Verification of the Observer Property in Discrete Event Systems


Abstract—The observer property is an important condition to be satisfied by abstractions of Discrete Event System (DES) models. This paper presents a new algorithm that tests if an abstraction of a DES obtained through natural projection has the observer property. The procedure, called OP-Verifier, can be applied to (potentially nondeterministic) automata, with no restriction on the existence of cycles of “non-relevant” events. This procedure has quadratic complexity in the number of states. The performance of the algorithm is illustrated by a set of experiments.

Index Terms—Discrete Event Systems, Natural Projections, Observer Property.

I. INTRODUCTION

Natural projections play a central role in the computation of abstractions for Discrete Event Systems (DES) models. Abstractions obtained by natural projections have been extensively used in the Supervisory Control Theory of DES [1] as, for example, in control with partial observation, in hierarchical control [2]–[6], in modular synthesis [2], [7], [8], and in compositional verification of the nonblocking property [2], [9], [10], among many problem domains. In several of the above cited works, the observer property is an important condition to be satisfied by the abstracted models. Abstractions satisfying this property are called OP-abstractions [11].

The observer property, or simply OP hereafter, was first introduced in the context of hierarchical control of DES. In [12], the abstraction is obtained in the form of a reporter map, that projects strings of events of the original (low-level) model, built from a set $\Sigma$, into high-level strings built from an independent set of events, denoted by $T$. Due to some difficulties with the use of reporter maps [13], most of the approaches subsequent to [12] focus on abstractions obtained by the natural projection, which maps strings of the original model into strings of the abstraction, by erasing events of $\Sigma$ that are not contained in a given subset of relevant events, denoted by $\Sigma_r$, with $\Sigma_r \subseteq \Sigma$; see [3]–[5], [14], [15].

The structure called OP-Verifier was first presented in [11]; it was inspired by an algorithm for testing diagnosability presented in [16]. Given an input automaton $G$, defined on the alphabet $\Sigma$, a set of relevant events $\Sigma_r \subseteq \Sigma$, and a natural projection $\theta$ from strings in $\Sigma$ to strings in $\Sigma_r$, the OP-Verifier algorithm checks whether the projection $\theta(L_m(G))$ is an OP-abstraction. The OP-Verifier algorithm does not require explicitly computing the abstraction to check for the OP and has been shown to have better computational performance when compared to other similar procedures [13], [17]–[19]. It runs in quadratic complexity in the number of states. A limitation of the OP-Verifier algorithm as proposed in [11], however, is that it can only be applied to automata that do not have cycles of non-relevant events.

A different algorithm to test the OP is proposed in [13], [17]. This algorithm relies on the computation of a coarsest observation equivalence relation and runs in cubic complexity in the number of states. Yet another algorithm for testing so-called “observerness” for a system $G$ and a mask $M$ is presented in [18]. This procedure may give false negatives as stated and needs to be modified to address this problem [20].

This paper presents a modified version of the OP-Verifier algorithm of [11] that subsumes the preliminary results in [21]. This algorithm can be applied to automata with no restriction on the existence of cycles of non-relevant events. The algorithm operates on a modified automaton $G_{nr}$, obtained from the input automaton $G$, by aggregating states connected by cycles of non-relevant events. It overcomes the limitations of the previously proposed verifier [11], [21], while retaining its quadratic complexity. The modified OP-Verifier algorithm has been implemented in Supremica [22].

This paper is organized as follows. Section II introduces the necessary background. Section III describes the construction of the OP-Verifier automaton and its properties. Then Section IV presents an algorithm to construct the OP-Verifier and check the observer property. This section also contains a complexity analysis and experimental results to demonstrate the performance of the algorithm in comparison with [17]. Finally, concluding remarks are given in Section V.

II. PRELIMINARIES

This paper is set in the supervisory control framework. The reader is referred to [1] for a detailed introduction to the theory. Behaviors of DES are modeled using strings of events...
taken from a finite alphabet $\Sigma$. $\Sigma^*$ is the set of all finite strings of events in $\Sigma$, including the empty string $e$. The concatenation of strings $s, u \in \Sigma^*$ is written as $su$. A string $s \in \Sigma^*$ is called a prefix of $t \in \Sigma^*$, written $s \leq t$, if there exists $u \in \Sigma^*$ such that $su = t$. A subset $L \subseteq \Sigma^*$ is called a language. The prefix closure $T_{\ell}$ of a language $L \subseteq \Sigma^*$ is the set of all prefixes of strings in $L$, i.e., $T_{\ell} = \{ s \in \Sigma^* | s \leq t \text{ for some } t \in L \}$. Regular languages are represented by (possibly nondeterministic) finite-state automata as follows.

**Definition 1:** A (nondeterministic) finite-state automaton is a tuple $G = \langle \Sigma, Q, \rightarrow, Q^o \rangle$, 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 $x \rightarrow y_1 \text{ and } x \rightarrow y_2$ always implies $y_1 = y_2$.

The transition relation is written in infix notation $x \rightarrow^\alpha y$, and is extended to traces in $\Sigma^*$ by letting $x \rightarrow^\alpha z$ for all $x \in Q$, and $x \rightarrow^\alpha z$ if $x \rightarrow^\alpha y$ and $y \rightarrow^\beta z$ for some $y \in Q$. Furthermore, $x \rightarrow^\alpha y$ means $x \rightarrow^\alpha y$ for some $y \in Q$, and $x \rightarrow^\alpha y$ means $x \rightarrow^\alpha y$ for some $s \in \Sigma^*$. These notations also apply to state sets: $X \rightarrow^\alpha Y$ for $X, Y \subseteq Q$ means $x \rightarrow^\alpha y$ for some states $x \in X$ and $y \in Y$. Also, if $G$ is an automaton, then $G \rightarrow^\alpha x, G \rightarrow^\alpha X$, and $G \rightarrow^\alpha \sigma$ stand for $Q^o \rightarrow^\alpha x, Q^o \rightarrow^\alpha X$, and $Q^o \rightarrow^\alpha \sigma$, respectively. For example, $G \rightarrow^\alpha X$ means that the automaton $G$ can reach some state in the set $X \subseteq Q$ on execution of trace $s \in \Sigma^*$. Finally, the generated language of automaton $G$ is $L(G) = \{ s \in \Sigma^* | G \rightarrow^\alpha s \}$.

To express the marking of strings, the alphabet $\Sigma$ is assumed to contain the marking event $\omega \in \Sigma$, which may only appear on self-loops, i.e., $x \rightarrow^\omega y$ always implies $y = x$. In this notation, the marked language of $G$ is defined as $L_{m}(G) = \{ s \in (\Sigma \setminus \{\omega\})^* | s \omega \in L(G) \}$. This paper uses natural projections, it is written as follows.

Given an automaton $G = \langle \Sigma, Q, \rightarrow, Q^o \rangle$, a state $x \in Q$ is called reachable if $G \rightarrow x$, and coreachable if $x \rightarrow^\alpha y$ for some $t \in \Sigma^*$. The automaton $G$ is called reachable if every state $x \in Q$ is reachable, and nonblocking if every reachable state $x \in Q$ is coreachable.

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

**Definition 2:** Let $G = \langle \Sigma, Q, \rightarrow, Q^o \rangle$ be an automaton and let $\sim \subseteq Q \times Q$ be an equivalence relation. The quotient automaton of $G$ modulo $\sim$ is $G/\sim = \langle \Sigma, Q/\sim, \rightarrow/\sim, Q^o/\sim \rangle$, where $\rightarrow/\sim = \{ (x, y) \mid x' \rightarrow^\alpha y' \text{ for some } x' \in [x] \text{ and } y' \in [y] \}$ and $Q^o/\sim = \{ [x^o] \mid x^o \in Q^o \}$. 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.

An operation over languages that is very important for abstraction is natural projection. For this purpose, the event alphabet is partitioned into $\Sigma = \Sigma_r \cup \Sigma_{nr}$, where $\Sigma_r$ denotes the set of relevant events, while $\Sigma_{nr}$ denotes the set of non-relevant events. For $\Sigma_r \subseteq \Sigma$, the natural projection $\theta: \Sigma^* \rightarrow \Sigma_r^*$ maps strings in $\Sigma^*$ to strings in $\Sigma_r^*$ by erasing all events not contained in $\Sigma_r$. The concept is extended to languages by defining $\theta(L) = \{ t \in \Sigma_r^* \mid t = \theta(s) \text{ for some } s \in L \}$.

This paper is concerned with the property of projections known as the observer property, which was first introduced in the context of reporter maps in [12] and [14]. In the context of natural projections, it is written as follows.

**Definition 3:** [14] Let $L \subseteq \Sigma^*$ be a language, let $\Sigma_r \subseteq \Sigma$, and let $\theta: \Sigma^* \rightarrow \Sigma_r^*$ be the natural projection. If for all $l \in \Sigma^*$ and all $t \in \Sigma_r^*$ such that $\theta(s) t \in \theta(L)$, there exists $t' \in \Sigma^*$ such that $\theta(st') = \theta(st)$ and $st' \in L$, then $\theta(L)$ has the observer property.

The observer property ensures that, if two states can be reached by traces with the same projection, i.e., $G \xrightarrow{a} x_1$ and $G \xrightarrow{b} x_2$ with $\theta(s_1) = \theta(s_2)$, then these states can also achieve termination by traces with equal projection, i.e., $x_1 \xrightarrow{a} x' \implies x_2 \xrightarrow{b} x''$ with $\theta(t_1) = \theta(t_2)$. If the observer property is satisfied for an automaton, then its natural projection is “observation equivalent” to that automaton, which means that all branching in the automaton remains visible in its projection [12].

Projections can also be applied to automata. Given a deterministic and nonblocking automaton $G$, its projection $\theta(G)$ is the minimal deterministic recognizer of the language $\theta(L_m(G))$ [23]. Then it is said that $\theta(G)$ has the observer property if $\theta(L_m(G))$ has the observer property. In this case $\theta(G)$ is also called an OP-abstraction.

**Example 1:** Automaton $G$ in Fig. 1 models the behavior of a simple manufacturing transfer line with material feedback, adapted from [21], [24]. After starting to manufacture a workpiece $(a)$, the transfer line can either finish production successfully $(b)$, or decide to retain the workpiece $(r)$ for one or more rework cycles $(y)$, and eventually finish production with a reworked workpiece $(c)$. Assume that, in some hierarchical control approach, as in [3], [5], [6], one is concerned only with the input-output behavior of the line. Then it is of interest to construct the abstraction $\theta(G)$ with respect to relevant events $\Sigma_r = \{a, b, c, \omega\}$ and non-relevant events $\Sigma_{nr} = \{r, y\}$, which is shown in Fig. 1. In this case, $\theta(G)$ is not an OP-abstraction.

To see this, let $s = ar$ and $t = b$ in Definition 3. Then $\theta(st) = ab \in \theta(L_m(G))$, but there is no trace $t' \in \Sigma^*$ such that $st' = ab \in L_m(G)$ and $\theta(st') = \theta(st) = ab$.

The OP-Verifier algorithm [11] can check for certain projections whether or not they satisfy the observer property. This algorithm, which was inspired by the verifier [16] for testing the property of diagnosability, can only be applied to deterministic automata that do not have cycles of non-relevant events. The automaton $G$ in Fig. 1 has a cycle of non-relevant events involving states 1, 2, and 3. Because of this cycle, the example cannot be classified correctly by the algorithm [11].
III. VERIFICATION OF THE OBSERVER PROPERTY

In this section, the OP-Verifier algorithm is presented. It extends the algorithm in [11] by adding the ability to handle cycles of non-relevant events.

A. Strongly $\Sigma_{nr}$-Connected Components Automaton $G_{nr}$

In order to deal with cycles of non-relevant events, a strongly $\Sigma_{nr}$-connected components automaton is introduced. Let $G = (\Sigma, Q, \rightarrow, Q^0)$ be an automaton, and let $\Sigma_{nr} \subseteq \Sigma$ be a set of non-relevant events. Define the following relations on the state set $Q$:

\[
x_{nr} \rightarrow y \iff x \xrightarrow{s} y \text{ for some } s \in \Sigma_{nr}; \quad (2)
\]

\[
x_{nr} \leftrightarrow y \iff x_{nr} \rightarrow y \text{ and } y_{nr} \rightarrow x. \quad (3)
\]

If $x_{nr} \rightarrow y$, then the states $x$ and $y$ are called strongly $\Sigma_{nr}$-connected ($\Sigma_{nr}$-SC), because it is possible to reach each state from the other using only non-relevant events. If $G$ does not contain two distinct $\Sigma_{nr}$-SC states it is said to be $\Sigma_{nr}$-acyclic.

A set of $\Sigma_{nr}$-SC states is called a strongly $\Sigma_{nr}$-connected component ($\Sigma_{nr}$-SCC). If each $\Sigma_{nr}$-SCC is contracted to a single state, the resulting automaton is $\Sigma_{nr}$-acyclic. This contracted automaton is called the strongly $\Sigma_{nr}$-connected components automaton ($\Sigma_{nr}$-SCC automaton) of $G$ in the following. Formally, the $\Sigma_{nr}$-SCC of state $x \in Q$ is

\[
[x] = \{ y \in Q \mid x_{nr} \rightarrow y \}, \quad (4)
\]

and the $\Sigma_{nr}$-SCC automaton of $G$ is the quotient automaton constructed by merging the $\Sigma_{nr}$-SCCs in $G$,

\[
G_{nr} = G/_{nr}. \quad (5)
\]

Remark 1: In graph theory, the $\Sigma_{nr}$-SCC automaton is called a condensation graph, which is known to be acyclic [25], i.e., it does not contain any cycles of non-relevant events except for self-loops. For a finite state set, it follows that for every state $x \in Q$, there exists a state $y \in Q$ such that $x_{nr} \rightarrow y$, with $[y]$ a terminal component, i.e., a component with no further $\Sigma_{nr}$-transitions outgoing to other components.

Definition 4: Let $G = (\Sigma, Q, \rightarrow, Q^0)$ be an automaton, and let $\Sigma_{nr} \subseteq \Sigma$. For $y \in Q$, the component $[y]$ is $\Sigma_{nr}$-terminal if, for all $\sigma \in \Sigma_{nr}$ and all $z \in Q$ such that $[y] \xrightarrow{\sigma} [z]$, it holds that $[y] = [z]$.

The strongly connected components of a graph can be computed efficiently using Tarjan’s Algorithm [26]. This algorithm has a worst-case time complexity of $O(|V|)$, i.e., it is linear in the number of transitions. Tarjan’s Algorithm can be easily adapted to compute the $\Sigma_{nr}$-SCC automaton.

B. OP-Verifier $V_G$

Based on the $\Sigma_{nr}$-SCC automaton, the OP-Verifier $V_G$ is constructed. The OP-Verifier is a nondeterministic automaton that is used to determine whether or not the observer property is satisfied for the original automaton $G$ and non-relevant events $\Sigma_{nr}$. It is constructed in a similar way to the previous OP-Verifier for $\Sigma_{nr}$-acyclic automata in [11], except that it is based on the $\Sigma_{nr}$-SCC automaton $G_{nr}$ instead of $G$.

Definition 5: Let $G = (\Sigma, Q, \rightarrow, Q^0)$ be a deterministic automaton with $\Sigma_{nr}$-SCC automaton $G_{nr} = (\Sigma, Q_{nr}, \rightarrow_{nr}, Q_{nr}^0)$, and let $\Sigma = \Sigma_r \cup \Sigma_{nr}$. The OP-Verifier $V_G$ for $G$ is

\[
V_G = (\Sigma, Q_V, \rightarrow, Q_V^0) \quad (6)
\]

where

- $Q_V = \{ P \subseteq Q/_{nr} \mid 1 \leq |P| \leq 2 \} \cup \{ \bot \}$.

The state set of the verifier consists of sets of $\Sigma_{nr}$-SCCs of $G$ of cardinality one or two, i.e., a single $\Sigma_{nr}$-SCC and pairs of $\Sigma_{nr}$-SCCs, plus the special state $\bot$.

- $\rightarrow$ consists of the following transitions:

\[
\{ [x], [y] \} \xrightarrow{\sigma} \{ [x'], [y'] \} \quad \text{if } \sigma \in \Sigma_r, [x] \xrightarrow{\sigma_{nr}} [x'], \quad (7)
\]

\[
\{ [x], [y] \} \xrightarrow{\sigma} [y] \quad \text{if } \sigma \in \Sigma_{nr} \text{ and } [x] \xrightarrow{\sigma_{nr}} [x'], \quad (8)
\]

\[
[x], [y] \xrightarrow{\sigma} [z] \quad \text{if } \sigma \in \Sigma_r, [x] \xrightarrow{\sigma_{nr}} [y], \quad (9)
\]

\[
\emptyset \subseteq \emptyset \quad \text{is terminal, and } [y] \not\xrightarrow{\sigma_{nr}}.
\]

- $Q_V^0 = \{ \{ [x^0], [y^0] \} \mid x^0, y^0 \in Q^0 \}$.

The initial state set of the verifier contains all pairs of $\Sigma_{nr}$-SCCs of initial states of $G$.

Example 2: The $\Sigma_{nr}$-SCC automaton corresponding to $G$ in Example 1 is $G_{nr}$ shown in Fig. 2. The $\Sigma_{nr}$-SCCs are $\{ 0 \} = \{ 0 \}$, $\{ 1 \} = \{ 1 \}$ and $\{ 2 \} = \{ 2 \}$, $\{ 3 \} = \{ 3 \}$. Notice that $\{ 2 \}$ is $\Sigma_{nr}$-terminal. The verifier $V_G$, shown in Fig. 2, contains the following transitions: from (7), $\{ 0 \} = \{ 0 \} \xrightarrow{\sigma} \{ 0 \}$, $\{ 1 \} \xrightarrow{\sigma_{nr}} \{ 0 \}$, and $\{ 2 \} \xrightarrow{\sigma_{nr}} \{ 0 \}$; from (8), $\{ 1 \} \xrightarrow{\sigma_{nr}} \{ 1, 2 \}$, $\{ 2 \} \xrightarrow{\sigma_{nr}} \{ 2 \}$, and $\{ 2 \} \xrightarrow{\sigma_{nr}} \{ 2 \}$; and, from (9), $\{ 1 \} \xrightarrow{\sigma_{nr}} \{ 1 \}$, $\{ 2 \} \xrightarrow{\sigma_{nr}} \{ 2 \}$, and $\{ 2 \} \xrightarrow{\sigma_{nr}} \{ 2 \}$, since $\{ 1 \} \not\xrightarrow{\sigma_{nr}}$.

C. Properties of the OP-Verifier

This section establishes a key property of the OP-Verifier. The special state $\bot$ is reachable in the OP-Verifier if and only if the observer property is not satisfied. The main result in Theorem 3 depends on two lemmas to relate traces with the same projection to the states of the verifier: the OP-Verifier contains all pairs of $\Sigma_{nr}$-SCCs that can be reached by traces that project to the same relevant events.

Lemma 1: Let $V_G$ be the verifier for automaton $G$. Let $a, b \in \Sigma_r$ such that $\theta(a) = \theta(b)$ and $G \xrightarrow{a} x_a$ and $G \xrightarrow{b} x_b$. Then there exists $s \in \Sigma^*$ such that $\theta(a) = \theta(b) = \theta(s)$ and $V_G \xrightarrow{\{ s \} | \{ s \} | \{ s \}}$.
Proof: The claim is shown by induction on $n = |a| + |b|$. In the base case, $n = 0$ and thus $a = b = \varepsilon$. Then $x_a, x_b \in Q^0$ and thus $\{[x_a], [x_b]\} \in Q^0_V$, i.e., $V_G \xrightarrow{} \{[x_a], [x_b]\}$.

Now assume the claim has been shown for all $a, b \in \Sigma^*$ such that $\sigma(a) = \theta(b)$ and $|a| + |b| \leq n$. Consider $a, b \in \Sigma^*$ such that $\sigma(a) = \theta(b)$ and $|a| + |b| = n + 1$ and $G \xrightarrow{a} x_a$ and $G \xrightarrow{b} x_b$. As $\sigma(a) = \theta(b)$ and $|a| + |b| > 0$, either at least one of the traces $a$ or $b$ ends with an event in $\Sigma_{nr}$, or both end with the same event in $\Sigma_r$.

In the first case, assume without loss of generality $a = a'\sigma$ for some $\sigma \in \Sigma_{nr}$. Then $G \xrightarrow{a} x_a \xrightarrow{a'} x_a$ and $\sigma(a') = \theta(a') = \theta(b)$, and by inductive assumption there exists $s \in \Sigma^*$ such that $\sigma(s) = \theta(a') = \theta(b)$ and $V_G \xrightarrow{s} \{[x_a], [x_b]\}$. Given $x_a \xrightarrow{a'} x_a$ with $\sigma \in \Sigma_{nr}$, it follows by construction of $V_G$ (8) that $V_G \xrightarrow{s} \{[x_a], [x_b]\} \xrightarrow{a'} \{[x_a], [x_b]\}$, i.e., $V_G \xrightarrow{s} \{[x_a], [x_b]\}$ with $\sigma(s) = \theta(s) = \theta(a') = \theta(b)$.

In the second case, $a = a'\sigma$ and $b = b'\sigma$ for some $\sigma \in \Sigma_r$. Then $G \xrightarrow{a} x_a \xrightarrow{a'} x_a$ and $G \xrightarrow{b} x_b \xrightarrow{b'} x_b$ and $\sigma(a') = \theta(b')$. By inductive assumption there exists $s' \in \Sigma^*$ such that $\sigma(s') = \theta(a') = \theta(b')$ and $V_G \xrightarrow{s'} \{[x_a], [x_b]\}$. Given $x_a \xrightarrow{a'} x_a$ and $x_b \xrightarrow{b'} x_b$ with $\sigma \in \Sigma_r$, it follows by construction of $V_G$ (7) that $V_G \xrightarrow{s'} \{[x_a], [x_b]\} \xrightarrow{a'} \{[x_a], [x_b]\}$, i.e., $V_G \xrightarrow{s'} \{[x_a], [x_b]\}$ with $\sigma(s') = \theta(s') = \theta(a') = \theta(b')$.

Lemma 2: Let $V_G = (\Sigma, \Omega_G, V, Q_G)$ be the verifier for automaton $G$. Let $s \in \Sigma^*$ and $A, B \in Q_G$. Then $V_G \xrightarrow{s} \{A, B\}$ if and there exist $a, b \in \Sigma^*$ such that $\sigma(a) = \theta(b)$ and $G \xrightarrow{a} x_a \xrightarrow{b} x_b$. Proof: The claim is shown by induction on $n = |s|$.

In the base case, $n = 0$ and thus $s = \varepsilon$. Note that $A, B \in Q_G$. By Def. 5 there exist $x_a \xrightarrow{a} x_a$ in $A$ such that $x_a \xrightarrow{b} x_b$ and $x_b \xrightarrow{B} x_b$ such that $x_b \xrightarrow{b} x_b \in Q^0$, which is enough to show $G \xrightarrow{a} A$ and $G \xrightarrow{b} B$.

Now consider $s = s'\sigma$ such that $V_G \xrightarrow{s'} \{A', B'\}$ and $\sigma(a') = \theta(b') = \theta(s')$ and $G \xrightarrow{a'} A'$ and $G \xrightarrow{b'} B'$. Consider two cases.

If $\sigma \in \Sigma_{nr}$, then by construction of $V_G$ (8), without loss of generality, there exist $x_a' \xrightarrow{a'} x_a'$ and $x_b' \xrightarrow{b'} x_b'$ such that $x_a' \xrightarrow{a'} x_a'$ and $b' \xrightarrow{B'} x_b'$. As $G \xrightarrow{a'} A'$, there exists $y'_a \xrightarrow{A'} x'_a$ such that $G \xrightarrow{a'} y'_a$. Furthermore, $x'_a, y'_a \in A'$ implies $x'_a \xrightarrow{a'} y'_a$, i.e., there exists $t \in \Sigma^*$ such that $x'_a \xrightarrow{a'} y'_a$. Thus $G \xrightarrow{a'} y'_a \xrightarrow{t} x'_a \xrightarrow{b'} x_b' \xrightarrow{B'} x_b'$. If $\sigma \in \Sigma_{nr}$, then by construction of $V_G$ (7) there exist $x_a' \xrightarrow{a'} x_a'$ and $x_b' \xrightarrow{b'} x_b'$ such that $x_a' \xrightarrow{a'} x_a'$ and $x_b' \xrightarrow{b'} x_b'$. As $G \xrightarrow{a'} A'$, there exists $y'_a \xrightarrow{A'} x'_a$ such that $G \xrightarrow{a'} y'_a$. Furthermore, $x'_a, y'_a \in A'$ implies $x'_a \xrightarrow{a'} y'_a$, i.e., there exists $t \in \Sigma^*$ such that $x'_a \xrightarrow{a'} y'_a$. Thus $G \xrightarrow{a'} y'_a \xrightarrow{t} x'_a \xrightarrow{b'} x_b' \xrightarrow{B'} x_b'$.

IV. IMPLEMENTATION

A. The OP-Verifier Algorithm

Algorithm 1 shows the pseudo-code of the OP-Verifier algorithm; this pseudo-code is the basis of the implementation of the OP-verifier algorithm within Supremica [22], which is further discussed in Section IV-C. The algorithm explores the state space of the verifier until a transition to $\bot$ is encountered, or until all possible verifier states have been constructed.

Verifier states are represented as ordered pairs $([x], [y])$ to represent a set $\{[x], [y]\} \in Q^0_S$, with singletons $[x]$ represented as $([x], [x])$. To exploit the symmetry, all pairs are ordered such that $|x| < |y|$ based on a fixed but arbitrary ordering of the $\Sigma_{nr}$-SCC. The algorithm maintains the set
Algorithm 1 OP-Verifier algorithm
1: input $G = (\Sigma, Q, \rightarrow, Q^c)$
2: calculate $G_{nr} = (\Sigma, Q / \epsilon^c, \rightarrow_{nr}, Q^c_{nr})$
3: queue $\leftarrow$ (empty queue)
4: visited $\leftarrow$ (empty hash set)
5: for all $[x] \in Q / \epsilon^c$ do
6: expand([x], [x])
7: end for
8: while queue not empty do
9: remove ([x], [y]) from queue
10: expand([x], [y])
11: end while
12: stop “The observer property is satisfied.”
13: procedure expand([x], [y])
14: for all $\sigma \in \Sigma_{nr}$ do
15: if $[x] \sigma_{nr} [x']$ and $[y] \sigma_{nr} [y']$ then
16: for all $[x] \sigma_{nr} [x']$ and $[y] \sigma_{nr} [y']$ do
17: enqueue([x'], [y'])
18: end for
19: else if $[x] \sigma_{nr} [y]$ and $[x] \sigma_{nr} [x']$ and $y$ is terminal or $[y] \sigma_{nr} [x]$ and $x$ is terminal then
20: stop “The observer property is not satisfied.”
21: end if
22: end for
23: for all $\sigma \in \Sigma_{nr}$ do
24: for all $[x] \sigma_{nr} [x']$ do
25: enqueue([x'], [y])
26: end for
27: for all $[y] \sigma_{nr} [y']$ do
28: enqueue([x], [y'])
29: end for
30: end for
31: procedure enqueue([x], [y])
32: if $[x] = [y]$ then
33: return
34: else if $[x] > [y]$ then
35: enqueue([y], [x])
36: else if $([x], [y]) \notin$ visited then
37: add ([x], [y], to visited)
38: add ([x], [y]) to queue
39: end if

visited containing all pairs $([x], [y])$ discovered so far, and a queue containing those pairs that still need to be explored.

After construction of the $\Sigma_{nr}$-SCC automaton using Tarjan’s Algorithm [26], the loop in line 5 examines every $\Sigma_{nr}$-SCC $[x] \in Q / \epsilon^c$ and records the successors of the verifier state $([x])$ according to (7) and (8). This step assumes that all states are reachable. Afterwards, the loop in line 8 visits and expands all verifier states $([x], [y])$ resulting from the previous loop in line 5, and their successors.

Procedure expand checks for transitions originating from a verifier state $([x], [y])$. For relevant events, the loop in line 14 checks in line 15 for successor pairs according to (7), and then checks in line 19 whether condition (9) is satisfied. If so, the verifier clearly contains the state $\bot$, so the algorithm terminates and reports that the observer property is not satisfied. For non-relevant events, the loop in line 23 constructs successor pairs according to (8). Procedure enqueue adds new state pairs to the set visited and to the queue, ensuring the ordering and using the hash set visited to prevent any pairs from being processed more than once.

B. Complexity

The complexity of the OP-Verifier algorithm is determined by the complexity to construct the verifier. If the input is a deterministic automaton $G = (\Sigma, Q, \rightarrow, Q^c)$ then the number of reachable states of $V_G$ is bounded by $|Q|^2 + |Q| + 1 = O(|Q|^2)$. To estimate the number of transitions of $V_G$, consider a transition $x \xrightarrow{\sigma} x'$ in the input automaton $G$, and let $y \in Q$ be an arbitrary state. If $\sigma \in \Sigma_{nr}$, then this produces at most one transition $([x], [y]) \xrightarrow{\sigma} ([x'], [y])$ or $([x], [y]) \xrightarrow{\bot}$ according to (7) or (9), and if $\sigma \in \Sigma_{nr}$, then there is one transition $([x], [y]) \xrightarrow{\sigma} ([x'], [y'])$ according to (8). That is, every transition of $G$ produces up to $|Q|$ transitions in $V_G$. The deterministic automaton $G$ has up to $|\Sigma||Q|$ transitions, so the total number of transitions of $V_G$ is bounded by $|\rightarrow||Q| \leq |\Sigma||Q|^2 = O(|\Sigma||Q|^2)$.

(11)

Tarjan’s algorithm to identify the $\Sigma_{nr}$-SCC runs in $O(|\rightarrow|) = O(|\Sigma||Q|^2)$ time [26], so it is dominated by the verification construction. Therefore, (11) gives the worst-case time complexity of the OP-Verifier algorithm.

C. Experimental Results

The OP-Verifier algorithm has been implemented in Java and integrated in the discrete event systems tool Supremica [22]. Table 1 shows some experimental results to demonstrate the performance of the implementation. All experiments were run on a standard desktop computer using a single core 2.33 GHz CPU and 3 GB of RAM.

The test suite consists of 23 automata obtained as intermediate results during compositional nonblocking verification [9], and variations of such automata. The table shows for each automaton that was checked, the number of states $|Q|$, the number of events $|\Sigma|$, the total number of transitions $|\rightarrow|$, and the number of non-relevant transitions $|\rightarrow_{nr}|$. Then the table shows the number of states $|Q_{nr}|$ constructed by the OP-Verifier algorithm, and the time taken to check the observer property. Furthermore, the time taken by Supremica [22] to compute a coarsest observation equivalence relation using the method in [27] is shown. This is the crucial step of the observer property verification algorithm proposed in [17] and is indicative of its performance. Finally, the table shows the verdict whether or not the given automaton satisfies the observer property.

Table 1 shows that the OP-Verifier algorithm can easily check automata with more than 100,000 states in a few seconds. With one exception, the number of verifier states is of the same order of magnitude as the number of states of the automaton, and the OP-Verifier algorithm runs significantly faster than observation equivalence. This is particularly true.
Table I

<table>
<thead>
<tr>
<th>Automaton</th>
<th>OP-Verifier</th>
<th>Time</th>
<th>Verdict</th>
</tr>
</thead>
<tbody>
<tr>
<td>14934</td>
<td>38 118793 16087</td>
<td>31023</td>
<td>0.01 s</td>
</tr>
<tr>
<td>105619</td>
<td>20 1273580 249928</td>
<td>387611</td>
<td>0.17 s</td>
</tr>
</tbody>
</table>

when the observer property is not satisfied, because the OP-Verifier algorithm can terminate early as soon as the state $\bot$ is encountered during construction of the verifier. The case where the OP-Verifier is slower than observation equivalence has the largest number of non-relevant transitions among the examples that satisfy the observer property, while the observation equivalence algorithm quickly finds a good partition in this case. In all other cases, the OP-Verifier algorithm gives an answer in less than two seconds.

V. CONCLUSIONS

The OP-Verifier algorithm presented in this paper allows to efficiently check whether an abstraction obtained by a natural projection has the observer property. The procedure is a modified version of a previous one [11], which removes a restriction on the existence of cycles of non-relevant events while still ensuring quadratic complexity in the number of states. The new version of the verifier first merges all states connected by cycles of non-relevant events. The resulting (non-deterministic) automaton is then translated into a transition structure, in which the observer property is checked by verifying the reachability of a specific state. We are currently investigating how the OP-Verifier can be used to improve the OP-Search algorithm [19] in order to help computing reduced OP-abstractions.

REFERENCES