sigma^*$ and $L(G \parallel G_{\Sigma^*}) = L(G)$ for any automaton G.

D. Relevant Event Set

The synchronous product as defined above can only be used for the composition of automata with the same event alphabet. In order to compose two automata

$$\begin{aligned} G_1 &= (\Sigma_1, Q_1, \delta_1, q_{0,1}) \\ G_2 &= (\Sigma_2, Q_2, \delta_2, q_{0,2}) \end{aligned}$$

with different event alphabets Σ_1 and Σ_2, these must be extended to consider the alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$ by adding selfloops with the missing events to all their states. Fig. 2 shows how the automaton $Buffer$ of Fig. 1, originally with alphabet $\{s_1, s_2, f_1\}$, has been extended to consider the alphabet $\{s_1, s_2, f_1, f_2, b_1, b_2, r_1, r_2\}$ by adding selfloops with corresponding event labels.

Given such an extended automaton, it is interesting to determine which events are selflooped at every state of the automaton, being effectively irrelevant to the automaton's behaviour, and which events are associated with "meaningful" state transitions relevant to the automaton's behaviour. Accordingly, we introduce the notion of relevant and irrelevant events. For a language $L \subseteq \Sigma^*$, an event $\sigma \in \Sigma$ is said to be *irrelevant* for L, if we have for all $s, t \in \Sigma^*$

$$st \in L \quad \text{if and only if} \quad s\sigma t \in L; \tag{2}$$

otherwise σ is called *relevant* for L. The set of all relevant events for a language $L \subseteq \Sigma^*$ is denoted by

$$\operatorname{rel} L \;=\; \{\, \sigma \in \Sigma \mid \sigma \text{ is relevant for } L \,\}. \tag{3}$$

Accordingly, an event will be said to be relevant or irrelevant for an automaton G if it is relevant or correspondingly irrelevant for the language $L(G)$. For convenience, we will picture automata with relevant events only, unless otherwise stated.

E. Supervisory Control

In order to introduce the notion of supervisory control, the set Σ of events is partitioned into two disjoint subsets of events: the subset Σ_{c} of *controllable events* and the subset Σ_{u} of *uncontrollable events*. Controllable events may be enabled or disabled by an external agent; uncontrollable events are spontaneous.

Let L be a prefix-closed language describing a possible system behaviour, and let K be another prefix-closed language describing a desired system behaviour. K is defined to be *controllable* with respect to L if

$$K\,\Sigma_{\mathrm{u}} \cap L \;\subseteq\; K. \tag{4}$$

In other words, a language K is controllable with respect to L if there is no string in K that can be followed by an uncontrollable event possible in L but not possible in K. This means that, given a possible system behaviour L, the behaviour given by K can be achieved by disabling controllable events only. Note that the languages $\emptyset$, L, and Σ^* are all trivially controllable with respect to L.

We introduce the set $\mathcal{C}(K, L)$ of all sublanguages of a language K that are controllable with respect to L:

$$\begin{aligned} \mathcal{C}(K, L) \;=\; \{\, & K' \subseteq K \mid \\ & K' \text{ is controllable with respect to } L \,\}. \end{aligned} \tag{5}$$

It is easy to show that the union of any number of controllable languages is again controllable [4]. Therefore, the set $\mathcal{C}(K, L)$ contains a unique supremal element

$$\sup \mathcal{C}(K, L) \;=\; \bigcup_{K' \in \mathcal{C}(K, L)} K'. \tag{6}$$

This supremal element is known as the *supremal controllable sublanguage* of K with respect to L [12]. It is the maximally permissive behaviour achievable through control within K.

III. Verification and Synthesis

A. Behavioural Inclusion

Let a system G and requirements R be modelled as automata. We define *behavioural inclusion* as follows.

Definition 1: Let G and R be two automata using the same alphabet Σ. We say that G *satisfies* R if $L(G) \subseteq L(R)$.

Thus, G satisfies R if every string of events accepted by the automaton G is also accepted by the requirements automaton R. In order to test whether this condition is true for G and R, we can construct the synchronous product of G and R, and check at each reachable combination of states whether there exists an event possible in G which is not allowed in R. If such a state combination exists, G does not satisfy the requirements R, otherwise it does.

In practice however, the system G and the requirements R are typically specified as the synchronous composition of several automata, and not as single automata, i.e.

$$\begin{aligned} G &= G_1 \parallel \cdots \parallel G_n, \qquad \text{and} \\ R &= R_1 \parallel \cdots \parallel R_m. \end{aligned}$$

Note that the synchronous composition of the automata G and R may be large and impossible to construct explicitly.

The language of a composed system consists of the intersection of the languages of its components[1], i.e.

$$\begin{aligned} L(G) &= L(G_1) \cap \cdots \cap L(G_n), \\ L(R) &= L(R_1) \cap \cdots \cap L(R_m). \end{aligned}$$

Synchronous composition by definition always restricts the system behaviour and never enlarges it. This leads to the following simple result.

Proposition 1: Let $K, L \subseteq \Sigma^*$ be two languages, and let $L = L_1 \cap L_2$. If $L_1 \subseteq K$ then $L \subseteq K$.

Assume we want to check whether

$$G = G_1 \parallel \cdots \parallel G_n \quad \text{satisfies} \quad R. \tag{7}$$

According to Proposition 1, in order to show that G satisfies R, it is enough to find a subsystem G' of G satisfying R. In particular, in order to prove (7), it is enough to show that there exists a set of indexes $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ such that

$$G' = G_{i_1} \parallel \cdots \parallel G_{i_k} \quad \text{satisfies} \quad R. \tag{8}$$

In case the requirements R are specified as the synchronous product of several automata, the following result can be used to simplify checking behavioural inclusion.

Proposition 2: Let $K, L \subseteq \Sigma^*$ be two languages, and let $K = K_1 \cap K_2$. Then we have $L \subseteq K$ if and only if $L \subseteq K_1$ and $L \subseteq K_2$.

Accordingly, in order to prove that a system G satisfies multiple requirements, we can prove that the system satisfies each requirement in turn. In particular, in order to prove that

$$G \quad \text{satisfies} \quad R = R_1 \parallel \cdots \parallel R_m$$

we can equivalently prove that

$$G \quad \text{satisfies} \quad R_j, \quad \text{for each } j = 1, \ldots, m.$$

[1] See Equation (1).

Finally, Propositions 1 and 2 can also be used to prove that a system G, specified as the synchronous product of several automata, satisfies multiple requirements by finding different subsystems of G satisfying each requirement individually. How to find suitable subsystems of G automatically is addressed in Section IV.

B. Controllability

System behaviours cannot always be shown to satisfy behavioural requirements. If the requirements are not satisfied, we can try to enforce them through control, i.e. through the disablement of controllable events.

The notion of controllability introduced in (4) allows us to characterise the languages which may be kept within another language through control. In the following, we will also refer to requirements R as being controllable with respect to a system G, if $L(R)$ is controllable with respect to $L(G)$. Assume a requirements automaton R to be controllable with respect to a system G. Then, when composed with G, R will restrict the behaviour of G to remain within the behaviour of R, such that $L(G \parallel R) \subseteq L(R)$.

For future reference, the definition of controllability is extended to consider any set of events.

Definition 2: Let $K, L \subseteq \Sigma^*$ be two prefix-closed languages, and let $\Sigma' \subseteq \Sigma$. K is called Σ'-*controllable* with respect to L if $K\Sigma' \cap L \subseteq K$.

This definition extends (4) by replacing the fixed set Σ_{u} of uncontrollable events by an arbitrary subset $\Sigma' \subseteq \Sigma$, enabling us to consider any set of events as the set of uncontrollable events.

In order to check whether R is controllable with respect to G, we can build the synchronous product and check at each reachable combination of states whether there exists an uncontrollable event possible in G but not in R. Fortunately, as in the case of behavioural inclusion, G and R are typically specified as the synchronous composition of several automata, allowing subsystems of G and R to be considered instead for verification purposes, as described in the following propositions.

Proposition 3: Let $K, L_1, L_2 \subseteq \Sigma^*$ be prefix-closed languages, and let $L = L_1 \cap L_2$. Also let $\Sigma' \subseteq \Sigma$. If K is Σ'-controllable with respect to L_1, then K is Σ'-controllable with respect to $L = L_1 \cap L_2$.

Proof: Simply observe that $K\Sigma' \cap L = K\Sigma' \cap L_1 \cap L_2 \subseteq K\Sigma' \cap L_1 \subseteq K$. ∎

Thus, as in the case of behavioural inclusion (Proposition 1), to show that R is controllable with respect to G it is enough to find a subsystem G' of G such that R is controllable with respect to G'. In particular, in order to check whether

$$R \quad \text{is controllable with respect to} \quad G = G_1 \parallel \cdots \parallel G_n \tag{9}$$

it is enough to show that there exists a set of indexes $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ such that

$$R \quad \text{is controllable with respect to} \quad G' = G_{i_1} \parallel \cdots \parallel G_{i_k}. \tag{10}$$

In case requirements R are specified as the synchronous product of several automata, the following extension of the

corresponding result of [4] can be used to simplify checking controllability.

Proposition 4: Let $K_1, K_2, L \subseteq \Sigma^*$ be prefix-closed languages, and let $K = K_1 \cap K_2$. Also let $\Sigma' \subseteq \Sigma$. If K_1 and K_2 are both Σ'-controllable with respect to L, then K is Σ'-controllable with respect to L.

Proof: Let $s \in K\Sigma' \cap L$. Then we have $s \in (K_1 \cap K_2)\Sigma' \cap L = K_1\Sigma' \cap K_2\Sigma' \cap L$. By controllability of K_1 and K_2, this implies $s \in K_1\Sigma' \cap L \subseteq K_1$ and $s \in K_2\Sigma' \cap L \subseteq K_2$. Therefore we have $s \in K_1 \cap K_2 = K$. Since $s \in K\Sigma' \cap L$ was chosen arbitrarily, this implies $K\Sigma' \cap L \subseteq K$, i.e. K is Σ'-controllable with respect to L. ■

This result can be paraphrased by saying that the intersection of two controllable languages is itself controllable. In other words, if requirements to be checked for controllability are specified as the synchronous product of several automata, it is enough to prove that each automaton is itself controllable. In particular, in order to check whether

$$R = R_1 \parallel \cdots \parallel R_m \quad \text{is controllable with respect to} \quad G \tag{11}$$

it is enough to show that

$$R_j \quad \text{is controllable with respect to} \quad G \tag{12}$$

for each $j = 1, \ldots, m$.

Note that the converse of Proposition 4 is not true. Since the composition of two uncontrollable behaviours may be controllable, checking (12) provides only a sufficient, but not a necessary condition for the controllability of the composed behaviour. Nevertheless, Proposition 4 can be used to develop interesting ways of checking controllability. For example, requirements R specified as the synchronous composition of several automata, can be shown to be controllable with respect to a system G, if R can be grouped into sets of automata controllable with respect to G. Thus, in order to check condition (11), we can try finding index sets $I_1, \ldots, I_{m'} \subseteq \{1, \ldots, m\}$, covering all the automata constituting R, such that each corresponding subsystem of R is controllable with respect to G, i.e. such that

$$\parallel_{i \in I_j} R_i \quad \text{is controllable with respect to} \quad G \tag{13}$$

for each $j = 1, \ldots, m'$, with

$$\bigcup_{j=1}^{m'} I_j \;=\; \{1, \ldots, m\}.$$

Note that in (13), subsystems G' of G can be considered instead of G using (10).

If some requirements from R are known to be controllable with respect to G, the following result allows us to modify (11).

Proposition 5: Let $K_1, K_2, L \subseteq \Sigma^*$ be prefix-closed languages, and let $K = K_1 \cap K_2$. Also let $\Sigma' \subseteq \Sigma$. If K_1 is Σ'-controllable with respect to $K_2 \cap L$, and K_2 is Σ'-controllable with respect to L, then K is Σ'-controllable with respect to L.

Proof: Let $s \in K\Sigma' \cap L = K_1\Sigma' \cap K_2\Sigma' \cap L$. Since K_2 is Σ'-controllable with respect to L, we have $K_2\Sigma' \cap L \subseteq K_2$, and therefore $s \in K_2$. This also implies $s \in K_1\Sigma' \cap K_2 \cap L$. Since K_1 is Σ'-controllable with respect to $K_2 \cap L$, we also have $K_1\Sigma' \cap K_2 \cap L \subseteq K_1$, and thus $s \in K_1$. So we have $s \in K_1 \cap K_2 = K$, and since $s \in K\Sigma' \cap L$ was chosen arbitrarily, this implies that K is Σ'-controllable with respect to L. ■

That is, if one of the requirements of R in (11), for example R_m, is known to be controllable with respect to G, (11) can be shown by verifying that

$$R_1 \parallel \cdots \parallel R_{m-1} \quad \text{is controllable with respect to} \quad G \parallel R_m \tag{14}$$

which is likely to be simpler than proving (11), because fewer requirements $R_1 \parallel \cdots \parallel R_{m-1}$ need to be shown to be controllable with respect to the extended system $G \parallel R_m$.

While Propositions 3, 4 and 5 above show that given requirements can be verified to be controllable with respect to a given system, by verifying requirement subsets to be controllable with respect to one or more subsystems, we will now consider the use of subsets of uncontrollable events.

Proposition 6: Let $K, L \subseteq \Sigma^*$ be two prefix-closed languages, and let $\Sigma', \Sigma'_1, \Sigma'_2 \subseteq \Sigma$ be alphabets such that $\Sigma' = \Sigma'_1 \cup \Sigma'_2$. Then K is Σ'-controllable with respect to L if and only if K is Σ'_1-controllable and Σ'_2-controllable with respect to L.

Proof: First let K be Σ'-controllable with respect to L. We have $\Sigma'_i \subseteq \Sigma'$ for each $i \in \{1, 2\}$. Since K is Σ'-controllable with respect to L, this implies $K\Sigma'_i \cap L \subseteq K\Sigma' \cap L \subseteq K$. Therefore, K is Σ'_i-controllable for each $i \in \{1, 2\}$.

For the converse implication, let K be Σ'_i-controllable for each $i \in \{1, 2\}$. Now let $s \in K\Sigma' \cap L$. Then we have $s \in K\Sigma'$ and $s \in L$. Since $\Sigma' = \Sigma'_1 \cup \Sigma'_2$, this implies that $s \in K\Sigma'_i$ for some $i \in \{1, 2\}$, and $s \in L$. Then we have $s \in K\Sigma'_i \cap L \subseteq K$, since K is Σ'_i-controllable. Since $s \in K\Sigma' \cap L$ was chosen arbitrarily, this implies $K\Sigma' \cap L \subseteq K$, i.e. K is Σ'-controllable with respect to L. ■

Thus, instead of considering all controllable events at once and verifying requirements to be controllable with respect to a given system, we can verify multiple times the requirements to be controllable, each time considering different subsets of uncontrollable events. In particular, it is possible to consider each uncontrollable event on its own. That is, in order to verify whether

$$R \quad \text{is } \{v_1, \ldots, v_l\}\text{-controllable with respect to} \quad G \tag{15}$$

we can equivalently check that

$$R \quad \text{is } \{v_j\}\text{-controllable with respect to} \quad G \tag{16}$$

for each $j = 1, \ldots, l$, i.e. l potentially simpler checks are carried out, assuming each time all uncontrollable events but one to be controllable, instead of one single more complex check, considering all uncontrollable events at once. Note that Proposition 6 should typically be used together with Propositions 3, 4, and 5.

C. Synthesis

If both the behavioural inclusion and the controllability checks fail, (i) the system considered is known not to satisfy

the requirements of interest, and (ii) its behaviour cannot be restricted through control to satisfy the corresponding requirements. Typically, the system will require a redesign. For this purpose, supervisory control [4] provides a means to synthesise (when possible) *least restrictive* and controllable components in the form of automata which composed with the original system, guarantee that the composed system satisfies behavioural inclusion and controllability requirements. Such synthesised components are sometimes also referred to as supervisors [4].

Similarly to the results of the above Sections, the following results show how the modular structure of a system and requirements composed of several automata can be used for synthesis purposes.

Proposition 7: Let $K_1, K_2, L \subseteq \Sigma^*$ be prefix-closed languages, and let $K = K_1 \cap K_2$. Furthermore let

$$\begin{aligned} \hat{K} &= \sup \mathcal{C}(K, L), \\ \hat{K}_1 &= \sup \mathcal{C}(K_1, L), \\ \hat{K}_2 &= \sup \mathcal{C}(K_2, L). \end{aligned}$$

Then $\hat{K} = \hat{K}_1 \cap \hat{K}_2$.

Proof: First, we show that $\hat{K} \subseteq \hat{K}_1 \cap \hat{K}_2$. Note that $\hat{K} = \sup \mathcal{C}(K, L)$ is controllable with respect to L, and $\hat{K} \subseteq K = K_1 \cap K_2$. The above implies that $\hat{K} \in \mathcal{C}(K_i, L)$ for each $i \in \{1, 2\}$. Then we have $\hat{K} \subseteq \sup \mathcal{C}(K_i, L) = \hat{K}_i$, i.e. $\hat{K} \subseteq \hat{K}_1 \cap \hat{K}_2$.

We now show that $\hat{K}_1 \cap \hat{K}_2 \subseteq \hat{K}$. Since $\hat{K}_i = \sup \mathcal{C}(K_i, L)$ is controllable with respect to L, for each $i \in \{1, 2\}$, we have from Proposition 4 that $\hat{K}_1 \cap \hat{K}_2$ is controllable with respect to L. Furthermore, we have $\hat{K}_i \subseteq K_i$ for each $i \in \{1, 2\}$, and therefore $\hat{K}_1 \cap \hat{K}_2 \subseteq K_1 \cap K_2 = K$. Then we have $\hat{K}_1 \cap \hat{K}_2 \in \mathcal{C}(K, L)$, i.e. $\hat{K}_1 \cap \hat{K}_2 \subseteq \sup \mathcal{C}(K, L) = \hat{K}$. ■

Consider a system behaviour G, and a requirements specification R consisting of several automata, i.e.

$$R = R_1 \parallel \cdots \parallel R_m.$$

Proposition 7 enables us to obtain synthesised components $\hat{R}_j$, for each $j = 1, \ldots, m$, such that $L(\hat{R}_j) = \sup \mathcal{C}(L(R_j), L(G))$, whose composition is equivalent to the synthesised least restrictive component $\hat{R}$ such that $L(\hat{R}) = \sup \mathcal{C}(L(R), L(G))$.

Proposition 8: Let $K, L_1, L_2 \subseteq \Sigma^*$ be prefix-closed languages, and let $L = L_1 \cap L_2$. Then $\hat{K}_1 = \sup \mathcal{C}(K, L_1)$ is controllable with respect to L.

Proof: By construction of the supremal controllable language, we have that $\hat{K}_1 = \sup \mathcal{C}(K, L_1)$ is controllable with respect to L_1. Then it follows from Proposition 3 that $\hat{K}_1$ is controllable with respect to L. ■

Proposition 8 is useful in case the system behaviour G is specified as the synchronous product of several automata

$$G = G_1 \parallel \cdots \parallel G_n.$$

In this case, we can try synthesising components $\hat{R}_i$, for each $i = 1, \ldots, n$, such that $L(\hat{R}_i) = \sup \mathcal{C}(L(R), L(G_i))$. If this fails, i.e. some $L(\hat{R}_i)$ is empty, we can try grouping system behaviours and synthesising supervisors for groups of subsystems of G. Proposition 8 tells us that the composition of all supervisors obtained in such a way is again controllable.

However, it is important to note that components which have been synthesised for a subsystem G' of G will be least restrictive for G', but not necessarily for G itself. Accordingly, the composition of such components may also not be least restrictive for G.

Proposition 8 has been extended in [9] to show that the composition of components synthesised with respect to two different system behaviours yields a least restrictive supervisor if the two system behaviours do not share any uncontrollable events. An alternative proof of the result in [9] is provided below.

Proposition 9: Let $K, L_1, L_2 \subseteq \Sigma^*$ be prefix-closed languages, and let $L = L_1 \cap L_2$. Also write $\hat{K} = \sup \mathcal{C}(K, L)$ and $\hat{K}_1 = \sup \mathcal{C}(K, L_1)$, and assume that $(\operatorname{rel} K \cup \operatorname{rel} L_1) \cap \operatorname{rel} L_2 \cap \Sigma_u = \emptyset$. Then we have $\hat{K}_1 \cap L = \hat{K} \cap L$.

Proof: First we show that $\hat{K}_1 \cap L \subseteq \hat{K} \cap L$. By Proposition 8, $\hat{K}_1 = \sup \mathcal{C}(K, L_1)$ is controllable with respect to L. Since we also have $\hat{K}_1 \subseteq K$, this implies $\hat{K}_1 \in \mathcal{C}(K, L)$, and therefore $\hat{K}_1 \cap L \subseteq \sup \mathcal{C}(K, L) \cap L = \hat{K} \cap L$.

Next we show that $\hat{K} \cap L \subseteq \hat{K}_1 \cap L$. We write $\Sigma_{u1} = (\operatorname{rel} K \cup \operatorname{rel} L_1) \cap \Sigma_u$. Note that Σ_{u1} is irrelevant for L_2. Let $s \in \hat{K} \cap L$. We prove by induction on the length of s that $s \in \hat{K}_1$.

Base case: $s = \varepsilon$. If $\varepsilon \in \hat{K}$, then we have $\Sigma_u^* \cap L \subseteq \hat{K}$ by controllability of $\hat{K}$. Since Σ_{u1} is irrelevant for L_2, we also have $\Sigma_{u1}^* \cap L_2 = \Sigma_{u1}^*$. These facts together imply $\Sigma_{u1}^* \cap L_1 = \Sigma_{u1}^* \cap L_1 \cap L_2 = \Sigma_{u1}^* \cap L \subseteq \Sigma_u^* \cap L \subseteq \hat{K} \subseteq K$. This means that $\Sigma_u^* \cap L_1 \subseteq K$ because $\Sigma_u - \Sigma_{u1}$ is irrelevant for K. But then we have $s = \varepsilon \in \hat{K}_1$, because $\hat{K}_1$ is a supremal controllable language with respect to K.

Inductive step: $s = t\sigma$. Let $s \in \hat{K} \cap L$. Then we have $t \in \hat{K}$, and by inductive assumption also $t \in \hat{K}_1$. Since $s \in \hat{K}$, we also have $s\,\Sigma_u^* \cap L \subseteq \hat{K}$ by controllability of $\hat{K}$. Since Σ_{u1} is irrelevant for L_2, and $s \in L_2$, we also have $s\,\Sigma_{u1}^* \cap L_2 = s\,\Sigma_{u1}^*$. These facts together imply $s\,\Sigma_{u1}^* \cap L_1 = s\,\Sigma_{u1}^* \cap L_1 \cap L_2 = s\,\Sigma_{u1}^* \cap L \subseteq s\,\Sigma_u^* \cap L \subseteq \hat{K} \subseteq K$. This means that $s\,\Sigma_u^* \cap L_1 \subseteq K$ because $\Sigma_u - \Sigma_{u1}$ is irrelevant for K. We also have $s = t\sigma$ with $t \in \hat{K}_1$, and therefore $s \in \hat{K}_1$, because $\hat{K}_1$ is a supremal controllable language with respect to K. ■

Again, we assume the system behaviour G to be composed of several automata

$$G = G_1 \parallel \cdots \parallel G_n.$$

In order to apply Proposition 9, we split the index set $\{1, \ldots, n\}$ into two disjoint sets J_1 and J_2. In addition, we require the subsystems

$$G' = \parallel_{i \in J_1} G_i \qquad \text{and} \qquad G'' = \parallel_{i \in J_2} G_i$$

to be such that the automata constituting G'' do not have any relevant uncontrollable events in common with any of the automata constituting G', nor with the requirement R. If this is the case, we can synthesise a component $\hat{R}'$ such that $L(\hat{R}') = \sup \mathcal{C}(L(R), L(G'))$. Proposition 9 guarantees that the behaviour of G composed with $\hat{R}'$ is identical to the

behaviour of G composed with the component $\hat{R}$, such that $L(\hat{R}) = \sup \mathcal{C}\,(L(R), L(G))$.

In case both the system behaviour G and the requirements R are specified as the synchronous product of several automata, we can use Propositions 7 and 9 jointly, by considering each requirement R_j in turn, applying Proposition 9 finding for R_j an appropriate subsystem G'_j of G, and synthesising a component $\hat{R}_j$ such that $L(\hat{R}_j) = \sup \mathcal{C}\,(L(R_j), L(G'_j))$. The composed behaviour $L(\hat{R}_1 \parallel \cdots \parallel \hat{R}_m)$ will be the same as the behaviour $L(\hat{R}) = \sup \mathcal{C}\,(L(R), L(G))$.

This approach works well if for each requirement R_j, a small set of system behaviours constituting the subsystem G'_j of G can be found. In practice, this is often the case since system behaviours typically only share few uncontrollable events. A simple algorithm for finding the smallest set of system behaviours needed for a given requirement is provided in [9].

IV. Proof Search by Counter-Examples

We have seen in Section III that it is possible to verify that a system satisfies given behavioural requirements, such as language inclusion and controllability, by verifying that one or more subsystems satisfy these requirements. The identification of suitable subsystems, in an automated fashion, remains one of the key challenges of such verification approaches.

Verification algorithms, able to determine whether given behavioural requirements are satisfied, will also produce *counter-examples* showing why the requirements are not satisfied. Furthermore, such diagnostic information may be used, as is shown below, to identify suitable subsystems, thereby guiding an automated proof-search.

A. Incremental Behavioural Inclusion Check

Assume we want to check whether a system G satisfies some behavioural requirements R, and consider an arbitrary subsystem G' of G. If the subsystem G' satisfies the behavioural requirements R, we know by Proposition 1 that the entire system G also satisfies the requirements R. Otherwise there must exist a counter-example showing that G' does not satisfy R, i.e. there must exist a string

$$s \in L(G') \quad \text{such that} \quad s \notin L(R).$$

In this case, we can check whether the remaining components of G accept the string s. In particular, we consider in turn each automaton G_i of G not belonging to the subsystem G', and check whether $s \in L(G_i)$. Note that checking whether a string is accepted by a deterministic automaton is a simple task with complexity bounded in the length of the string: it is enough to trace a path through the automaton using the string's events, starting at the initial state.

If s is accepted by all the automata G_i of G not belonging to G', we can immediately conclude that the behavioural requirement R is not satisfied: we have found a string

$$s \in L(G) \quad \text{such that} \quad s \notin L(R).$$

Thus, s is a counter-example showing that G does not satisfy R.

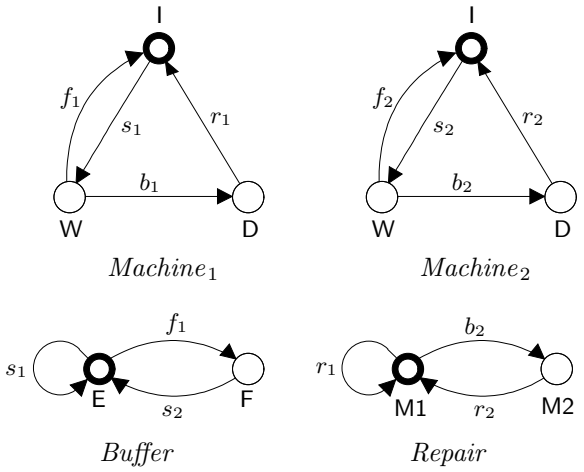

Fig. 1. Manufacturing example automata

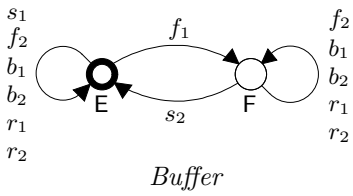

Fig. 2. Manufacturing example—*Buffer* automaton showing implicit selfloops for events f_2, b_1, b_2, r_1, and r_2

Otherwise there must be at least one automaton G_i not accepting s, i.e.

$$s \notin L(G_i).$$

In this case, we augment the subsystem G' with the automaton G_i, and check whether

$$G' \parallel G_i \quad \text{satisfies} \quad R.$$

Note that augmenting G' with an automaton G_j accepting s makes no sense since we already know there exists a counter-example $s \in L(G' \parallel G_j)$ such that $s \notin L(R)$.

Accordingly, the subsystem G' is augmented incrementally until it is either shown to satisfy the requirements R, or a counter-example accepted by G is found, showing that G itself does not satisfy R.

In order to illustrate the above procedure, we consider a simple manufacturing system consisting of two machines and a buffer of size one [4]. The system components are modelled as automata and are shown in Fig. 1. $Machine_1$ is initially in its idle state I. It may start (s_1), entering its working state W. When working, the machine may finish (f_1), returning to its idle state I, or it may break down (b_1) entering its down state D. It can then be repaired (r_1), returning to its idle state I. $Machine_2$ works in an identical fashion. The two machines are linked by a buffer of size one, and whenever $Machine_1$ finishes (f_1), it places a workpiece in the buffer, and whenever $Machine_2$ starts (s_2), it removes a workpiece from the buffer.

The automata *Buffer* and *Repair* specify the following behavioural requirements. The buffer must never overflow or underflow, i.e. $Machine_1$ must never finish operating while a workpiece is present in the buffer, and $Machine_2$ must never

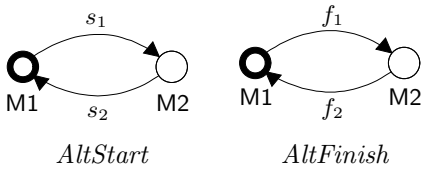

Fig. 3. Requirements for manufacturing example

start operating unless a workpiece is present in the buffer. Furthermore, $Machine_2$ has repair and return-to-service priority over $Machine_1$, i.e. in case both machines are down, $Machine_2$ must be repaired and returned to service first.

Please recall that, for convenience, automata are shown without irrelevant events, i.e. the automaton $Buffer$ in Fig. 1 is understood to behave like the automaton in Fig. 2.

We will check whether our manufacturing system

$$G = Machine_1 \parallel Machine_2 \parallel Buffer \parallel Repair$$

satisfies the behavioural requirement $AltStart$ of Fig. 3, specifying that the two machines always start alternatingly, i.e. we will check whether

$$L(Machine_1 \parallel Machine_2 \parallel Buffer \parallel Repair) \subseteq L(AltStart).$$

By Proposition 1, it is enough to find a subsystem G' of G satisfying $AltStart$. We start by considering an automaton able to generate the set of all finite strings, i.e. we let $G' = G_{\Sigma^*}$, and check whether

$$\Sigma^* \subseteq L(AltStart). \tag{17}$$

This is easily found not to be the case, producing the counter-example s_2 which is not accepted by $AltStart$. Note that in our implementation, the checking of (17) is simply performed by inspecting the automaton $AltStart$: we do not construct the automaton G_{Σ^*}.

Having obtained the counter-example s_2, we check whether this is a counter-example for the entire system, i.e. we check whether each of the automata of Fig. 1 accepts the string s_2. It turns out that all automata except $Buffer$ accept s_2. Accordingly, we augment G' with $Buffer$ so that $G' = Buffer$, and check whether

$$L(Buffer) \subseteq L(AltStart). \tag{18}$$

This is also not the case, yielding the counter-example $f_1 s_2$. Checking whether $f_1 s_2$ is a counter-example for the entire system, we find $Machine_1$ not accepting it. Accordingly, we augment G' with $Machine_1$ so that $G' = Buffer \parallel Machine_1$, now check whether

$$L(Buffer \parallel Machine_1) \subseteq L(AltStart) \tag{19}$$

and obtain the counter-example $s_1 b_1 r_1 s_1$ accepted by the entire system G. Accordingly, we conclude that G does not satisfy the behavioural requirement $AltStart$. Inspecting the counter-example $s_1 b_1 r_1 s_1$, we can see that the two machines cannot be started alternatingly in case $Machine_1$ breaks down, since the latter will have to be started again.

Note that we only needed to compose two out of four automata to show that G does not satisfy $AltStart$, and that these were identified automatically through the use of counter-examples.

B. Incremental Controllability Check

Assume we want to check whether

$$\begin{gathered} R = R_1 \parallel \cdots \parallel R_m \\ \text{is controllable with respect to} \\ G = G_1 \parallel \cdots \parallel G_n. \end{gathered} \tag{20}$$

According to Proposition 4 it is enough to show all requirement automata R_i, $i = 1, \ldots, m$, to be controllable with respect to the complete system behaviour G, either on their own or composed with other requirement automata. This can in turn be simplified considering Proposition 3, and identifying suitable subsystems G' of G to be used instead of G. In the following, let R' be the composition of several requirement automata for which we try to prove controllability with respect to a subsystem G' of G. Initially, we let $R' = R_j$, for some $j = 1, \ldots, m$.

We start by checking whether R' is controllable with respect to the set of all finite strings, i.e. we let $G' = G_{\Sigma^*}$, and check whether R' is controllable with respect to G_{Σ^*}. If this is the case, by Proposition 3, R' is also controllable with respect to G. Otherwise R' is found not to be controllable with respect to $G' = G_{\Sigma^*}$, yielding a counter-example s. There are now two possibilities.

(i) We can look for a system automaton G_k, $k = 1, \ldots, n$, not accepting s. If such an automaton is found, G' is augmented with G_k to check whether R' is controllable with respect to $G' \parallel G_k$. If this is the case, by Proposition 3, R' is controllable with respect to G. Otherwise we obtain a counter-example s showing that R' is not controllable with respect to $G' \parallel G_k$. Note that according to Proposition 5, a requirement R_l, $l = 1, \ldots, m$, controllable with respect to a subsystem of G and not accepting s could have been used instead of G_k to augment G'.

(ii) We can look for a requirement automaton R_l, $l = 1, \ldots, m$, not accepting s such that the first event of s not accepted by R_l is controllable, since requirement automata cannot disable uncontrollable events but only controllable events, in contrast to system automata which can disable both uncontrollable and controllable events. If such an automaton is found, we augment R' with R_l and check whether $R' \parallel R_l$ is controllable with respect to G'. If this is the case, by Propositions 3 and 4, $R' \parallel R_l$ is controllable with respect to G. Otherwise we obtain a counter-example s showing that $R' \parallel R_l$ is not controllable with respect to G'.

Note that in (i) above, if all G_k, $k = 1, \ldots, n$, accept s, the composed requirements R may still be controllable with respect to G as described in (ii).

Steps (i) and (ii) above are repeated, augmenting G' and R', until either R' is shown to be controllable with respect to

G', or G' and R' can no longer be augmented: a counter-example s has been found, accepted by the system and requirement automata $G_1, \ldots, G_n, R_1, \ldots, R_m$, showing R not to be controllable with respect to G. In case R' is found to be controllable with respect to G, the remaining requirement automata in $R - R'$ must next be shown to be controllable.

Note that checking R' to be controllable with respect to $G' \parallel G_k$ in (i) is usually easier than checking $R' \parallel R_l$ to be controllable with respect to G' in (ii), since in this case more requirements ($R' \parallel R_l$ compared to R') are considered with respect to comparatively fewer system automata (G' compared to $G' \parallel G_k$). Accordingly and if possible, it may be preferable to select (i) over (ii). We have implemented both, i.e. preferring (i) over (ii) and not preferring (i) over (ii); the experimental results are shown in Table III and discussed in Section V-B.

We now consider Proposition 6, which allows us to divide up the task of proving controllability in simpler subtasks, i.e. instead of carrying out one check considering the complete set of uncontrollable events, several checks are carried out instead, considering subsets of uncontrollable events. Proving controllability of a requirement with few relevant uncontrollable events will typically require considering few system components sharing these events. In particular, requirements with no relevant uncontrollable event are controllable per definition.

Consider again the small manufacturing example of the previous Section. We will check whether the automata $Buffer$ and $Repair$ of Fig. 1, and the automaton $AltFinish$ of Fig. 3 can actually implement the behaviour they describe. Recall that the automata $Buffer$ and $Repair$ specify respectively that the buffer must never overflow or underflow, and that $Machine_2$ has repair and return-to-service priority over $Machine_1$. The automaton $AltFinish$ specifies that the two machines must always finish their work alternatingly. Accordingly, we will check whether

$$\begin{gathered} L(Buffer \parallel Repair \parallel AltFinish) \\ \text{is } \{b_1, b_2, f_1, f_2\}\text{-controllable with respect to} \\ L(Machine_1 \parallel Machine_2). \end{gathered} \tag{21}$$

We consider in turn each requirement $Buffer$, $Repair$, and $AltFinish$. First we check whether $Buffer$ is controllable with respect to $G' = G_{\Sigma^*}$, i.e. whether

$$\begin{gathered} L(Buffer) \\ \text{is } \{b_1, b_2, f_1, f_2\}\text{-controllable with respect to} \\ \Sigma^*. \end{gathered}$$

This fails, producing the counter-example $f_1 f_1$, showing that the $Buffer$ requirement cannot be implemented with G'. We search for automata not accepting the counter-example $f_1 f_1$, and find $Machine_1$.

Note that, although the requirement automaton $AltFinish$ does not accept $f_1 f_1$, the first event of $f_1 f_1$ not accepted by $AltFinish$ is f_1 which is an uncontrollable event. Being a requirement automaton, $AltFinish$ cannot disable the uncontrollable event f_1 although it does not accept the string $f_1 f_1$. Therefore $AltFinish$ is not considered in order to augment G'.

Accordingly, G' is augmented with $Machine_1$ in order to check whether

$$\begin{gathered} L(Buffer) \\ \text{is } \{b_1, b_2, f_1, f_2\}\text{-controllable with respect to} \\ L(Machine_1). \end{gathered}$$

This check succeeds, showing that $Buffer$ is controllable with respect to $Machine_1$, and therefore with respect to $Machine_1 \parallel Machine_2$. Similarly, $Repair$ is shown to be controllable with respect to $Machine_2$, and therefore with respect to $Machine_1 \parallel Machine_2$.

As described above and according to Proposition 5, $Buffer$ and $Repair$ being controllable with respect to $Machine_1 \parallel Machine_2$, we can now show $AltFinish$ to be controllable with respect to $Machine_1 \parallel Machine_2$ by showing that

$$\begin{gathered} L(AltFinish) \\ \text{is } \{b_1, b_2, f_1, f_2\}\text{-controllable with respect to} \\ L(Machine_1 \parallel Machine_2 \parallel Buffer \parallel Repair). \end{gathered} \tag{22}$$

First we check $AltFinish$ to be controllable with respect to $G' = G_{\Sigma^*}$, obtaining the counter-example f_2. $Machine_2$ does not accept the counter-example f_2 and is therefore used to augment G' so that $G' = Machine_2$. Unfortunately, $AltFinish$ is not controllable with respect to G', yielding the counter-example $f_1 f_1$. Since both $Machine_1$ and $Buffer$ do not accept the counter-example $f_1 f_1$, we arbitrarily choose to augment G' with $Machine_1$ so that $G' = Machine_1 \parallel Machine_2$, and check whether

$$\begin{gathered} L(AltFinish) \\ \text{is } \{b_1, b_2, f_1, f_2\}\text{-controllable with respect to} \\ L(Machine_1 \parallel Machine_2) \end{gathered}$$

resulting in the counter-example $s_2 f_2$. $Buffer$ is the only automaton not accepting $s_2 f_2$, and is used to augment G' so that $G' = Machine_1 \parallel Machine_2 \parallel Buffer$, in order to check whether

$$\begin{gathered} L(AltFinish) \\ \text{is } \{b_1, b_2, f_1, f_2\}\text{-controllable with respect to} \\ L(Machine_1 \parallel Machine_2 \parallel Buffer) \end{gathered}$$

producing the new counter-example $s_1 f_1 s_2 s_1 f_1$ accepted by $Machine_1$, $Machine_2$, $Buffer$, and $Repair$. We conclude that the requirement $AltFinish$ is not controllable with respect to $Machine_1 \parallel Machine_2 \parallel Buffer \parallel Repair$. Inspecting the counter-example, we see that that the two machines can finish in any order, when simultaneously in their working state W.

The above proof-searches are shown in tree form in Fig. 4. Each node of the tree represents a step of the proof, and lists the requirement and system automata composed in that step, together with the counter-example obtained if any. We note that, in the second step of the controllability proof of $AltFinish$, had we selected $Buffer$ instead of $Machine_1$, then $Machine_1$ would have been selected in the next step.

The above proof is now simplified using Proposition 6 and considering one uncontrollable event at a time. Accordingly,

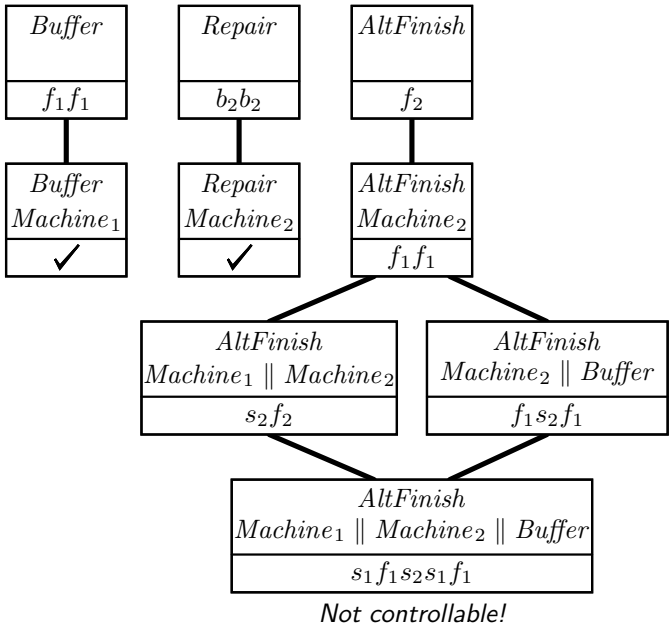

Fig. 4. Incremental controllability proof-search by component

in order to prove (21), it is enough to check whether

$$L(Buffer \parallel Repair \parallel AltFinish)$$
$$\text{is } \{\sigma_{\mathrm{u}}\}\text{-controllable with respect to}$$
$$L(Machine_1 \parallel Machine_2),$$
$$\text{for each } \sigma_{\mathrm{u}} \in \{b_1, b_2, f_1, f_2\}.$$

We need to check each of the three requirement automata to be controllable with respect to $Machine_1 \parallel Machine_2$, each time assuming all uncontrollable events but one to be controllable. Fortunately, most of these checks turn out to be trivial. For example, the event b_1 is seen to be irrelevant for any of the requirement automata, automatically implying these to be $\{b_1\}$-controllable with respect to $Machine_1 \parallel Machine_2$. Similarly, for $\sigma_{\mathrm{u}} = b_2$ only the $Repair$ requirement must be shown to be $\{b_2\}$-controllable with respect to $Machine_1 \parallel Machine_2$.

The non-trivial proofs are described in tree form in Fig. 5. Note that the requirement $AltFinish$ is considered twice, once for $\sigma_{\mathrm{u}} = f_1$ and once for $\sigma_{\mathrm{u}} = f_2$. Also note that, in the $\{f_1\}$-controllability proof of $AltFinish$, only $Machine_1$ and $Buffer$ need to be considered. In comparison, recall that $Machine_1$, $Machine_2$, and $Buffer$ were needed in the previous proof considering the complete set of uncontrollable events.

C. Heuristics

The incremental verification algorithms presented in this paper are guided by counter-examples obtained from failed analysis attempts to verify that a system satisfies given behavioural requirements, by verifying that one or more subsystems satisfy the same requirements. Although there may exist several counter-examples showing why one or more subsystems do not satisfy the requirements considered, the implementation presented always uses the first minimal-length counter-example found. We have not investigated in this work the use of alternative counter examples, which remains an interesting question for future research.

Given a counter-example, the incremental verification algorithms presented look for automata not accepting the particular counter-example to include one or more of these automata in the next analysis attempt. As seen in the previous Section, there may be cases in which multiple automata are found not to accept a particular counter-example. Experience shows that choosing the right automata may be crucial, and can make the difference between a quick proof or no proof, i.e. a proof-search blow-up.

A number of heuristics for selecting automata not accepting a counter-example of interest are presented below. They have all been implemented, and thoroughly tested on the industrial examples of Section V-B.

- All. This heuristic simply selects all the automata not accepting the counter-example.
- EarlyNotAccept. This heuristic selects the automaton rejecting the counter-example as early as possible, i.e. the automaton accepting as little as possible of the counter-example.
- LateNotAccept. This heuristic selects the automaton rejecting the counter-example as late as possible, i.e. the automaton accepting as much as possible of the counter-example.
- MaxCommonEvents. This heuristic selects the automaton with the maximum number of relevant events in common with the system considered so far. Recall that relevant events are defined by (2). An automaton sharing a large number of relevant events with the system considered is likely to interact more closely with it, and is therefore more likely to contribute to the analysis.
- MaxCommonUncontrollables. This heuristic selects the automaton with the maximum number of relevant uncontrollable events in common with the system considered so far. If two automata have the same number of relevant uncontrollable events in common, the automaton with the highest number of relevant controllable events in common is considered. This heuristic is similar to the MaxCommonEvents heuristic, but is adapted to controllability checks.
- MinEvents. This heuristic selects the automaton with the smallest number of relevant events, the motivation being selecting the simplest automata to construct the smallest possible synchronous product.
- MinNewEvents. This heuristic selects the automaton adding the minimum number of additional relevant events to the set of events of the system considered. This heuristic is similar to the MaxCommonEvents heuristic, which looks for an automaton interacting closely with the system considered.
- MinStates. This heuristic selects the automaton with the minimum number of states. This heuristic is similar to the MinEvents heuristic, the idea being constructing the smallest possible synchronous product.
- MinTransitions. This heuristic selects the automaton with the minimum number of transitions. This heuristic is very similar to the MinStates heuristic but considers the

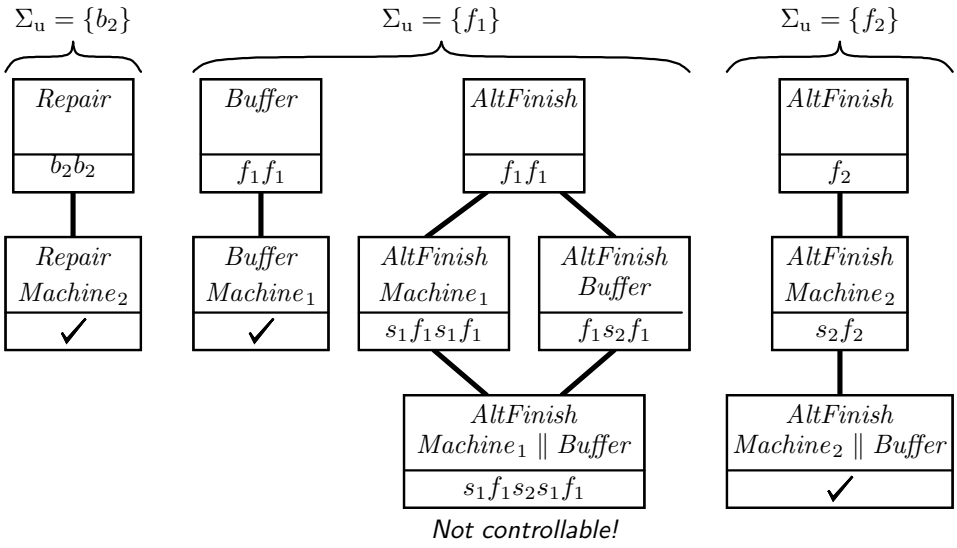

Fig. 5. Incremental controllability proof-search by component and event

number of transitions instead of the number of states as a measure of size.

- One. This heuristic simply selects the first automaton found.
- RelMaxCommonEvents. This heuristic selects the automaton with the maximum ratio of shared relevant events with the system considered to the automaton relevant alphabet. This heuristic is similar to the MaxCommonEvents strategy, but tries to avoid adding complex automata with large relevant alphabets, sharing few relevant events with the system considered. For example, an automaton with four relevant events, of which two are shared with the system considered (ratio: 2 to 4), is preferred to an automaton with twenty relevant events, sharing five relevant events with the system considered (ratio: 5 to 20).

As discussed in Section IV-B, it may be preferable, if possible, to augment subsystems (G') instead of the compositions of requirements (R') when checking controllability. Accordingly, a further heuristic consists of selecting requirement automata only in case all system automata are found to accept the counter-example under consideration. This heuristic has been implemented in combination with the above heuristics, resulting in two variations for each, one variation preferring system automata over requirement automata whenever possible, the other variation not preferring system automata over requirement automata.

V. Experimental Results

A scalable transfer line is considered first to (i) compare the incremental verification approaches presented with straightforward state exploration, and (ii) measure the computation efforts for increasing system complexity. A number of complex industrial applications are considered next to compare different heuristics, and assess the applicability of incremental verification to complex industrial examples.

A. Transfer Line Example

A scalable transfer line [5] is considered. It consists of functional blocks, shown in Fig. 6, which can be combined into a large, scalable system with regular structure.

Each block i consists of a machine $\mathbf{M}_i$ followed by a test unit $\mathbf{TU}_i$, linked by two buffers $\mathbf{B1}_i$ and $\mathbf{B2}_i$. The machines $\mathbf{M}_i$ are a simplified version of the machines of the manufacturing example of Section IV-A. They can start (s_i) and finish (f_i) operating. Workpieces can be loaded into the test unit $\mathbf{TU}_i$ (t_i), which in turn accepts (a_i) or rejects (r_i) them. If accepted, a workpiece is released and transferred to the next block, if rejected it is returned to the buffer $\mathbf{B1}_i$ for processing by $\mathbf{M}_i$. The buffer capacities for $\mathbf{B1}_i$ and $\mathbf{B2}_i$ are 3 and 1, respectively.

A functional block labelled i, modelled using five automata, is shown in Fig. 7. The machine and test unit behaviours are modelled by the automata $Machine_i$ and $TestUnit_i$. The behavioural requirements $B1Sup1_i$, $B1Sup2_i$, and $B2Sup_i$ specify that the two buffers must not underflow and overflow. A loading unit, used to load work pieces into the first block of the transfer line, is also shown in Fig. 7, modelled by the automaton $Init$.

In spite of appearances, system complexity increases dramatically with the number of combined functional blocks. The total number of reachable states of the synchronous product can be calculated explicitly to be

$$(1 + \tfrac{7}{58}\sqrt{58})(7 + \sqrt{58})^n + (1 - \tfrac{7}{58}\sqrt{58})(7 - \sqrt{58})^n \approx 1.92 \cdot 14.62^n$$

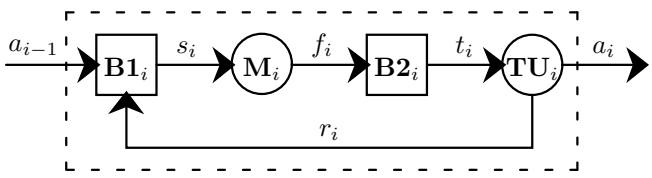

Fig. 6. Functional block for transfer line example

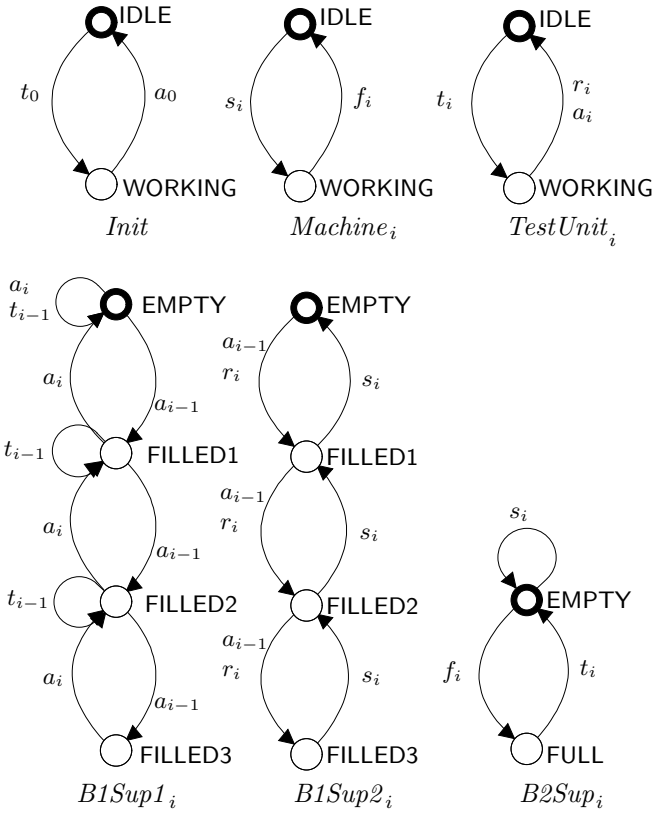

Fig. 7. Automata for transfer line example

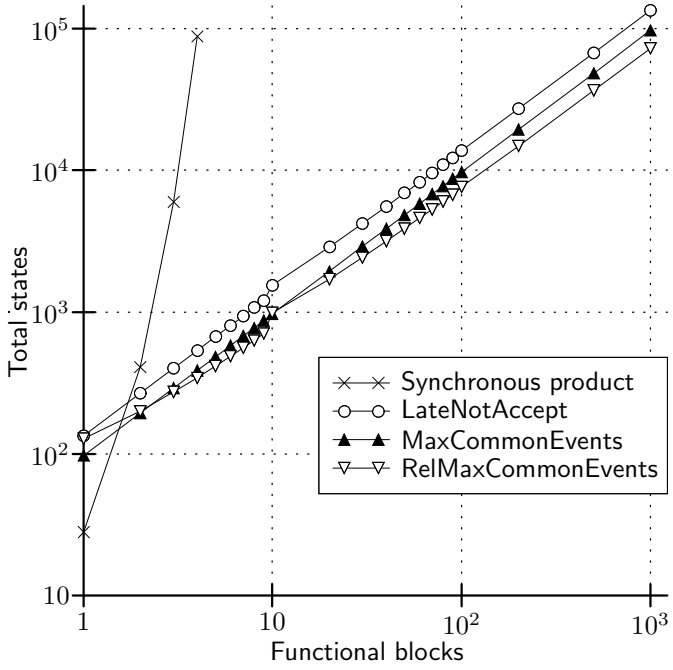

Fig. 8. Evaluation of transfer line example

where n is the number of combined functional blocks involved.

Transfer line models with 1 to 1 000 functional blocks were verified to be controllable using on the one hand, incremental verification as proposed in Section IV-B, and on the other hand straightforward state exploration, i.e. building the synchronous product. Fig. 8 shows the number of states constructed vs. the number of functional blocks considered.

The synchronous product grows so quickly that it is impossible to construct it explicitly for $n > 6$ combined functional blocks, and symbolically using BDDs [13] for $n > 8$ combined functional blocks. Incremental verification based on the LateNotAccept, MaxCommonEvents, or RelMaxCommonEvents heuristics, handles $n = 1\,000$ combined functional blocks easily (less than ten minutes of CPU time on a 600 MHz Pentium III processor with 512 MB of RAM), the number of states constructed growing linearly.

Further results are shown in Table I comparing the heuristics of Section IV-C. Verification runs were performed for different numbers n of functional blocks, recording the total number of states constructed in all proof steps, as well as the maximum number of automata composed in a single step. The number of states constructed in a verification step was limited to 1 000 000, if exceeded the run was aborted, and the corresponding table entry left blank. No preference was given to the selection of system automata or requirement automata.

Some heuristics are seen to perform poorly, constructing more states than the synchronous product. However, most heuristics succeeded in proving controllability of the entire system, never composing more than 6 or 8 automata at a time.

In summary, we conclude that the incremental verification approaches presented, while having a worst-case exponential complexity, can solve some problems in linear time.

B. Industrial Application Examples

The incremental verification approaches presented have been tested with complex industrial examples and case studies taken from various application areas such as manufacturing systems, communication protocols, and automotive body-electronics. All examples considered are listed below, together with the corresponding automata models, also referred to in Tables II and III.

- BMW E65 CAS window lift controller [14], [15]: big_cmft_kl50, big_fh_cmftreq1, big_manual_cmft, big-cmft_reg, big_fh_cmftreq0, big_bmw.
- Case study production cell I [16]: fzelle.
- Case study production cell II [17]: ftechnik, ftechnik-nocoll.
- PROFIsafe field bus protocol [18]–[20]: profisafe-i4_host_to, profisafe_o4_host_to, profisafe_i4_slave, profisafe_o4_slave, profisafe_i4, profisafe_o4.
- AIP automated manufacturing system [21]–[23]: rhone-alps.
- Train testbed [24]: tbed_uncont, tbed_nocoll, tbed-noderail, tbed_ctct, tbed_valid.
- Central locking system (KORSYS project): verriegel4-vrprop, verriegel4_erprop, verriegel4.

Considering in turn all the heuristics of Section IV-C, behavioural inclusion checks were carried out for 14 different behavioural requirements, and all examples were checked to be controllable. Note that controllability was checked

(i) not preferring system automata over requirement automata,

TABLE I

EXPERIMENTAL DATA FROM TRANSFER LINE EXAMPLE

n	Incremental controllability, not preferring system behaviours																					
	All		Early NotAccept		Late NotAccept		MaxCommon Events		MaxCommon Uncontr		Min Events		Min NewEvents		Min States		Min Transitions		One		RelMax Common	
	Aut	States	Aut	States	Aut	States	Aut	States	Aut	States	Aut	States	Aut	States	Aut	States	Aut	States	Aut	States	Aut	States
1	6	114	6	97	6	134	6	97	6	97	6	97	6	97	6	97	6	97	6	97	6	128
2	7	256	11	1787	6	268	6	194	6	194	8	384	11	1787	8	384	11	1787	8	384	6	200
3	7	398	16	26016	6	402	6	291	6	291	8	671	16	26016	8	671	16	26016	8	671	6	272
4	7	540	21	295258	6	536	6	388	6	388	8	958	21	295258	8	958	21	295258	8	958	6	344
5	7	682			6	670	6	485	6	485	8	1245			8	1245			8	1245	6	416
6	7	824			6	804	6	582	6	582	8	1532			8	1532			8	1532	6	488
7	7	966			6	938	6	679	6	679	8	1819			8	1819			8	1819	6	560
8	7	1108			6	1072	6	776	6	776	8	2106			8	2106			8	2106	6	632
9	7	1250			6	1206	6	873	6	873	8	2393			8	2393			8	2393	6	704
10	8	1609			8	1542	6	970	6	970	8	2490			8	2490			8	2490	8	992
20	8	3029			8	2882	6	1940	6	1940	8	5360			8	5360			8	5360	8	1712
30	8	4449			8	4222	6	2910	6	2910	8	8230			8	8230			8	8230	8	2432
40	8	5869			8	5562	6	3880	6	3880	8	11100			8	11100			8	11100	8	3152
50	8	7289			8	6902	6	4850	6	4850	8	13970			8	13970			8	13970	8	3872
60	8	8709			8	8242	6	5820	6	5820	8	16840			8	16840			8	16840	8	4592
70	8	10129			8	9582	6	6790	6	6790	8	19710			8	19710			8	19710	8	5312
80	8	11549			8	10922	6	7760	6	7760	8	22580			8	22580			8	22580	8	6032
90	8	12969			8	12262	6	8730	6	8730	8	25450			8	25450			8	25450	8	6752
100	8	14606			8	13804	6	9700	6	9700	8	28130			8	28130			8	28130	8	7688
200	8	28806			8	27204	6	19400	6	19400	8	56830			8	56830			8	56830	8	14888
500	8	71406			8	67404	6	48500	6	48500	8	142930			8	142930			8	142930	8	36488
1000	8	142623			8	134606	6	97000	6	97000	8	286240			8	286240			8	286240	8	72704

Aut is the maximum number of automata composed in a single step; States is the total number of states constructed in all steps together.

(ii) preferring system automata over requirement automata, and

(iii) considering uncontrollable events one at a time and not preferring system automata over requirement automata.

The results of all test runs are shown in Tables II and III. 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. Each Table is split into two blocks: the examples above the horizontal line satisfy the property considered (language inclusion or controllability), whereas the examples below the horizontal line do not satisfy the property considered.

The number of states constructed in a single verification step was limited to 2 000 000, if exceeded the run was aborted, and the corresponding table entry left blank. This number was chosen small to keep the test run-times reasonably short, increasing it was seen to have no impact on the test results.

In spite of the complexity of the models, these could all be shown to satisfy or not to satisfy behavioural inclusion and controllability, requiring less than ten minutes of CPU time using a 600 MHz Pentium III processor with 512 MB of RAM. This amount of memory is enough to explicitly construct state spaces with approximately 10^7 reachable states. Note that all the examples considered have state spaces in excess of 10^7 reachable states, which could not be constructed explicitly.

Fig. 9 and 10 respectively summarise the data from Tables II and III. They show the total number of automata constituting the model being checked versus the maximum number of automata used by the incremental verification algorithms presented. Note that for each model, only the maximum number of automata, used by successful heuristics with the minimum use of automata, is shown. Note that in Fig. 10 for each model, the data points, corresponding to the three alternative controllability checks considered, are connected by a vertical line.

The maximum number of automata used by incremental verification is seen to remain constant as the total number of automata constituting the model increases. This is particularly the case in Fig. 9. In comparison, straightforward state exploration requires using all the automata constituting the model being checked.

From Fig. 10, we can see that preferring system automata over requirements automata does not usually bring the expected gain in performance, although it can sometimes help as seen in Table III. Closer investigation of the models for which preferring system automata over requirements automata did not help, showed that the corresponding requirements could only be shown to be controllable if combined together. Furthermore, Fig. 10 suggests that checking controllability one uncontrollable event at a time typically reduces the maximum number of automata used. Nevertheless, inspection of Table III shows this approach does not perform much better than the other approaches presented, typically requiring more checks to be carried out, thereby increasing the chances of failed proof-searches.

Closer inspection of Tables II and III reveals further insights into the behaviour of the different heuristics. The MaxCommonEvents heuristic, not preferring system behaviours, can be seen to be the only consistently successful heuristic, but not necessarily the most efficient, sometimes constructing more states than for example the MaxCommonUncontrollables heuristic for controllability. The heuristics EarlyNotAccept and LateNotAccept outperform MaxCommonEvents in some cases. In contrast, the heuristics MinEvents, MinStates, and MinTransitions which select automata according to a measure of size, are seen to perform poorly; apparently size is not a good criterion when selecting

TABLE II

EXPERIMENTAL DATA FROM INDUSTRIAL EXAMPLES: INCREMENTAL BEHAVIOURAL INCLUSION CHECK

Model		Behavioural inclusion																			
		All		Early NotAccept		Late NotAccept		MaxCommon Events		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
big_cmft_kl50	32	7	753	6	118	5	77	9	1669	8	3868	8	1309	5	53	6	118	7	3305	6	432
big_fh_cmftreq1	32	3	23	2	7	2	7	2	7	4	52	9	2684	2	7	2	7	4	52	4	52
big_manual_cmft	32	7	765	11	6323	4	40	5	60	7	438	11	8789	3	13	4	40	7	1178	7	438
ftechnik_nocoll	42	30	271242	10	4309			13	2666552												
profi safe_i4_host_to	76	10	419					12	13571			11	2409	20	277605			15	77038	15	41148
profi safe_i4_slave	76	14	24612	18	475262	11	4450	12	11496	16	36377	18	274723	17	64953	18	454711			12	12635
profi safe_o4_host_to	85	10	436					12	13575			11	2454	20	277160			15	77044	15	40899
profi safe_o4_slave	85	12	2907			11	1690	11	10611	22	1079247									12	17662
tbed_nocoll	109			21	5144173			11	335791												
tbed_noderail	96	6	112	9	3287	24	8395939	6	623					11	4273			7	1245		
verriegel4_vrprop	66	9	34326	8	23709	8	23709	8	22614	8	23709	8	23709	7	11000	8	23709	7	11000	8	23709
big_cmft_req	32	7	1465	7	1548	7	1572	7	1548	7	1572	7	1385	7	1572	7	1572	7	1548	7	1572
big_fh_cmftreq0	32	9	2296	9	3584	10	5809	9	3851	9	3851	11	6261	11	6276	9	3894	9	4715	9	3851
verriegel4_erprop	66	7	538	7	759	7	759	7	759	10	1716	10	1716	7	759	7	759	10	1571	7	700

TABLE III

EXPERIMENTAL DATA FROM INDUSTRIAL EXAMPLES: INCREMENTAL CONTROLLABILITY CHECK

Model		Incremental controllability, not preferring system behaviours																			
		All		Early NotAccept		Late NotAccept		MaxCommon Events		MaxCommon Uncontr		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
big_bmw	31	7	1096	4	190	4	190	6	411	4	223	7	1281	4	190	4	190	4	190	9	5219
fzelle	67	9	7793	13	18385	7	5366	5	4072	5	4072	14	7101	13	8990	13	8732	11	4901	8	3394
profi safe_i4	75	6	688	3	351	5	245	3	155	5	160	4	369	5	244	5	245	5	160	3	224
profi safe_o4	84	6	691	3	354	5	248	3	158	5	163	4	372	5	247	5	248	5	163	3	227
rhone_alps	35	11	224616	15	1035198	9	16021	11	224838	10	16432	9	16063	6	903	6	903	6	955	13	1037531
tbed_ctct	84	10	119934	7	29092			5	18522	5	18522			12	3557956						
tbed_valid	84							26	609040	30	3733989										
verriegel4	65			10	32027	6	655	9	23142	6	1730	9	19103	7	7859	7	7859	13	75704	7	2956
ftechnik	36	23	159118	10	9879	3	221	3	221	3	221			20	6557213	20	6571455	20	4547036	6	683
tbed_uncont	58	44	821906	40	310893			44	5772225			47	2158804	46	1536444					49	999512

Model		Incremental controllability, preferring system behaviours																			
		All		Early NotAccept		Late NotAccept		MaxCommon Events		MaxCommon Uncontr		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
big_bmw	31	5	346	4	190	4	190	4	223	4	223	5	339	4	190	4	190	4	190	7	1110
fzelle	67	13	9711	11	9733	13	7799	10	6316	11	6826	13	7882	13	8990	13	8732	11	4901	13	5658
profi safe_i4	75	6	688	3	351	5	245	3	155	5	160	4	369	5	244	5	245	5	160	3	224
profi safe_o4	84	6	691	3	354	5	248	3	158	5	163	4	372	5	247	5	248	5	163	3	227
rhone_alps	35	11	224614	6	903	9	16021	8	8367	8	8333	9	16063	6	903	6	903	6	955	10	37030
tbed_ctct	84	10	119934	7	29092			5	18522	5	18522			12	3557956						
tbed_valid	84																				
verriegel4	65	12	1227894	10	31666	7	7859	7	9454	7	8956	11	42931	7	7859	7	7859	13	75704	8	16605
ftechnik	36	23	1089179	19	2376834	20	6568826	17	3428725	17	3429005	20	6783622	20	6557213	20	6571455	20	4547036	20	5854300
tbed_uncont	58	45	281589	48	6517643									46	1536444						

Model		Incremental controllability, considering one event at a time																			
		All		Early NotAccept		Late NotAccept		MaxCommon Events		MaxCommon Uncontr		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
big_bmw	31	7	1410	4	571	4	565	6	757	6	757	7	1940	4	571	4	571	4	571	9	5918
fzelle	67	10	7743	12	63838	7	5480	4	3771	4	3839	15	82276	14	65032	14	67070	14	98735	6	3008
profi safe_i4	75	7	48518			4	11287	3	3923	5	3945			8	46523	6	58483	4	8956		
profi safe_o4	84	7	48508			4	11457	3	4093	5	4115	11	3768020	9	63453	6	58653	4	9126		
rhone_alps	35	7	11572	8	30651	5	1951	6	4763	4	952	8	31748	8	31724	8	31473	8	31702	4	1112
tbed_ctct	84	9	69909	7	43248			4	5641	3	2992										
tbed_valid	84							10	315204	15	820620										
verriegel4	65	11	2179279	8	35151	4	893	9	30342	6	2260	7	15855	7	15375	7	15353	8	35376	4	861
ftechnik	36			8	216267			12	3116021	13	883573										
tbed_uncont	58			37	6311478			39	4348353	39	4356942										

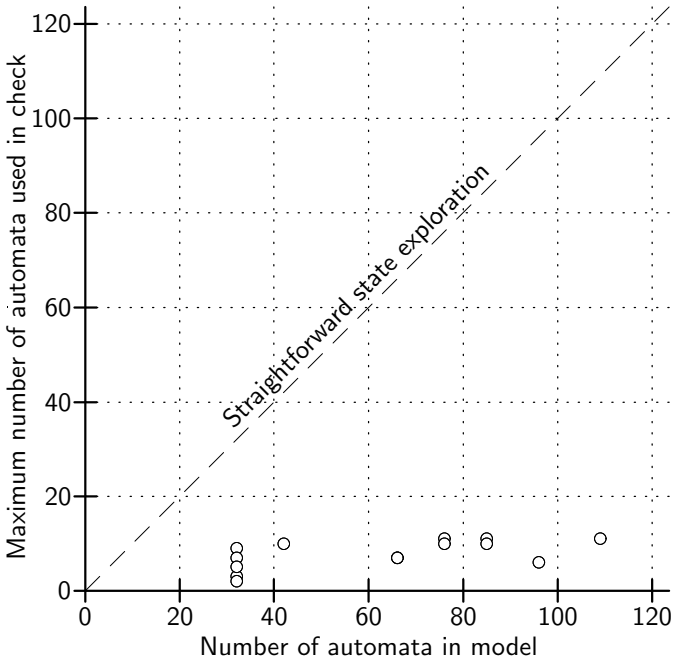

Fig. 9. Number of automata used in incremental behavioural inclusion check

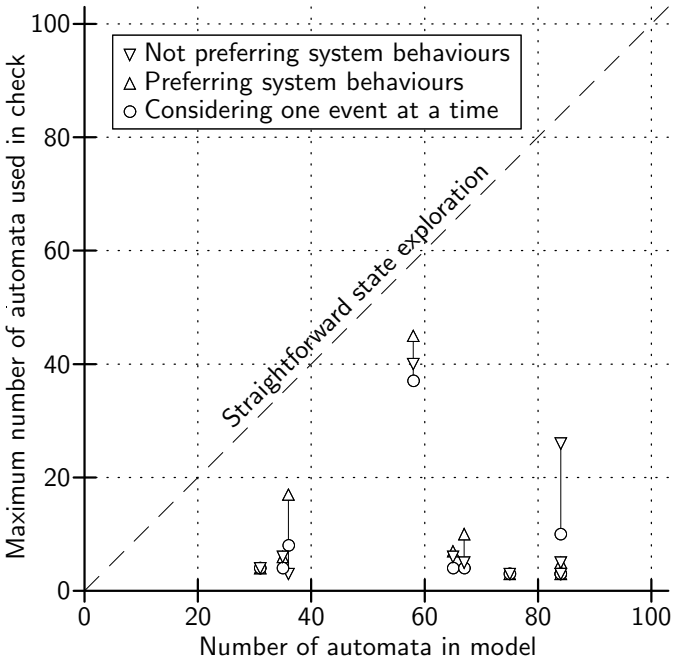

Fig. 10. Number of automata used in incremental controllability check

an automaton on the basis of counter-examples acceptance.

For all the examples considered, we either obtained a proof (positive or negative) in a few minutes of CPU time with a certain upper-limit, or this upper-limit was exceeded and the proof-search failed. This suggests that, to maximise the chances of obtaining a proof, it may make sense not to focus on one heuristic in particular, but to consider several heuristics, letting each run up to an empirical time upper-limit.

In summary, the incremental verification approaches presented were seen to be successful in checking behavioural inclusion and controllability for a number of complex industrial examples and case studies.

VI. Conclusions

This article presented new approaches to system verification and synthesis based on subsystem verification and the combined use of counter-examples and heuristics to identify suitable subsystems incrementally, thereby bringing about important computational advantages.

The scope of safety properties considered was limited in this article to behavioural inclusion and controllability.

The examples considered provided a comparison of the approaches presented with straightforward state exploration, and an understanding of their applicability in an industrial context. While having a worst-case exponential complexity, these were shown to have the potential of solving some problems in linear time, and were shown to be successful in checking behavioural inclusion and controllability for a number of complex industrial applications.

Two areas for future work have been identified: (i) the extension of the approaches presented to discrete event system synthesis, and in particular to the development of automatic incremental synthesis procedures, and (ii) the investigation of alternative subsystem selection heuristics in combination with state-space reduction techniques, such as symbolic representations or partial-order reduction.

References

[1] N. Halbwachs, F. Lagnier, and P. Raymond, “Synchronous observers and the verification of reactive systems,” in *Proc. 3rd Int. Conf. Algebraic Methodology and Software Technology*, ser. Workshops in Computing, M. Nivat, C. Rattray, T. Rus, and G. Scollo, Eds. Twente: Springer-Verlag, June 1993.

[2] A. Aziz, F. Balarin, R. K. Brayton, and A. Sangiovanni-Vincentelli, “Sequential synthesis using S1S,” in *Proc. Int. Conf. CAD*, 1995, pp. 621–617.

[3] T. Bultan, J. Fischer, and R. Gerber, “Compositional verification by model checking for counter-examples,” *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3, pp. 224–238, 1996.

[4] 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.

[5] W. M. Wonham, “Notes on control of discrete event systems,” Dept. of Electrical Engineering, University of Toronto, Ontario, Canada, 1999. [Online]. Available: http://www.control.utoronto.ca/ under “Research”

[6] W. M. Wonham and P. J. Ramadge, “Modular supervisor control of discrete event systems,” *Math. Control, Signals and Systems*, vol. 1, no. 1, pp. 13–30, Jan. 1988.

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

[8] B. A. Brandin, R. Malik, and P. Dietrich, “Incremental system verification and synthesis of minimally restrictive behaviours,” in *American Control Conf.*, Chicago, IL, USA, 2000, pp. 4056–4061.

[9] K. Åkesson, H. Flordal, and M. Fabian, “Exploiting modularity for synthesis and verification of supervisors,” in *Proc. 15th IFAC World Congress on Automatic Control*, Barcelona, Spain, 2002.

[10] G. Milne and R. Milner, “Concurrent processes and their syntax,” *J. ACM*, vol. 26, pp. 302–321, 1979.

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

[12] W. M. Wonham and P. J. Ramadge, “On the supremal controllable sublanguage of a given language,” *SIAM J. Control and Optimization*, vol. 25, no. 3, pp. 637–659, May 1987.

[13] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, 1986.

[14] P. Dietrich, “Projekt BMW E65 CAS — FH-Master — eine Modellierung in DCD,” Siemens AG, Corporate Technology, Software and Engineering 4, Munich, Germany, Tech. Rep., 2000.

[15] P. Malik, "From supervisory control to nonblocking controllers for discrete event systems," Ph.D. dissertation, University of Kaiserslautern, Kaiserslautern, Germany, 2003.
[16] C. Lewerentz and T. Linder, *Case Study "Production Cell"*, ser. LNCS. Springer-Verlag, 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, 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 International Conference on 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 Inter-établissement de Productique," Laboratoire d'Automatique de Grenoble, Grenoble, France, Tech. Rep., 1994.
[23] R. J. Leduc, "Hierarchical interface-based supervisory control," Ph.D. dissertation, 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.

Bertil A. Brandin (M'95) received the Bachelor's degree in mechanical engineering from the University of New South Wales, Australia, in 1984. He received the Master of Applied Science and Doctorate degrees in electrical engineering from the University of Toronto, Canada, in 1989 and 1993, respectively. From 1993 to 1995, he was project leader in a collaboration project on the supervisory control of automated manufacturing systems between the Province of Ontario, Canada, and Région Rhône-Alpes, France. In 1994 he was invited Scientist at the Rockwell Science Centre, Thousand Oaks, CA. From 1995 to the present day, he managed a research and development group of Siemens Corporate Technology, located in Munich, Germany, on formal verification techniques for software and business process applications. His research interests include discrete event systems, supervisory control, formal verification, model based testing and diagnostics. Safety and operation critical software applications have been the focus of his implementation efforts.

Robi Malik received the Masters degree in Computer Science at the University of Kaiserslautern, Germany, in 1993. He then obtained a Ph.D. scholarship and completed his Ph.D. degree at the University of Kaiserslautern in 1997. From 1998 to 2002, he continued his work in a research and development group at Siemens Corporate Research in Munich, Germany, where he was involved in the development and application of formal modelling and verification software. In 2003, he joined the University of Waikato in Hamilton, New Zealand, as a lecturer in Computer Science.

Dr. Malik has been working in the areas of logic programming, and formal specification, verification, and synthesis of finite-state and discrete-event systems. His current research interests are in the application of formal verification and abstraction techniques for the analysis of state-machine based specifications.

Petra Malik received the Masters degree in Computer Science at the University of Kaiserslautern, Germany, in 1999. After completing her Masters, she joined a research group at Siemens Corporate Research in Munich, Germany, as a Ph.D. student. During the research for her thesis, she investigated the issue of obtaining nonblocking implementations from discrete-event models. In 2001, she was a visitor at the Systems Control Group of the University of Toronto, Canada. Since 2003, she is working as a research assistant at the Department of Computer Science of the University of Waikato in Hamilton, New Zealand. Her research interests include verification of discrete-event systems, model checking, and theorem proving.