An Algorithm for Weak Synthesis Observation Equivalence for Compositional Supervisor Synthesis

Sahar Mohajerani * **Robi Malik** ** **Martin Fabian** *

* *Department of Signals and Systems, Chalmers University of Technology, Göteborg, Sweden, (e-mail: {mohajera,fabian}@chalmers.se)*
** *Department of Computer Science, University of Waikato, Hamilton, New Zealand, (e-mail: robi@waikato.ac.nz)*

Abstract: This paper proposes an algorithm to simplify automata in such a way that compositional synthesis results are preserved in every possible context. It relaxes some requirements of synthesis observation equivalence from previous work, so that better abstractions can be made. The paper describes the algorithm, adapted from known bisimulation equivalence algorithms, for the improved abstraction method. The algorithm has been implemented in the DES software tool Supremica and has been used to compute modular supervisors for several large benchmark examples. It successfully computes modular supervisors, even for systems with more than 10^{12} reachable states.

Keywords: Discrete event systems, supervisory control theory, abstraction, compositional synthesis

1. INTRODUCTION

Compositional methods are of great interest in *supervisory control theory* (Ramadge and Wonham, 1989), firstly in order to find more comprehensible supervisor representations, and secondly to overcome the problem of *state-space explosion* for systems with a large number of components.

Compositional synthesis (Flordal *et al.*, 2007; Malik and Flordal, 2008; Mohajerani *et al.*, 2011*a*) seeks to compute a supervisor for a large discrete event system by *abstraction*. Individual system components are replaced by simpler versions obtained from abstraction, and synchronous composition is computed step-by-step on abstracted components. At each step, partial supervisors are computed, which in the end give a modular supervisor for the original system. In this way, the state-space explosion is mitigated, making synthesis possible for very large systems.

Several methods of compositional synthesis exist that differ in how abstractions are computed. *Natural projection* is easy to compute, but it is restrictive and additional conditions must be imposed to ensure synthesis of least restrictive nonblocking supervisors (Feng and Wonham, 2006; Schmidt and Breindl, 2008). *Conflict-preserving* abstractions and *observation equivalence* are adequate for the synthesis of nonblocking supervisors, but least restrictiveness is only guaranteed if all observable events are retained in the abstraction (Malik *et al.*, 2007; Su *et al.*, 2010).

More recently (Mohajerani *et al.*, 2011*b*) proposed *synthesis observation equivalence*, a stronger version of observation equivalence that is adequate for compositional synthesis of least restrictive supervisors. The approach has been integrated in a framework with other abstraction methods and can be used to compute supervisors for practical applications (Mohajerani *et al.*, 2011*a*).

This paper proposes a relaxation of synthesis observation equivalence, called *weak synthesis observation equivalence*, which allows to achieve more abstraction. A polynomial complexity algorithm to compute the abstraction is presented. After the preliminaries in Sect. 2, weak synthesis observation equivalence is defined in Sect. 3. The algorithm to compute it is given in Sect. 4, followed by experimental results in Sect. 5. Then Sect. 6 adds some concluding remarks.

2. PRELIMINARIES AND NOTATION

2.1 Events and Languages

Discrete event systems are modelled using events and languages (Ramadge and Wonham, 1989). Events are taken from a finite alphabet Σ, which is partitioned into two disjoint subsets, the set Σ_c of *controllable* events and the set Σ_u of *uncontrollable* events. Uncontrollable events are prefixed by an exclamation mark (!) in this paper. The special event $\omega \in \Sigma_c$ denotes *termination* and does not appear anywhere else but to mark such completions.

The set of all finite *traces* of elements of Σ, including the *empty trace* ε, is denoted by Σ^*. A subset $L \subseteq \Sigma^*$ is called a *language*. The concatenation of two traces $s, t \in \Sigma^*$ is written as st. A trace $s \in \Sigma^*$ is called a *prefix* of $t \in \Sigma^*$, written $s \sqsubseteq t$, if $t = su$ for some $u \in \Sigma^*$. For $\Omega \subseteq \Sigma$, the *natural projection* $P_\Omega \colon \Sigma^* \to \Omega^*$ is the operation that removes from traces $s \in \Sigma^*$ all events not in Ω.

2.2 Nondeterministic Automata

System behaviours are typically modelled by deterministic automata, but nondeterministic automata may arise as intermediate results during abstraction.

Definition 1. A (nondeterministic) finite-state automaton is a tuple $G = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$, where Σ is a finite set of events, Q is a finite set of *states*, $\rightarrow \subseteq Q \times \Sigma \times Q$ is the *state transition relation*, and $Q^\circ \subseteq Q$ is the set of *initial states*. G is *deterministic*, if $|Q^\circ| \leq 1$ and $x \xrightarrow{\sigma} y_1$ and $x \xrightarrow{\sigma} y_2$ always implies $y_1 = y_2$.

The transition relation is written in infix notation $x \xrightarrow{\sigma} y$, and is extended to traces in Σ^* by letting $x \xrightarrow{\varepsilon} x$ for all $x \in Q$, and $x \xrightarrow{s\sigma} z$ if $x \xrightarrow{s} y$ and $y \xrightarrow{\sigma} z$ for some $y \in Q$. Furthermore, $x \xrightarrow{s}$ means $x \xrightarrow{s} y$ for some $y \in Q$, and $x \rightarrow y$ means $x \xrightarrow{s} y$ for some $s \in \Sigma^*$. These notations also apply to state sets and to automata: $X \xrightarrow{s} Y$ for $X, Y \subseteq Q$ means $x \xrightarrow{s} y$ for some $x \in X$ and $y \in Y$, and $G \xrightarrow{s}$ means $Q^\circ \xrightarrow{s}$, etc. The language of automaton G is $\mathcal{L}(G) = \{\, s \in \Sigma^* \mid G \xrightarrow{s} \,\}$.

A special requirement is that states reached by the termination event ω do not have any outgoing transitions. This ensures that the termination event, if it occurs, is always the final event of any trace. The traditional set of marked states is $Q^\omega = \{\, x \in Q \mid x \xrightarrow{\omega} \,\}$ in this notation. For graphical simplicity, states in Q^ω are shaded in the figures of this paper instead of explicitly showing ω-transitions.

When automata are brought together to interact, lock-step synchronisation in the style of (Hoare, 1985) is used.

Definition 2. Let $G_1 = \langle \Sigma_1, Q_1, \rightarrow_1, Q_1^\circ \rangle$ and $G_2 = \langle \Sigma_2, Q_2, \rightarrow_2, Q_2^\circ \rangle$ be two automata. The *synchronous composition* of G_1 and G_2 is defined as

$$G_1 \parallel G_2 = \langle \Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, \rightarrow, Q_1^\circ \times Q_2^\circ \rangle \qquad (1)$$

where

$$\begin{aligned} &(x,y) \xrightarrow{\sigma} (x',y') \text{ if } \sigma \in (\Sigma_1 \cap \Sigma_2),\ x \xrightarrow{\sigma}_1 x',\ y \xrightarrow{\sigma}_2 y' ; \\ &(x,y) \xrightarrow{\sigma} (x',y) \text{ if } \sigma \in (\Sigma_1 \setminus \Sigma_2),\ x \xrightarrow{\sigma}_1 x' ; \\ &(x,y) \xrightarrow{\sigma} (x,y') \text{ if } \sigma \in (\Sigma_2 \setminus \Sigma_1),\ y \xrightarrow{\sigma}_2 y' . \end{aligned}$$

Another common automaton operation is the *quotient* modulo an equivalence relation on the state set.

Definition 3. Let $G = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$ be an automaton and let $\sim \subseteq Q \times Q$ be an equivalence relation. The *quotient automaton* of G modulo $\sim$ is

$$G/\!\sim = \langle \Sigma, Q/\!\sim, \rightarrow/\!\sim, \tilde{Q}^\circ \rangle \ , \qquad (2)$$

where $\rightarrow/\!\sim = \{\, [x] \xrightarrow{\sigma} [y] \mid x \xrightarrow{\sigma} y \,\}$ and $\tilde{Q}^\circ = \{\, [x^\circ] \mid x^\circ \in Q^\circ \,\}$. Here, $[x] = \{\, x' \in Q \mid x \sim x' \,\}$ denotes the *equivalence class* of $x \in Q$, and $Q/\!\sim = \{\, [x] \mid x \in Q \,\}$ is the set of all equivalence classes modulo $\sim$.

2.3 Supervisory Control Theory

Given a *plant* automaton G and a *specification* automaton K, *supervisory control theory* (Ramadge and Wonham, 1989) provides a method to synthesise a supervisor that restricts the behaviour of the plant such that the specification is always fulfilled. Two common requirements for the supervisor are *controllability* and *nonblocking*.

Definition 4. Let G and K be two automata using the same alphabet Σ. K is *controllable* with respect to G if, for every trace $s \in \Sigma^*$, every state x of K, and every uncontrollable event $\upsilon \in \Sigma_u$ such that $K \xrightarrow{s} x$ and $G \xrightarrow{s\upsilon}$, it holds that $x \xrightarrow{\upsilon}$ in K.

Definition 5. An automaton $G = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$ is *nonblocking*, if for every state $x \in Q$ and every trace $s \in (\Sigma \setminus \{\omega\})^*$ such that $G \xrightarrow{s} x$ there exists $t \in \Sigma^*$ such that $x \xrightarrow{t\omega}$.

For a deterministic plant G and specification K, it is shown in (Ramadge and Wonham, 1989) that there exists a *least restrictive* controllable sublanguage

$$\sup\mathcal{C}_G(K) \subseteq \mathcal{L}(K) \qquad (3)$$

such that $\sup\mathcal{C}_G(K)$ is controllable with respect to G and nonblocking, and this language can be computed using a fixpoint iteration. For nondeterministic automata, synthesis produces a *subautomaton* instead of a language, and the controllability condition is modified accordingly (Malik and Flordal, 2008).

Definition 6. (Malik and Flordal, 2008) $G_1 = \langle \Sigma, Q_1, \rightarrow_1, Q_1^\circ \rangle$ is a *subautomaton* of $G_2 = \langle \Sigma, Q_2, \rightarrow_2, Q_2^\circ \rangle$, written $G_1 \subseteq G_2$, if $Q_1 \subseteq Q_2$, $\rightarrow_1 \subseteq \rightarrow_2$, and $Q_1^\circ \subseteq Q_2^\circ$.

Definition 7. (Malik and Flordal, 2008) Let $G = \langle \Sigma, Q_G, \rightarrow_G, Q_G^\circ \rangle$ and $K = \langle \Sigma, Q_K, \rightarrow_K, Q_K^\circ \rangle$ be automata such that $K \subseteq G$. Then K is called *controllable* in G if, for all states $x \in Q_K$ and $y \in Q_G$ and for every uncontrollable event $\upsilon \in \Sigma_u$ such that $x \xrightarrow{\upsilon}_G y$, it also holds that $x \xrightarrow{\upsilon}_K y$.

Traditionally, a supervisor synthesis problem involves both plants and specifications. However, a simple transformation (Flordal *et al.*, 2007) can make specifications regardable as plants, which makes it possible to consider plant-only synthesis problems.

Theorem 1. Let $G = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$. There exists a unique subautomaton $\sup\mathcal{CN}(G) \subseteq G$ such that $\sup\mathcal{CN}(G)$ is nonblocking and controllable in G, and such that for every subautomaton $S \subseteq G$ that is also nonblocking and controllable in G, it holds that $S \subseteq \sup\mathcal{CN}(G)$.

The synthesis result $\sup\mathcal{CN}(G)$ can be computed by iteratively removing blocking and uncontrollable states of the plant, until a fixpoint is reached, and restricting the original automaton G to these states.

Definition 8. (Malik and Flordal, 2008) The *restriction* of $G = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$ to $X \subseteq Q$ is

$$G_{|X} = \langle \Sigma, Q, \rightarrow_{|X}, Q^\circ \cap X \rangle \ , \qquad (4)$$

where $\rightarrow_{|X} = \{\, (x,\sigma,y) \in \rightarrow \mid x, y \in X \,\}$.

Definition 9. (Malik and Flordal, 2008) The *synthesis step operator* $\Theta_G \colon 2^Q \rightarrow 2^Q$ for $G = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$ is defined as $\Theta_G(X) = \Theta_G^{\text{cont}}(X) \cap \Theta_G^{\text{nonb}}(X)$, where

$$\Theta_G^{\text{cont}}(X) = \{\, x \in X \mid \forall \sigma \in \Sigma_u,\ x \xrightarrow{\sigma} y \text{ implies } y \in X \,\} \ ;$$

$$\Theta_G^{\text{nonb}}(X) = \{\, x \in X \mid x \xrightarrow{t\omega}_{|X} \text{ for some } t \in \Sigma^* \,\} \ .$$

Theorem 2. (Malik and Flordal, 2008) Let $G = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$. The synthesis step operator Θ_G has a greatest fixpoint $\text{gfp}\Theta_G = \hat{\Theta}_G \subseteq Q$, such that $G_{|\hat{\Theta}_G}$ is the greatest subautomaton of G that is both controllable in G and coreachable, i.e.,

$$\sup\mathcal{CN}(G) = G_{|\hat{\Theta}_G} \ . \qquad (5)$$

If the state set Q is finite, the sequence $X^0 = Q$, $X^{i+1} = \Theta_G(X^i)$ reaches this fixpoint in a finite number of steps, i.e., $\hat{\Theta}_G = X^n$ for some $n \geq 0$.

2.4 Compositional Synthesis

Most discrete event systems are *modular* and consist of several interacting components. Then the synthesis problem is to find a least restrictive, controllable and nonblocking supervisor for a set of plants,

$$G = \{G_1, G_2, \cdots, G_n\} \ . \tag{6}$$

Compositional methods seek to build the synchronous product incrementally, replacing individual components G_i by simpler *abstractions* G_i'. Such simplification typically exploits a set $\Upsilon \subseteq \Sigma$ of *local* events. These events are used only in the automaton being abstracted and contribute substantially to its simplification.

The abstraction relation must ensure that the results obtained from the abstracted model are the same as for the original model. An appropriate condition that works for compositional synthesis is *synthesis abstraction*.

Definition 10. (Mohajerani *et al.*, 2011*b*) Let G and H be deterministic automata with alphabet Σ. Then H is a *synthesis abstraction* of G with respect to $\Upsilon \subseteq \Sigma$, written $G \lesssim_{\text{synth},\Upsilon} H$, if for every deterministic automaton $T = \langle \Sigma_T, Q_T, \rightarrow_T, Q_T^\circ \rangle$ such that $\Sigma_T \cap \Upsilon = \emptyset$ the following holds,

$$\mathcal{L}(G \,\|\, T \,\|\, \text{sup}\mathcal{CN}(H \,\|\, T)) = \mathcal{L}(G \,\|\, T \,\|\, \text{sup}\mathcal{CN}(G \,\|\, T)) \ . \tag{7}$$

Synthesis abstraction requires that the supervisor synthesised from the abstracted automaton H together with the rest of the system T, yields the same language when controlling the system, as would the supervisor synthesised from the original automaton G together with T.

3. SYNTHESIS OBSERVATION EQUIVALENCE

Synthesis abstraction describes in a general way what kind of abstraction is feasible for a compositional synthesis. For practical use, it is necessary to have algorithmic means to simplify a given automaton in such a way that synthesis abstraction is preserved.

Bisimulation and *observation equivalence* are standard examples of branching equivalences, which are easy to compute (Milner, 1989). For two states to be equivalent, they must have the same nondeterministic future, which is described as an equivalence relation that is stable with respect to certain transition relations.

Definition 11. Let $\rightarrow \subseteq X \times X$ be a relation on a set X. An equivalence relation $\sim \subseteq X \times X$ is *stable* with respect to $\rightarrow$, if for all $x_1, x_2, y_1 \in X$ such that $x_1 \sim x_2$ and $x_1 \rightarrow y_1$ there exists $y_2 \in X$ such that $x_2 \rightarrow y_2$ and $y_1 \sim y_2$.

Definition 12. Let $G = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$ be an automaton. An equivalence relation $\sim \subseteq Q \times Q$ is called a *bisimulation* on G, if $\sim$ is stable with respect to $\xrightarrow{\sigma}$ for all $\sigma \in \Sigma$.

Definition 13. Let $G = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$ be an automaton with $\Sigma = \Omega \,\dot\cup\, \Upsilon$. An equivalence relation $\sim \subseteq Q \times Q$ is called an *observation equivalence* on G with respect to Υ, if $\sim$ is stable with respect to $\stackrel{\sigma}{\Rightarrow}$ for all $\sigma \in \Sigma$, where $x \stackrel{\sigma}{\Rightarrow} y$ if and only if $x \xrightarrow{t_1 P_\Omega(\sigma) t_2} y$ for some $t_1, t_2 \in \Upsilon^*$.

Unlike bisimulation, observation equivalence takes local events into account. Both equivalences preserve all temporal logic properties (Milner, 1989). Once an appropriate equivalence $\sim$ on G is found, the quotient automaton $G/\!\sim$ can be considered as an abstraction. For bisimulation this results in a synthesis abstraction, but it does not for observation equivalence (Mohajerani *et al.*, 2011*b*).

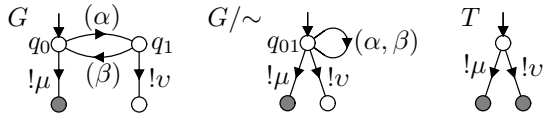

Fig. 1. Example of observation equivalence.

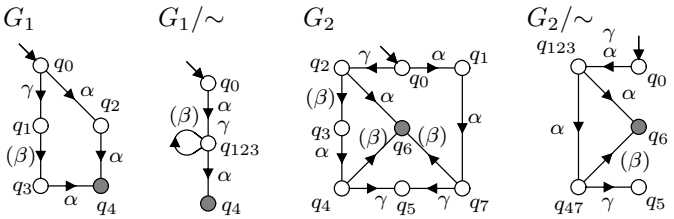

Fig. 2. Examples of synthesis observation equivalence.

Example 1. (Mohajerani *et al.*, 2011*b*) Consider automata G and T in Fig. 1, where $\Upsilon = \{\alpha, \beta\}$ and $\Sigma_u = \{!\mu, !\upsilon\}$. States q_0 and q_1 are observation equivalent and merging them results in $G/\!\sim$. However, $G/\!\sim \,\|\, T$ does not have the same least restrictive supervisor as $G \,\|\, T$. A supervisor for $G \,\|\, T$ can disable α to prevent blocking via $!\upsilon$, but after merging q_0 and q_1, disabling α is not enough to prevent the dangerous uncontrollable event $!\upsilon$.

While observation equivalence does not lead to synthesis abstraction in general, it can be strengthened such that synthesis results are preserved.

Definition 14. Let $G = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$ be an automaton with $\Sigma = \Omega \,\dot\cup\, \Upsilon$. An equivalence relation $\sim \subseteq Q \times Q$ is a *synthesis observation equivalence* on G with respect to Υ, if $\sim$ is stable with respect to $\stackrel{\Upsilon}{\Rightarrow}_{\text{soe}}$, to $\stackrel{\sigma}{\Rightarrow}_{\text{soe}}$ for each $\sigma \in \Sigma_c \cap \Omega$, and to $\stackrel{\upsilon}{\Rightarrow}_{\text{u}}$ for each $\upsilon \in \Sigma_u$, defined as follows.

- $x \stackrel{\Upsilon}{\Rightarrow}_{\text{soe}} y$ if there exists a path $z_0 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_k} z_k$, such that $z_0 = x$, and $z_k = y$, and $\tau_1, \ldots, \tau_k \in \Upsilon$, and $\tau_j \in \Sigma_c$ implies $x \sim z_j$ or $j = k$.
- $x \stackrel{\sigma}{\Rightarrow}_{\text{soe}} y$ if there exists a path $x = z_0 \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_k} z_k \xrightarrow{\sigma} y$ such that $\tau_1, \ldots, \tau_k \in \Upsilon$, and $\tau_j \in \Sigma_c$ implies $x \sim z_j$.
- $x \stackrel{\upsilon}{\Rightarrow}_{\text{u}} y$ if $x \xrightarrow{t_1 P_\Omega(\upsilon) t_2} y$ for some $t_1, t_2 \in (\Sigma_u \cap \Upsilon)^*$.

Example 2. Consider automaton G_1 in Fig. 2, where all events are controllable and $\Upsilon = \{\beta\}$. The equivalence relation $\sim$ with $q_1 \sim q_2 \sim q_3$ is a synthesis observation equivalence relation. For example, the transition $q_2 \xrightarrow{\alpha} q_4$ is matched by $q_1 \xrightarrow{\beta} q_3 \xrightarrow{\alpha} q_4$ where state q_3, reached by the local controllable event β, is equivalent to q_2. Merging the equivalent states results in the synthesis observation equivalent abstraction $G_1/\!\sim$ shown in Fig. 2.

Synthesis observation equivalence implies synthesis abstraction (Mohajerani *et al.*, 2011*b*). Def. 14 modifies observation equivalence based on event types. For uncontrollable events, $\Rightarrow_{\text{u}}$ is observation equivalence restricted to uncontrollable events. For controllable events, $\Rightarrow_{\text{soe}}$ does not allow local events *after* the controllable event. It is shown in the following how this condition can be relaxed to allow some local events after the controllable event.

Definition 15. Let $G = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$ be an automaton with $\Sigma = \Omega \dot\cup \Upsilon$. An equivalence relation $\sim \subseteq Q \times Q$ is a *weak synthesis observation equivalence* on G with respect to Υ, if $\sim$ is stable with respect to $\overset{\Upsilon}{\Rightarrow}_{\text{wsoe}}$, to $\overset{\sigma}{\Rightarrow}_{\text{wsoe}}$ for each $\sigma \in \Sigma_\text{c} \cap \Omega$, and to $\overset{\upsilon}{\Rightarrow}_\text{u}$ for each $\upsilon \in \Sigma_\text{u}$.

- $x \overset{\Upsilon}{\Rightarrow}_{\text{wsoe}} y$ if $x \overset{\Upsilon}{\Rightarrow}_{\text{soe}} z \overset{\Upsilon}{\Rightarrow}_\text{c} y$ for some $z \in Q$.
- $x \overset{\sigma}{\Rightarrow}_{\text{wsoe}} y$ if $x \overset{\sigma}{\Rightarrow}_{\text{soe}} z \overset{\Upsilon}{\Rightarrow}_\text{c} y$ for some $z \in Q$.
- $x \overset{\Upsilon}{\Rightarrow}_\text{c} y$ if there exists a path $z_0 \overset{\tau_1}{\rightarrow} \cdots \overset{\tau_k}{\rightarrow} z_k$, such that $z_0 = x$, and $z_k = y$, and $\tau_1, \ldots, \tau_k \in \Upsilon$, and $z_j \overset{u}{\rightarrow} z'$ for $u \in (\Sigma_\text{u} \cap \Upsilon)^*$ implies $z' \sim z_i$ for some $0 \leq i \leq k$, and $z_j \overset{\upsilon}{\Rightarrow}_\text{u} z'$ for $\upsilon \in \Sigma_\text{u} \cap \Omega$ implies $y \overset{\upsilon}{\Rightarrow}_\text{u} z''$ for some $z'' \sim z'$.

The modified relation $\Rightarrow_{\text{wsoe}}$ allows for a path of local events after a controllable event, if local uncontrollable transitions outgoing from the path lead to a state equivalent to a state on the path, and shared uncontrollable transitions are also possible in the end state of the path.

Example 3. Consider automaton G_2 in Fig. 2, with all events controllable and $\Upsilon = \{\beta\}$. An equivalence relation with $q_1 \sim q_2 \sim q_3$ and $q_4 \sim q_7$ is a weak synthesis observation equivalence. For example, transition $q_2 \overset{\alpha}{\rightarrow} q_6$ is matched by $q_1 \overset{\alpha}{\rightarrow} q_7 \overset{\beta}{\rightarrow} q_6$, and state q_7 has no uncontrollable transitions outgoing. Merging the equivalent states results in the synthesis observation equivalent abstraction $G_2/\!\sim$ shown in Fig. 2. Note that these states are not synthesis observation equivalent.

Weak synthesis observation equivalence can be shown to be more general than synthesis observation equivalence. Therefore, the following result confirms that both methods are feasible for compositional synthesis.

Theorem 3. Let $G = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$ and $\Upsilon \subseteq \Sigma$, and let $\sim$ be a weak synthesis observation equivalence on G with respect to Υ. Then $G \lesssim_{\text{synth},\Upsilon} G/\!\sim$.

4. ALGORITHM

Given an automaton $G = \langle \Sigma, Q, \rightarrow, Q^\circ \rangle$ and a set Υ of local events, a coarsest weak synthesis observation equivalence relation can be computed by a partition refinement algorithm similar to (Fernandez, 1990). This algorithm represents an equivalence relation as a *partition*, i.e., a set of *equivalence classes* each representing a set of equivalent states. The algorithm starts with an *initial partition* consisting of a single equivalence class, which is iteratively refined until a stable partition is reached. At each step, a *split* is performed on each known equivalence class C for each relation $\Rightarrow$ for which stability is required, separating states with $x \Rightarrow C$ from other states. This principle is shown in Algorithm 1.

The bisimulation algorithm (Fernandez, 1990) performs clever bookkeeping when classes are split, which reduces the need to check whether further splits are necessary and ensures an overall time complexity of $O(|\rightarrow| \log |Q|)$. For observation equivalence, the *transitive closure* of the local event transitions needs to be computed, and this transitive closure computation dominates complexity. A partition based on observation equivalence can be computed in $O(|\Sigma||Q|^3)$ time complexity (Bolognesi and Smolka, 1987).

The partition refinement algorithm uses several data structures to facilitate the splitting of classes (Fernandez, 1990).

Algorithm 1 Weak Synthesis Observation Equivalence

```
1: input G = ⟨Σ, Q, →, Q°⟩
2: partition ← {Q}
3: repeat
4:    for all splitter ∈ partition do
5:       for all σ ∈ Σ do
6:          SplitOn(partition, splitter, σ)
7:       end for
8:    end for
9: until there has been no further split
10: return partition
```

Algorithm 2 *SplitOn*($partition \subseteq 2^Q$, $splitter \subseteq Q$, $\sigma \in \Sigma$)

```
1: if σ ∈ Σ_u then
2:    for all end ∈ splitter do
3:       for all src --t1 P_Ω(σ) t2--> end with t1, t2 ∈ (Σ_u ∩ Υ)* do
4:          move src to split list in [src]
5:       end for
6:    end for
7: else
8:    for all end ∈ splitter do
9:       BS(σ, end)
10:   end for
11: end if
12: for all class ∈ partition do
13:   if class has a non-trivial split list then
14:      split class and update partition
15:   end if
16: end for
```

Each equivalence class is an object containing a list of the states in the class, and each state has a reference back to the class containing it. In addition, each equivalence class has a *split list* containing states to be split off from it.

The *SplitOn* algorithm (Algorithm 2) performs the splitting for paths leading to a target class, called a *splitter*. States with a path to the *splitter* based on each relation $\Rightarrow_{\text{wsoe}}$ and $\Rightarrow_\text{u}$ in Def. 15 are separated from states without such a path. This is done by visiting each state *end* in the *splitter* and searching backwards for all states *src* with appropriate paths to *end*. These states are put in the split list of their class. After exploring the predecessors of all *end* states, the split lists are checked in lines 12–16. Classes with an empty split list or a split list containing all states in the class are left unchanged, other classes are split and replaced by two new classes.

For uncontrollable events, the source states for $\Rightarrow_\text{u}$ are found by a standard backwards search using the precomputed transitive closure of the local uncontrollable transitions (lines 2–6), whereas for controllable events a special procedure *BS* is used to follow the paths generated by $\Rightarrow_{\text{wsoe}}$ (lines 8–10).

The *BS* algorithm (Algorithm 3) performs a backward search for a given controllable event σ and *end* state. It uses a *queue* of *search records* $\langle current, part, startclass \rangle$ containing a *current* state, whether the search is in the first or second *part* of the path, and the *startclass* of the path. The search starts with the *end* state, in the second part of the path, with an unassigned *startclass*. Thus, the initial search record $\langle end, 2, \mathsf{none} \rangle$ is added to the queue.

Algorithm 3 Backward search $BS(\sigma \in \Sigma_c, end \in Q)$

```
1:  queue ← ∅
2:  add ⟨end, 2, none⟩ to queue
3:  while queue ≠ ∅ do
4:    remove ⟨current, part, startclass⟩ from queue
5:    if part = 1 then
6:      if startclass ∈ {[current], none} then
7:        move current to split list in [current]
8:      end if
9:      for all transitions src -υ-> current with υ ∈ ϒ do
10:       if υ ∈ Σ_u then
11:         add ⟨src, 1, startclass⟩ to queue
12:       else if startclass ∈ {[current], none} then
13:         add ⟨src, 1, [current]⟩ to queue
14:       end if
15:     end for
16:   else
17:     for all transitions src -υ-> current with υ ∈ ϒ do
18:       controllable ← true
19:       for all src -u-> succ with u ∈ (Σ_u ∩ ϒ)* do
20:         if succ ∉ [src] ∪ [current] ∪ [end] then
21:           controllable ← false
22:         else
23:           for all succ -γ-> succ' with γ ∈ Σ_u ∩ Ω do
24:             if not [end] ⇒^γ_u [succ'] then
25:               controllable ← false
26:             end if
27:           end for
28:         end if
29:       end for
30:       if controllable then
31:         add ⟨src, 2, none⟩ to queue
32:       end if
33:     end for
34:     if σ ∈ ϒ then
35:       add ⟨current, 1, none⟩ to queue
36:     else
37:       for all transitions src -σ-> current do
38:         add ⟨src, 1, none⟩ to queue
39:       end for
40:     end if
41:   end if
42: end while
```

When exploring a *current* state in the first part of the path, it is first checked whether this state can be the start of a path generated by $\Rightarrow_{soe}$. This is possible if it belongs to the *startclass*, or if the *startclass* is unassigned, and in this case the *current* state is marked as a candidate to be split off from its class (lines 6–8).

Afterwards the loop in line 9 scans all local transition leading to the *current* state. If the event is uncontrollable, a new search record with the previous *startclass* is created in line 11. If the event is controllable, then based on Def. 14 the *current* state must be equivalent to the yet unknown start state x of the path. If the *startclass* is unassigned or the same as the class of *current*, then *current* can potentially be x, so its class is used to form a new search record in line 13.

If the algorithm is in the second part of the path, it also scans the transition leading to the *current* state. First it checks in lines 18–32 whether the source state *src* is *controllable*. This is done by exploring all successors reachable by local uncontrollable events. If one of these states is not equivalent to the *src*, *current*, or *end* state, or a state has a shared uncontrollable outgoing transition to a state with no matching state reachable from the *end* class, then the *src* state is not *controllable*. Otherwise, a new search record is created in line 31. The condition checked here is stronger than $\Rightarrow_c$ in Def. 15, which allows the target states of uncontrollable local transitions to be anywhere along the second part of the path. An exact implementation of $\Rightarrow_c$ requires search records to store complete paths, making the algorithm exponential.

Next the algorithms scans for σ-transition to the *current* state, and depending on whether σ is a local event or not, the algorithm moves to appropriate states in the first part of the path (lines 34–40). Synthesis observation equivalence can be checked by the same algorithm if lines 17–33 are deleted from *BS*.

The algorithm terminates when the *queue* of search records is empty. To prevent duplicates, the *queue* is linked to a hash set to ensure that search records that have been enqueued once are never added to the *queue* again. The hash set is reset for each split operation, i.e., before line 8 in Algorithm 2.

Complexity. In the worst case, the main loop in line 3 of Algorithm 1 is executed once for each state, giving up to $|Q|$ iterations. Inside the loop, a split on each class is performed. This causes each state to be processed once for each event, using either the loop in lines 2–6 or 8–10 of Algorithm 2. The bodies of these loops are executed $|\Sigma||Q|$ times in total during each iteration of the main loop of Algorithm 1. The splitting of classes after line 12 can be executed in lower complexity using the data structures outlined above.

The loop in line 2 of Algorithm 2 can be executed in $O(|Q|^2)$ time, by performing a search that visits each state at most twice, and each time checks all incoming transitions. This is dominated by the loop in line 8 which calls Algorithm *BS*.

In the worst case, Algorithm *BS* visits two search records for each combination of a state and class, i.e., up to $2|Q|^2$ search records. Each time, it executes either the loop in lines 9–15 or 17–33. The loop in lines 9–15 visits all local incoming transitions to a state, up to $|Q|$ operations if the local transitions are appropriately stored in advance. The loop in lines 17–33 also processes up to $|Q|$ local predecessor states, however each time the loop in lines 19–29 must be executed, potentially increasing complexity. Fortunately, this can be avoided by caching. The $\Rightarrow_u$-successors of the *end* class can be computed in advance, and it can be checked for each state x whether it has exactly one successor class reachable by local uncontrollable events that is different from the class of x and from the *end* class. By caching this class, it is possible to execute the loop in lines 19–29 only once for each state x during the execution of the Algorithm 3. With this caching, the complexity of Algorithm *BS* is $O(|Q|^3)$.

Therefore, the execution of Algorithm 1 involves $O(|Q|)$ iterations of the main loop, each performing $O(|\Sigma||Q|)$ search operations with of $O(|Q|^3)$ complexity. The worst-case time complexity of Algorithm 1 is $O(|\Sigma||Q|^5)$.

Table 1. Experimental Results

			SOE		WSOE	
Model	**Aut**	**States**	**Time**	**States**	**Time**	**States**
agv	16	$2.6{\cdot}10^7$	17.8 s	107747	18.2 s	106169
agvb	17	$2.3{\cdot}10^7$	11.7 s	83577	11.5 s	82353
aip0alps	35	$3.0{\cdot}10^8$	0.9 s	867	0.9 s	867
fencaiwon09b	31	$8.9{\cdot}10^7$	0.1 s	73	0.1 s	73
fms_2003	31	$1.4{\cdot}10^7$	83.6 s	673868	69.7 s	444922
koordwsp_b	24	$1.1{\cdot}10^7$	0.5 s	756	0.4 s	743
tbed_noderailb	84	$3.1{\cdot}10^{12}$	5.7 s	18134	4.4 s	18134
tbed_uncont	84	$3.6{\cdot}10^{12}$	5.0 s	9148	4.4 s	9148

5. EXPERIMENTAL RESULTS

The synthesis observation equivalence and weak synthesis observation equivalence algorithms have been implemented in the DES software tool *Supremica* (Åkesson *et al.*, 2006). They are used for abstraction within a compositional supervisor synthesis algorithm that computes modular supervisors for large systems (Mohajerani *et al.*, 2011*a*).

This program has been used to compute synthesis abstractions for a set of benchmark examples that include complex industrial models and case studies taken from various application areas such as manufacturing systems and automotive body electronics. The automata in each example are iteratively composed and simplified, until a final abstraction is obtained and passed on to standard synthesis. All tests were run on a standard desktop PC using a single core 2.66 GHz microprocessor.

Table 1 shows for each test case the number of automata (Aut) in the model and the size of the reachable state space (States). It also shows the total runtime of compositional synthesis (Time) and the number of states in the final abstraction passed on to standard synthesis (States), when using synthesis observation equivalence (SOE) or weak synthesis observation equivalence (WSOE).

Supervisors can be calculated for all models in less than two minutes, with memory usage no more than 600 MB. The size of the models are substantially reduced compared to the size of the original systems. A closer look at the table reveals that weak synthesis observation equivalence gives slightly more abstraction with about the same computational cost.

All examples are too large for supervisors to be computed by standard synthesis alone, and abstraction using only bisimulation results in a final abstraction with at least $2 \cdot 10^6$ states for all test cases.

6. CONCLUSIONS

Weak synthesis observation equivalence has been introduced as a means of abstraction for compositional synthesis algorithms. Weak observation equivalence allows for more abstraction than previously possible with synthesis observation equivalence. A polynomial complexity algorithm for synthesis observation equivalence and weak synthesis observation equivalence has been proposed and implemented in the DES software tool Supremica. The experimental results show that the algorithm can compute abstractions of automata with several thousand states, making it possible to construct modular supervisors for systems with more than 10^{12} reachable states.

REFERENCES

Åkesson, Knut, Martin Fabian, Hugo Flordal and Robi Malik (2006). 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. pp. 384–385.

Bolognesi, Tommaso and Scott A. Smolka (1987). Fundamental results for the verification of observational equivalence: a survey. In: *Protocol Specification, Testing and Verification VII: Proc. IFIP WG6.1 7th Int. Conf. Protocol Specification, Testing and Verification* (Harry Rudin and Colin H. West, Eds.). North Holland. Amsterdam, The Netherlands. pp. 165–179.

Feng, Lei and W. M. Wonham (2006). Computationally efficient supervisor design: Abstraction and modularity. In: *Proc. 8th Int. Workshop on Discrete Event Systems, WODES '06*. Ann Arbor, MI, USA. pp. 3–8.

Fernandez, Jean-Claude (1990). An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming* **13**, 219–236.

Flordal, Hugo, Robi Malik, Martin Fabian and Knut Åkesson (2007). Compositional synthesis of maximally permissive supervisors using supervision equivalence. *Discrete Event Dyn. Syst.* **17**(4), 475–504.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Malik, Petra, Robi Malik, David Streader and Steve Reeves (2007). Modular synthesis of discrete controllers. In: *Proc. 12th IEEE Int. Conf. Engineering of Complex Computer Systems, ICECCS '07*. Auckland, New Zealand. pp. 25–34.

Malik, Robi and Hugo Flordal (2008). Yet another approach to compositional synthesis of discrete event systems. In: *Proc. 9th Int. Workshop on Discrete Event Systems, WODES '08*. Göteborg, Sweden. pp. 16–21.

Milner, Robin (1989). *Communication and concurrency*. Series in Computer Science. Prentice-Hall.

Mohajerani, Sahar, Robi Malik and Martin Fabian (2011*a*). Nondeterminism avoidance in compositional synthesis of discrete event systems. In: *Proc. 7th Int. Conf. Automation Science and Engineering, CASE 2011*. Trieste, Italy. pp. 19–24.

Mohajerani, Sahar, Robi Malik, Simon Ware and Martin Fabian (2011*b*). On the use of observation equivalence in synthesis abstraction. In: *Proc. 3rd IFAC Workshop on Dependable Control of Discrete Systems, DCDS 2011*. Saarbrücken, Germany. pp. 84–89.

Ramadge, Peter J. G. and W. Murray Wonham (1989). The control of discrete event systems. *Proc. IEEE* **77**(1), 81–98.

Schmidt, Klaus and Christian Breindl (2008). On maximal permissiveness of hierarchical and modular supervisory control approaches for discrete event systems. In: *Proc. 9th Int. Workshop on Discrete Event Systems, WODES '08*. Göteborg, Sweden. pp. 462–467.

Su, Rong, Jan H. van Schuppen and Jacobus E. Rooda (2010). Model abstraction of nondeterministic finite-state automata in supervisor synthesis. *IEEE Trans. Autom. Control* **55**(11), 2527–2541.