Transformational Nonblocking Verification

Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnLine 55-28 (2022) 256–263

Transformational Nonblocking Verification [★]

Sander Thuijsman [*] Michel Reniers [*] Kai Cai [**]

[*] Eindhoven University of Technology, Eindhoven, The Netherlands
(e-mail: {s.b.thuijsman, m.a.reniers}@tue.nl)
[**] Osaka Metropolitan University, Osaka, Japan
(e-mail: cai@omu.ac.jp)

Abstract: Nonblocking verification can be applied to evaluate the behavior of discrete event systems. Performing nonblocking verification can be computationally costly. In this work, we consider discrete event systems that evolve over time. We study how to reuse results from a previous nonblocking verification, to more efficiently perform nonblocking verification when the system is adapted. We call this approach transformational nonblocking verification, and present an algorithm for the method. The efficiency of the method is evaluated by applying an academic and an industrial use case.

Keywords: Formal verification, discrete event systems, system evolution, nonblocking

1. INTRODUCTION

Supervisory control theory, as introduced by Ramadge and Wonham (1987, 1989), is a model-based approach to control discrete event (dynamic) systems. Typically cyber physical systems are modeled, where the physical system consists out of actuators and sensors. A supervisory controller can enable or disable particular events to occur in the actuators to steer the system. When the supervisory controller receives an event from a sensor, its state is updated which may then result in a new control command. These cyber physical discrete event systems are often safety-critical. Therefore, correct functioning of the system is vital. In order to guarantee correct behavior, model checking can be performed. Model checking is a formal verification technique to establish behavioral properties on the basis of a suitable model of a system through systematic inspection of all states in the model (Baier and Katoen (2008)).

In this paper, we focus on verification of the *nonblocking* property (Ramadge and Wonham (1989)) in discrete event systems. Nonblockingness indicates that the system can always progress to some marked state. The marking of states is chosen by the modeler, and typically indicates a situation where the system is stable or has progressed in some sense. Verification of nonblockingness is common in the supervisory control framework. A method to obtain a (nonblocking) supervisory controller is to apply supervisory controller synthesis and algorithmically compute the supervisor based on an uncontrolled system formulation and a set of formalized requirements (Ramadge and Wonham (1989)). Supervisory controller synthesis may require considerable computational effort because it suffers from *state space explosion*. Technically, all possible combinations of states of components in the system must be taken into account. Therefore, adding a small component to the model might induce a large increase to the total system state space. A common way of mitigating state space explosion is by applying modular supervisory controller synthesis techniques, that split the problem into multiple sub-problems. Some popular methods are modular (Ramadge and Wonham (1989)), decentralized (Rudie and Wonham (1992)), hierarchical (Zhong and Wonham (1990)), compositional (Flordal et al. (2007)), coordinated (Komenda et al. (2012), distributed (Su et al. (2010)), and multilevel supervisory control (Komenda et al. (2016)). Unfortunately, for these methods often nonblockingness of the synchronous controlled system cannot be guaranteed (or only under certain conditions). In such a circumstance, one may choose to apply nonblocking verification. Unfortunately, system-wide verification of nonblockingness also suffers from state space explosion.

Cyber physical systems generally evolve over time. Lehman (1996) has defined the *laws of software evolution*, these describe what changes typically occur during a software's lifetime. The laws themselves have evolved over the years, but *the law of continuing change* has consistently been a part of them. This law states that a controller must continually be adapted, otherwise it becomes progressively less satisfactory. Frequent modifications of the supervisory control of an industrial system are also observed in the use case we study in this paper, based on van der Schriek (2018).

In case *nonblocking verification* (NBV) is performed for a particular system, its outcome is not valid anymore once this system is adapted. In this paper, we investigate how to reduce the computational cost of NBV in the case of an evolving system. We assume a *base model*, on which NBV is already performed. After NBV on the base model, the base model is adapted such that a *variant model* is created. The goal is to use the verification result of the base system, to more efficiently perform NBV for the variant model. We call this *transformational nonblocking verification* (TNBV).

[★] Research leading to these results has received funding from the EU ECSEL Joint Undertaking under grant agreement n° 826452 (project Arrowhead Tools) and from the partners national programs/funding authorities.

10.1016/j.ifacol.2022.10.351

1.1 Related work

The idea we present here originates from the *transformational supervisor synthesis* framework introduced in Thuijsman and Reniers (2020, 2022), where the result of the synthesis of a base system and the difference between the base and variant system (defined in a so-called model delta) are used to more efficiently compute the supervisor for the variant system.

For related work in the area of model checking, where information of previous verification runs is reused in new verifications, an overview of regression verification is given in Beyer and Wendler (2013). Conditional model checking (Beyer et al. (2012)) is used to reuse partial verification results when an earlier verification run did not verify the complete system. A condition can be input to direct the verifier to the parts of the program that still need to be verified. Some approaches reuse witnesses, that are counterexamples provided by the verifier, to reverify if the result of the model checker is still correct when the system is adapted (Henzinger et al. (2003)). Other approaches use auxiliary information from a previous verification, for instance reuse of abstractions that have been applied is discussed in Beyer et al. (2013). In Yang et al. (2009) state mappings are generated to aid identification of states of interest in new verifications. One or more of the following aspects generally contrast our work to existing work: We use state sets to pass information between the verifications, we do not create any additional data structures specifically for usage in some next computation (i.e., the state sets we store always need to be computed for verification), we provide a theoretical bound where the transformational approach is always equal or less costly than the conventional approach, and our approach is specifically designed for verification of nonblockingness.

There is related work that studies efficient formal verification without considering system evolution. For instance, symbolic model checking (Clarke et al. (1996)) using binary decision diagrams where explicit enumeration of the state space is avoided. In compositional verification (Graf and Steffen (1990)), and the hierarchical approaches of Leduc et al. (2005) and Feng and Wonham (2008), the system is abstracted or divided before or during verification to reduce state space sizes but still give the same verification result. In Mohajerani et al. (2015) a compositional verification approach is introduced that is specifically tailored to nonblocking verification. We deliberately do not apply symbolic or abstraction methods, in order to prevent this work becoming too convoluted. In the future, we believe synergistic verification methods may be developed using notions from transformational, symbolic, abstracted, and other efficient approaches.

1.2 Structure

In Section 2 we discuss preliminaries on automata and reachability searches. An NBV and a TNBV algorithm are introduced and discussed in Section 3. Some experiments to evaluate the efficiency of the methods are discussed in Section 4. Finally, conclusions are provided in Section 5.

2. PRELIMINARIES

We consider finite state automaton A defined as a *5-tuple*: $A = (X, \Sigma, T, X_0, X_m)$, where X is the finite set of states, of which $X_0 \subseteq X$ is the set of initial states and $X_m \subseteq X$ is the set of marked states. Σ is the finite set of events. T is the finite set of transitions, a transition is a triple: $(x_{or}, \sigma, x_{tar}) \in X \times \Sigma \times X$, specifying a transition from origin state x_{or} to target state x_{tar} over event σ.

A state is *coreachable* if from it a sequence of transitions can be followed that leads to a marked state. A state is *reachable* if from some initial state a sequence of transitions can be followed that leads to that state. For computation of coreachable and reachable states we can use breadth-first search reachability algorithms provided in Algorithms 1 and 2, taken from Kleinberg and Tardos (2005) and adapted in Thuijsman and Reniers (2022). The coreachable states X_{cr} can be computed through a Backward Reachability Search (BRS) from the marked states: $X_{cr} = \texttt{BRS}(X, \Sigma, T, X_m)$. The reachable states X_r can be computed through a Forward Reachability Search (FRS) from the initial states: $X_r = \texttt{FRS}(X, \Sigma, T, X_0)$. An automaton is called (co-)reachable if all its states can be defined as such. An automaton for which all reachable states are coreachable is called *nonblocking*, i.e., $X_r \subseteq X_{cr}$. The introduced reachability algorithms are used in the NBV algorithms that are discussed next. They have linear complexity: $\mathcal{O}(|T|)$ (Kleinberg and Tardos (2005)).

Algorithm 1 Backward Reachability Search `BRS`

```
Input: State set $X$, alphabet $\Sigma$, finite set of transitions $T$, starting set $X_\alpha$
Output: State set $X_\omega$ in $X$ from which a sequence of transitions in $T$ exists through states in $X$, using events in $\Sigma$, to a state in $X_\alpha \cap X$
 1: $T_p = T \cap (X \setminus X_\alpha) \times \Sigma \times X$
 2: $X_\omega = X_\alpha \cap X$
 3: $currentLayer = X_\alpha \cap X$
 4: while $currentLayer \neq \emptyset$
 5:     $nextLayer = \emptyset$
 6:     for all $x \in currentLayer$ do
 7:         for all $(x_{or}, \sigma, x) \in T_p$ do
 8:             if $x_{or} \notin X_\omega$
 9:                 $X_\omega = X_\omega \cup \{x_{or}\}$
10:                 $nextLayer = nextLayer \cup \{x_{or}\}$
11:             end if
12:         end for
13:     end for
14:     $currentLayer = nextLayer$
15: end while
16: return $X_\omega$
```

Algorithm 2 Forward Reachability Search `FRS`

```
Input: State set $X$, alphabet $\Sigma$, finite set of transitions $T$, starting set $X_\alpha$
Output: State set $X_\omega$ in $X$ to which a sequence of transitions in $T$ exists through states in $X$, using events in $\Sigma$, from a state in $X_\alpha \cap X$
 1: $T^R = \{(x_{tar}, \sigma, x_{or}) | (x_{or}, \sigma, x_{tar}) \in T\}$
 2: $X_\omega = \texttt{BRS}(X, \Sigma, T^R, X_\alpha)$
 3: return $X_\omega$
```

3. TRANSFORMATIONAL NONBLOCKING VERIFICATION

In this section we assume NBV is first applied to base model A. We store information, specifically sets of states, of this base verification. A variant model A' is given. The goal is to use the information of the NBV of base model A to perform efficient NBV for variant model A', instead of performing a whole new verification afresh.

Note that for our method any pair of well-defined automata is permissible as base and variant model, i.e., there are no restrictions on (the difference between) the automata. The method also permits non-deterministic automata.

In this work we assume that one may sometimes know (before any verification) for certain that an automaton is reachable. For instance, this may be the case when synchronous composition (Cassandras and Lafortune (2008)) is performed to obtain a single automaton from a network of automata, for which many algorithms in practice will yield a reachable automaton by construction. If it is known beforehand that an automaton is reachable, some steps in NBV may be skipped. In the algorithms we present, we use a Boolean *reachable* that is true when it is known beforehand that the input automaton is reachable. Our algorithms are also applicable to automata that are not reachable, or for which the reachability is unknown. In that case, the Boolean *reachable* is set to false.

In this section we first introduce a basic NBV algorithm. Then, we introduce the TNBV algorithm. Next, some statements on computational cost of the algorithms are given. Finally some examples of the functioning of the algorithms are provided.

3.1 Nonblocking verification algorithm

We use Example 1 to introduce the concepts and algorithms.
Example 1. Let us consider automaton A given in Fig. 1. States are represented by circles. Initial states have a dangling incoming arrow. Marked states have a double circle representation. Transitions are shown by arrows between states, with the event label displayed next to them.

We observe that states $\{x_0, x_1, x_3, x_4, x_5, x_6\}$ are reachable and states $\{x_0, x_3, x_6\}$ are coreachable. Since there are reachable states that are not coreachable, automaton A is blocking. ▲

The basic NBV algorithm given in Algorithm 3 can be used to determine nonblockingness of an automaton. In

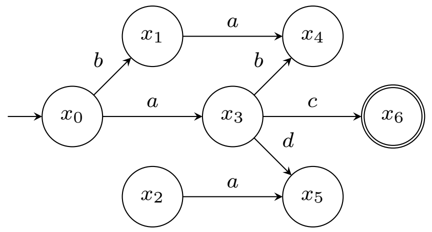

Fig. 1. Automaton Example 1: A

Algorithm 3 Nonblocking verification `NBV`

```
Input: Automaton $A = (X, \Sigma, T, X_0, X_m)$, Boolean reachable
Output: Boolean result, nonblocking states $N$, reachable states $Q$
1: if not reachable then
2:     $Q \leftarrow$ FRS$(X, \Sigma, T, X_0)$
3:     $(X, \Sigma, T, X_0, X_m) \leftarrow$ restrict$(A, Q)$
4: else
5:     $Q \leftarrow X$
6: end if
7: $N \leftarrow$ BRS$(Q, \Sigma, T, X_m)$
8: return $(N = Q, N, Q)$
```

Algorithm 4 Restrict automaton `restrict`

```
Input: Automaton $A = (X, \Sigma, T, X_0, X_m)$, states $Q$
Output: Restricted automaton $A'$
1: return $(X \cap Q, \Sigma, T \cap (Q \times \Sigma \times Q), X_0 \cap Q, X_m \cap Q)$
```

case automaton A is not (known to be) reachable, first the reachable states Q are computed in line 2. Then, using the `restrict` function provided in Algorithm 4, the automaton is reduced to its maximal reachable part. For automaton A of Example 1, this would result in state x_2 and transition (x_2, a, x_5) being pruned away. For the reachable automaton, the set of nonblocking states N is computed that represents all states from which a marked state can be reached. The algorithm returns true as a result when $N = Q$, otherwise it will return result false. Next to the result, NBV outputs the state sets N and Q, which we will use later in our transformational approach.

3.2 Transformational nonblocking verification algorithm

In this section we discuss a TNBV algorithm that computes the nonblocking result based on the output of a previous NBV. We illustrate the idea using Example 2. We discuss conceptually how a modification can influence nonblockingness, and then present the TNBV algorithm based on those concepts.
Example 2. Let us consider variant automaton A' in Fig. 2. We observe that automaton A' is obtained by making a modification to automaton A of Example 1, transition (x_4, c, x_6) is added.

First, we evaluate how the addition of this transition might have influenced the reachability of states. We know that the origin and target state of this transition were already reachable. Therefore, all states that are reachable from these states were already evaluated in the NBV of base automaton A, so there are no new reachable states in A'. Also, the addition of a transition cannot lead states that

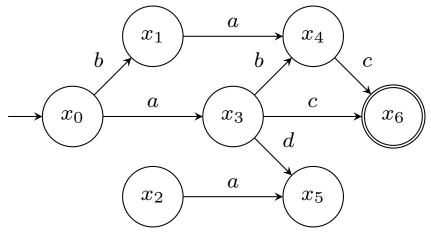

Fig. 2. Automaton Example 2: A'

were reachable before to become unreachable. Therefore, without any further analysis, we can state that the set of reachable states of A' are the same as those of A.

Second, we evaluate how the addition of this transition might have influenced the coreachability of states. The origin state, x_4, of the added transition was not coreachable in A, while the target state, x_6, was coreachable. Because x_4 now has a path to a coreachable state, it is now coreachable in the variant automaton. In turn, more states that were non-coreachable before may now be coreachable in the variant automaton. Therefore, we can perform a BRS for the variant automaton to find all coreachable states. In this search, we do not have to evaluate transitions between states that were already coreachable in the base automaton, because we know they will still be coreachable. In the end, the set of coreachable states is not the same as the set of reachable states, and A' is therefore blocking. ▲

In Algorithm 5, a TNBV algorithm is provided. As input it requires a base automaton, the nonblocking states of the base automaton, a variant automaton, and the Boolean reachable variable for the variant automaton. It will output the nonblocking result of the variant automaton, as well as its nonblocking states and reachable states.

Algorithm 5 follows the same structure as Algorithm 3: First, if the (variant) automaton is not reachable, it is restricted to its reachable part. For the remaining reachable automaton, the coreachable states are computed. If all reachable states are coreachable, the automaton is nonblocking. In contrast to NBV, TNBV uses the reachable and nonblocking state sets from a previous NBV to more efficiently compute the reachable and nonblocking state sets for the variant automaton.

Let us first consider the case that the variant automaton A' is reachable, i.e., *reachable=true*. As a result of the if-statement, lines 2-12 are not performed. Since A' is reachable, $Q' = X'$.

In line 16, the base automaton is restricted to only the reachable part. For the remaining lines in the algorithm, we can reason on a base and variant automaton that are reachable.

In line 17, the added and removed transitions and the states for which the marked property is added or removed are computed.

In lines 18-23 the algorithm computes the set of states that are nonblocking for the variant automaton and that also were nonblocking in the base automaton. States that were nonblocking might become blocking because of states with removed marked property, or removed transitions. A state set X_a is computed that contains all states with removed marked property (these are always in N since marked states are always nonblocking in the (reachable) base model), and all states in N from which a transition is removed that leads to some state in N. If there are no states in X_a, we know that all previously nonblocking states are still nonblocking for the variant model. If there are states in X_a, we perform a BRS over the state space spanned by N from the marked states X'_m to find all nonblocking states in N. When during this BRS all states in X_a are found, we do not have to finish the search; we

Algorithm 5 Transformational nonblocking verification `TNBV`

Input: Base automaton $A = (X, \Sigma, T, X_0, X_m)$, nonblocking states N, reachable states Q, variant automaton $A' = (X', \Sigma', T', X'_0, X'_m)$, Boolean *reachable*
Output: Boolean result, nonblocking states N', reachable states Q' of variant model

```
1:  if not reachable then
2:      $T^+ \leftarrow T' \setminus T$, $T^- \leftarrow T \setminus T'$, $X_0^+ \leftarrow X'_0 \setminus X_0$, $X_0^- \leftarrow X_0 \setminus X'_0$
3:      $X_b \leftarrow (X_0^- \cap Q) \cup \{x_{tar} \in Q | \exists x_{or} \in Q,\ \sigma \in \Sigma : (x_{or}, \sigma, x_{tar}) \in T^-\}$
4:      if $X_b \neq \emptyset$ then
5:          $Q' \leftarrow$ FRS$(Q, \Sigma', T', X'_0)$: break after line 9 of BRS when $X_b \subseteq X_\omega$, do $Q' \leftarrow Q$
6:      else
7:          $Q' \leftarrow Q$
8:      end if
9:      if $T^+ \cap (Q' \times \Sigma' \times (X' \setminus Q')) \neq \emptyset \vee X_0^+ \cap (X' \setminus Q') \neq \emptyset$ then
10:         $Q' \leftarrow$ FRS$(X', \Sigma', T', Q' \cup X'_0)$
11:     end if
12:     $(X', \Sigma', T', X'_0, X'_m) \leftarrow$ restrict$(A', Q')$
13: else
14:     $Q' \leftarrow X'$
15: end if
16: $(X, \Sigma, T, X_0, X_m) \leftarrow$ restrict$(A, Q)$
17: $T^+ \leftarrow T' \setminus T$, $T^- \leftarrow T \setminus T'$, $X_m^+ \leftarrow X'_m \setminus X_m$, $X_m^- \leftarrow X_m \setminus X'_m$
18: $X_a \leftarrow X_m^- \cup \{x_{or} \in N | \exists x_{tar} \in N,\ \sigma \in \Sigma : (x_{or}, \sigma, x_{tar}) \in T^-\}$
19: if $X_a \neq \emptyset$ then
20:     $N' \leftarrow$ BRS$(N, \Sigma', T', X'_m)$: break after line 9 of BRS when $X_a \subseteq X_\omega$, do $N' \leftarrow N$
21: else
22:     $N' \leftarrow N$
23: end if
24: if $T^+ \cap ((X' \setminus N') \times \Sigma' \times N') \neq \emptyset \vee X_m^+ \cap (X' \setminus N') \neq \emptyset$ then
25:     $N' \leftarrow$ BRS$(X', \Sigma', T', N' \cup X'_m)$
26: end if
27: return $(N' = Q', N', Q')$
```

know that all states that were nonblocking in the base model are nonblocking in the variant model. So each time after line 9 in BRS we can check whether all states in X_a have been found, if this is the case we break (terminate BRS) and perform $N' \leftarrow N$. If not all states in X_a are found, BRS will terminate as normal and return all nonblocking states within the state space spanned by N.

Next, in line 24-26 the algorithm finds all states outside N' that are nonblocking. If there are no added transitions that have an origin state outside N' and a target state within N', and there are no added marked states outside N', it is not necessary to perform a new search as no nonblocking states outside N' will exist. These states would have already been found as nonblocking for the base model. In case a new BRS is performed, it is already initiated with N' in the initial state set. This part of the state space is not searched again, because of the pruning that is performed in line 1 of BRS.

Algorithm 5 returns true as a result when $N' = Q'$, otherwise it will return result false. The algorithm also outputs the state sets N' and Q' as respectively the nonblocking and reachable states of the variant automaton.

Let us now consider the case that the variant automaton is not (known to be) reachable, i.e., *reachable* is false.

In lines 1-11 the reachable states of the variant automaton are computed, based on modifications that were made from A to A'. State set X_b contains all states in Q that are states from which the initial property is removed, and all states in Q to which a transition has been removed that originates from a state in Q. An FRS is performed to find all states in Q that are reachable in the variant automaton. This `FRS` breaks when all states in X_b are found and then $Q' \leftarrow Q$ is performed. Otherwise, `FRS` terminates as normal. Next, in lines 9-11, the reachable states outside Q' are also found. They are only searched for when there are added transitions from Q' to a state outside Q', or if there is a state with added initial property outside Q'. Otherwise, the search does not need to take place and Q' remains unchanged.

After computing the reachable states Q', both the base and variant automaton are restricted to their reachable part, and the algorithm continues as discussed.

Theorem 1 states that the result of TNBV and NBV are the same.

Theorem 1. Given automata A and A', NBV output $(\mathit{result}, N, Q) = \texttt{NBV}(A, \mathit{reachable})$, and Booleans *reachable* and *reachable'* that imply their respective automaton A or A' is reachable; then $\texttt{TNBV}(A, N, Q, A', \mathit{reachable}') = \texttt{NBV}(A', \mathit{reachable}')$.

Proof. Algorithm 5 terminates because the reachability searches are known to terminate and there are no other loops. It follows from the explanations above that the result of Algorithms 3 and 5 is the same. □

Example 3. Let us consider the case that a variant automaton A' in Fig. 3 is constructed. We assume that without any computations it is known that A' is reachable. This could be the case when, e.g., the automaton is created by performing a synchronous composition algorithm that always yields a reachable automaton.

For TNBV, we use automaton A from Example 1 as the base automaton. The algorithm can be performed with *reachable=true*, so no FRS will be performed and $Q' \leftarrow X'$. We note that after restricting the base automaton to its reachable part, it is the same as the variant automaton. Therefore, in relation to the restricted automaton, the variant automaton has no added or removed transitions or marked states. No BRS is performed and $N' = N$. Since $N' \neq Q'$, the variant automaton is blocking. ▲

3.3 Computational cost

The purpose of TNBV is to verify the nonblocking property for some variant automaton when a (T)NBV has been performed for some base automaton, with reduced computational cost compared to NBV. Both NBV and TNBV make calls to the BRS and/or FRS algorithms. Because the BRS and FRS algorithms have linear complexity, NBV and TNBV in turn also have linear complexity. However, because TNBV performs these reachability searches over smaller state spaces, or the searches terminate earlier, it is still more computationally efficient than NBV. In this section we discuss how the computational cost of TNBV is always lower than (or at worst equal to) the cost of NBV.

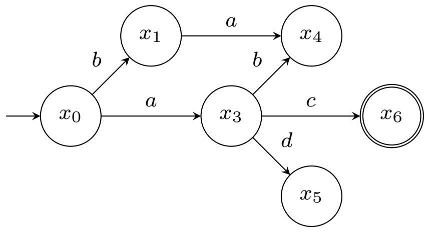

Fig. 3. Automaton Example 3: A'

We use the *transition evaluation count* θ from Thuijsman and Reniers (2020) to express the computational cost for performing (T)NBV. When performing (T)NBV, we start with a count of $\theta = 0$. Every time we reach line 8 in Algorithm 1 (BRS), we increment θ by one. The total count at the end of (T)NBV indicates the effort that is spent searching the state space. Because NBV and TNBV both consist of performing BRS (sometimes indirectly through FRS), we can compare the computational costs of the algorithms by comparing their transition evaluation counts in BRS.

Example 4. We demonstrate the transition evaluation count for the previous examples. For NBV of A in Example 1: $\theta = 8$ (6 during FRS and 2 during BRS). For the TNBV of A' discussed in Example 2: $\theta = 2$. If we were to perform NBV for A' of Example 2, then $\theta = 13$. For TNBV of A' discussed in Example 3: $\theta = 0$. If we were to perform NBV for A' of Example 3, then $\theta = 8$. ▲

In the following we show that the computational cost of performing TNBV is lower or equal to the computational cost of performing NBV. We denote $A = (X, \Sigma, T, X_0, X_m)$ and $A' = (X', \Sigma', T', X'_0, X'_m)$.

Essentially, both NBV and TNBV consist of two stages: First an FRS stage and second a BRS stage. Let us discuss the FRS stage first. If *reachable'=true*, then no transitions are evaluated in the FRS stage for both NBV and TNBV. If *reachable'=false*, then for FRS in NBV all transitions in $\{(x_{or}, \sigma, x_{tar}) \in T' | x_{or} \in Q' \wedge x_{tar} \in Q' \wedge x_{tar} \notin X'_0 \wedge \sigma \in \Sigma'\}$ are evaluated, where Q' are all reachable states.

For either FRS call in TNBV, at most all transitions in the previously mentioned set are evaluated, since all transitions with $x_{tar} \in X'_0$ are pruned away in line 1 of BRS, and evaluating any transition with $x_{tar} \notin Q'$ is impossible, since its evaluation would add it to Q'. Furthermore, no transition is evaluated twice when computing Q' through FRS in TNBV. By construction of the reachability search algorithm, no transition is evaluated twice during one call to the algorithm, because the transitions of a state are iterated only once.

Let us call the set of reachable states found in line 5 of TNBV Q'_1. At most, all transitions in $T'_1 = \{(x_{or}, \sigma, x_{tar}) \in T' | x_{or} \in Q'_1 \wedge x_{tar} \in Q'_1 \wedge x_{tar} \notin X'_0 \wedge \sigma \in \Sigma'\}$ will have been evaluated. In the second FRS call (line 10), all transitions in $T'_2 = \{(x_{or}, \sigma, x_{tar}) \in T' | x_{or} \in Q' \wedge x_{tar} \in Q' \wedge x_{tar} \notin (Q'_1 \cup X'_0) \wedge \sigma \in \Sigma'\}$ are evaluated. We note that $T'_1 \cap T'_2 = \emptyset$, since in T_1 all transitions have $x_{tar} \in Q'_1$, and in T_2 all transitions have $x_{tar} \notin Q'_1$.

In conclusion, for the FRS stage there are no transitions that are evaluated in TNBV that would not be evaluated in NBV, and no transition is evaluated twice in TNBV.

Therefore, the FRS stage of TNBV never has more transition evaluations than the FRS stage of NBV. The same logic can be applied for the BRS stage. Hence, the computational cost (expressed in transition evaluation count) of TNBV is always equal to or lower than the computational cost of NBV, as is stated in Theorem 2.

Theorem 2. Given a base automaton A, its NBV output $(\mathit{result}, N, Q) = \texttt{NBV}(A, \mathit{reachable})$, some variant automaton A', and a Boolean $\mathit{reachable}'$ that implies A' is reachable; the computational cost of $\texttt{TNBV}(A, N, Q, A', \mathit{reachable}')$ is lower or equal to $\texttt{NBV}(A', \mathit{reachable}')$, i.e., the transition evaluation count θ after performing TNBV is equal to or lower than θ after performing NBV.

Proof. It follows from the explanations above that the computational cost of TNBV is equal to or lower than the computational cost of NBV. □

The result that TNBV has equal or lower computational cost to NBV is particularly notable, because for the similar method in supervisor synthesis, the computational cost of transformational supervisor synthesis may be much higher than 'ordinary' supervisor synthesis (Thuijsman and Reniers (2022)).

4. EXPERIMENTS

Even though we showed in the previous section that the computational cost expressed in transition evaluation count is always equal or lower for TNBV compared to NBV, this does not mean that an implementation of the TNBV algorithm is always more efficient than NBV because of practical reasons how the algorithms are implemented. Therefore we present some experiments with wall-clock time measurements to evaluate the benefit of the approach we present in this paper.

For the experiments, a proof-of-concept implementation of the above verification algorithms, and models of the case studies we describe below, have been made in Matlab [1]. Wall-clock time is used to represent the time effort of performing verification. The experiments were performed on an HP ZBook Studio G4 laptop, using an Intel i7 processor clocked at 2.8 GHz. Matlab used around 1 GB of memory, regardless of the model size. Filesizes to store the models ranged from a few KB to a few MB.

In Section 4.1 we consider the Transfer Line model as an academic case study, and in Section 4.2 we consider a Lithography Machine Wafer Logistics controller as an industrial case study. These are the same case studies as presented in Thuijsman and Reniers (2022).

The monolithic models that we use for these experiments are obtained through synchronous composition of a network of automata. In Thuijsman and Reniers (2022), a sink-state was introduced in the model, which is not done here as it is only required for supervisor synthesis and not NBV. The monolithic models are computed in preparation of the experiments. The preparatory computations are not included in the computational effort measurements, because this matches the experiments to the monolithic level discussed in the theoretical part, and because computing the synchronous composition is required for both NBV and TNBV discussed here.

[1] The algorithms and models can be found here: `https://github.com/sbthuijsman/WODES_TNBV`

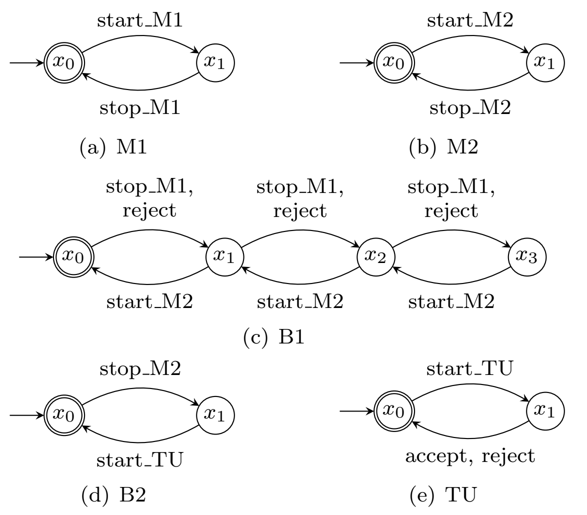

Fig. 4. Transfer Line automata

The construction of A can be done by computing a synchronous composition that also contains non-reachable states. Practically however, often only the reachable part of A is constructed. We consider both options in the case studies below.

4.1 Transfer Line

We first consider the Transfer Line model from Wonham and Cai (2019) as an academic case study. In this model, products are being processed by two machines. Machine M1 takes products from the environment, and processes them. After processing, M1 places the product in buffer B1, which can hold up to three products. Machine M2 takes products from B1, processes them, and places them in buffer B2, which can hold only one product. Test unit TU takes products from B2, and tests them. If the product is accepted, it is released from the system. If the product is rejected, it goes back to B1. Controllability of events is irrelevant for the method and not taken into account. The automata models are shown in Fig. 4. The synchronous product over all automata is taken. The resulting base automaton TL has 64 states and 168 transitions.

The model evolution cases we present are the same as presented in Thuijsman and Reniers (2022). The following five variant automata, TL'_1 to TL'_5, have been generated by making adaptations to TL.

- TL'_1: Reduced capacity of B1 to two products: state x_3 of automaton $B1$ removed, and transitions $(x_2, \text{stop\_M1}, x_3)$, $(x_2, \text{reject}, x_3)$, $(x_3, \text{start\_M2}, x_2)$ removed.

Table 1. Transfer Line experimental results

Evolution	**Variant model size**		`NBVr`		`TNBVr`		$\lvert N'\rvert$
	$\lvert X'\rvert$	$\lvert T'\rvert$	θ	ms	θ	ms	
TL to TL'_1	64	120	119	0.3	119	0.4	64
TL to TL'_2	96	268	267	0.6	84	0.5	96
TL to TL'_3	64	168	167	0.5	0	0.4	64
TL to TL'_4	64	184	183	0.5	0	0.4	64
TL to TL'_5	96	192	191	0.5	179	0.6	96

Table 2. Wafer logistics experimental results

Evolution	**Complete**							**Reachable**						
	Variant model size	NBV		TNBV				Variant model size	NBV		TNBV			
	$\vert \longrightarrow' \vert$	θ	sec	θ		sec		$\vert \longrightarrow' \vert$	θ	sec	θ		sec	
B1-B2	1022004	991460	130	0	-100%	0	-100%	873516	118028	17	0	-100%	0	-100%
B2-B3	1133460	1101482	177	72096	-93%	26	-85%	968604	132972	22	13808	-90%	14	-36%
B3-B4	1082052	1050722	158	924650	-12%	116	-27%	924740	126072	19	126072	0%	3	-84%
B4-B5	1100484	1068850	162	0	-100%	0	-100%	940436	128504	20	0	-100%	0	-100%
B5-B6	1100484	1068850	162	753727	-29%	91	-44%	940436	128504	20	109964	-14%	3	-85%
B6-B7	1109124	1077346	163	759853	-29%	92	-44%	947796	129640	20	110882	-14%	2	-90%
B7-B8	1179092	1145451	185	814894	-29%	118	-36%	1007344	138200	23	118757	-14%	17	-26%
B8-B9	1082052	1050722	159	924650	-12%	116	-27%	924740	126072	19	126072	0%	3	-84%
B9-B10	1179092	1145451	186	54946	-95%	26	-86%	1007344	138200	23	7912	-94%	15	-35%
B10-B11	1244164	1208737	211	54304	-96%	29	-86%	1062724	146108	28	7260	-95%	17	-39%

- TL'_2: Increased capacity of B2 to two products: added a state x_2 and transitions $(x_1, \text{stop\_M2}, x_2)$, $(x_2, \text{start\_TU}, x_1)$ to B2.
- TL'_3: B1 initially holds one product instead of zero: removed initial property of state x_0, and added initial property to state x_1 in automaton B1.
- TL'_4: TU may send the product to B2 upon completion: added event 'retest', added transition $(x_1, \text{retest}, x_2)$ to TU, and added transition $(x_0, \text{retest}, x_1)$ to B2.
- TL'_5: Capacity of B1 and B2 is two products each: removed state x_3 and transitions $(x_2, \text{stop\_M1}, x_3)$, $(x_2, \text{reject}, x_3)$, $(x_3, \text{start\_M2}, x_2)$ from B1, state x_2 and transitions $(x_1, \text{stop\_M2}, x_2)$, $(x_2, \text{start\_TU}, x_1)$ are added to B2.

As the TNBV methods are on a monolithic state space, the adaptations for the individual automata are converted to adaptations on the synchronous state space for the experiments, as discussed before. For the base and variant models, all states constructed during the synchronous composition are reachable. Therefore we only apply the algorithms with *reachable=true* for this case study.

For each of the variant automata, the number of states and transitions are given in Table 1. For each variant model, an NBV is performed in two ways. The first by doing a completely new verification given in Algorithm 3 and the second by using the verification result of the base automaton and applying TNBV (Algorithm 5). For each verification, the computational cost is shown in Table 1 by means of θ as well as measured runtime shown in milliseconds. Percentage changes are shown that compare the computational cost of TNBV to NBV for each model. The runtime is the mean from 100 runs for each verification. The runtimes and percentages are rounded to the nearest integer.

We observe that all variant models are nonblocking, since $|X'| = |N'|$ for each model. For NBV, the computational cost in θ can be lower than the number of transitions in the model because of the pruning that occurs in line 1 of Algorithm 1. In this case, for every variant model there was a single transition from the marked state that was pruned away. All other transitions were evaluated. We observe that TNBV has a lower or equal computational cost in θ than NBV, which was also expected from the theoretical result. Because the runtimes are very low, conclusions cannot be made based on the measured wall-clock time for this case study.

4.2 Lithography Machine Wafer Logistics

Next we present an industrial case study. This case study is performed using models from ASML. ASML is the world-leading manufacturer of lithography machines, which are used in the semiconductor industry to produce integrated circuits. These circuits are printed on silicon wafers. The movement of these wafers through the machine is called the Wafer Logistics, which is studied in van der Sanden et al. (2015) and van der Schriek (2018). van der Schriek (2018) presents a study on how the components of the Wafer Logistics controller evolve over time. In this study equivalent automata models of the component controllers are constructed. These automata models are constructed for the variation points that the components evolved to. We use these automata models here, to investigate the efficiency of TNBV in this industrial setting.

'Component B' of van der Schriek (2018) is selected to perform the experiments on, based on its large but manageable state space size, the number of variation points, and the variety of changes between the models. The first 11 variation points of this model are taken, to investigate 10 adaptations. Opposed to the Transfer Line experiment, where each variant model was an adaptation of the same base model, we now consider incremental adaptations. So we start with the evolution from B1 to B2, then from B2 to B3, from B3 to B4, and so on. The same models are used in Thuijsman and Reniers (2022).

Unlike the Transfer Line model, for the models of Component B not all states are reachable. Therefore we perform two pairs of experiments, one for the complete model containing unreachable states (i.e., *reachable=false*), and one with only reachable states (*reachable=true*). The first model, B1, of the complete system consists of 69 120 states and 1 017 684 transitions. The model of the reachable system is smaller: 59 184 states and 869 836 transitions.

The experimental results are summarized in Table 2. The computational cost is displayed for NBV and TNBV for both systems. Each runtime value is the mean over 10 verification runs and is rounded to the nearest second. There is only a weak correlation between wall-clock time and the computational cost in θ, due to other steps in the computation such as pruning and finding adjacent transitions for the states in the current layer. None of the models were nonblocking.

NBV for the reachable wafer logistics model is quicker than the complete model because the model is smaller, and the reachability search only needs to be performed in backwards direction rather than both forwards and backwards. For convenience, percentages are added to Table 2 that compare the cost of TNBV to NBV for the same model. We observe that for all models the computational cost of TNBV is lower than or equal to the computational cost of NBV. The efficiency is case specific, TNBV had better reductions in computational cost for some models than others. Regardless, TNBV is more efficient than NBV for both measurement methods.

5. CONCLUSION

We present an algorithm for TNBV, that reuses the result of a previous NBV to efficiently generate the nonblocking result for a discrete event system every time it is adapted. The algorithms are explained and examples are provided. It is shown that the computational cost, expressed in transition evaluation count, of TNBV is always equal to or lower than the computational cost of NBV. The method is evaluated by means of an academic and an industrial use case. From the experiments, it is shown that the runtime of TNBV is indeed lower than NBV.

This work is based on a basic NBV algorithm. At the moment TNBV is likely less efficient than NBV methods that use, e.g., abstractions or symbolic representations. However, in the future NBV approaches may be investigated that use concepts from various methods, among which the concepts of a transformational approach that are introduced in this work.

REFERENCES

Baier, C. and Katoen, J. (2008). *Principles of Model Checking (Representation and Mind Series)*. The MIT Press.

Beyer, D., Henzinger, T., Keremoglu, M., and Wendler, P. (2012). Conditional model checking. In *Proceedings of the ACM International Symposium on the Foundations of Software Engineering*. ACM Press.

Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., and Wendler, P. (2013). Precision reuse for efficient regression verification. In *Proceedings of Joint Meeting on Foundations of Software Engineering*. ACM Press.

Beyer, D. and Wendler, P. (2013). Reuse of verification results. In *Model Checking Software*, 1–17. Springer Berlin Heidelberg.

Cassandras, C. and Lafortune, S. (2008). *Introduction to Discrete Event Systems*. Springer, Boston, MA, USA, 2nd edition.

Clarke, E., McMillan, K., Campos, S., and Hartonas-Garmhausen, V. (1996). Symbolic model checking. In *Computer Aided Verification*, 419–422. Springer Berlin Heidelberg.

Feng, L. and Wonham, W. (2008). Supervisory control architecture for discrete-event systems. *IEEE Transactions on Automatic Control*, 53(6), 1449–1461.

Flordal, H., Malik, R., Fabian, M., and Åkesson, K. (2007). Compositional synthesis of maximally permissive supervisors using supervision equivalence. *Journal of Discrete Event Dynamic Systems*, 17(4), 475–504.

Graf, S. and Steffen, B. (1990). Compositional minimization of finite state systems. In *Lecture Notes in Computer Science*, 186–196. Springer-Verlag.

Henzinger, T., Jhala, R., Majumdar, R., and Sanvido, M. (2003). Extreme model checking. In *Lecture Notes in Computer Science*, 332–358. Springer Berlin Heidelberg.

Kleinberg, J. and Tardos, E. (2005). *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA.

Komenda, J., Masopust, T., and van Schuppen, J. (2012). Supervisory control synthesis of discrete-event systems using a coordination scheme. *Automatica*, 48(2), 247–254.

Komenda, J., Masopust, T., and van Schuppen, J. (2016). Control of an engineering-structured multilevel discrete-event system. In *Workshop on Discrete Event Systems*, 103–108.

Leduc, R., Brandin, B., Lawford, M., and Wonham, W. (2005). Hierarchical interface-based supervisory control-part I: serial case. *IEEE Transactions on Automatic Control*, 50(9), 1322–1335.

Lehman, M. (1996). Laws of software evolution revisited. In *Software Process Technology*, 108–124.

Mohajerani, S., Malik, R., and Fabian, M. (2015). A framework for compositional nonblocking verification of extended finite-state machines. *Journal of Discrete Event Dynamic Systems*, 26(1), 33–84.

Ramadge, P. and Wonham, W. (1987). Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1), 206–230.

Ramadge, P. and Wonham, W. (1989). The control of discrete event systems. *Proceedings of the IEEE*, 77(1), 81–98.

Rudie, K. and Wonham, W. (1992). Think globally, act locally: decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11), 1692–1708.

Su, R., van Schuppen, J., and Rooda, J. (2010). Aggregative synthesis of distributed supervisors based on automaton abstraction. *IEEE Transactions on Automatic Control*, 55(7), 1627–1640.

Thuijsman, S. and Reniers, M. (2020). Transformational supervisor synthesis for evolving systems. In *Workshop on Discrete Event Systems*, 309–316.

Thuijsman, S. and Reniers, M. (2022). Transformational supervisor synthesis for evolving systems. *Discrete Event Dynamic Systems*, 32(2), 317–358.

van der Sanden, B., Reniers, M., Geilen, M., Basten, T., Jacobs, J., Voeten, J., and Schiffelers, R. (2015). Modular model-based supervisory controller design for wafer logistics in lithography machines. In *ACM/IEEE Conference on Model Driven Engineering Languages and Systems*, 416–425.

van der Schriek, Y. (2018). *Evaluation of supervisory control theory based on requirement evolution of LOPW*. Master's thesis, Eindhoven University of Technology, Deptartment of Mechanical Engineering.

Wonham, W. and Cai, K. (2019). *Supervisory Control of Discrete-Event Systems*. Springer, Cham.

Yang, G., Dwyer, M., and Rothermel, G. (2009). Regression model checking. In *IEEE Conference on Software Maintenance*, 115–124.

Zhong, H. and Wonham, W. (1990). On the consistency of hierarchical supervision in discrete-event systems. *IEEE Transactions on Automatic Control*, 35(10), 1125–1134.