The Use of Language Projection for Compositional Verification of Discrete Event Systems

Simon Ware Robi Malik
Department of Computer Science
University of Waikato, Hamilton, New Zealand
{siw4,robi}@cs.waikato.ac.nz

***Abstract*— This paper proposes the use of abstraction by language projection to improve the performance of compositional verification to prove or disprove that a large system of composed finite-state machines satisfies a given safety property. Algorithms are presented for the automatic verification of language inclusion and controllability for discrete event systems, and are applied to a set realistic industrial examples. The experimental results suggest that the method can improve performance considerably, particularly in cases where previous methods of compositional verification fail because a large number of automata need to be considered.**

I. Introduction

With the continuously increasing size and complexity of reactive systems software, the automatic verification of large reactive systems is and remains a challenging problem, in the field of model checking in general, and in the area of discrete event systems in particular. This paper proposes a compositional approach based on language projection to verify safety properties of large systems composed of many automata running in parallel.

The scope of safety properties considered in this paper is limited to *language inclusion* and *controllability*. Language inclusion [1] is the question whether the composed behaviour of a set of automata is included in a given requirements language. Controllability [2], [3] can be seen as an extension of language inclusion: a system is controllable with respect to given requirements if its behaviour can be restricted to remain within the requirements by means of control.

The standard method to address these two questions involves the explicit composition of all the automata involved, and is limited by the well-known *state-space explosion* problem. *Symbolic model checking* techniques have been used successfully to reduce the amount of memory required by representing the state space symbolically rather the enumerating it explicitly [4]. As an alternative, *compositional* or *incremental* verification tries to avoid constructing large state spaces by reducing the number of automata to be composed: this method has been used very successfully to verify very large discrete event systems [1], [5].

This paper proposes to further enhance incremental verification as described in [1] by introducing *language projection* to compute *abstractions* of automata, reducing the size of the automata to be composed, and thereby further reducing the synchronous product state space. The projection-based method has been successfully applied to the same set of large-scale industrial examples as in [1], and is capable of solving at least one verification problem not solvable by the algorithms in [1].

This paper summarises some of the work presented in [6], which contains more detailed proofs and experimental results. In the following, Sect. II briefly introduces the needed terminology of languages, automata, and projection. Next, Sect. III presents the results needed to apply projection for the verification of language inclusion and controllability, and describes algorithms to verify these properties in a compositional way. Afterwards, Sect. IV discusses the experimental results, and Sect. V adds some concluding remarks.

II. Preliminaries

This section summarises the notation of automata and supervisory control theory used in this paper. For full details of the concepts, the reader is referred to [3], [7], [8].

A. Languages and Automata

Event sequences and languages are a simple means to describe discrete system behaviours. Their basic building blocks are *events*, 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, including the *empty string* ε. The length of a string $s \in \Sigma^*$ is denoted $|s|$, and the *concatenation* of two strings $s, t \in \Sigma^*$ is written as st.

A *language* over Σ is any subset $L \subseteq \Sigma^*$. A language L is called *prefix-closed* if for all $s, t \in \Sigma^*$, it holds that $st \in L$ implies $s \in L$. In this paper, which discusses safety properties, only prefix-closed languages are considered.

This paper considers models expressed as *deterministic automata* $G = \langle \Sigma, X, \delta, x^\circ \rangle$ where Σ is the alphabet, X is the set of *states*, $\delta\colon X \times \Sigma \to X$ is the (partial) *transition function*, and $x^\circ \in X$ is the *initial state*. The transition function is extended in the natural way to accept strings $s \in \Sigma^*$. Given this notation, the *language* or *prefix-closed behaviour* of an automaton G is

$$\mathcal{L}(G) \;=\; \{\, s \in \Sigma^* \mid \delta(x^\circ, s) \text{ is defined} \,\}. \qquad (1)$$

If two automata are running in parallel, lock-step synchronisation in the style of [9] is used. The *synchronous product* of $G_1 = \langle \Sigma, X_1, \delta_1, x_1^\circ \rangle$ and $G_2 = \langle \Sigma, X_2, \delta_2, x_2^\circ \rangle$ is

$$G_1 \parallel G_2 \;=\; \langle \Sigma, X_1 \times X_2, \delta, (x_1^\circ, x_2^\circ) \rangle \qquad (2)$$

where

$$\delta((x_1, x_2), \sigma) \;=\; (\delta_1(x_1, \sigma), \delta_2(x_2, \sigma)) \qquad (3)$$

provided that $\delta_1(x_1, \sigma)$ and $\delta_2(x_2, \sigma)$ are both defined.

In synchronous composition, shared events must be executed by all automata together. If two automata to be composed have different alphabets, in order to apply the above definition, the alphabets first have to be *extended* by adding selfloops on the missing events.

B. Projection

In large systems composed of several automata, there typically exist some events that are used exclusively by only one automaton or can be abstracted away for other reasons. To this end, the alphabet Σ is partitioned into the set Υ of events to be abstracted away and the set Ω of events to be retained. Typically, Υ consists of the events used exclusively by the automaton considered.

Natural projection

$$P_{\Sigma\to\Omega}\colon\ \Sigma^* \to \Omega^* \tag{4}$$

is the operation that removes all events not in Ω (i.e., all events in Υ) from a string. This operation is naturally extended to operate on languages as well. *Inverse projection*, defined for languages,

$$P^{-1}_{\Sigma\leftarrow\Omega}\colon\ 2^{\Omega^*} \to 2^{\Sigma^*}\ , \tag{5}$$

inserts events in Υ into all strings at all possible positions. Where there is no danger of ambiguity, the source alphabet may be omitted, and P_Ω or P^{-1}_Σ may be written instead of $P_{\Sigma\to\Omega}$ or $P^{-1}_{\Sigma\leftarrow\Omega}$, respectively.

Projection and inverse projection can also be applied to automata. Projection is typically implemented by first replacing all occurrences of the events to be hidden (the events in Υ) by the *silent* event τ, and subsequently using a determinisation algorithm [8] to make the resultant nondeterministic automaton deterministic. Inverse projection is achieved by adding selfloops with the hidden events to all the states of the automaton.

C. Language Inclusion and Controllability

When modelling reactive systems as automata or languages, it is of interest to check whether a system behaviour $L \subseteq \Sigma^*$ is contained in a given requirements language $K \subseteq \Sigma^*$. This is the question of *language inclusion* [1], formally written as

$$L \subseteq K\ , \tag{6}$$

which is the simplest form of the *safety properties* considered in this paper.

When control is taken into account, the slightly more complicated question of *controllability* arises. To describe this question, the set Σ of events is partitioned into the set Σ_c of *controllable events* and the set 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* [3], [7] with respect to L if

$$K\Sigma_u \cap L \subseteq K\ . \tag{7}$$

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.

III. Algorithms

This section outlines a method of automatically simplifying a model using projection, such that given language inclusion properties are preserved in the abstracted model. After introducing the underlying theoretical result in Sect. III-A, algorithms to check language inclusion are explained in Sect. III-B, III-C, and III-D. In addition, Sect. III-E discusses how to use these algorithms to verify controllability.

A. Theoretical Background

When checking whether a system behaviour L satisfies requirements K, the requirements typically do not use all the events in the system behaviour L. Then those events that occur only in L can be projected out, transforming the language inclusion problem (6) into the equivalent but simpler problem of checking whether

$$P(L) \subseteq K\ . \tag{8}$$

This observation is formalised in the following simple proposition, which forms the main justification for the algorithms in this section.

Proposition 1: Let $\Omega \subseteq \Omega' \subseteq \Sigma$, and let $L \subseteq \Sigma^*$ and $K \subseteq \Omega^*$ be two languages. The following two statements are equivalent.

$$L \subseteq P^{-1}_{\Sigma\leftarrow\Omega}(K) \iff P_{\Sigma\to\Omega'}(L) \subseteq P^{-1}_{\Omega'\leftarrow\Omega}(K) \tag{9}$$

Proof: First assume $L \subseteq P^{-1}_{\Sigma\leftarrow\Omega}(K)$. By monotonicity of projection, it follows immediately that $P_{\Sigma\to\Omega'}(L) \subseteq P_{\Sigma\to\Omega'}P^{-1}_{\Sigma\leftarrow\Omega}(K) = P^{-1}_{\Omega'\leftarrow\Omega}(K)$.

Second assume $P_{\Sigma\to\Omega'}(L) \subseteq P^{-1}_{\Omega'\leftarrow\Omega}(K)$, and let $s \in L$. Then it follows that $P_{\Sigma\to\Omega'}(s) \in P_{\Sigma\to\Omega'}(L) \subseteq P^{-1}_{\Omega'\leftarrow\Omega}(K)$, and therefore $s \in P^{-1}_{\Sigma\leftarrow\Omega'}P^{-1}_{\Omega'\leftarrow\Omega}(K) = P^{-1}_{\Sigma\leftarrow\Omega}(K)$. ■

In practical applications, the system to be checked for language inclusion can be very complex. It then is to be checked whether a composed system $G = G_1 \parallel \cdots \parallel G_n$ satisfies the behaviour of a given property automaton R, i.e., whether

$$\mathcal{L}(G_1 \parallel \cdots \parallel G_n) \subseteq \mathcal{L}(R)\ . \tag{10}$$

According to proposition 1, this check can be simplified by first calculating the projection of the left-hand side in the above expression. The following section presents an algorithm to compute this projection without first calculating the synchronous product $G_1 \parallel \cdots \parallel G_n$.

B. Iterative Projection

Most system models are designed in a structured way, such that most events are used only in a part of the model. Such structure can be exploited to compute the projection of composed systems. Using the well-known fact that

$$P_\Omega(G_1 \parallel G_2) = P_\Omega(G_1) \parallel G_2 \tag{11}$$

Let $\mathbf{G} = \{G_1, \ldots, G_n\}$ be the set of automata to be simplified, and let $\Omega \subseteq \Sigma$ be the set of events to be retained. The algorithm uses a set $\mathbf{C}$ of candidate sets of automata, where each element in $\mathbf{C}$ represents a minimal set of automata to cover at least one event.

```
1. For all events $\sigma \in \Sigma \setminus \Omega$, find the set $C_\sigma$ of all automata in $\mathbf{G}$ whose alphabet contains $\sigma$. Then let $\mathbf{C} := \{\, C_\sigma \mid \sigma \in \Sigma \setminus \Omega$ and $C_\sigma \neq \mathbf{G} \,\}$.
2. If $\mathbf{C} = \emptyset$, then stop and return $\mathbf{G}$.
3. Choose and remove a candidate $C_\sigma \in \mathbf{C}$ and calculate its synchronous product $C_{\mathrm{prod}} = \|_{G_i \in C_\sigma} G_i$. Abort this computation if $10 \cdot \mathtt{max}_{\mathrm{states}}$ states are exceeded, in which case go to 8.
4. Let the set $\Omega_\sigma$ of kept events be equal to $\Omega$ plus all events which occur in some automaton in $\mathbf{G} \setminus C_\sigma$.
5. Compute projection and obtain the determinised automaton $C_{\mathrm{det}} = P_{\Omega_\sigma}(C_{\mathrm{prod}})$. Abort this computation if $\mathtt{max}_{\mathrm{states}}$ states are exceeded, in which case go to 8.
6. Compute the minimal version $C_{\min}$ of $C_{\mathrm{det}}$.
7. Set $\mathbf{G} := (\mathbf{G} \setminus C_\sigma) \cup \{C_{\min}\}$ and go to 1.
8. If the maximum number of states has been exceeded $\mathtt{max}_{\mathrm{attempts}}$ times, stop and return $\mathbf{G}$. Otherwise go to 2.
```

Fig. 1. Iterative Projection Algorithm

if the automaton G_2 only uses events in Ω, the projection in (10) can be computed in a step-by-step way.

Fig. 1 presents an *Iterative Projection Algorithm* to simplify an automata model based on this idea. This algorithm simply identifies all candidate sets of automata, from which an event can be projected out. This is done in step 1. In step 3, a candidate set is chosen heuristically, based on its expected synchronous product size after projection. The heuristic takes into account the number of states of the automata in the candidate set, and the number of events that can be projected or need to be retained. Once a candidate has been selected, its automata are composed and projected, determinised using the subset construction algorithm [8], and minimised using Hopcroft's minimisation algorithm [10]. Then the selected set of candidate automata is replaced by the minimised automaton, and the process is started again.

Although the size of automata can often be reduced by means of projection, this is not in general the case. Therefore, the *Iterative Projection Algorithm* is parametrised by the thresholds $\mathtt{max}_{\mathrm{states}}$ and $\mathtt{max}_{\mathrm{attempts}}$. If the number of states becomes too large during step 3 or 5, the computation is aborted and the next best candidate set is tried instead. This algorithm therefore does not always find the projection $P_\Omega(G_1 \parallel \cdots \parallel G_n)$: in general, it returns an approximation $P_{\Omega'}(G_1 \parallel \cdots \parallel G_n)$ where $\Omega \subseteq \Omega' \subseteq \Sigma$, which is completely sufficient to apply proposition 1.

To estimate the complexity of the *Iterative Projection Algorithm*, it is observed that there may be one projection step for each event in $\Upsilon = \Sigma \setminus \Omega$, but each step may require a failed attempt for all other events. Thus, the number of projection attempts is bounded by $\frac{1}{2}|\Upsilon|(|\Upsilon| + 1)$. For each attempt, the number of states is bounded by the $\mathtt{max}_{\mathrm{states}}$ threshold, so synchronous composition in step 3 visits at most $10 \cdot \mathtt{max}_{\mathrm{states}} \cdot |\Sigma|$ transitions, determinisation in step 5 visits at most $\mathtt{max}_{\mathrm{states}} \cdot |\Sigma|$ transitions, and the complexity of minimisation in step 6 is $O(|\Upsilon| \cdot \mathtt{max}_{\mathrm{states}} \cdot \log \mathtt{max}_{\mathrm{states}})$ using the algorithm of [10]. Therefore, the overall complexity of the *Iterative Projection Algorithm* is bounded by

$$O(\mathtt{max}_{\mathrm{states}} \cdot \log \mathtt{max}_{\mathrm{states}} \cdot |\Upsilon|^2 \cdot |\Sigma|) \ . \tag{12}$$

Let $G = \langle \Sigma, X, \delta, x^\circ \rangle$ and $t' = \sigma_0 \sigma_1 \ldots \sigma_{n-1} \in P_\Omega \mathcal{L}(G)$. The algorithm uses a queue $\mathbf{Open}$ of triples $\langle x, i, t \rangle \in X \times \mathbb{N} \times \Sigma^*$ where i represents the number of events processed in t', and t represents the trace in G built up so far. It also uses a set $\mathbf{Visited}$ of pairs $\langle x, i \rangle \in X \times \mathbb{N}$.

```
1. Append $\langle x^\circ, 0, \varepsilon \rangle$ to $\mathbf{Open}$, and add $\langle x^\circ, 0 \rangle$ to $\mathbf{Visited}$.
2. Remove the first triple $\langle x, i, t \rangle$ from $\mathbf{Open}$.
3. If $i = n$, then stop and return $t$.
4. If $\delta(x, \sigma_i) = y$, then append $\langle y, i+1, t\sigma_i \rangle$ to $\mathbf{Open}$, and add $\langle y, i+1 \rangle$ to $\mathbf{Visited}$ if not already present.
5. For every event $\upsilon \in \Sigma \setminus \Omega$, if $\delta(x, \upsilon) = y$, then append $\langle y, i, t\upsilon \rangle$ to $\mathbf{Open}$, and add $\langle y, i \rangle$ to $\mathbf{Visited}$ if not already present.
6. Go to 2.
```

Fig. 2. Extend Trace Algorithm

C. Extracting Counterexamples

If a requirement is not satisfied, it is of great interest to provide a *counterexample* that explains to the user why the requirement is not satisfied. A language inclusion counterexample simply is a trace $t \in \Sigma^*$ accepted by the system G but not by the requirements R,

$$t \in \mathcal{L}(G) \quad \text{but} \quad t \notin \mathcal{L}(R) \ . \tag{13}$$

The computation of a counterexample is straightforward for monolithic state exploration, but after projection has been applied to the model, some additional effort is needed.

If proposition 1 is used, and the projected model $P_\Omega(G)$ is found not to satisfy the requirements, then only a trace $t' \in \mathcal{L}(P_\Omega(G))$ is available, which is not necessarily a counterexample in the original system. Yet, it is known that, for any trace $t' \in \mathcal{L}(P_\Omega(G))$ there exists a trace $t \in \mathcal{L}(G)$ such that $t' = P_\Omega(t)$. Since the requirements R do not use any of the events projected out, this trace t is a counterexample to the original system.

Fig. 2 shows the *Extend Trace Algorithm* for finding such a trace. This algorithm basically consists of a breadth-first search through the automaton G. All possibilities of executing the events of the projected trace t', interleaved with events in $\Upsilon = \Sigma \setminus \Omega$, are attempted until a trace t that matches t' is found. To save time on string copying, instead of creating a new trace for each step, a trace $t\sigma$ can be represented by the new event σ plus a pointer to the old trace t. Because the algorithm only examines a state if it has

Let $\mathbf{G} = \{G_1, \dots, G_n\}$ be a set of automata to be checked, and let R be a requirements automaton.

1. Use the *Iterative Projection Algorithm* (Fig. 1) to simplify $\mathbf{G}$ into $\mathbf{G}'$.
2. Use a monolithic state-space search to determine whether $G' = \|_{G_i \in \mathbf{G}'} G_i$ satisfies R.
3. If G' does not satisfy R, use the *Extend Trace Algorithm* (Fig. 2) to extend the counterexample t' found in step 2 to a counterexample t for the original system.

Fig. 3. Monolithic Projecting Language Inclusion Check Algorithm

Let $\mathbf{G} = \{G_1, \dots, G_n\}$ be a set of automata to be checked, and let R be a requirements automaton.
The algorithm uses a subset $\mathbf{H} \subseteq \mathbf{G}$ of automata used at the current step, which is initially empty.

1. Invoke the *Monolithic Projecting Language Inclusion Check Algorithm* (Fig. 3) to check whether $\mathbf{H}$ satisfies R.
2. If $\mathbf{H}$ satisfies R, then $\mathbf{G}$ satisfies R: return `true`.
3. Let t be the counterexample obtained in step 1, and let $\mathbf{N} := \{\, G_i \in \mathbf{G} \setminus \mathbf{H} \mid t \notin \mathcal{L}(G_i) \,\}$.
4. If $\mathbf{N} = \emptyset$, then $\mathbf{G}$ does not satisfy R: return `false` along with the counterexample t.
5. Pick a subset $\mathbf{N}' \subseteq \mathbf{N}$, let $\mathbf{H} := \mathbf{H} \cup \mathbf{N}'$, and go to 1.

Fig. 4. Compositional Projecting Language Inclusion Check Algorithm

not already been explored for the same depth through t', this algorithm has a worst-case complexity of $O(|X| \cdot |\Upsilon| \cdot |t'|)$.

The *Extend Trace Algorithm* can be used to find a trace in the original model obtained from a single projection step. If the model has been simplified in several steps by the *Iterative Projection Algorithm* (Fig. 1), the trace needs to be extended several times. That is, the algorithm in Fig. 2 is invoked once for each projection step. In the end, this results in a trace accepted by the original, unprojected model.

D. Verifying Language Inclusion

A simple way to verify language inclusion using proposition 1 is to first apply the *Iterative Projection Algorithm* (Fig. 1) to all the automata in the system in order to reduce their size. After projection, the synchronous product hopefully is small enough to be searched exhaustively. The only requirement for this approach to work is that none of the events in the alphabet of the property may be hidden during projection, because these events are essential for the final check. This *Monolithic Projecting Language Inclusion Check Algorithm* is shown in Fig. 3.

If the synchronous product state space after projection is still too large, the algorithm may be combined with the method of incremental verification proposed in [1]. This method is based on the fact that, in order to prove or disprove language inclusion, it is enough to compose only a subset of all the automata in the system. The *Compositional Projecting Language Inclusion Check Algorithm* is shown in Fig. 4.

According to [1], this algorithm tries to find a subsystem $\mathbf{H}$ of the system $\mathbf{G}$ to be checked, which satisfies the requirements R. If such a subsystem $\mathbf{H}$ can be found, the requirements are also satisfied for the complete system. At each step, if the subsystem $\mathbf{H}$ does not satisfy the requirements, then there is a counterexample accepted by $\mathbf{H}$ but not by R. Clearly, if this counterexample is accepted by all automata in $\mathbf{G}$, then $\mathbf{G}$ does not satisfy the requirements. Otherwise some of the automata not accepting the counterexample are added to the subsystem $\mathbf{H}$, and the check is attempted again. A more detailed discussion and justification for these steps can be found in [1], along with a list of heuristics for picking automata to add in step 5. The only new aspect in Fig. 4 is that a *Monolithic Projecting Language Inclusion Check Algorithm* is used instead of a monolithic state-space search in step 1.

Obviously, the *Compositional Projecting Language Inclusion Check Algorithm* algorithm requires several runs of the *Iterative Projection Algorithm* to simplify different sets of automata. However, all these sets are similar and have many automata in common. This suggests the use of caching to avoid the repeated computation of the projections of the same automata. The potential of caching in this algorithm is discussed in more detail in [6].

The complexity of the *Monolithic Projecting Language Inclusion Check Algorithm* is dominated by the complexity of the monolithic state-space search in step 2, which is $O(|\Omega| \cdot |X'|)$, where X' is the state space of the synchronous product of $\mathbf{G}'$. As shown in (12), the complexity of the *Iterative Projection Algorithm* in step 1 is determined by the number of events and the value of $\mathtt{max}_{\mathrm{states}}$, which are significantly smaller then the number of states in the synchronous product. Thus, the projection step does not adversely impact complexity, but if the state space can be reduced by projection, i.e., if $|X'| \ll |X|$, there is the chance for significant improvement.

E. Verifying Controllability

The algorithms outlined in this section can only verify language inclusion, and are not directly applicable to controllability, which is more common in the context of discrete event systems. This section shows how controllability problems can be converted into equivalent language inclusion problems, so the projecting language inclusion check algorithms can be used to solve these problems as well.

The problem of verifying controllability consists of checking whether a given specification language K is controllable with respect to a given plant language L. The two languages typically are given in a modular fashion,

$$K = \mathcal{L}(R) = \mathcal{L}(R_1 \| \cdots \| R_m) \; ; \tag{14}$$

$$L = \mathcal{L}(G) = \mathcal{L}(G_1 \| \cdots \| G_n) \; . \tag{15}$$

To transform the problem of checking whether $\mathcal{L}(R)$ is controllable with respect to $\mathcal{L}(G)$ into a language inclusion problem, for each uncontrollable event υ and each specification automaton R_i, a new event $\gamma_{R_i,\upsilon}$ is introduced with the

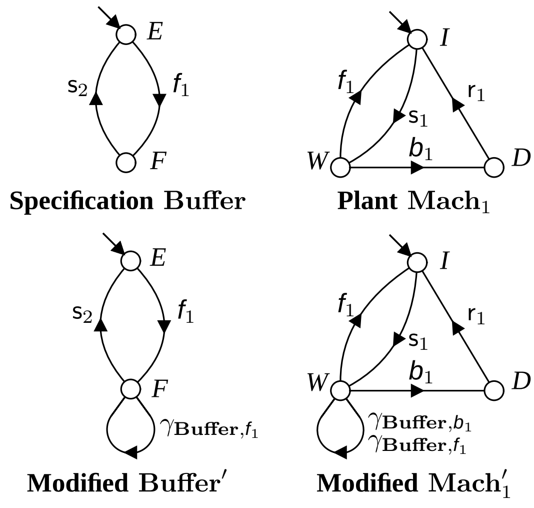

Fig. 5. Transformation of a controllability problem into language inclusion. Events f_1 and b_1 are uncontrollable, all other events are controllable.

intended meaning that a controllability problem can occur with respect to event υ in specification R_i. The set of all introduced events is denoted by Γ.

Each specification automaton R_i is transformed into a modified specification R_i' by adding the event $\gamma_{R_i,\upsilon}$ to the alphabet for each uncontrollable event υ in the alphabet of R_i, and by adding selfloop transitions

$$x \xrightarrow{\gamma_{R_i,\upsilon}} x \tag{16}$$

to each state x of R_i *without* outgoing υ-transition. Similarly, each plant automaton G_j is translated into a modified plant G_j' by adding the events $\gamma_{R_i,\upsilon}$ to the alphabet for each uncontrollable event υ in the alphabet of G_j and for all specification automata R_i, and by adding selfloop transitions

$$x \xrightarrow{\gamma_{R_i,\upsilon}} x \tag{17}$$

to each state x of G_j *with* an outgoing υ-transition. Fig. 5 shows the result of applying this transformation to two automata of the classical "small factory" example from [2].

Proposition 2: Let $R = R_1 \parallel \cdots \parallel R_m$ and $G = G_1 \parallel \cdots \parallel G_n$ be composed of automata with alphabet Σ, and let $R' = R_1' \parallel \cdots \parallel R_m'$ and $G' = G_1' \parallel \cdots \parallel G_n'$ be transformed automata as explained above, with introduced events Γ such that $\Sigma \cap \Gamma = \emptyset$. Then R is controllable with respect to G if and only if $\mathcal{L}(R' \parallel G') \subseteq \Sigma^*$.

Proof: First assume that R is not controllable with respect to G. According to (7), there exist $s \in \Sigma^*$ and $\upsilon \in \Sigma_u$ such that $s\upsilon \in \mathcal{L}(R)\Sigma_u \cap \mathcal{L}(G)$ but $s\upsilon \notin \mathcal{L}(R)$. Thus, $s\upsilon \notin \mathcal{L}(R_k)$ for some k. By construction, $s\gamma_{R_k,\upsilon} \in \mathcal{L}(R_k')$ and, since $\gamma_{R_k,\upsilon}$ is not in the alphabet of any other R_j', $s\gamma_{R_k,\upsilon} \in P_{\Sigma\cup\Gamma}^{-1}\mathcal{L}(R_j)$ for all $j \neq k$. Furthermore, since $s\upsilon \in \mathcal{L}(G)$, it follows by construction that $s\gamma_{R_k,\upsilon} \in \mathcal{L}(G')$. Thus $s\gamma_{R_k,\upsilon} \in \mathcal{L}(R' \parallel G')$ but $s\gamma_{R_k,\upsilon} \notin \Sigma^*$, i.e., $\mathcal{L}(R' \parallel G') \not\subseteq \Sigma^*$.

Second assume that $\mathcal{L}(R' \parallel G') \not\subseteq \Sigma^*$. Then there exist $s \in \Sigma^*$ and $\gamma_{R_k,\upsilon} \in \Gamma$ such that $s\gamma_{R_k,\upsilon} \in \mathcal{L}(R' \parallel G')$, where $\upsilon \in \Sigma_u$. By construction of R' and G' it follows that $s \in \mathcal{L}(R \parallel G)$, $s\upsilon \in \mathcal{L}(G)$, and $s\upsilon \notin \mathcal{L}(R_k) \supseteq \mathcal{L}(R)$. Then, R is not controllable with respect to G by (7). ■

TABLE I

TIME TO SOLVE MODELS

Model	Aut	Incremental	Compositional Projecting	Monolithic Projecting
big_bmw	31	0.04 s	0.18 s	0.21 s
tbed_ctct	84	0.13 s	0.22 s	
tbed_nocoll	84	0.61 s	12.97 s	1.42 s
tbed_uncont	84		30.87 s	1.45 s
profisafe_i4	80	0.07 s	0.13 s	
rhone_tough	61		8.60 s	2.34 s

In this way, the problem of checking whether R is controllable with respect to G is transformed into the problem of checking whether behaviour of the modified system $R' \parallel G'$ is contained in Σ^*. This is a very simple language inclusion check—the language Σ^* can be represented by the one-state requirement automaton $R_{\Sigma^*} = \langle \Gamma, \{x^\circ\}, \emptyset, x^\circ \rangle$.

The translation of the models R and G into R' and G' can be implemented in a single preprocessing pass over the states of the automata and therefore does not have any impact on the complexity of the *Monolithic* or *Compositional Projecting Language Inclusion Check Algorithm*.

IV. Experimental Results

The *Monolithic* and *Compositional Projecting Language Inclusion Check Algorithms* have been implemented in the DES software tool Supremica [11] and tested on the same set of industrial-scale models as used previously in [1]. All these problems have been solved, and in addition, the projection-based algorithms have successfully computed a counterexample for the model called rhone_tough, which cannot be solved by the algorithms in [1]. This model represents a faulty variant of the AIP automated manufacturing system [12], [13]. It is not controllable, and its shortest counterexample has 107 steps and affects almost all the 61 automata in the model.

Table I shows the results for verifying controllability of selected models. For each model, the number of automata in the model and the times taken by the incremental method of [1] and the *Monolithic* and *Compositional Projecting Language Inclusion Check Algorithms* are shown. In all cases, if it was required to explicitly compose a synchronous product with more than $2 \cdot 10^6$ states, the run was aborted: this is represented by an empty entry in the table. All tests were run on a 1.8 GHz desktop computer with 1 GB of RAM.

Fig. 6 represents the data in Table I graphically. It shows that the incremental method [1] outperforms both projection-based methods in all cases where it finds a solution. Apparently, synchronous products are much easier to compute than projection, so it is better to construct large state spaces as long as they fit in memory. However, the projecting algorithms find counterexamples for the tbed_uncont and rhone_tough models, which cannot be solved by the incremental method. These models require the majority of their automata to be composed, and the incremental method fails to find a subsystem small enough to fit in memory.

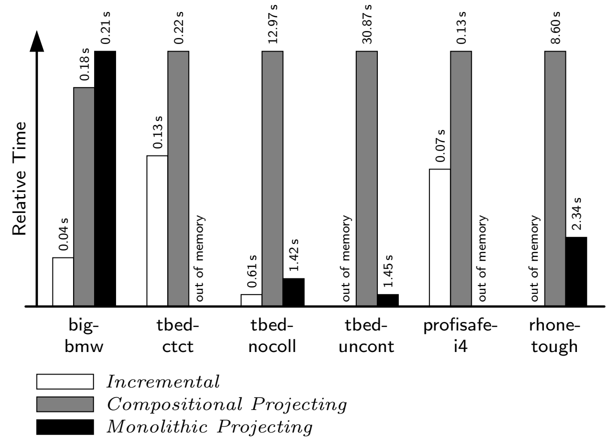

Fig. 6. Relative Time to Solve Models

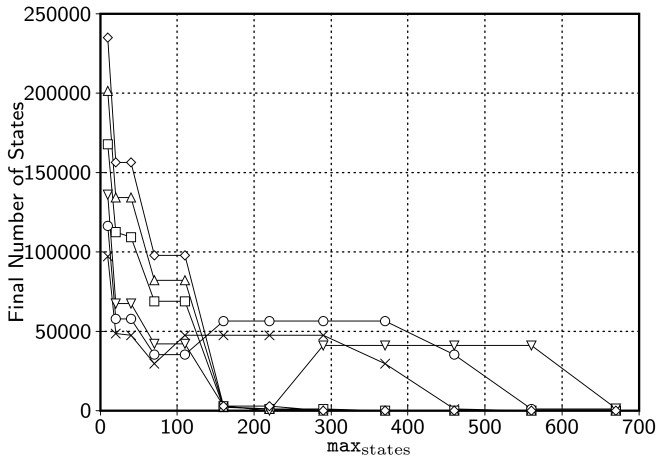

Fig. 7. States in simplified PROFIsafe models vs. $\mathtt{max}_{\mathrm{states}}$.

While the *Compositional Projecting Language Inclusion Check Algorithm* is the only method consistently successful at solving all the problems considered within the given resource limit, the *Monolithic Projecting Language Inclusion Check Algorithm* tends to be faster whenever it is successful. Again, the need to compose a large portion of the system seems to cause high overhead for the compositional algorithm in spite of caching, so it is better to project and compose the complete model as long as this is possible. Maybe the models in the test suite are not big enough to give the compositional projecting algorithm an advantage.

Fig. 7 shows the effect of increasing the $\mathtt{max}_{\mathrm{states}}$ threshold of the *Iterative Projection Algorithm* (Fig. 1) on the number of states in the final synchronous product to verify language inclusion for some large PROFIsafe models [14], [15]. The $\mathtt{max}_{\mathrm{states}}$ parameter is the largest size the algorithm allows a projected automaton to get before giving up. The chart shows a steep decrease in the number of states constructed for relatively small values of $\mathtt{max}_{\mathrm{states}}$ that quickly fades off. A small value of $\mathtt{max}_{\mathrm{states}}$ in the range 50–200 seems to give the best returns, while larger values tend to increase the effort for projection and the number of aborted attempts without achieving much more state-space reduction.

V. Conclusions

This paper presents algorithms to enhance the performance of the verification of safety properties of automata models using language projection. While not changing the complexity of monolithic verification algorithms, the experimental results show that language projection has the potential to reduce state spaces and verify some models that cannot be solved otherwise. This is particularly the case when very long counterexamples need to be computed that involve a large part of the system verified.

In the future, the authors would like to further improve the projecting verification algorithms. There still are many ways to be explored how projection can be combined with monolithic and compositional verification, or with symbolic algorithms [4]. In addition, there are some possibilities to analyse and simplify a model and make it possible to project out more events. Finally, the author's present projection algorithm, which is the main bottleneck of the method, can be improved using more efficient data structures.

References

[1] 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.

[2] 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.

[3] W. M. Wonham, "Supervisory control of discrete-event systems," Systems Control Group, Dept. of Electrical Engineering, University of Toronto, Ontario, Canada; at http://www.control.utoronto.edu/DES/, 2007.

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

[5] 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.

[6] S. Ware, "Modular finite-state machine analysis," Honours project report, Dept. of Computer Science, University of Waikato, 2007.

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

[8] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.

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

[10] J. E. Hopcroft, "An $n \log n$ algorithm for minimizing states in a finite automaton," in *Theory of Machines and Computations*, Z. Kohavi and A. Paz, Eds. New York, NY, USA: Academic Press, 1971, pp. 189–196.

[11] K. Åkesson, M. Fabian, H. Flordal, and R. Malik, "Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems," in *Proc. 8th Int. Workshop on Discrete Event Systems, WODES'06*, Ann Arbor, MI, USA, July 2006, pp. 384–385.

[12] 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.

[13] R. Song, "Symbolic synthesis and verification of hierarchical interface-based supervisory control," Master's thesis, Dept. of Computing and Software, McMaster University, Hamilton, ON, Canada, 2006.

[14] 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.

[15] ——, "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.