Efficiently enforcing mutual state exclusion requirements in symbolic supervisor synthesis

Efficiently enforcing mutual state exclusion requirements in symbolic supervisor synthesis

Sander Thuijsman, Michel Reniers, and Dennis Hendriks

***Abstract*— Given a model of an uncontrolled system and a requirement specification, a supervisory controller can be synthesized so that the system under control adheres to the requirements. There are several ways in which informal behavioral safety requirements can be formalized, one of which is using mutual state exclusion requirements. In current implementations of the supervisor synthesis algorithm, synthesis may be inefficient when mutual state exclusion requirements are used. We propose a method to efficiently enforce these requirements in supervisor synthesis. We consider symbolic supervisor synthesis, where Binary Decision Diagrams are used to represent the system. The efficiency of the proposed method is evaluated by means of an industrial and academic case study.**

I. Introduction

A challenge in control software development is satisfying the ever-increasing demand for quality, performance, and safety. As a result, the control software becomes progressively complex to design. Traditionally, the requirements for the software are specified informally, and the engineers try to manually design a controller that satisfies the requirements. This is a time-consuming and error-prone task. To tackle this issue, supervisory control theory [1] can be applied, which provides a framework to control discrete-event systems. In this framework a plant model is used to define all possible system behavior. Additionally, a formal requirement specification is constructed to define what plant behavior is allowed or not. With these inputs, a *supervisory controller* can be computed algorithmically (*synthesized*) so that it restricts the plant's behavior in accordance with the specification.

Some examples where supervisory control theory is applied to controller design are; Theme park vehicles in [2], a patient support table of a magnetic resonance imaging scanner in [3], and a waterway lock and movable bridge combination in [4]. Despite the advantages of applying this technique, and the examples thereof shown in case studies, industrial acceptance is scarce. [5] points to *state space explosion* as one of the barriers to industrial acceptance. When the size of the system grows, the time- and space (memory) required for synthesis grows exponentially. This is often mitigated by splitting the problem in smaller subproblems, e.g., in modular synthesis [6]. Another state of the art approach to mitigate state space explosion is symbolically representing the system using Binary Decision Diagrams (BDDs) [7]–[12]. This paper investigates a method to improve the efficiency of this symbolic approach.

We use CIF [13] as part of the Eclipse Supervisory Control Engineering Toolkit (Eclipse ESCET™) to model plants and requirements and symbolically synthesize a supervisor[1]. In CIF, requirements can be specified in three ways:

- *requirement automata*, prescribing allowed behavior as sequences of events.
- *mutual state exclusion*, forbidding a particular combination of states to be reached.
- *state-transition exclusion*, requiring an event to be disabled for a given combination of states.

The latter two requirement types are discussed in [14]. These are also the requirement types that we consider in this paper.

One can imagine that a particular informal requirement, can be modeled in multiple ways using the different options described above. We provide an example.

Example: We consider two traffic lights, each is regulating the traffic for their road at a two-way intersection. The plant behavior can be modeled by two automata, given in Fig. 1.

The informal requirement is that the traffic lights should not be green at the same time, as this may result in a collision. This requirement can be formalized by a requirement automaton, given in Fig. 2.

Alternatively, the requirement specification can be given by a mutual state exclusion requirement:

`not(LightA.Green and LightB.Green)`

As another option, the modeler may give two state-transition exclusion requirements, specifying that one light can only be turned green if the other light is red:

$green\_A$ `needs LightB.Red`
$green\_B$ `needs LightA.Red`

Through general usage of CIF, it has been noticed empirically that the manner in which the requirements are modeled can impact the efficiency of performing supervisor synthesis, even if they represent the same informal requirement specification and the same controlled behavior is achieved. Notably the usage of mutual state exclusion requirements would lead to computations that required a lot of time and memory. Consequently, this type of requirement specification was sometimes avoided when modeling larger systems. This can be observed in, e.g., the model in [4].

For the purpose of modeling ease and model clarity, in a number of cases it might be useful to use mutual state

Research leading to these results has received funding from the EU ECSEL Joint Undertaking under grant agreement nº 826452 (project Arrowhead Tools) and from the partners national programs/funding authorities.

Sander Thuijsman (s.b.thuijsman@tue.nl) and Michel Reniers are with the Department of Mechanical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands.

Dennis Hendriks is with ESI (TNO), Eindhoven, The Netherlands.

[1]The ESCET toolset and documentation is open source and freely available at `https://www.eclipse.org/escet/`. 'Eclipse', 'Eclipse ESCET' and 'ESCET' are trademarks of Eclipse Foundation, Inc.

(a) LightA (b) LightB

Fig. 1. Traffic light plant automata

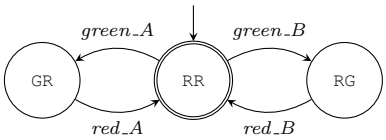

Fig. 2. Traffic light requirement automaton

exclusion requirements. For the traffic light example above, the mutual state exclusion requirement is arguably the most straightforward formalization of the informal requirement specification. Ideally, the usage of mutual state exclusion requirements would not be penalized by a higher computational effort of synthesis. This introduces the problem statement discussed in this paper: How can mutual state exclusion requirements be enforced (more) efficiently in symbolic supervisor synthesis? Our answer to this question is structured as follows: In Section II the preliminaries are provided on Extended Finite Automata (EFA), requirement specification, and symbolic supervisor synthesis. We introduce a method to convert mutual state exclusion requirements to state-transition exclusion requirements in Section III. This method is evaluated for an industrial and an academic case study in Section IV. Concluding remarks are given in Section V.

A. Related work

[15] and [16] investigate finding a partitioning of BDDs to represent transition relation systems to reduce time effort of reachable state computation. [17] extracts verification constraints from circuits and properties and exploits them to efficiently perform symbolic model checking algorithms. In the above works some of the principles about transition systems and BDDs are common with what we discuss here. However, they do not consider the enforcement of requirements through supervisor synthesis. Also, we already mentioned some works on symbolic supervisor synthesis, [9]–[12]. However, none of these works specifically investigate efficient enforcement of mutual state exclusion requirements.

II. Preliminaries

In this section we first introduce EFA and their symbolic version. Next, we formalize the specification of requirements. Finally, we discuss symbolic supervisor synthesis.

A. Automata

We consider an EFA *A* defined as *8-tuple*

$$A = (L, D, \Sigma, T, L_0, D_0, L_m, D_m)$$

where L is the domain of locations, $D{=}D^1{\times}...{\times}D^p$ is the domain of variables and Σ is the set of events, usually called the alphabet. The alphabet is split into two disjoint subsets; Σ_c and Σ_u representing controllable and uncontrollable events respectively. L_0 is the set of possible initial locations, $D_0{=}D_0^1{\times}...{\times}D_0^p$ is the set of possible initial variable values, L_m is the set of marked locations, and D_m is the set of marked variable values. T is a set of transitions where a transition t is defined as *5-tuple*

$$t = (l_o, l_t, \sigma, \gamma, \upsilon)$$

where l_o and l_t are the origin and target location in L, σ is an event in Σ, $\gamma : D \rightarrow \{false, true\}$ is the guard evaluation function and $\upsilon : D \rightarrow D$ is the update function that assigns new values to the variables.

$L(A)$ denotes the language/behavior of automaton A. $A_1||A_2$ denotes the synchronous composition of automata A_1 and A_2 [18]. The *state* of the EFA consists of the current automaton location and the value of each of the variables as a pair $(l, d) \in L{\times}D$. Consequently the initial states are pairs from $L_0{\times}D_0$ and the marked states are pairs from $L_m{\times}D_m$.

As noted in the introduction, we perform *symbolic* supervisor synthesis, i.e., a symbolic representation of the EFA is constructed to perform synthesis on. The symbolic representation makes use of predicates that can be represented by BDDs. Therefore, we will consider performing synthesis on a *Symbolic EFA (SEFA)* defined as a *5-tuple*

$$A_S = (X, \Sigma, E, X_0(X), X_m(X))$$

in which X is the set of symbols representing the automaton locations and the variables declared in that automaton. The state is defined by a valuation of these symbols. Σ is the alphabet. X_0 and X_m are predicates over the symbols that respectively represent the initial and marked states. Note that for a predicate $P(X)$ we may simply write P when the used set of symbols is clear from the context. E is the set of *edges*, with edge e defined as *triple*: $e = (\sigma, g(X), u(X, X^+))$, where σ is an event, g is a guard predicate, expressing from what states the event may occur, and u is an update predicate over current state symbols and *new state symbols* $X^+ = \{x^+ | x \in X\}$, representing what state will be reached when the edge occurs from a particular current state. In this way, an edge e may correlate to multiple transitions in T. [19] discusses construction of an SEFA from a (set of) EFA.

Example: For the traffic light plant model given in Section I, the set of symbols would be {`LightA`, `LightB`}. The initial state predicate would be `LightA.Red` $\wedge$ `LightB.Red`, where `LightA.Red` expresses that symbol `LightA` is valued `Red`. The SEFA edge for `LightA` turning green is given by ($green\_A$, `LightA.Red`, `LightA`$^+$`.Green` $\wedge$ `LightB`$^+$ =`LightB`).

B. Requirements

A mutual state exclusion requirement I is an expression defining a predicate over states in X that always needs to hold in the controlled system. A state-transition exclusion requirement is defined by $J \Rightarrow t$, where J is an expression defining a predicate over states in X that needs to hold in the controlled system for transition t to occur. Examples

for these requirement types are provided in the introduction. The modeler may define multiple mutual state- and state-transition exclusion requirements. Even though we define the safe states or states from which the transition is allowed to occur, we call it an 'exclusion' requirement because it is a restriction on the plant behavior.

In this paper we will regard *state-edge exclusion requirements*, essentially a set of state-transition exclusion requirements for all transitions t that pertain to an edge e. In this case we will write $J \Rightarrow e$ directly. We will refer to the set of all mutual state exclusion requirements as *MS*, and to the set of all state-edge exclusion requirements as *SE*. For simplicity we will consider specifications that do not contain any requirement automata. Enforcing the requirements expressed by automata in supervisor synthesis is well known [1], [18], [20].

C. Symbolic Supervisor Synthesis

The purpose of applying supervisor synthesis is to generate a supervisor automaton such that the synchronous product between the plant automaton and supervisor is *safe*, *nonblocking*, *controllable*, and *maximally permissive*. Safe means that the requirements always are adhered to. Nonblocking indicates that from every reachable state, a marked state can be reached. Controllable means that when the plant can execute an uncontrollable transition, this transition can also be executed in the synchronous product between supervisor and plant. In other words, the supervisor does not stop any uncontrollable events from occurring. Maximally permissive says that these properties are ensured without disabling any events that don't strictly need to be disallowed.

In Algorithm 1 a supervisor synthesis algorithm is presented. This synthesis algorithm is based on the algorithm introduced in [20]. The application of BDDs in this algorithm is studied in [21], [22]. We have split up the algorithm over multiple algorithms to enable reuse of parts later.

In the synthesis algorithm, first requirements are applied by Algorithm 2. In this algorithm, a safe state predicate P is computed by first taking the conjunction of all mutual state exclusion requirements. This predicate returns true only for states for which all mutual state exclusion requirements hold. As a convention we will use that the empty-conjunction is true. Next, the state-edge exclusion requirements are enforced by computing a safe state predicate and safe edges in Algorithm 3. When these requirements consider edges with controllable events, the guard can simply be strengthened by taking the conjunction with the predicate, so that the edge only occurs when the requirement holds. This is not possible when the edge is labeled by an uncontrollable event. In that case, the safe state predicate is modified to exclude states from which the edge can take place, but the state-edge exclusion requirement does not hold. The predicate $g \rightarrow J$ specifies the states where the state-exclusion requirement is adhered to, where '$\rightarrow$' denotes logical implication.

Algorithm 1 iteratively calculates a set of nonblocking states N, followed by a set of bad states B. Only the safe edges with strengthened guards computed in Algorithm 3 will be considered in the supervisor synthesis. N is initiated with the safe states found by Algorithm 2. The calculation to obtain N and B is done by the means of a backward reachability search, given in Algorithm 4. This search is performed on the predicates by using the existential quantification operator [22], [23]. The bad states are removed from N. The removal of these states can induce other states to become blocking. Therefore, the algorithm repeats these steps until the fixpoint is reached, i.e., no further bad states get removed. Next, the guards of the edges labeled by controllable events are strengthened such that the guard can only be true when the nonblocking predicate is true for the state that is reached after taking the edge. Finally, the supervisor SEFA is constructed. The conjunction is taken between the initial state predicate and the nonblocking predicate, so that the supervised system is only initialized in nonblocking states. The supervised system will remain in nonblocking states, as the strengthened guards prevent transitions from nonblocking to bad states.

Algorithm 1 `SS` (Supervisor Synthesis)

Input: Plant SEFA $A_S = (X, \Sigma, E, X_0, X_m)$, mutual state exclusion requirements *MS*, state-edge exclusion requirements *SE*
Output: Supervisor SEFA S

1: $(N, E_S) = \texttt{applyRequirements}(MS, SE, E)$
2: **repeat**
3: $N' = N$
4: $N = \texttt{BRS}(N, E_S, X_m)$
5: $B = \texttt{BRS}(\texttt{true}, \{(\sigma, g, u) \in E | \sigma \in \Sigma_u\}, \neg N)$
6: $N = N \wedge \neg B$
7: **until** $N = N'$
8: **for all** $(\sigma, g, u) \in E_S$ with $\sigma \in \Sigma_c$
9: $g(X) = g(X) \wedge \exists_{X^+}[N(X^+) \wedge u(X, X^+)]$
10: **end**
11: $S = (X, \Sigma, E_S, X_0 \wedge N, X_m \wedge N)$

Algorithm 2 `applyRequirements`

Input: Mutual state exclusion requirements *MS*, state-edge exclusion requirements *SE*, edges E
Output: Safe state predicate P, safe edges E

1: $P = \bigwedge_{I \in MS} I$
2: $(P, E) = \texttt{applyEdgeRequirements}(P, E, SE)$

III. Conversion of mutual state exclusion requirements to state-edge exclusion requirements

As stated in the introduction, it has been found empirically that synthesis on models containing state exclusion requirements was inefficient, and therefore they are sometimes manually converted to state-edge exclusion requirements. From practice it has been found that this can solve the inefficiency problem. Therefore this is also the direction in which we seek our solution; we convert the state exclusion requirements to state-edge exclusion requirements. For practical reasons we

Algorithm 3 `applyEdgeRequirements`

Input: State predicate P, edges E, state-edge exclusion requirements *SE*
Output: Safe state predicate P, safe edges E

1: **for all** $(\sigma, g, u) \in E,\ (J \Rightarrow (\sigma, g, u)) \in SE$
2: **if** $\sigma \in \Sigma_c$
3: $g = g \wedge J$
4: **else**
5: $P = P \wedge (g \rightarrow J)$
6: **end if**
7: **end for**

Algorithm 4 `BRS` (Backward Reachability Search)

Input: Restriction predicate P_r, edges E, start predicate P
Output: Coreachable predicate P'

1: **repeat**
2: $P' = P$
3: $P'(X) = P_r(X) \wedge (P(X) \vee \bigvee_{(\sigma,g,u)\in E} \exists_{X^+}[P(X^+) \wedge g(X) \wedge u(X, X^+)])$
4: **until** $P' = P$

do so automatically rather than manually.

In Algorithm 5 the mutual state exclusion requirements are first converted into state-edge exclusion requirements. This is done by iterating over all the mutual state exclusion requirements and edges. For each edge and mutual state exclusion requirement a predicate R is constructed that expresses the states from which the edge can be performed and the requirement holds after executing the edge. In the controlled behavior, the edge can only be performed from the states indicated by $g \wedge I$, because states where I does not hold are not reached by a safe supervisor. In all cases that the edge can be performed, R must hold so that a safe state is reached. In line 4 it is checked whether there are any states for which this does not hold. If that is the case, a state-edge exclusion requirement is added that requires R to hold for the edge to occur. Next, the initial state predicate is modified so that all mutual state exclusion requirements hold in the initial state. Thus, the system starts in a safe state, and does not leave the safe states as a result of the generated state-edge exclusion requirements. The generated and existing state exclusion requirements are applied in line 10, in the same manner as Algorithm 2.

In Algorithm 1, we can substitute line 1 with the following line, to apply the introduced efficient enforcement of the requirements:
1: (N, E_S, X_0)=`applyRequirementsEfficient`(MS, SE, E, X_0)

After this substitution, the behavior of the controlled system, i.e. synchronous composition between plant and synthesized supervisor, remains the same:

Theorem 1: $L(\texttt{SS}'(A_S, \mathit{MS}, \mathit{SE})||A_S) = L(\texttt{SS}(A_S, \mathit{MS}, \mathit{SE})||A_S)$, where $\texttt{SS}'$ is Algorithm 1 with line 1 substituted as indicated above.

Algorithm 5 `applyRequirementsEfficient`

Input: Mutual state exclusion requirements *MS*, state-edge exclusion requirements *SE*, edges E, initial states X_0
Output: Safe state predicate P, safe edges E, safe initial states X_0

1: **for all** $I \in MS$
2: **for all** $(\sigma, g, u) \in E$
3: $R(X) = \exists_{X^+}[I(X^+) \wedge g(X) \wedge u(X, X^+)]$
4: **if** $(g \wedge I \rightarrow R) \neq \texttt{true}$
5: $SE = SE \cup \{R \Rightarrow (\sigma, g, u)\}$
6: **end if**
7: **end for**
8: **end for**
9: $X_0 = X_0 \wedge \bigwedge_{I\in MS} I$
10: $(P, E) =$ `applyEdgeRequirements`$(\texttt{true}, E, SE)$

IV. Experiments

We perform some experiments to evaluate the new method. In Section IV-A we discuss how the computational effort is measured, and some relevant settings for the synthesis algorithm that were used for the experiments. Then we introduce an industrial and academic case study in Sections IV-B and IV-C respectively[2]. For both case studies we perform measurements to express the computational effort for the current *baseline* method (`SS` using Algorithm 2) and the introduced *converted* method ($\texttt{SS}'$ using Algorithm 5).

A. Measuring computational effort

For symbolic supervisor synthesis, the computational effort can be expressed by *peak used BDD nodes* and *BDD operation count* [21]. The first metric is the maximal combined size of all BDDs during synthesis. Computer memory is always finite, so this is the main limiting factor for successful synthesis. The second metric is the number of times a recursive call is made to any BDD operation, indicating the time effort. These metrics allow measuring computational effort in a deterministic, platform-independent way and include no overhead in their measurements, opposed to more traditional metrics such as computer memory usage and wall-clock time. Performing synthesis with constant parameter settings results in the same result each time for the BDD-based metrics.

The BDD variable order and the order in which the edges are iterated over in the for loops have a large impact on the computational effort of a particular synthesis [21], [22]. The variable and edge order influence the synthesis efficiency that is found when comparing the two methods. I.e., if for both methods the same variable and edge order are used, a relative difference in effort can be computed. This relative difference can be different for different variable and edge orders. Note that the variable- and edge order only influence the computational effort of a synthesis, the resulting supervisor is not affected. Therefore, the experiments are executed as follows. 10 random variable orders and 10

[2]The used CIF models are available here: `https://github.com/sbthuijsman/CASE21_enforce_requirements`

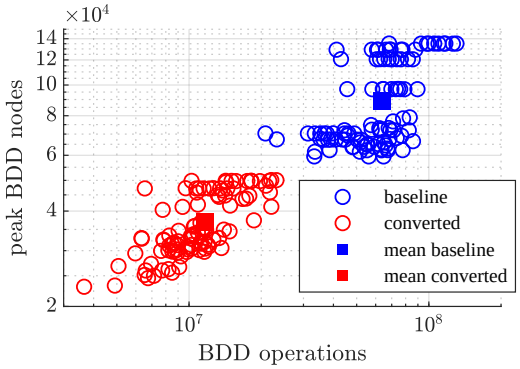

Fig. 3. Computational effort results lithography machine initialization

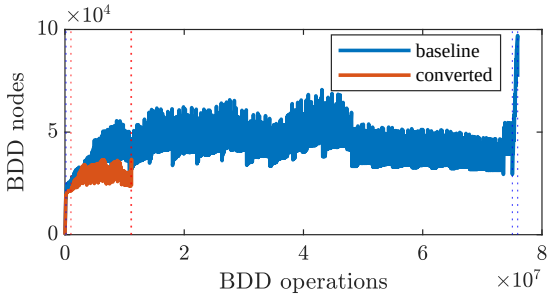

Fig. 4. BDD evolution lithography machine initialization

TABLE I
BDD OPERATIONS FOR SYNTHESIS PROGRESSION

Progression of synthesis		**BDD operations**
Finished applying requirements	baseline	171 448
	converted	1 001 676
Finished computing nonblocking predicate	baseline	75 040 338
	converted	11 058 198
Finished synthesis	baseline	75 920 257
	converted	11 156 718

random edge orders are computed. For each combination of these, synthesis is performed for both methods, so 200 experiments per model. The efficiency of the method is evaluated by comparing the average results. Variable ordering heuristics FORCE and sliding window [21] are applied to the randomly generated variable orders, as this is the most realistic usage circumstance.

The authors note that CIF has some functionality to simplify the guard expressions in the resulting supervisor. For example when the supervisor guard is exactly the same as the plant guard, the supervisor guard for this edge can be 'true' as this results in the same behavior in the controlled system. Such simplifications can make the supervisor guard restrictions more readable. They however do require computational effort and are performed in none of the experiments that we present to enable proper quantitative comparisons between the results. The supervisor guards are thus computed as presented in Algorithm 1.

B. Lithography machine initialization

[24] discusses an approach for initialization and termination of flexible manufacturing systems. In this approach a discrete-event model represents the system in a modular manner. Constraints can be enforced, and the existence of successful initialization and termination sequences under these constraints can be proven from any state. [24] includes an industrial use case on lithography machines. These machines are used in the semiconductor industry to manufacture integrated circuits. In [24], existing machine artifacts are used as the modular specification, which are converted into a network of automata in CIF.

We present the results of a case study on the industrial lithography initialization model. [24] also constructed a termination model, for which the results are practically the same. The complete model is much too large to perform (monolithic) synthesis. Therefore a partial model is constructed in [24]. This model is specified in CIF by 16 automata, the product of their number of states (worst-case state space size) is $1.85 \cdot 10^{16}$. There are 51 mutual state exclusion requirements, each requirement being an expression that refers to two automata. Applying Algorithm 5 results in generation of 74 state-edge exclusion requirements.

The experiments are performed as discussed in Section IV-A. The results are shown in Fig. 3. Of each synthesis the computational effort is shown using the BDD metrics on a logarithmic scale. Quite some variance can be observed as a result of the random variable- and edge orders. Nevertheless, it can be concluded from the results that for this model the introduced method, where the mutual state exclusion requirements are converted into state-edge exclusion requirements, has a beneficial influence on the computational effort. Each synthesis where Algorithm 5 was applied instead of Algorithm 2 required less memory, and for almost all syntheses less time as well. The mean computational effort of the syntheses is reduced by a factor 2.4 (from $8.9 \cdot 10^4$ to $3.7 \cdot 10^4$) for peak BDD nodes, and a factor 5.4 (from $6.3 \cdot 10^7$ to $1.2 \cdot 10^7$) for BDD operation count.

We study how this reduction in effort is achieved by the new method. For this we use Fig. 4. This image shows how the combined size of all BDDs evolves during synthesis. This is constructed from measurements on the lithography initialization model with the variable- and edge orders chosen such that the effort is close to the mean for both the baseline and converted method found previously. The metrics that we use, peak used BDD nodes and BDD operation count, are the maxima along both axes in this figure. Some vertical lines are drawn that indicate progression points in the synthesis algorithm. These are detailed in Table I.

As expected, the new method takes longer to apply the requirements as this includes performing the conversion too. However this effort is won back in the calculation of the nonblocking and guards predicates. Because of the conversion, not all the state exclusion requirements are in the nonblocking predicate, but rather in the guards of the edges. Even though the nonblocking predicate might express a larger set of states, the required space to express it (number of BDD nodes) is smaller. This is beneficial,

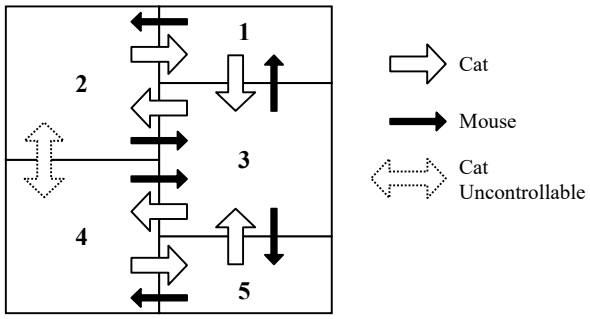

Fig. 5. CMT room layout of a level

because the nonblocking predicate is constantly used during the synthesis; these computations are now on a smaller BDD resulting in less computational effort. We also observe that in the baseline the memory usage spikes when the guards are computed (lines 8-10 of Algorithm 1). This spike does not occur for the new method, as the conditions where the requirements are not met were already largely included in the guards of the edges during the conversion.

C. Cat and Mouse Tower

As academic use case we take the Cat and Mouse Tower (CMT) from [25]. This model has been selected as it includes mutual state exclusion requirements, and additionally can be scaled in size.

On each floor of the tower there are five rooms as shown in Fig. 5. Cats and mice can move between the rooms as indicated by the arrows. All doors can be controlled, except for the bidirectional cat door between rooms 2 and 4. There are n levels. Between each level there is a controllable connection that both cats and mice can use. This connection is between room j of level $5{\cdot}i{+}j$ to room j of level $5{\cdot}i{+}j{+}1$, for $i \in \mathbb{N}_0$, $j \in \{1,2,3,4,5\}$, and $5 \cdot i + j < n$. There are k cats and k mice, and consequently each room can also hold between 0 and k cats and/or mice. The cats start in room 1 of level 1, and the mice start in room 5 of level n. The informal requirement of this system is that there can never be a cat and a mouse in the same room at the same time.

Same as [25], the model is constructed as a network of finite state automata. For each room two automata with $k{+}1$ states each are used that represent the amount of cats and mice present. The state of the automata is updated accordingly when the event occurs that a cat or a mouse moves between rooms. For each room a mutual state exclusion requirement is specified, expressing that there is either no cat or no mouse.

The same experiments as in Section IV-B have been repeated for this model for some combinations of n and k. The results are summarized in Table II. Table II additionally shows how many state-edge exclusion requirements are generated by Algorithm 5 during each synthesis.

For this model, the computational effort is lower when using the baseline method, rather than the proposed conversion method. We study the evolution of the BDD size during the synthesis for why this might be the case. Fig. 6 shows the

TABLE II
CMT EXPERIMENTAL RESULTS

n	*k*	method	mean peak BDD nodes	mean BDD operations	generated s-e reqs.
5	*1*	baseline	$8.8 \cdot 10^3$	$5.3 \cdot 10^5$	n.a.
		converted	$9.0 \cdot 10^3$	$6.6 \cdot 10^5$	86
5	*3*	baseline	$3.2 \cdot 10^5$	$3.5 \cdot 10^8$	n.a.
		converted	$3.0 \cdot 10^5$	$4.5 \cdot 10^8$	258
10	*1*	baseline	$3.2 \cdot 10^4$	$5.4 \cdot 10^6$	n.a.
		converted	$3.2 \cdot 10^4$	$6.6 \cdot 10^6$	176
10	*2*	baseline	$2.9 \cdot 10^5$	$2.3 \cdot 10^8$	n.a.
		converted	$2.6 \cdot 10^5$	$2.7 \cdot 10^8$	352

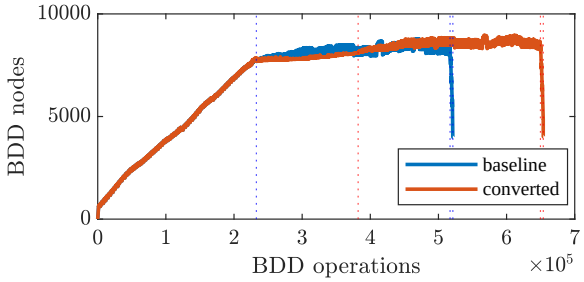

Fig. 6. BDD evolution CMT

evolution of the BDD during synthesis of a CMT model with $n = 5$ and $k = 1$. This is constructed from measurements with a variable- and edge order chosen such that the effort is close to the mean for both the baseline and converted method. Again we observe that for the new method it takes longer to apply the requirements; it requires $1.5 \cdot 10^5$ more BDD operations. However, it can be seen that this invested effort is not won back later during synthesis; the total synthesis requires $1.3{\cdot}10^5$ more BDD operations. Essentially, not many additional nodes are required after applying the requirements to compute the nonblocking predicate. I.e., for the CMT model the nonblocking predicate is relatively simple to express when compared to the other predicates representing the system. The same was observed for the CMT model for larger values for n and k.

On the contrary, for the lithography machine initialization model we saw the expression of the nonblocking predicate was relatively complex. The new method focuses on reducing the BDD size of the nonblocking predicate. As the nonblocking predicate has a larger impact on the computational effort of the lithography machine model compared to the CMT model, there is less efficiency to be gained for the CMT model by applying the conversion method. As it turns out, for the CMT model the efficiency gain was too little to win back the invested effort for applying the conversion.

To further support the statement above, why the method might work for one model and not the other, we create a modified CMT model so that the construction of the nonblocking predicate becomes relatively complex. The modification is as follows: Initially there are no cats or mice in the tower; cats can enter room 1 level 1, and mice can enter room 5 level n uncontrollably whenever there is space; each room can hold

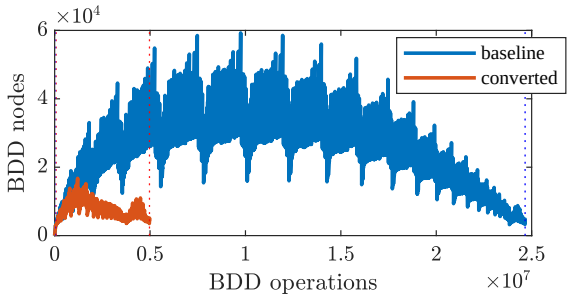

Fig. 7. BDD evolution modified CMT

k cats or mice; the total number of cats and mice is now only limited to the combined capacity of all the rooms. The same experiment as before is performed for this model for $n = 3$, $k = 1$. In each synthesis, 52 state-edge exclusion requirements are generated from the mutual state exclusion requirements. For this model, using the proposed method reduces the mean peak BDD nodes by a factor 3.7 (from $5.9 \cdot 10^4$ to $1.6 \cdot 10^4$) and the BDD operation count by 4.7 (from $2.6 \cdot 10^7$ to $5.6 \cdot 10^6$). So for this model the conversion from state exclusion requirements to state-edge exclusion requirements is beneficial. In Fig. 7 we have once more plotted the evolution of the BDD sizes during two average sample syntheses of this model. Once more, converting the requirements requires more computational effort for the new method, but this is won back when computing the nonblocking predicate. Unfortunately, at the moment we have no way to predict which method will be more efficient.

V. Conclusions

Mutual state exclusion requirements and state-edge exclusion requirements can be used to formalize a requirement specification. The use of mutual state exclusion requirements was found to have a negative impact on the computational effort to apply synthesis, and were therefore omitted in some cases. We present a method to enforce mutual state exclusion requirements by converting them to state-edge exclusion requirements first. The gain in computational efficiency of synthesis is evaluated for the industrial lithography machine initialization model, and the academic CMT model. For the lithography machine model, the method is shown to be beneficial. However, for the CMT model it is not. The proportion of computational effort required to compute the nonblocking predicate after applying the requirements is shown to influence the efficiency of the method. It is shown how the method can be efficient for a modified version of the CMT model.

Acknowledgment

The authors thank Bart Wetzels for his initial efforts on the topic.

References

[1] P. Ramadge and W. Wonham, "Supervisory control of a class of discrete event processes," *SIAM J. Control*, vol. 25, no. 1, pp. 206–230, 1987.

[2] S. Forschelen, J. van de Mortel-Fronczak, R. Su, and J. Rooda, "Application of supervisory control theory to theme park vehicles," *Discrete Event Dynamic Syst.*, vol. 22, no. 4, pp. 511–540, 2012.

[3] R. Theunissen, M. Petreczky, R. Schiffelers, D. van Beek, and J. Rooda, "Application of supervisory control synthesis to a patient support table of a magnetic resonance imaging scanner," *IEEE Trans. Autom. Sci. Eng.*, vol. 11, no. 1, pp. 20–32, 2014.

[4] F. Reijnen, M. Goorden, J. van de Mortel-Fronczak, and J. Rooda, "Modeling for supervisor synthesis – a lock-bridge combination case study," *Discrete Event Dynamic Syst.*, vol. 30, no. 3, pp. 499–532, 2020.

[5] W. Wonham, K. Cai, and K. Rudie, "Supervisory control of discrete-event systems: A brief history," *Annu. Rev. Control*, vol. 45, pp. 250 – 256, 2018.

[6] W. Wonham and P. Ramadge, "Modular supervisory control of discrete-event systems," *Math. Control Signal Systems*, vol. 1, pp. 13–30, 1988.

[7] S. Akers, "Binary decision diagrams," *IEEE Trans. Comput.*, vol. 27, no. 6, pp. 509–516, 1978.

[8] C. Lee, "Representation of switching circuits by binary-decision programs," *The Bell System Tech. J.*, vol. 38, no. 4, pp. 985–999, 1959.

[9] A. Vahidi, M. Fabian, and B. Lennartson, "Efficient supervisory synthesis of large systems," *Control Eng. Practice*, vol. 14, no. 10, pp. 1157–1167, 2006.

[10] R. Leduc, P. Dai, and R. Song, "Synthesis method for hierarchical interface-based supervisory control," *IEEE Trans. Autom. Control*, vol. 54, no. 7, pp. 1548–1560, 2009.

[11] S. Miremadi, B. Lennartson, and K. Åkesson, "A BDD-based approach for modeling plant and supervisor by extended finite automata," *IEEE Trans. Control Syst. Technol.*, vol. 20, no. 6, pp. 1421–1435, 2012.

[12] R. Malik, K. Åkesson, H. Flordal, and M. Fabian, "Supremica-an efficient tool for large-scale discrete event systems," *Proc. IFAC World Congr.*, vol. 50, no. 1, pp. 5794 – 5799, 2017.

[13] D. van Beek, W. Fokkink, D. Hendriks, A. Hofkamp, J. Markovski, J. van de Mortel-Fronczak, and M. Reniers, "CIF 3: Model-based engineering of supervisory controllers," *Tools Algorithms Construction Anal. Syst.*, vol. 8413, pp. 575–580, 2014.

[14] J. Markovski, D. van Beek, R. Theunissen, K. Jacobs, and J. Rooda, "A state-based framework for supervisory control synthesis and verification," in *IEEE Conf. Decision Control*, 2010, pp. 3481–3486.

[15] R. Hojati, S. Krishnan, and R. Brayton, "Early quantification and partitioned transition relations," in *Proc. Conf. Comput. Des.*, 1996, pp. 12–19.

[16] G. Cabodi, P. Camurati, and S. Quer, "Improving the efficiency of BDD-based operators by means of partitioning," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 5, pp. 545–556, 1999.

[17] G. Cabodi, P. Camurati, L. Garcia, M. Murciano, S. Nocco, and S. Quer, "Speeding up model checking by exploiting explicit and hidden verification constraints," in *Des. Automat. Test Eur. Conf.*, 2009, pp. 1686–1691.

[18] C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Springer, 2008.

[19] D. Nadales Agut, D. van Beek, and J. Rooda, "Syntax and semantics of the compositional interchange format for hybrid systems," *J. Logic Algebr. Progr.*, vol. 82, no. 1, pp. 1–52, 2013.

[20] L. Ouedraogo, R. Kumar, R. Malik, and K. Åkesson, "Nonblocking and safe control of discrete-event systems modeled as extended finite automata," *IEEE Trans. Autom. Sci. Eng.*, vol. 8, no. 3, pp. 560–569, 2011.

[21] S. Thuijsman, D. Hendriks, R. Theunissen, M. Reniers, and R. Schiffelers, "Computational effort of BDD-based supervisor synthesis of extended finite automata," in *IEEE Conf. Autom. Sci. Eng.*, 2019, pp. 486–493.

[22] S. Lousberg, S. Thuijsman, and M. Reniers, "DSM-based variable ordering heuristic for reduced computational effort of symbolic supervisor synthesis," in *IFAC Workshop Discrete Event Syst.*, 2020.

[23] R. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Comput. Surv.*, vol. 24, no. 3, pp. 293–318, 1992.

[24] Z. Vos, "Initialization and termination of flexible manufacturing systems: a formal approach," Master's thesis, Eindhoven Univ. Techn., Dept. Elect. Eng., 2019.

[25] C. Ma and W. Wonham, "STSLib and its application to two benchmarks," in *IEEE Workshop Discrete Event Syst.*, 2008, pp. 119–124.