An Algorithm for Weak Synthesis Observation Equivalence for Compositional Supervisor Synthesis

This document has been downloaded from Chalmers Publication Library (CPL). It is the author’s version of a work that was accepted for publication in:

Proc. 11th International Workshop on Discrete Event Systems (WODES'12)

Citation for the published paper:

http://dx.doi.org/10.3182/20121003-3-MX-4033.00040
Downloaded from: http://publications.lib.chalmers.se/publication/165507

Notice: Changes introduced as a result of publishing processes such as copy-editing and formatting may not be reflected in this document. For a definitive version of this work, please refer to the published source. Please note that access to the published version might require a subscription.

Chalmers Publication Library (CPL) offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all types of publications: articles, dissertations, licentiate theses, masters theses, conference papers, reports etc. Since 2006 it is the official tool for Chalmers official publication statistics. To ensure that Chalmers research results are disseminated as widely as possible, an Open Access Policy has been adopted. The CPL service is administrated and maintained by Chalmers Library.

(article starts on next page)
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., 2011a) 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., 2011b) 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., 2011a).

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 \(\Sigma\), which is partitioned into two disjoint subsets, the set \(\Sigma_c\) of controllable events and the set \(\Sigma_u\) 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 \(\Sigma\), including the empty trace \(\varepsilon\), is denoted by \(\Sigma^*\). 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 \(P_\Omega: \Sigma^* \rightarrow \Omega^*\) is the operation that removes from traces \(s \in \Sigma^*\) all events not in \(\Omega\).

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 $G = (\Sigma, Q, \rightarrow, Q^o)$, where $\Sigma$ 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 $Q^o \subseteq Q$ is the set of initial states. $G$ is deterministic, if $|Q^o| \leq 1$ 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 $\Sigma^*$ by letting $x \xrightarrow{\omega} x$ for all $x \in Q$, and $x \xrightarrow{\sigma^a} z$ if $x \xrightarrow{\sigma} y$ and $y \xrightarrow{\sigma} z$ for some $y \in Q$. Furthermore, $x \xrightarrow{\epsilon}$ means $x \xrightarrow{\epsilon} y$ for some $y \in Q$, and $x \xrightarrow{\omega}$ means $x \xrightarrow{\omega} y$ for some $s \in \Sigma^*$. These notations also apply to state sets and to automata: $X \xrightarrow{\sigma} Y$ for $X, Y \subseteq Q$ means $x \xrightarrow{\sigma} y$ for some $x \in X$ and $y \in Y$, and $G \xrightarrow{\sigma}$ means $Q \xrightarrow{\sigma}$, etc. The language of automaton $G$ is $L(G) = \{ s \in \Sigma^* \mid G \xrightarrow{\omega} \}$. A special requirement is that states reached by the termination event $\omega$ 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 $Q^\omega$ are shaded in the figures of this paper instead of explicitly showing $\omega$-transitions.

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

Definition 2. Let $G_1 = (\Sigma, Q_1, \rightarrow_1, Q^o_1)$ and $G_2 = (\Sigma, Q_2, \rightarrow_2, Q^o_2)$ be two automata. The synchronous composition of $G_1$ and $G_2$ is defined as

$$G_1 \parallel G_2 = (\Sigma \cup \Sigma, Q_1 \times Q_2, \rightarrow, Q^o_1 \times Q^o_2)$$

where

$$(x, y) \xrightarrow{\sigma} (x', y')$$

if $\sigma \in (\Sigma_1 \cap \Sigma_2)$, $x \xrightarrow{\sigma_1} x'$, $y \xrightarrow{\sigma_2} y'$;

$$(x, y) \xrightarrow{\sigma} (x, y')$$

if $\sigma \in (\Sigma_1 \setminus \Sigma_2)$, $x \xrightarrow{\sigma_1} x'$;

$$(x, y) \xrightarrow{\sigma} (x', y)$$

if $\sigma \in (\Sigma_1 \setminus \Sigma_2)$, $y \xrightarrow{\sigma_2} y'$. Another common automaton operation is the quotient modulo an equivalence relation on the state set.

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

$$G/\sim = (\Sigma, Q/\sim, \rightarrow/\sim, \hat{Q}^o)$$,

where $\rightarrow/\sim = \{ [x] \xrightarrow{\sigma} [y] \mid x \xrightarrow{\sigma} y \}$ and $\hat{Q}^o = \{ [x^o] \mid x^o \in Q^o \}$. 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 $\Sigma$. $K$ is controllable with respect to $G$ if, for every trace $s \in \Sigma^*$, every state $x \in K$, and every uncontrollable event $v \in \Sigma^*$ such that $K \xrightarrow{v} x$ and $G \xrightarrow{\sigma} s$, it holds that $x \xrightarrow{v}$ in $K$.

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

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

$$\sup\mathcal{L}(K) \subseteq L(K)$$

such that $\sup\mathcal{L}(K)$ 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) $G_1 = (\Sigma, Q_1, \rightarrow_1, Q^o_1)$ is a subautomaton of $G_2 = (\Sigma, Q_2, \rightarrow_2, Q^o_2)$, written $G_1 \subseteq G_2$, if $Q_1 \subseteq Q_2$, $\rightarrow_1 \subseteq \rightarrow_2$, and $Q^o_1 \subseteq Q^o_2$.

Definition 7. (Malik and Flordal, 2008) Let $G = (\Sigma, Q, \rightarrow, Q^o)$ and $K = (\Sigma, Q_K, \rightarrow_K, Q^o_K)$ 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 $v \in \Sigma^*$ such that $x \xrightarrow{v} y$, it also holds that $x \xrightarrow{\omega v} y$.

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 $G = (\Sigma, Q, \rightarrow, Q^o)$. There exists a unique subautomaton $\sup\mathcal{N}(G) \subseteq G$ such that $\sup\mathcal{N}(G)$ 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 $S \subseteq \sup\mathcal{N}(G)$.

The synthesis result $\sup\mathcal{N}(G)$ 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 $G = (\Sigma, Q, \rightarrow, Q^o)$ to $X \subseteq Q$ is

$$G|_X = (\Sigma, Q, \rightarrow|_X, Q^o \cap X),$$

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

Definition 9. (Malik and Flordal, 2008) The synthesis step operator $\Theta_G : 2^Q \rightarrow 2^Q$ for $G = (\Sigma, Q, \rightarrow, Q^o)$ is defined as

$$\Theta_G(X) = \Theta_G^{\text{cont}}(X) \cap \Theta_G^{\text{nonb}}(X),$$

where

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

$$\Theta_G^{\text{nonb}}(X) = \{ x \in X \mid x \xrightarrow{t \omega} |_X \text{ for some } t \in \Sigma^* \}.$$ Theorem 2. (Malik and Flordal, 2008) Let $G = (\Sigma, Q, \rightarrow, Q^o)$. The synthesis step operator $\Theta_G$ has a greatest fixpoint $\text{gfp}\Theta_G = \Theta_G \subseteq G$, such that $G|_{\text{gfp}\Theta_G}$ is the greatest subautomaton of $G$ that is both controllable in $G$ and coreachable, i.e.,

$$\sup\mathcal{N}(G) = G|_{\text{gfp}\Theta_G}.$$ If the state set $Q$ is finite, the sequence $X^0 = Q$, $X^{n+1} = \Theta_G(X^n)$ reaches this fixpoint in a finite number of steps, i.e., $\Theta_G = X^n$ for some $n \geq 0$. 

2
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,

\[
G = \{G_1, G_2, \ldots, G_n\}.
\]

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 \(\mathcal{Y} \subseteq \Sigma\) 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., 2011b) Let \(G\) and \(H\) be deterministic automata with alphabet \(\Sigma\). Then \(H\) is a synthesis abstraction of \(G\) with respect to \(\mathcal{Y} \subseteq \Sigma\), written \(G \preceq_{\text{synth}, \mathcal{Y}} H\), if for every deterministic automaton \(T = (\Sigma_T, Q_T, \rightarrow_T, Q^*_T)\) such that \(\Sigma_T \cap \mathcal{Y} = \emptyset\) the following holds,

\[
\mathcal{L}(G\|T) = \mathcal{L}(G'\|T) \iff \mathcal{L}(H\|T) = \mathcal{L}(H'\|T).
\]

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 \(x_1 \rightarrow x_2\) and \(x_1 \rightarrow y_1\) there exists \(y_2 \in X\) such that \(x_2 \rightarrow y_2\) and \(y_1 \rightarrow y_2\).

Definition 12. Let \(G = (\Sigma, Q, \rightarrow, Q^0)\) be an automaton. An equivalence relation \(\sim \subseteq Q \times Q\) is called a bisimulation on \(G\), if \(\sim\) is stable with respect to \(\rightarrow\) for all \(\sigma \in \Sigma\).

Definition 13. Let \(G = (\Sigma, Q, \rightarrow, Q^0)\) be an automaton with \(\Sigma = \Omega \cup \mathcal{Y}\). An equivalence relation \(\sim \subseteq Q \times Q\) is called an observation equivalence on \(G\) with respect to \(\mathcal{Y}\), if \(\sim\) is stable with respect to \(\Rightarrow\) for all \(\sigma \in \Sigma\), where \(x \Rightarrow y\) if and only if \(x \xrightarrow{t_1 \cdot \Phi_i(t_2)} y\) for some \(t_1, t_2 \in \mathcal{Y}^*\).

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., 2011b).

Example 1. (Mohajerani et al., 2011b) Consider automata \(G\) and \(T\) in Fig. 1, where \(\mathcal{Y} = \{\alpha, \beta\}\) and \(\Sigma_u = \{\mu, \nu\}\). 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 \(\alpha\) to prevent blocking via \(\nu\), but after merging \(q_0\) and \(q_1\), disabling \(\alpha\) is not enough to prevent the dangerous uncontrollable event \(\nu\).

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

Definition 14. Let \(G = (\Sigma, Q, \rightarrow, Q^0)\) be an automaton with \(\Sigma = \Omega \cup \mathcal{Y}\). An equivalence relation \(\approx \subseteq Q \times Q\) is a synthesis observation equivalence on \(G\) with respect to \(\mathcal{Y}\), if \(\approx\) is stable with respect to \(\Rightarrow_{\text{soe}}\) to \(\equiv_{\text{soe}}\) for each \(\sigma \in \Sigma \cup \Omega\), and to \(\Rightarrow_{\sw}\) for each \(\nu \in \Sigma_u\) defined as follows.

- \(x \Rightarrow_{\text{soe}} y\) if there exists a path \(z_0 \Rightarrow z_1 \leadsto \cdots \leadsto z_k\), such that \(z_0 = x\), and \(z_k = y\), and \(\tau_1, \ldots, \tau_k \in \mathcal{Y}\), and \(\tau_j \in \Sigma\) implies \(x \Rightarrow z_j\) or \(j = k\).
- \(x \equiv_{\text{soe}} y\) if there exists a path \(x = z_0 \Rightarrow z_1 \leadsto \cdots \leadsto z_k \Rightarrow y\) such that \(\tau_1, \ldots, \tau_k \in \mathcal{Y}\), and \(\tau_j \in \Sigma\) implies \(x \Rightarrow z_j\).
- \(x \Rightarrow_{\sw} y\) if \(x \xrightarrow{t_1 \cdot \Phi_i(t_2)} y\) for some \(t_1, t_2 \in (\Sigma_u \cap \mathcal{Y})^*\).

Example 2. Consider automaton \(G_1\) in Fig. 2, where all events are controllable and \(\mathcal{Y} = \{\beta\}\). The equivalence relation \(\approx\) with \(q_1 \approx q_2 \approx q_3\) is a synthesis observation equivalence relation. For example, the transition \(q_2 \Rightarrow q_4\) is matched by \(q_1 \Rightarrow q_3\) and \(q_3 \Rightarrow q_4\) where state \(q_3\), reached by the local controllable event \(\beta\), is equivalent to \(q_2\). Merging the equivalent states results in the synthesis observation equivalent abstraction \(G_1/\approx\) shown in Fig. 2.

Synthesis observation equivalence implies synthesis abstraction (Mohajerani et al., 2011b). Def. 14 modifies observation equivalence based on event types. For uncontrollable events, \(\Rightarrow_{\sw}\) 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.
Observation equivalence, the transitive closure

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

\[ x \mathrel{\Rightarrow_{\text{wsoe}}} y \text{ if } x \mathrel{\Rightarrow_{\text{wsoe}}} z \mathrel{\Rightarrow_{\text{wsoe}}} y \text{ for some } z \in Q, \]

\[ x \mathrel{\Rightarrow_{\text{wsoe}}} y \text{ if } x \mathrel{\Rightarrow_{\text{wsoe}}} z \mathrel{\Rightarrow_{\text{wsoe}}} y \text{ for some } z \in Q, \]

\[ x \mathrel{\Rightarrow_{\text{wsoe}}} y \text{ if there exists a path } z_0 \xrightarrow{t_1} \cdots \xrightarrow{t_k} z_k \text{, such that } \]

\[ x = x, \text{ and } z_k = y, \text{ and } t_1, \ldots, t_k \in \Upsilon, \text{ and } z_j \mathrel{\Rightarrow_{\text{wsoe}}} z' \text{ for } u \in (\Sigma \cap \Upsilon)^* \text{ implies } z' \sim z_k \text{ for some } 0 \leq i \leq k, \]

\[ \text{and } z_j \mathrel{\Rightarrow_{\text{wsoe}}} z' \text{ for } v \in (\Sigma \cap \Upsilon)^* \text{ implies } y \mathrel{\Rightarrow_{\text{wsoe}}} z'' \text{ for some } \]

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.

\[ \text{Example 3. Consider automaton } G_2 \text{ in Fig. 2, with all events controllable and } \Upsilon = \{ \beta \}. \]

The partition refinement 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 \( \Rightarrow_u \) 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 \( \Rightarrow_u \) 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 \( \sigma \) and end state. It uses a queue of search records \( (\text{current}, \text{part}, \text{startclass}) \) 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 \( (\text{end}, 2, \text{none}) \) is added to the queue.
Algorithm 3: Backward search $BS(\sigma \in \Sigma_c, \text{end} \in Q)$

1: $\text{queue} \leftarrow \emptyset$
2: add $(\text{end}, 2, \text{none})$ to $\text{queue}$
3: while $\text{queue} \neq \emptyset$ do
4:   remove $(\text{current}, \text{part}, \text{startclass})$ from $\text{queue}$
5:   if $\text{part} = 1$ then
6:     if $\text{startclass} \in \{(\text{current}), \text{none}\}$ then
7:       move $\text{current}$ to split list in $[\text{current}]$
8:     end if
9:   for all transitions $\sigma \xrightarrow{v} \text{current}$ with $v \in \Upsilon$ do
10:      if $v \in \Sigma_u$ then
11:        add $(\sigma, 1, \text{startclass})$ to $\text{queue}$
12:      else if $\text{startclass} \in \{(\text{current}), \text{none}\}$ then
13:        add $(\sigma, 1, [\text{current}])$ to $\text{queue}$
14:     end if
15:   end for
16: else
17:   for all transitions $\sigma \xrightarrow{v} \text{current}$ with $v \in \Upsilon$ do
18:     if $\text{controllable} \leftarrow \text{true}$
19:       for all $\sigma \xrightarrow{u} \text{succ}$ with $u \in (\Sigma_u \cap \Upsilon)^*$ do
20:         if $\text{succ} \notin [\sigma] \cup \{\text{current}\} \cup [\text{end}]$ then
21:           $\text{controllable} \leftarrow \text{false}$
22:         else for all $\text{succ} \xrightarrow{\gamma} \text{succ}'$ with $\gamma \in \Sigma_u \cap \Omega$ do
23:            if not $[\text{end}] \xrightarrow{\gamma} [\text{succ}']$ then
24:              $\text{controllable} \leftarrow \text{false}$
25:           end if
26:         end for
27:       end if
28:     end for
29:   end if
30:   if $\text{controllable}$ then
31:     add $(\sigma, 2, \text{none})$ to $\text{queue}$
32:   end if
33: end for
34: if $\sigma \in \Upsilon$ then
35:   add $(\text{current}, 1, \text{none})$ to $\text{queue}$
36: else
37:   for all transitions $\sigma \xrightarrow{\gamma} \text{current}$ do
38:     add $(\sigma, 1, \text{current})$ to $\text{queue}$
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 $\Rightarrow_{\text{soc}}$. 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 algorithm scans for $\sigma$-transition to the current state, and depending on whether $\sigma$ 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

<table>
<thead>
<tr>
<th>Model</th>
<th>Aut</th>
<th>States</th>
<th>SOE Time</th>
<th>SOE States</th>
<th>WSOE Time</th>
<th>WSOE States</th>
</tr>
</thead>
<tbody>
<tr>
<td>agv</td>
<td>16</td>
<td>$2.6 \times 10^9$</td>
<td>17.8 s</td>
<td>107747</td>
<td>18.2 s</td>
<td>106169</td>
</tr>
<tr>
<td>agvb</td>
<td>17</td>
<td>$2.3 \times 10^7$</td>
<td>11.7 s</td>
<td>83577</td>
<td>11.5 s</td>
<td>82353</td>
</tr>
<tr>
<td>aip0alps</td>
<td>35</td>
<td>$3.9 \times 10^8$</td>
<td>0.9 s</td>
<td>867</td>
<td>0.9 s</td>
<td>867</td>
</tr>
<tr>
<td>fencaiwon09b</td>
<td>31</td>
<td>$8.9 \times 10^7$</td>
<td>0.1 s</td>
<td>73</td>
<td>0.1 s</td>
<td>73</td>
</tr>
<tr>
<td>fms_2003</td>
<td>31</td>
<td>$1.4 \times 10^7$</td>
<td>83.6 s</td>
<td>673568</td>
<td>69.7 s</td>
<td>444922</td>
</tr>
<tr>
<td>koordwsp_b</td>
<td>24</td>
<td>$1.1 \times 10^7$</td>
<td>0.5 s</td>
<td>756</td>
<td>0.4 s</td>
<td>743</td>
</tr>
<tr>
<td>tbbed_noderailb</td>
<td>84</td>
<td>$3.1 \times 10^{12}$</td>
<td>5.7 s</td>
<td>18134</td>
<td>4.4 s</td>
<td>18134</td>
</tr>
<tr>
<td>tbbed_uncont</td>
<td>84</td>
<td>$3.6 \times 10^{12}$</td>
<td>5.0 s</td>
<td>9148</td>
<td>4.4 s</td>
<td>9148</td>
</tr>
</tbody>
</table>

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., 2011a).

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


