Discrete Event Dynamic Systems (2024) 34:689–732
https://doi.org/10.1007/s10626-024-00403-4

Reducing the computational effort of symbolic supervisor synthesis

Sander Thuijsman[1] · **Dennis Hendriks**[2,3] · **Michel Reniers**[1]

Received: 16 May 2023 / Accepted: 29 August 2024 / Published online: 12 September 2024

Abstract
Supervisor synthesis is a means to algorithmically derive a supervisory controller from a discrete-event model of a system and a requirements specification. For large systems, synthesis suffers from state space explosion. To mitigate this, synthesis can be applied to a symbolic representation of the models by using Binary Decision Diagrams (BDDs). Peak used BDD nodes and BDD operation count are introduced as deterministic and platform independent metrics to express the computational effort of a symbolic synthesis. These BDD-based metrics are useful to analyze the efficiency of the synthesis algorithm. From this analysis, modifications can be made to how BDDs are handled during synthesis, improving synthesis efficiency. We demonstrate this approach by introducing and analyzing: DCSH, a variable ordering heuristic; several edge ordering heuristics; and an approach to efficiently enforce state exclusion requirements in synthesis. These methods were recently implemented in our open source supervisory control tool: Eclipse ESCET. The analysis is based on large scale experiments of performing synthesis on a variety of models from literature. We show that: (1) by using DCSH, synthesis with high computational effort can be avoided, and generally low computational effort is required, relative to the variable ordering heuristics that were used prior to this work; (2) applying reverse-model edge order realizes relatively low synthesis effort; and (3) state exclusion requirements can efficiently be enforced by restricting edge guards prior to synthesis. While these methods reduce computational effort in practice, it should be noted that they do not affect the theoretical (worst-case) complexity of synthesis.

Keywords Discrete-event systems · Supervisory control · Supervisor synthesis · Binary decision diagrams · Computational efficiency

✉ Michel Reniers
m.a.reniers@tue.nl

Sander Thuijsman
sbthuijsman@gmail.com

Dennis Hendriks
dennis.hendriks@tno.nl

[1] Eindhoven University of Technology, Eindhoven, Netherlands

[2] TNO-ESI, Eindhoven, Netherlands

[3] Radboud University, Nijmegen, Netherlands

1 Introduction

Supervisory control theory (Ramadge and Wonham 1987, 1989) is a model-based approach to control cyber-physical systems. Given a plant (a model that defines all possible system behavior) and a requirements specification (a model that defines what behavior is allowed), a supervisor can be computed algorithmically (synthesized) that restricts the plant's behavior so that it is in accordance with the requirements specification. Depending on the synthesis algorithm, the supervised system has some useful properties by construction, such as safety, nonblockingness, controllability, and maximal permissiveness. There are a number of formal modeling frameworks to which supervisory control theory can be applied. The framework of *extended finite automata* (EFAs) (Sköldstam et al. 2007) is an extension to *finite state automata* that augments them with variables, guard expressions and updates, which enables more convenient modeling of systems.

The power of supervisory control theory has been demonstrated in literature. There are many examples where it is applied to controller design. We refer to Table 1 further down this paper for a selection. Despite the advantages of supervisory control theory, and demonstration thereof in case studies, industrial acceptance is scarce. Wonham et al. (2018) point to the *state space explosion* as one of the barriers to industrial acceptance. 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 way to mitigate state space explosion, is by representing the system model using *binary decision diagrams* (BDDs) (Akers 1978; Lee 1959), and performing supervisor

Table 1 Benchmark models

Name		Worst case state space size
Robotic swarm aggregation	(Lopes et al. 2016)	$1.0 \cdot 10^{0}$
Robotic swarm clustering	(Lopes et al. 2016)	$1.0 \cdot 10^{0}$
Robotic swarm segregation	(Lopes et al. 2016)	$6.4 \cdot 10^{1}$
Robotic swarm formation	(Lopes et al. 2016)	$8.0 \cdot 10^{1}$
Multi agent formation	(Cai and Wonham 2014)	$1.0 \cdot 10^{3}$
Automatic guided vehicles	(Wonham and Cai 2019)	$3.1 \cdot 10^{3}$
Ball sorting system	(Čengić and Åkesson 2008)	$7.4 \cdot 10^{4}$
Theme park vehicles	(Forschelen et al. 2012)	$2.9 \cdot 10^{5}$
Cluster tool	(Su et al. 2010)	$2.7 \cdot 10^{8}$
Production cell	(Feng et al. 2008)	$7.5 \cdot 10^{8}$
Modified cat and mouse tower *(n=3, k=1)*	(Thuijsman et al. 2021)	$1.1 \cdot 10^{9}$
Advanced driver assistance system	(Korssen et al. 2018)	$3.4 \cdot 10^{9}$
Cat and mouse tower *(n=3, k=2)*	(Ma and Wonham 2008)	$2.1 \cdot 10^{14}$
Lithography machine initialization	(Vos 2020)	$1.8 \cdot 10^{16}$
Bridge	(Reijnen et al. 2018b)	$2.8 \cdot 10^{27}$
FESTO production line	(Reijnen et al. 2018a)	$1.3 \cdot 10^{28}$
Waterway lock	(Reijnen et al. 2017)	$6.0 \cdot 10^{32}$

synthesis on this symbolic representation (Ma and Wonham 2006; Vahidi et al. 2006; Miremadi et al. 2012). This approach is considered state of the art to handle industrial-sized systems (Malik et al. 2017).

Symbolic supervisor synthesis has been shown to be able to deal with large-scale systems. For instance, monolithic synthesis was successfully performed for a system where the uncontrolled system and supervised system respectively had $2.3 \cdot 10^{57}$ and $4.5 \cdot 10^{34}$ states by Reijnen et al. (2020), which are much larger state spaces than non-symbolic monolithic synthesis could handle. However, as we will also show in this paper, the amount of time and memory required for symbolic synthesis is majorly impacted by the settings the algorithm is initiated with, and different ways the algorithm can be applied (Vahidi et al. 2006; Thuijsman et al. 2019). It is of practical benefit to optimize the application of the algorithm to minimize time and memory required to perform synthesis (of large-scale systems). Such optimization is difficult, as what techniques are beneficial is often case dependent, and even frequently counter-intuitive, since a BDD representing a small amount of states may require many more BDD nodes than a BDD representing a much larger amount of states (Ciardo and Siminiceanu 2002). Sufficient experimentation and validation is required to judge the efficiency of a method.

A tool that can be used to perform symbolic supervisor synthesis is *CIF* (van Beek et al. 2014). CIF is part of the *Eclipse Supervisory Control Engineering Toolkit* (Eclipse ESCET™)[1] since 2020 (Fokkink et al. 2023). As a result of this open source project, the intensity of the development of the CIF tool has recently greatly increased. Among the many developments that have been made, are implementations of recently proposed methods that aim to improve the computational efficiency of symbolic supervisor synthesis, such as the BDD variable ordering heuristic algorithm from Lousberg et al. (2020) and efficient enforcement of state exclusion requirements from Thuijsman et al. (2021). These methods were previously only available in local proof-of-concept implementations.

This paper is an extension to Thuijsman et al. (2019); Lousberg et al. (2020); and Thuijsman et al. (2021). Those papers contain many results of elaborate experiments. Because, as mentioned, many developments have taken place for the CIF tool, we re-perform and re-evaluate the results from Thuijsman et al. (2019); Lousberg et al. (2020); and Thuijsman et al. (2021). We use the BDD-based metrics of Thuijsman et al. (2019) to measure the computational effort of performing symbolic supervisor synthesis. We re-evaluate the impact of the variable order heuristic of Lousberg et al. (2020) and the requirement enforcement of Thuijsman et al. (2021) on the computational effort, now that they are implemented and available to all users. For further validation, the experiments are performed on a larger set of models. Additionally, we make all models publicly available, so that our experiments are repeatable[2]. In further extension to Thuijsman et al. (2019); Lousberg et al. (2020); and Thuijsman et al. (2021), we investigate various (simple) heuristics for edge ordering to improve synthesis efficiency. We also evaluate how these methods perform together, e.g., the edge ordering heuristics are evaluated using the variable ordering heuristic we introduce. Furthermore, this paper contains a proof of correctness of the efficient requirement enforcement method, that was not given in Thuijsman et al. (2021). Finally, in this paper the methods are presented in a more unified way.

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

[2] All experiments in this paper are performed using ESCET release v0.9, available here: https://eclipse.dev/escet/v0.9/. The models are available bundled in ESCET under "CIF Benchmarks", see https://eclipse.dev/escet/cif/examples.html. The files to run the same experiments as presented in this paper are available here: https://github.com/sbthuijsman/reduce_effort.

The authors note that there are many ways in which improvements can be made on computational efficiency of symbolic supervisor synthesis. Evidently, we restrict ourselves to a few options in this paper in order to keep this study contained. The methods that we evaluate have the following in common, they are:

- monolithic approaches: supervisor synthesis is not divided into multiple sub-problems;
- non-restrictive to the input model: the method can be used for synthesis of any plant and requirement specification that CIF supports synthesis of;
- "under the hood": the user does not have to supply additional parameters or modify their model;
- static: in the sense that optimization is performed only at a single stage, and not on-the-fly (dynamically) during/throughout synthesis.

All of the effort reducing methods we present in this paper are heuristic algorithms. For none of them, it can be proven or shown that definitely the computational effort will be reduced by applying them. Therefore, we present extensive experimental evaluations of these methods. The experiments are performed on a large scale: running all experiments sequentially would require several years.

In brief, this paper contains the following contributions:

- Introduction of BDD-based metrics to measure computational effort, and comparison of those to conventional metrics, such as time and memory usage or state space sizes, in Section 3.
- Introduction of the DCSH variable ordering heuristic, and comparison of its performance to order variable ordering heuristcs, in Section 4.
- Evaluation and comparison of various edge ordering heuristics to reduce computational effort, in Section 5.
- Introduction of a method to efficiently enforce requirements in synthesis, and comparison to the conventional approach, in Section 6.

Related work

The foundations of symbolic supervisor synthesis are discussed in Ma and Wonham (2006). Symbolic supervisor synthesis for (sets of) EFAs is discussed in Ouedraogo et al. (2011) and Fei et al. (2014). In Ziller and Schneider (2003) a supervisor synthesis algorithm is constructed that is based on μ-calculus, and a BDD-based implementation is made. In Vahidi et al. (2006) partitioning and ordering of the transition relation to efficiently perform BDD-based synthesis is investigated. This partitioning, as well as how supervisor guards can efficiently be generated for such a partitioning, is inspected in Fei et al. (2013). A symbolic synthesis approach using hierarchichal decomposition is presented in Song and Leduc (2006). In Miremadi and Lennartson (2016) an efficient synthesis algorithm is introduced that is based on forward reachability rather than backward reachability to avoid unnecessary exploration of states, of which also a BDD-based implementation is evaluated.

Efficient symbolic state space exploration is also a well-studied topic in the field of model checking. An overview of concepts and techniques for BDD-based model checking is provided in Chaki and Gurfinkel (2018). A BDD-based algorithm for computation tree logic model checking is introduced in Burch et al. (1994). Several variable ordering heuristics for state space exploration of interacting finite state machines are evaluated in Aziz et al. (1994). In Cabodi et al. (1999) the efficiency of BDD-based operators is improved by partitioning the BDDs. Zero-supressed BDDs are used in Minato (2001) to reduce computational effort of symbolic model checking in some applications.

The authors are not aware of existing works that study variable ordering heuristics specifically for supervisor synthesis, static edge ordering heuristics for supervisor synthesis (although dynamic edge ordering is investigated in Vahidi et al. (2006) and Fei et al. (2014)), or efficient enforcement of state-based requirements.

2 Symbolic supervisor synthesis

In this section we first introduce EFAs and their linearized version that can be symbolically encoded. Next, we discuss symbolic supervisor synthesis. Following, the encoding of a system in BDDs is considered. Finally, we introduce the relevant parts of the tool CIF, which we use for symbolic supervisor synthesis.

2.1 Automata

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

$$A = (L, V, \Sigma, T, L_0, V_0(V), L_m, V_m(V)),$$

where L is a finite set of locations, V is a finite set of discrete variables (each with a finite domain), and Σ is a finite set of events, usually called alphabet. The alphabet is split into two disjoint subsets: Σ_c and Σ_u, representing controllable and uncontrollable events respectively. $L_0 \subseteq L$ is a set of possible initial locations, $V_0(V)$ is an expression indicating possible initial values of all variables in V, $L_m \subseteq L$ is the set of marked locations, and $V_m(V)$ is an expression indicating marked values for variables V. 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, γ is a guard expression, indicating for which variable values the transition can take place, and υ is an update expression that indicates new values for the variables after the transition has occurred. If for some transition an update is not specified for some variable, then its value remains the same when taking the transition. If a guard is not specified, then it is assumed 'true'.

Essentially, locations can be modeled as variables, which we call *location pointer variables*, and transitions between locations can be modeled as guards and updates. Furthermore, expressions stated in an EFA can be encoded in (Boolean) predicates, that return true or false for a particular evaluation of variable values. Therefore, to simplify our explanations, and also stay consistent with the implementation of symbolic supervisor synthesis in CIF, we consider *Linearized Finite Automata (LFAs)*. We will shortly introduce them here, for more details on the linearization of EFAs we refer to Nadales Agut and Reniers (2011). An LFA is defined as a *5-tuple*:

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

in which X is a finite set of variables (which may contain a location pointer variable). A *state* is defined by a valuation over these variables. Σ is the alphabet. X_0 and X_m are predicates over variables from X that respectively represent the initial and marked states. Note that for a predicate $P(X)$ we may simply write P when it is clear from the context that it is a predicate over variables X. 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 variables X and *new state variables* $X^+ = \{x^+ | x \in X\}$, representing what state will be reached when the

edge is taken from a particular current state. I.e., for each variable x, we use variable x^+ to indicate the value of x after an event occurs. We assume $X \cap X^+ = \varnothing$.

Example 1 *We consider the EFA of* Fig. 1. *This EFA consists of two locations* $L=\{l_0, l_1\}$ *of which* l_1 *is marked:* $L_m=\{l_1\}$, *as indicated in* Fig. 1 *by a double circle. The initial location* $L_0 = \{l_0\}$ *is indicated by the dangling incoming arrow. We have two Boolean variables named a and b. Both variables are initially set to false, as indicated by the expression next to the dangling incoming arrow, i.e.,* $V_0(V) = \neg a \wedge \neg b$. *Events a_on and b_on can occur at* l_0, *the value of a and b will then respectively update to true. These updates are denoted in* Fig. 1 *by the keyword 'do'. An edge with event label continue can be taken from origin location* l_0 *to target location* l_1. *This can only happen if the guard* $a=b$ *evaluates to true. The guard is denoted by the keyword 'when'. All variable values are considered to be marked, i.e.,* $V_m(V) = true$.

The same model can be expressed as an LFA as follows: The set of variables is $\{l_s, a, b\}$, *where* l_s *is the location pointer variable with domain* $\{l_0, l_1\}$, *used to encode the current location. The initial state predicate is* $l_s=l_0 \wedge \neg a \wedge \neg b$. *The marked state predicate is* $l_s=l_1$. *There are three edges:*

$$
\begin{array}{lll}
(a\_on & , l_s=l_0 & , l_s^+=l_0 \wedge a^+ \wedge b^+=b); \\
(b\_on & , l_s=l_0 & , l_s^+=l_0 \wedge a^+=a \wedge b^+); \text{ and} \\
(continue & , l_s=l_0 \wedge a=b & , l_s^+=l_1 \wedge a^+=a \wedge b^+=b).
\end{array}
$$

2.2 Symbolic supervisor synthesis

The purpose of applying supervisor synthesis is to generate a supervisor automaton such that the parallel composition between the plant automaton and supervisor is *safe*, *nonblocking*, *controllable*, and *maximally permissive* (Cassandras and Lafortune 2021). Safe means that the requirements are always satisfied. How requirements are specified, and what it exactly means to satisfy them, is discussed in more detail in Section 6. Nonblocking indicates that from every reachable state in the controlled system, a marked state can be reached. Controllable means that from every reachable state in the controlled system, when the plant can execute an uncontrollable event, this event can also be executed in the parallel composition between

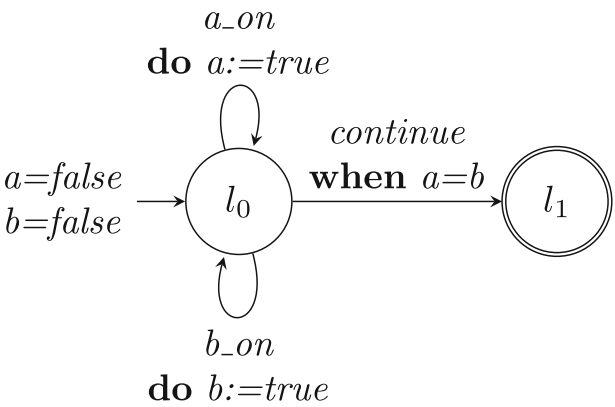


Fig. 1 Example EFA

supervisor and plant. In other words, the supervisor does not disallow any uncontrollable events. Maximal permissiveness says that these properties are ensured without disabling any events that do not strictly need to be disallowed.

In Algorithm 1 a supervisor synthesis algorithm is presented. This synthesis algorithm is strongly based on the algorithm introduced by Ouedraogo et al. (2011), simplified by using an LFA instead of an EFA. We will shortly introduce it here, for more details we refer to Ouedraogo et al. (2011). In line 1 the requirements are applied by using algorithm `applyRequirements`. We study the application of requirements in more detail in Section 6. For this preliminary section, it is only relevant to know that `applyRequirements` returns a predicate N that defines all states where the requirements are satisfied, a set of edges E_S which is the set of plant edges E with restricted guards of the controllable events such that the requirements are satisfied, and a predicate $X_{0,S}$ defining the initial states that satisfy the requirements, i.e., the initial states that are safe[3].

After applying the requirements, predicate N might still allow blocking states (states that cannot reach a marked state). Algorithm 1 iteratively calculates nonblocking states N, followed by bad states B. The calculation to obtain N and B is done by means of a backward reachability search, given in Algorithm 2. The algorithm loops over the edges, as defined in the *edge order*, discussed later in more detail. The bad states are removed from N, which may induce other states to become blocking. Therefore, the algorithm repeats these steps until a fixpoint is reached, i.e., no further bad states get removed. Next, the guards of the controllable edges are modified such that the edge is only taken when a nonblocking state will be reached. Here, notation $N(X^+)$ denotes predicate $N(X)$ in which each current state variable $x \in X$ is substituted by its new state counterpart x^+. Finally, the supervisor LFA is constructed.

Algorithm 1 `SS` (Supervisor Synthesis).

Input: Plant LFA $A_L = (X, \Sigma, E, X_0, X_m)$, state exclusion predicates SX, state-event exclusion predicates EX
Output: Supervisor LFA S
1: $(N, E_S, X_{0,S}) =$ `applyRequirements`(SX, EX, E, X_0)
2: **repeat**
3: $\quad N' = N$
4: $\quad N =$ `BRS`(N, E_S, X_m)
5: $\quad B =$ `BRS`$(true, \{(\sigma, g, u) \in E | \sigma \in \Sigma_u\}, \neg N)$
6: $\quad N = N \wedge \neg B$
7: **until** $N = N'$
8: **for all** $(\sigma, g, u) \in E_S$ with $\sigma \in \Sigma_c$
9: $\quad g(X) = g(X) \wedge \exists_{X^+}[N(X^+) \wedge u(X, X^+)]$
10: **end**
11: $S = (X, \Sigma, E_S, X_{0,S} \wedge N, X_m \wedge N)$

2.3 Binary decision diagrams

A Binary Decision Diagram (BDD) (Akers 1978) is a data structure that is used to represent Boolean functions and predicates, and can be used to represent and perform calculations on an LFA. BDDs are directed acyclic graphs that consist out of two types of nodes: decision- and terminal nodes. Each decision node is labeled by a Boolean variable b and has two edges

[3] In case of multiple initial states, it is assumed that the supervisor can restrict in which of those states the system can start.

Algorithm 2 BRS (Backward Reachability Search).

Input: Restriction predicate $P_R(X)$, edges E, start predicate $P(X)$
Output: Coreachable predicate $P'(X)$
1: **repeat**
2: $P'(X) = P(X)$
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** $P(X) = P'(X)$

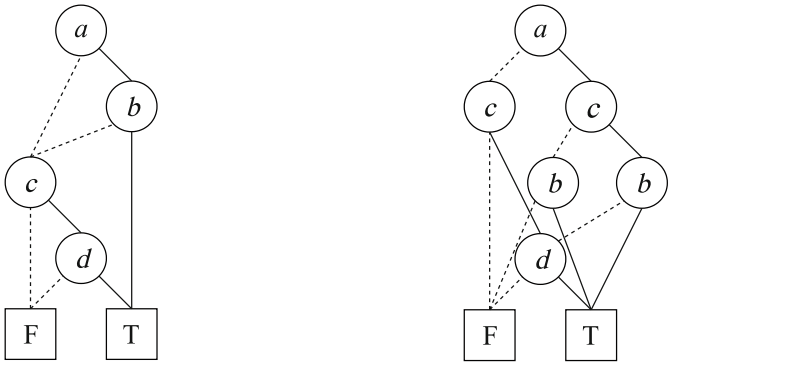
leading to child nodes, one edge labeled true and the other false. When evaluating b to true or false we take the respective edge. At the leaves of the BDD are terminal nodes, that are labeled by true or false, indicating the final result of evaluating the BDD.

When referring to BDDs in this paper, we implicitly always mean *reduced ordered* BDDs (Bryant 1992). This type of BDD imposes some additional restrictions such that the BDD is minimal in the number of decision nodes and canonical for a given order of the variables. This order is strictly imposed over all the variables in the BDD and is called the *variable order*. A variable order is denoted as $<$, where $b_1 < b_2$ indicates that decision node b_1 is placed closer to the root node than b_2.

The variable order can influence the number of decision nodes required to encode a Boolean expression, see Fig. 2 for an example. Visually, we represent true edges by solid lines and false edges by dashed lines. The size of a BDD is defined by the number of decision nodes and in worst-case this size can be exponential in the number of Boolean variables (Bryant 1992).

In our work, when we mention variable order, we refer to an order of the LFA variables, which corresponds to an order of Boolean variables in the BDD. How an ordering of LFA variables relates to an ordering of BDD variables is shown in Example 2.

Example 2 *We consider an LFA with integer variables y and z. De domains of y and z respectively are $\{0, 1, 2\}$ and $\{0, 1\}$. Variable y can be encoded using two Boolean variables: $b_{y,0}$ and $b_{y,1}$; variable z requires a single Boolean variable $b_{z,0}$. As for every current state variable, there is a new state variable: there is an integer y^+ corresponding to Boolean variables $b_{y^+,0}$ and $b_{y^+,1}$, and integer z^+ corresponding to Boolean variable $b_{z^+,0}$. Let us assume the LFA variables are ordered by $y < z$. This then corresponds to the following*



(a) Variable order $a < b < c < d$. (b) Variable order $a < c < b < d$.

Fig. 2 Two BDDs representing $(a \wedge b) \vee (c \wedge d)$ for different variable orders

variable order of the Boolean variables: $b_{y,0} < b_{y^+,0} < b_{y,1} < b_{y^+,1} < b_{z,0} < b_{z^+,0}$. *So, in the order of Boolean variables, a Boolean variable corresponding to a current state variable is always immediately succeeded by the respective Boolean variable corresponding to the new state variable (using default settings in CIF).*

2.4 CIF

There are several tools that allow modeling of plants and requirements with the ability to synthesize a supervisor. Of the tools considered in Reniers and van de Mortel-Fronczak (2018), the tools Supremica (Malik et al. 2017) and CIF (van Beek et al. 2014) allow for the use of EFAs and base their supervisor synthesis algorithm on the use of BDDs. In this paper synthesis is performed using the supervisor synthesis tool for EFAs of CIF. CIF has been used to synthesize supervisors for industrial sized systems (Theunissen et al. 2014; Reijnen et al. 2017, 2018a; Loose et al. 2018; Korssen et al. 2018; Reijnen et al. 2020).

CIF is part of the ESCET project, an Eclipse open-source project since 2020 (Fokkink et al. 2023). ESCET provides a model-based approach and toolkit for synthesis-based engineering of correct-by-construction supervisory controllers of discrete-event systems.

3 Evaluating computational effort in symbolic supervisor synthesis

The performance of algorithms is typically judged by their space- and time complexity (Meinel and Theobald 1998). In this section, we first introduce the metrics *peak used BDD nodes* and *BDD operation count* to quantitatively express the space- and time effort required for supervisor synthesis. Then we explain why these metrics are advantageous over conventional metrics such as peak random access memory and wall clock time, to assess the computational efficiency of symbolic computations. Finally in this section, we use the BDD-based metrics to demonstrate the impact of the variable order and edge order on the computational effort of synthesis.

We distinguish *complexity* from *effort*. Complexity regards classes of problems, and defines the generic trend of the (space/time) resources a computation requires for inputs of different sizes, often expressed using 'Big O' notation (Knuth 1976). Effort specifies the amount of resources used for one particular computation, where the complete input is considered rather than only its size. This input includes algorithm configuration settings and, in our case, variable and edge order.

3.1 Peak used BDD nodes

During symbolic supervisor synthesis, the number of BDD nodes used to describe the predicates generally fluctuates. Since *reduced* ordered BDDs are used, which are minimal representations, the used BDD nodes is the minimal amount of BDD nodes required to represent the predicates at that point during the computation. The space effort can be measured by the peak (maximal) number of BDD nodes used during synthesis (Meinel and Theobald 1998; Vahidi et al. 2006).

In CIF, BDD nodes are stored in a hash table. Each new node is allocated to an entry in the hash table. Once the hash table reaches a certain fill rate, garbage collection is employed to free no longer used entries. We only count the *used* BDD nodes, i.e., hash table entries that still contain relevant information for the BDDs that are still in use. Garbage collection is performed

by means of a standard mark-and-sweep algorithm. Functions from the implementation of this algorithm in the JavaBDD library[4] are reused to count the BDD nodes that are in use. This measurement is performed each time just before a BDD reference is deleted, which causes used nodes to become unused. Thereby ensuring the exact peak value of used nodes is found. For a more detailed explanation on BDD node references, used nodes, and unused (dead) nodes we refer to Somenzi (1999).

Peak used BDD nodes is a reproducible metric: Performing a supervisor synthesis twice with the same input yields exactly the same peak used BDD nodes.

3.2 BDD operation count

The time effort can be expressed in the number of steps/operations during a computation (Meinel and Theobald 1998). As supervisor synthesis is done by performing operations on BDDs, we use BDD operation count to express the time effort of performing supervisor synthesis. Since BDD operations (such as 'and' and 'or') are implemented as functions that employ structural recursion on BDD nodes, the number of invocations of such functions can be used to express time effort. Since the functions are deterministic, the results are reproducible.

Generally, these functions consist of three parts. First, a few checks are performed to see whether the requested calculation is a terminal case. Second, if it is a non-terminal case, it is checked whether the calculation has already been performed, and is still in the cache. Note that we do not mean hardware cache here, but a table actively storing results of previous calculations. If both previous cases do not apply, the function performs recursive expansion over the child nodes. Each time this recursive expansion is performed, i.e., when operations are applied on the BDD, we increment the BDD operation count. For more details about terminal cases, cache lookup and recursive expansion over child nodes we refer to Somenzi (1999).

3.3 Relevance of metrics

In order to compare the BDD-based metrics to conventional metrics, e.g., wall clock time, memory usage, and state space sizes, we perform a number of supervisor syntheses and extract these metrics. The data presented in this paper is acquired by performing supervisor syntheses to the models shown in Table 1. The models are selected to have a wide range of model sizes. Table 1 shows, and is sorted by, the worst case state space size of the uncontrolled plant for each model, which is the product of all location and variable domain sizes. The first two models have a worst-case state space of a single state because their plant models only contain EFAs with a single state. The requirement specifications of these models contain automata with more than one state.

For a supervisor synthesis of the Waterway lock model, Fig. 3 shows how the number of used BDD nodes evolves, as BDD operations are performed during synthesis. Intuitively, the horizontal axis represents the ever-increasing number of operations performed as synthesis progresses, and the vertical axis represents the fluctuating memory usage. The metrics presented in this paper are the maxima along both axes in this plot: the peak used BDD nodes and the final BDD operation count.

[4] The JavaBDD library is available at https://javabdd.sourceforge.net.

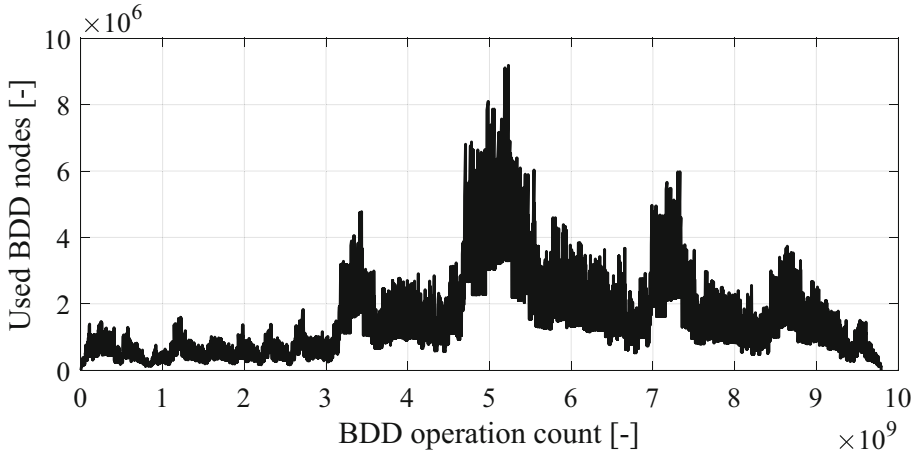
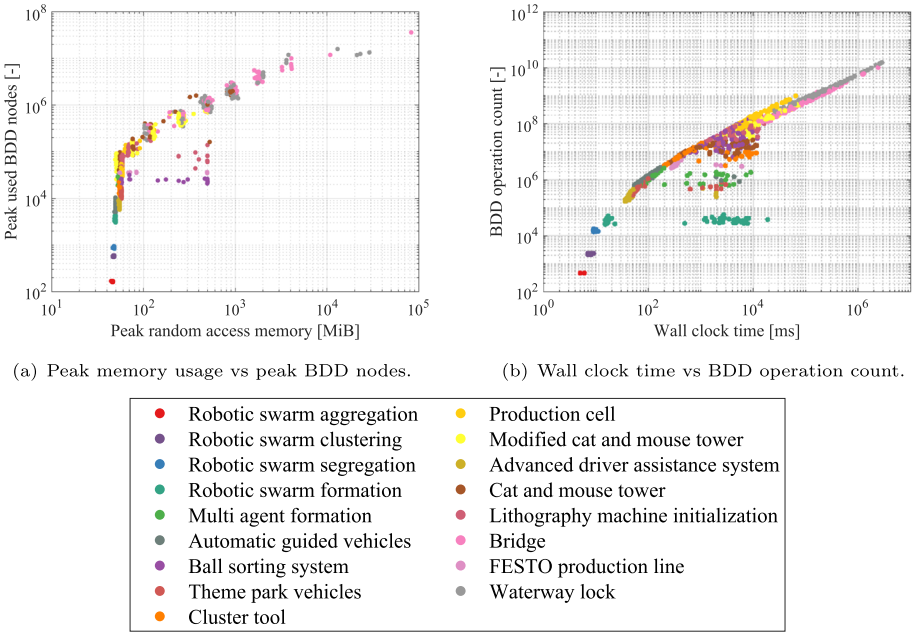


Fig. 3 Evolution of used BDD nodes during synthesis

Figure 4(a) and (b) show how peak random access memory and wall clock time respectively relate to peak used BDD nodes and BDD operation count. A supervisor was synthesized for each model of Table 1 for 100 pairs of random variable- and edge orders. Note that these variable orders have been re-ordered by heuristic algorithms FORCE and Sliding Window (SW), which we discuss later, to obtain the variable order that synthesis is actually applied with. The measurements were performed in sequence using two Intel Xeon Gold 6226 processors



(a) Peak memory usage vs peak BDD nodes.

(b) Wall clock time vs BDD operation count.

Fig. 4 BDD-based metrics against conventional metrics

clocked at 2.70 GHz, operating on Linux. The measurements for random access memory and wall clock time were done separately from the measurements of the BDD-based metrics to avoid them from interfering.

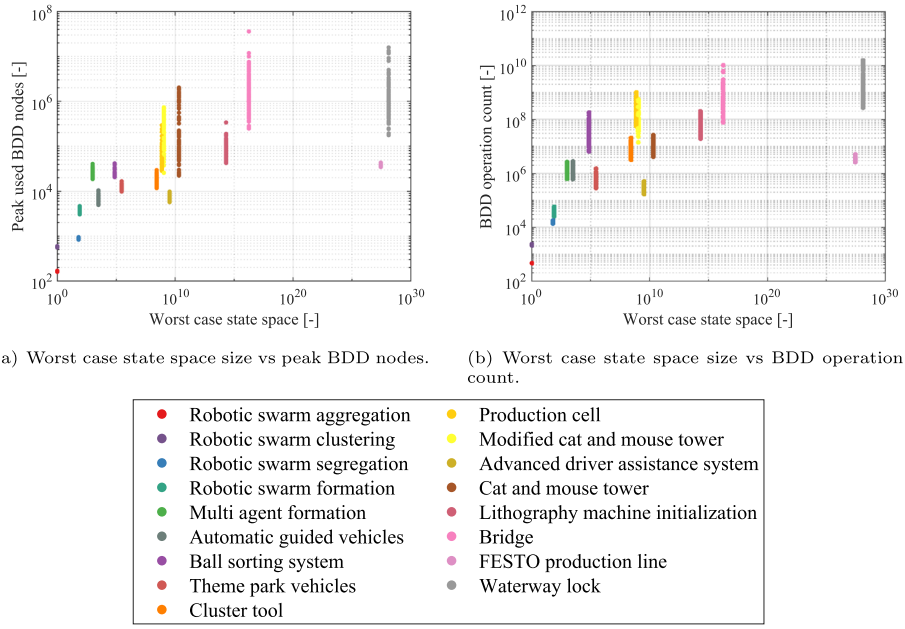
For small models, peak random access memory cannot indicate a difference in synthesis effort, as all results of the small models are grouped around 60 MiB. For some of the smaller and medium-sized models there is a significant amount of noise in the measured wall clock time and random access memory. For wall clock time, this noise typically originates from delays in I/O procedures (writing the output file). For memory, the noise originates from loaded in classes and the practically random intervals at which Java performs garbage collection.

For larger computations, a linear relation is visible between wall clock time and BDD operation count. The threshold at which this relation starts, and its slope, are dependent on the used hardware. The grouping that is seen for larger computations in Fig. 4(a) is a result of the manner in which the BDD space allocation takes place: when the current table is full, it gets doubled in size, the new free entries in this table will have an influence on the memory, but are not measured when counting the used BDD nodes. Also, when performing computations that require more memory, the Java Virtual Machine will perform garbage collection in the background to free memory. For separate measurements this will happen at different times, which impacts the peak random access memory, not the amount of used BDD nodes. Note that for the measurements in Fig. 4 additional garbage collection is performed before every measurement to achieve a consistent situation between measurements.

An advantage of wall clock time and peak random access memory is that a user performing supervisor synthesis is more likely to be familiar with these metrics. It gives a better idea whether their computer is able to perform the synthesis in an acceptable amount of time given the available memory. However, opposed to the BDD-based metrics, wall clock time and peak random access memory are not deterministic, so they will yield different results for every synthesis run. Their results are influenced by aspects including loaded classes, garbage collection, and I/O operations. The BDD-based metrics enable a distinction in effort for the actual synthesis portion of the computation. Also, the BDD-based metrics are platform independent. Particularly the wall clock time of some synthesis will be influenced by the used hardware, making it difficult to compare results.

Worst-case state space of the uncontrolled system is also frequently used to indicate (expected) synthesis effort. The advantage of using worst case state space size of the uncontrolled system over BDD-based metrics to indicate the synthesis effort, is that no supervisor synthesis or reachability computations are required to calculate this number. Figure 5(a) and (b) show how this state space size relates to the BDD-based metrics. From these figures we can conclude that the worst case state space size is not a very accurate indicator of the expected computational effort of synthesis: the correlation is very weak. Note that there are multiple ways to indicate state space sizes, some also taking the product with requirement automata, but we found that also in those cases the state space size does not accurately indicate the computational effort.

In summary, there are several advantages of the BDD-based metrics over the conventional metrics: They are deterministic: performing a supervisor synthesis twice with the same input and algorithm configuration will give the exact same result. This determinism also holds when doing the synthesis on two different platforms, even if one is a supercomputer and the other is a personal computer. As a result, it becomes easier to compare results from different measurements or publications. Also, there is no overhead in the measurement, loaded-in Java classes and other computer processes will not influence the measurement.



(a) Worst case state space size vs peak BDD nodes.

(b) Worst case state space size vs BDD operation count.

Fig. 5 BDD-based metrics against conventional metrics

3.4 Impact of variable- and edge order on computational effort

We presented two metrics that indicate the computational effort of a supervisor synthesis. However, we have to be careful when making conclusions based on this synthesis effort. The variable- and edge order have an influence on the results. This can also be seen in Figs. 4 and 5, where the results per model are scattered due to using different variable- and edge orders. Recall that since the BDD-based metrics are deterministic, re-performing a synthesis with the same variable- and edge orders would provide the exact same result.

It is well known that the variable order has a notable impact on the BDD size, and consequently on the computational effort. Therefore, the variable ordering heuristics FORCE and SW (Aloul et al. 2003) are implemented in CIF (already prior to this work). FORCE is supplied with a variable order and reorders it to group variables together that have high interaction, meaning they often appear together in guards and updates. Note that this algorithm finds a local optimum: initializing it with different orders, might give different resulting variable orders. SW starts from a variable order, and "slides a window" across the variables to locally optimize that part of the order. In this work, always a (default) window size of 4 is used. These heuristics can be sequenced. We denote *FORCE+SW* to indicate that first FORCE is applied to some initial variable order, and SW is performed on the variable order computed by FORCE, to produce the variable order used in synthesis.

In previous work (Thuijsman et al. 2019; Lousberg et al. 2020; Thuijsman et al. 2021) the definition of variable relations was different from the current implementation. For further details we refer to the CIF documentation. In contrast, the variable relations as used in Section 4 are reproducible with both the ESCET version used in this paper (version 0.9), as well as the current ESCET release (version 3.0). Hence, we repeat the experiments of Thuijsman et al. (2019); Lousberg et al. (2020) and Thuijsman et al. (2021) using the same

variable dependencies for all variable ordering heuristics, that we discuss below, such that we can accurately compare their efficiency.

We investigate to what extent the edge order and initial variable order influence the supervisor synthesis effort. For each model in Table 1, a supervisor has been synthesized for all combinations of 100 random edge orders and 100 random initial variable orders. These initial variable orders are re-ordered by FORCE+SW to produce the variable order used in synthesis. The effort of performing each synthesis is shown in Fig. 6.

It can be seen that there are major differences in computational effort by using different orders. For the Waterway lock model, the highest peak used BDD nodes is 658 times larger than the lowest peak used BDD nodes. For BDD operations this factor is 338. This is purely a result of changing the edge orders and initial variable orders: all other algorithm configurations were the same for all measurements.

Figure 6 also shows that measuring both peak used BDD nodes and BDD operation count is relevant. It would be difficult to distinguish the computational effort between some of the syntheses if only one of the metrics was used. For example, if we only measured the

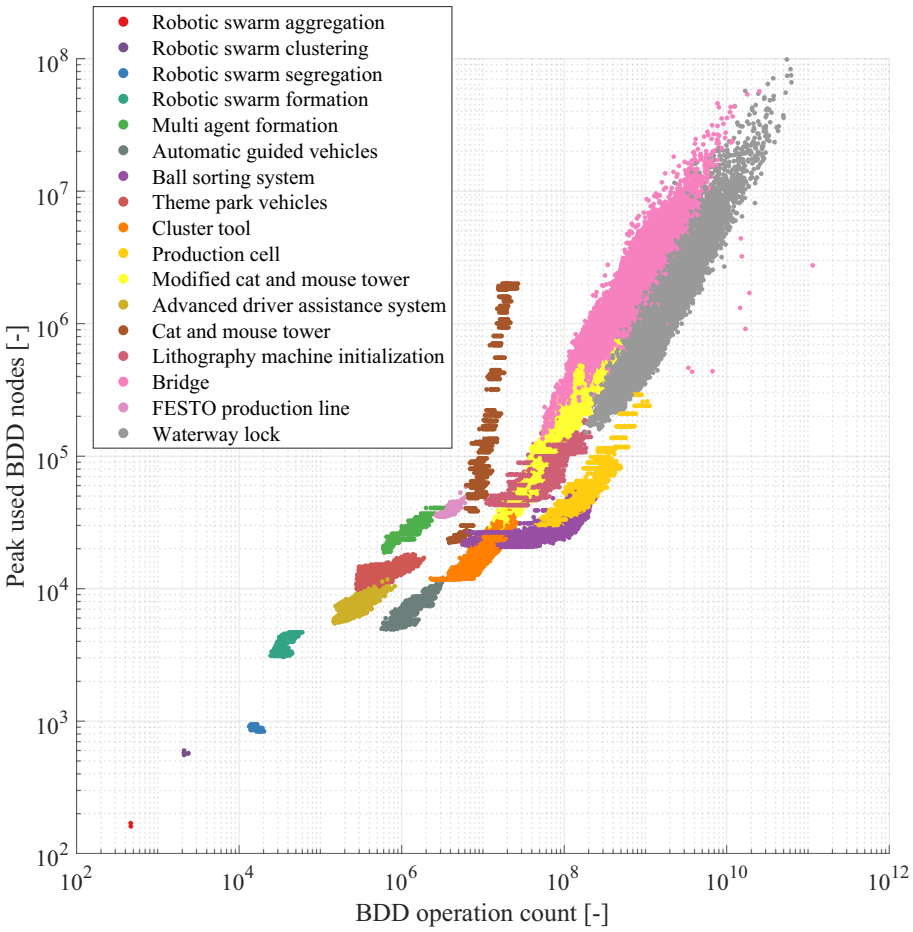


Fig. 6 Supervisor synthesis effort for all combinations of 100 edge- and 100 variable orders for each model

BDD operation count, we would not see much difference for the efforts of synthesis for Cat and mouse tower. If we only measured peak used BDD nodes. we would not see much difference for the effort of synthesis for Ball sorting system. Measuring both metrics enables us to differentiate between the efforts of synthesis for each model based on the various initial variable orders and edge orders.

Figure 7 shows the peak used BDD nodes for all syntheses of the Theme park vehicles model. Darker squares indicate a higher amount of peak used BDD nodes. All variable- and edge orders were given an index. A row shows the peak used BDD nodes of all syntheses that were performed with the same edge order and varying variable orders, and a column shows the same for a fixed variable order with varying edge orders. In Fig. 7, we see rows and columns where the elements are similarly colored, indicating that variable order and edge order both have a reasonable impact on the peak used BDD nodes for this particular model. There are other models where only the elements in columns are similarly colored, indicating that the variable order mainly influences their synthesis effort. We observe similar results for the BDD operation count.

Figure 7, along with analyzing the same plot for other models, shows that a relatively poor/good variable order generally performs relatively poor/good for all edge orders and vice versa. This means the variable order and edge order can be improved individually, which is what we respectively focus on in Sections 4 and 5.

If we define the peak used BDD nodes for a certain model as a deterministic function $f(o_{v,i}, o_{e,j})$, where $o_{v,i}$ is the i^{th} sample random variable order and $o_{e,j}$ the j^{th} sample

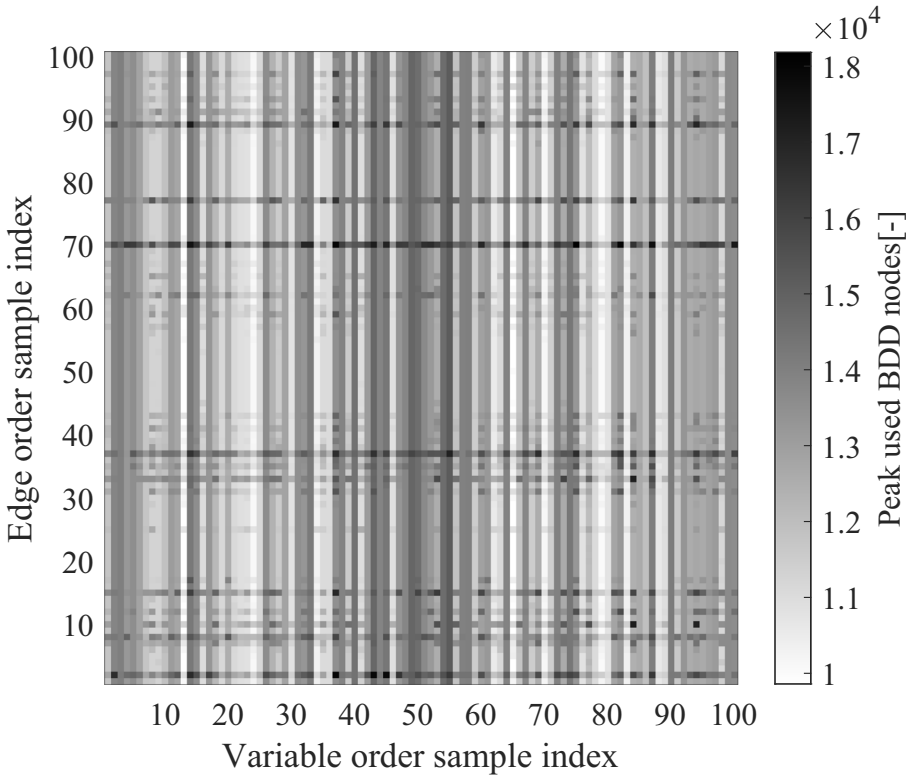


Fig. 7 Peak used BDD nodes for all supervisor syntheses of the theme park vehicles model

random edge order, the global sample mean (Montgomery and Runger 2018) of the peak used BDD nodes $\mu_G(f)$ is given by Eq. 1.

$$\mu_G(f) = \frac{1}{N \cdot M} \sum_{i=1}^{N} \sum_{j=1}^{M} f(o_{v,i}, o_{e,j}), \tag{1}$$

where N and M respectively are the total number of sampled variable- and edge orders. For our experiment, $N = M = 100$ for each model.

The global (unbiased) sample variance (Montgomery and Runger 2018) of the peak used BDD nodes $\sigma_G^2(f)$ is given by Eq. 2.

$$\sigma_G^2(f) = \frac{1}{N \cdot M - 1} \sum_{i=1}^{N} \sum_{j=1}^{M} \big(f(o_{v,i}, o_{e,j}) - \mu_G(f)\big)^2. \tag{2}$$

The sample variance $\sigma_{v,i}^2(f)$ of the peak used BDD nodes for the edge orders tested with a particular variable order $o_{v,i}$, is given by Eq. 3.

$$\sigma_{v,i}^2(f) = \frac{1}{M - 1} \sum_{j=1}^{M} \big(f(o_{v,i}, o_{e,j}) - \mu_{v,i}(f)\big)^2, \tag{3}$$

where $\mu_{v,i}(f) = \frac{1}{M} \sum_{j=1}^{M} f(o_{v,i}, o_{e,j})$ is the mean peak used BDD nodes of the edge orders tested with variable order $o_{v,i}$. The mean sample variance for fixed variable orders $\overline{\sigma_v^2}(f)$ is computed by Eq. 4:

$$\overline{\sigma_v^2}(f) = \frac{1}{N} \sum_{i=1}^{N} \sigma_{v,i}^2(f). \tag{4}$$

Equations 3 and 4 can analogously be applied to compute the sample variance of peak used BDD nodes for variable orders tested with particular edge orders $\sigma_{e,j}^2(f)$, and the mean sample variance for fixed edge orders $\overline{\sigma_e^2}(f)$. Likewise, we can define a function $g(o_{v,i}, o_{e,j})$ for the BDD operation count of a model and apply above computations to this.

When relating these characteristics to what we see in Fig. 7, a low mean sample variance for fixed variable orders $\overline{\sigma_v^2}(f)$ would indicate a similar amount of peak used BDD nodes for a given variable order. This would be visible in Fig. 7, as elements located in the same column would be similarly colored. This would indicate that the variable order mainly influences the peak used BDD nodes, and the edge order has little influence.

For each model, the global sample mean μ_G, global sample variance σ_G^2, mean sample variance for fixed variable orders $\overline{\sigma_v^2}$ and mean sample variance for fixed edge orders $\overline{\sigma_e^2}$ are given for peak used BDD nodes (f) and BDD operation count (g) in Table 2. For most of the models, the mean sample variance for fixed variable orders is smaller than the mean sample variance for fixed edge orders. This indicates that the variable order has a larger influence on the supervisor synthesis effort than the edge order. The effect of the variable order is particularly notable, when considering that for these experiments FORCE+SW is being applied to the variable order (to compute the variable order that synthesis is performed with). The variance is even higher when FORCE+SW is not applied (Lousberg et al. 2020). However, the mean variance for fixed variable orders is large enough that the edge order is still of considerable influence to the supervisor synthesis effort.

Models that require a relatively large amount of supervisor synthesis effort, also have a relatively large variance in effort. This would also be observed if we were to normalize

Table 2 Sample means and variances of all models

Name	$\mu_G(f)$	$\sigma^2_G(f)$	$\overline{\sigma^2_v}(f)$	$\overline{\sigma^2_e}(f)$	$\mu_G(g)$	$\sigma^2_G(g)$	$\overline{\sigma^2_v}(g)$	$\overline{\sigma^2_e}(g)$
Robotic swarm aggregation	$1.6 \cdot 10^2$	$2.0 \cdot 10^1$	0.0	$2.0 \cdot 10^1$	$4.6 \cdot 10^2$	1.0	0.0	1.0
Robotic swarm clustering	$5.7 \cdot 10^2$	$1.6 \cdot 10^2$	0.0	$1.6 \cdot 10^2$	$2.2 \cdot 10^3$	$1.7 \cdot 10^4$	0.0	$1.7 \cdot 10^4$
Robotic swarm segregation	$8.8 \cdot 10^2$	$9.7 \cdot 10^2$	0.0	$9.8 \cdot 10^2$	$1.6 \cdot 10^4$	$8.9 \cdot 10^5$	$5.6 \cdot 10^5$	$4.7 \cdot 10^5$
Robotic swarm formation	$3.9 \cdot 10^3$	$2.6 \cdot 10^5$	0.0	$2.6 \cdot 10^5$	$3.6 \cdot 10^4$	$4.5 \cdot 10^7$	$4.6 \cdot 10^6$	$4.2 \cdot 10^7$
Multi agent formation	$2.6 \cdot 10^4$	$2.3 \cdot 10^7$	0.0	$2.3 \cdot 10^7$	$1.2 \cdot 10^6$	$2.0 \cdot 10^{11}$	$5.8 \cdot 10^9$	$2.0 \cdot 10^{11}$
Automated guided vehicles	$6.0 \cdot 10^3$	$1.5 \cdot 10^6$	$7.1 \cdot 10^3$	$1.5 \cdot 10^6$	$1.1 \cdot 10^6$	$1.7 \cdot 10^{11}$	$1.4 \cdot 10^{10}$	$1.6 \cdot 10^{11}$
Ball sorting system	$2.4 \cdot 10^4$	$9.3 \cdot 10^6$	$8.4 \cdot 10^5$	$9.3 \cdot 10^6$	$3.4 \cdot 10^7$	$7.6 \cdot 10^{14}$	$1.5 \cdot 10^{14}$	$6.7 \cdot 10^{14}$
Theme park vehicles	$1.3 \cdot 10^4$	$2.0 \cdot 10^6$	$3.2 \cdot 10^5$	$1.8 \cdot 10^6$	$5.7 \cdot 10^5$	$3.8 \cdot 10^{10}$	$2.6 \cdot 10^{10}$	$1.7 \cdot 10^{10}$
Cluster tool	$1.7 \cdot 10^4$	$2.1 \cdot 10^7$	$2.2 \cdot 10^6$	$2.0 \cdot 10^7$	$8.5 \cdot 10^6$	$1.3 \cdot 10^{13}$	$1.2 \cdot 10^{12}$	$1.2 \cdot 10^{13}$
Production cell	$5.9 \cdot 10^4$	$1.9 \cdot 10^9$	$3.3 \cdot 10^6$	$1.9 \cdot 10^9$	$1.9 \cdot 10^8$	$2.4 \cdot 10^{16}$	$4.9 \cdot 10^{14}$	$2.4 \cdot 10^{16}$
Modified cat and mouser tower	$2.9 \cdot 10^5$	$4.5 \cdot 10^{10}$	$4.3 \cdot 10^8$	$4.5 \cdot 10^{10}$	$2.0 \cdot 10^8$	$2.5 \cdot 10^{16}$	$2.8 \cdot 10^{14}$	$2.5 \cdot 10^{16}$
Advanced driver assistance system	$7.4 \cdot 10^3$	$1.1 \cdot 10^6$	$2.6 \cdot 10^5$	$8.7 \cdot 10^5$	$2.8 \cdot 10^5$	$6.7 \cdot 10^9$	$4.6 \cdot 10^9$	$2.9 \cdot 10^9$
Cat and mouse tower	$7.3 \cdot 10^5$	$6.3 \cdot 10^{11}$	$4.1 \cdot 10^3$	$6.3 \cdot 10^{11}$	$1.4 \cdot 10^7$	$3.1 \cdot 10^{13}$	$4.8 \cdot 10^{11}$	$3.1 \cdot 10^{13}$
Lithography machine initialization	$8.5 \cdot 10^4$	$1.6 \cdot 10^9$	$2.1 \cdot 10^6$	$1.6 \cdot 10^9$	$6.6 \cdot 10^7$	$1.2 \cdot 10^{15}$	$8.4 \cdot 10^{13}$	$1.1 \cdot 10^{15}$
Bridge	$2.3 \cdot 10^6$	$8.7 \cdot 10^{12}$	$5.2 \cdot 10^{12}$	$6.5 \cdot 10^{12}$	$8.9 \cdot 10^8$	$2.5 \cdot 10^{18}$	$2.1 \cdot 10^{18}$	$2.2 \cdot 10^{18}$
FESTO production line	$3.6 \cdot 10^4$	$1.1 \cdot 10^6$	$5.1 \cdot 10^5$	$1.0 \cdot 10^6$	$3.2 \cdot 10^6$	$9.5 \cdot 10^{10}$	$3.7 \cdot 10^{10}$	$7.9 \cdot 10^{10}$
Waterway lock	$2.3 \cdot 10^6$	$1.9 \cdot 10^{13}$	$1.0 \cdot 10^{13}$	$1.8 \cdot 10^{13}$	$2.5 \cdot 10^9$	$1.5 \cdot 10^{19}$	$5.7 \cdot 10^{18}$	$1.3 \cdot 10^{19}$

to the mean values of the models, i.e., σ^2/μ or σ/μ. This indicates that applying a good variable- and edge order becomes more beneficial when considering models that require more supervisor synthesis effort.

4 DCSH variable ordering heuristic

In Section 3.4 we touched on the extent in which edge and variable order influence the computational effort. There is a large variance in the computational effort of synthesis as a result of the initial variable order, even if FORCE+SW is applied. As such, we want to find a variable order for which the computational effort is generally low. Unfortunately, finding the variable order that minimizes the BDD size is an NP-complete problem (Bryant 1992). This is why heuristic variable ordering algorithms are used.

In this section, we introduce a heuristic algorithm named *DSM-based Cuthill-McKee-Sloan variable ordering Heuristic* (DCSH) to find a variable order that reduces the computational effort required for symbolic supervisor synthesis compared to current implementation (FORCE+SW). This heuristic is based on two matrix ordering heuristics from Cuthill and McKee (1969) and Sloan (1989), that are used to minimize the *Weighted Event Span* (WES) (Siminiceanu and Ciardo 2006). It is shown in Meijer and van de Pol (2016) that these heuristics are able to reduce the WES, and thereby the computational effort for symbolic model checking. Since the approach is shown to work for symbolic model checking, we hypothesize it also might work for symbolic supervisor synthesis. These matrix reordering heuristics are applied to a Dependency Structure Matrix (DSM) that stores the number of times BDD-variables appear together in transition relations, to find a new variable order. Heuristics are used, since directly minimizing the WES is also an NP-complete problem (Siminiceanu and Ciardo 2006). The performance of DCSH in relation to the current implementation is experimentally evaluated in Section 4.3.

4.1 Transition relation, variable order, and computational effort

When studying the evolution of BDDs during synthesis, most computational effort is performed during the reachability searches (Algorithm 2). Specifically, during the existential quantification operation in line 3 of Algorithm 2. Because this operation is applied many times, the guard and update predicates are placed in a single predicate, which is the *transition relation*: $T_e(X, X^+) = g_e(X) \wedge u_e(X, X^+)$, where there is a transition relation T_e for each edge $e = (\sigma, g_e(X), u_e(X, X^+))$.

Existential quantification over the transition relations is frequently applied during synthesis. This operation can be executed by first computing $P(X^+) \wedge T_e(X, X^+)$ and then quantifying over X^+. However, this results in a large intermediate result of $P(X^+) \wedge T_e(X, X^+)$. Therefore, both the conjunction and existential quantification are computed in a single recursive pass over $P(X^+)$ and $T_e(X, X^+)$ by utilizing the relational product operation (Burch et al. 1994). This operation prevents computing the entire BDD $P(X^+) \wedge T_e(X, X^+)$ and quantifies early over X^+, thereby reducing memory usage and number of required operations. Nevertheless, computing the relational product is known to be an expensive computation (Burch et al. 1994).

We say sets of variables are *strongly related* if they appear together in many transitions. BDDs are overall small if strongly related BDD-variables are placed near each other in the variable order Minato 1996; Somenzi 1999. If we keep variables of each transition relation

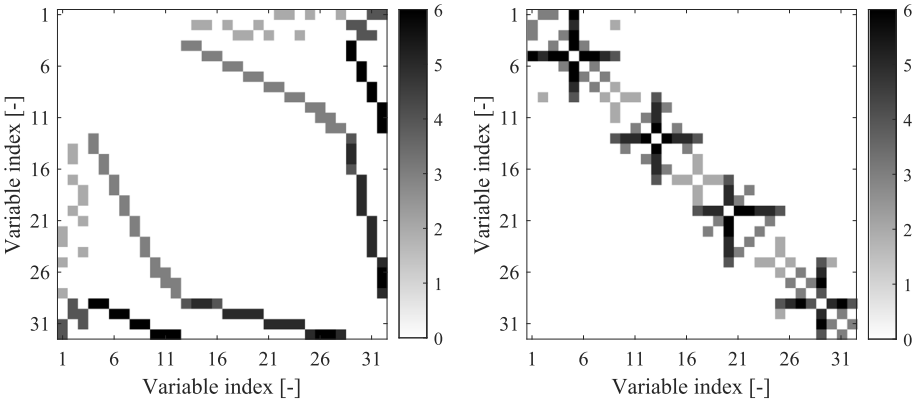
near each other in the variable order, it is likely that the resulting BDDs representing the (nonblocking and bad-state) predicates are kept small during synthesis, leading to reduced computational effort. Since we frequently apply computationally expensive operations to BDDs that represent a predicate $T_e(X, X^+)$ (for some $e \in E$), we base the relatedness of variables on these predicates.

4.2 Dependency structure matrix reordering

A DSM is a square $n \times n$ matrix representing dependencies between n aspects of a system or model (Browning 2016). We capture variables of the *LFA* along the rows and columns where each index represents a single variable. In our use, the variables are always ordered the same along the row and column axis. In this paper we utilize static Numerical DSMs (NDSMs). The off-diagonal elements can be non-negative integers, where the value indicates the number of times the respective variables appear together in a predicate expression T_e of an edge in the *LFA*. In our use, the diagonal elements are always zero. Furthermore, all dependencies in the NDSM are regarded as undirected, thus providing a symmetric matrix. Subsequently, the NDSM is manipulated by two matrix ordering heuristic algorithms that reorder the row and column indices such that non-zero values are placed towards the diagonal. The order in which the variables appear along the rows/columns is used as variable order for synthesis. Essentially, we are creating a variable order such that variables that often appear together in some T_e, are placed near each other in the variable order.

Before synthesis we extract the variables that appear in predicate expression T_e for each edge $e \in E$. For all occurrences of pairs of variables per T_e we increment the element in the NDSM by one, thus a higher value indicates a stronger dependency between the variables. The increment is executed for both combinations of the pair such that the resulting NDSM is symmetric.

Figure 8 shows an example of an NDSM before and after reordering. The before image has the variables ordered alphabetically, which is the default initial variable order in CIF. The variables are reordered, by applying the method we discuss next. In the after image we observe that variables that are closely related are clustered together.



(a) NDSM before reordering. (b) NDSM after reordering.

Fig. 8 NDSM before and after reordering of the cluster tool model

Algorithm 3 Weighted Cuthill-McKee ordering.

Input: NDSM M
Output: Variable order list R
1: Initialize empty list R, compute weighted adjacency graph A of M
2: **for each** Connected subgraph A' of A not connected to another node in A
3: Compute pseudo-peripheral node p of A'
4: Mark p and append p to R
5: **while** Unmarked nodes exist in A'
6: Find list C of unmarked neighbors of p
7: Sort list C such that the nodes are in descending weight
8: Sort list C such that nodes with equal weight are in ascending degree
9: Append C to R and mark all nodes in C
10: Set the next node in R as p
11: **end**
12: **end**

The use of DSMs in supervisory control theory is not new. In Goorden et al. (2020) DSMs are used to find clusters of highly interactive components for the purpose of applying multilevel synthesis. However, we are not looking for clusters, but interested in reordering the row and column indices such that higher valued elements are placed as close as possible towards the diagonal relative to lower valued elements. By finding such an order, we also find an order where variables that often appear together in transition relations are placed near each other.

In practice, the NDSMs constructed in our approach are sparse. We utilize existing variable ordering heuristics, that have been designed for bandwidth, profile, and/or wavefront reduction of symmetric sparse matrices. For an elaboration on these metrics we refer to Cuthill and McKee (1969) for bandwidth and Sloan (1989) for profile and wavefront. By minimizing any of these metrics, an order is achieved for which relatively low computational effort is expected. The effective use of these heuristics for static variable order optimization for BDDs is shown in Meijer and van de Pol (2016), where several bandwidth, profile and wavefront reducing node ordering heuristics have been compared. These heuristics apply a reordering to the adjacency graph that can directly be extracted from an NDSM. As we utilize an NDSM we append the graph's edges by weights resulting in a *weighted* adjacency graph. For an NDSM with row index i and column index j, we denote elements by $\eta_{i,j}$. For each row i we generate a node labeled by i. Subsequently, each non-zero element $\eta_{i,j}$ results in an undirected edge with weight $\eta_{i,j}$ between nodes i and j. This results in a weighted adjacency graph where the node labels are reordered using the heuristics Weighted Cuthill-McKee ordering and Sloan's ordering.

4.2.1 Weighted Cuthill-McKee ordering

The Cuthill-McKee (CM) ordering is a bandwidth reducing node ordering heuristic introduced by Cuthill and McKee (1969). The standard algorithm places non-zero elements near the diagonal to result in a matrix with a lower bandwidth. We introduce an adjustment to the standard algorithm, such that it is able to differentiate between non-zero elements. Higher valued elements are prioritized in being placed close to the diagonal over lower valued elements. We will refer to this algorithm as the *weighted* CM ordering, which is shown in Algorithm 3. Lines 7 and 8 are an adjustment of the standard algorithm. As a convention, the for-loop at line 2 selects sub-graphs in descending size.

4.2.2 Sloan's ordering

Sloan's ordering is a profile and wavefront reducing node ordering heuristic introduced by Sloan (1989). It places non-zero elements near the diagonal to result in a lower profile of the matrix. In this paper the standard algorithm is not adjusted to be able to differentiate between non-zero elements, although this is of interest for future work.

4.2.3 Weighted event span

We apply both ordering heuristics to the NDSM indicating related pairs of variables. This results in two orders. Furthermore, we notice that reversing the order can sometimes lead to significant differences in synthesis effort. Siminiceanu and Ciardo (2006) noticed that placing variables that result in more costly operations towards the bottom of the BDD resulted in less effort required in a similar application of BDDs. This resulted in the Weighted Event Span (WES) metric. Furthermore, the WES has been extensively tested by Meijer and van de Pol (2016), where a correlation is shown between peak BDD nodes, computation time, and the WES for several types of decision diagrams applied to symbolic model checking. Given a variable order, the WES is found by

$$\text{WES} = \sum_{e \in E} \frac{2x_l(e)}{|X|} \cdot \frac{x_l(e) - x_h(e) + 1}{|X||E|} \tag{5}$$

where $|X|$ and $|E|$ respectively indicate the total number of variables and edges. $x_l(e)$ and $x_h(e)$ are respectively the lowest- and highest variable index from the variables in $T_e(X)$. The first term in Eq. 5 increases as $x_l(e)$ is placed later in the variable order. The second term increases the WES when $x_l(e) - x_h(e)$ is large.

To estimate which of the four orders (two orders resulting from two different ordering heuristics and two reverse orders) should be used in synthesis, the WES is computed for each of the orders. The order that has the lowest WES is used in synthesis. This results in the proposed variable ordering heuristic, named DSM-based Cuthill-McKee-Sloan variable ordering Heuristic, abbreviated to DCSH for ease of reference.

4.3 Experiments

In Lousberg et al. (2020), the efficiency of variable orders computed by DCSH was compared to that of FORCE+SW. As mentioned in Section 3.4, in Lousberg et al. (2020) DCSH used different variable relations than FORCE and SW. We repeat the same experiments of Lousberg et al. (2020) here, using the same variable relations, as discussed above, for each heuristic algorithm, albeit in different formats: DCSH using DSMs, and FORCE and SW using adjacency graphs. Additionally, the experiments are now performed for all models in Table 1, which is a larger set of models than was used in Lousberg et al. (2020).

For each model in Table 1, 10,000 random initial variable orders are generated. Each of those orders, is ordered by FORCE+SW and DCSH (separately) and then synthesis is performed using the computed order. We noticed it may be beneficial to apply FORCE+SW on the order computed by DCSH, so essentially performing sequence DCSH+FORCE+SW. For clarity, the different ways the variable orders are computed are shown schematically in Fig. 9. We perform synthesis using each resulting variable order and measure the computational effort. Besides the initial variable order and turning on the BDD measurements, other settings in CIF were kept default in these experiments.

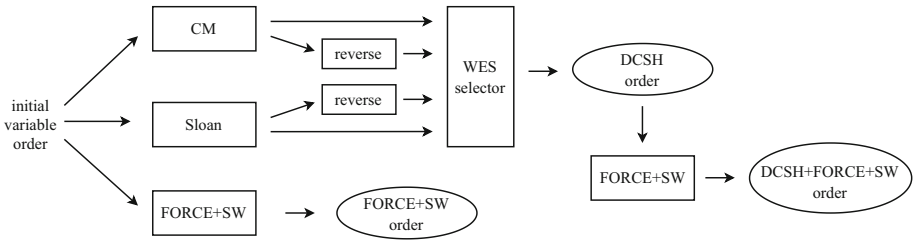


Fig. 9 Computation of the variable orders

For each heuristic method, the minimal, mean, and maximal value are shown in Table 3, for both peak used BDD nodes and BDD operation count. The same results are presented graphically in Fig. 10. In that figure, the values are normalized to the mean value for FORCE+SW for that model. E.g., a mean value of 0.8 ($8 \cdot 10^{-1}$) for DCSH+FORCE+SW for some model, indicates that for that model the mean computational effort of DCSH+FORCE+SW was 20% lower than the mean computational effort of FORCE+SW.

Summarizing the results, for both peak used BDD nodes and BDD operation count, for 16 out of 17 models, the maximal computational effort was found when using FORCE+SW. So, using FORCE+SW there is the likelihood to perform synthesis with relatively a really high computational effort. Generally, using DCSH or DCSH+FORCE+SW, removes most high-effort outliers: the maximal values are much lower. The mean of peak used BDD nodes over the measurements is less for DCSH+FORCE+SW compared to DCSH in 12 out of 17 models. For BDD operation count this is the case for 10 out of 17 models. Also, the maximal peak used BDD nodes using DCSH+FORCE+SW is less than the maximal peak used BDD nodes using just DCSH for 13 out of 17 models. For BDD operation count this is the case for 12 out of 17 models.

On average over all models, when using DCSH+FORCE+SW the peak used BDD nodes increase by 5%, and the BDD operation count lowers by 14%, relative to FORCE+SW. Relative to DCSH, it realizes 8% less peak used BDD nodes and a 12% lower BDD operation count. Even though DCSH+FORCE+SW realizes a slightly higher average amount of peak used BDD nodes than FORCE+SW, it is very effective at avoiding the high effort computations. In conclusion, using DCSH+FORCE+SW generally the least computational effort is required, and computations with relatively high computational effort are avoided. Therefore, using DCSH+FORCE+SW is advisable over just applying FORCE+SW or DCSH.

Even though generally a lower computational effort is achieved by using DCSH+FORCE+SW relative to FORCE+SW, it is clearly not always the case. The Cat and Mouse Tower (CMT) and Modified CMT model stand out, where the computational effort using DCSH+FORCE+SW is roughly double that of FORCE+SW. Relative to the other models, these models lack large automata but rather contain relatively many automata of smaller size and relatively many events with a low level of synchronization between the automata. These characteristics might be the cause of poorer performance of the suggested method. However, it is generally very difficult to characterize and compare models in this manner. Devising some way to predict what variable ordering heuristic may work well based on model characteristics remains future work.

The time to run the discussed variable ordering heuristic algorithms is negligible relative to the time to perform synthesis.

Table 3 Experimental results variable ordering heuristics

Name		Peak used BDD nodes			BDD operation count		
		FORCE+SW	DCSH	DCSH+ FORCE +SW	FORCE +SW	DCSH	DCSH+FORCE +SW
Swarm aggr.	min	$1.59 \cdot 10^2$	$1.59 \cdot 10^2$	$1.59 \cdot 10^2$	$4.60 \cdot 10^2$	$4.60 \cdot 10^2$	$4.60 \cdot 10^2$
	mean	$1.66 \cdot 10^2$	$1.66 \cdot 10^2$	$1.66 \cdot 10^2$	$4.94 \cdot 10^2$	$4.62 \cdot 10^2$	$4.62 \cdot 10^2$
	max	$1.70 \cdot 10^2$	$1.69 \cdot 10^2$	$1.69 \cdot 10^2$	$5.48 \cdot 10^2$	$4.65 \cdot 10^2$	$4.65 \cdot 10^2$
Swarm clust.	min	$5.55 \cdot 10^2$	$5.55 \cdot 10^2$	$5.55 \cdot 10^2$	$2.09 \cdot 10^3$	$2.09 \cdot 10^3$	$2.09 \cdot 10^3$
	mean	$5.74 \cdot 10^2$	$5.88 \cdot 10^2$	$5.88 \cdot 10^2$	$2.23 \cdot 10^3$	$2.09 \cdot 10^3$	$2.09 \cdot 10^3$
	max	$5.98 \cdot 10^2$	$5.98 \cdot 10^2$	$5.98 \cdot 10^2$	$2.38 \cdot 10^3$	$2.11 \cdot 10^3$	$2.11 \cdot 10^3$
Swarm segr.	min	$7.73 \cdot 10^2$	$8.02 \cdot 10^2$	$8.35 \cdot 10^2$	$1.31 \cdot 10^4$	$1.39 \cdot 10^4$	$1.42 \cdot 10^4$
	mean	$8.88 \cdot 10^2$	$8.62 \cdot 10^2$	$8.51 \cdot 10^2$	$1.44 \cdot 10^4$	$1.42 \cdot 10^4$	$1.43 \cdot 10^4$
	max	$1.03 \cdot 10^3$	$8.89 \cdot 10^2$	$8.89 \cdot 10^2$	$2.15 \cdot 10^4$	$1.44 \cdot 10^4$	$1.45 \cdot 10^4$
Swarm form.	min	$2.89 \cdot 10^3$	$3.26 \cdot 10^3$	$3.34 \cdot 10^3$	$2.25 \cdot 10^4$	$2.79 \cdot 10^4$	$2.52 \cdot 10^4$
	mean	$3.79 \cdot 10^3$	$4.14 \cdot 10^3$	$4.13 \cdot 10^3$	$3.60 \cdot 10^4$	$3.76 \cdot 10^4$	$3.71 \cdot 10^4$
	max	$4.71 \cdot 10^3$	$4.38 \cdot 10^3$	$4.32 \cdot 10^3$	$5.35 \cdot 10^4$	$3.95 \cdot 10^4$	$3.91 \cdot 10^4$
Multi agent form.	min	$1.83 \cdot 10^4$	$2.03 \cdot 10^4$	$2.09 \cdot 10^4$	$6.53 \cdot 10^5$	$5.86 \cdot 10^5$	$7.60 \cdot 10^5$
	mean	$2.50 \cdot 10^4$	$2.20 \cdot 10^4$	$2.18 \cdot 10^4$	$1.37 \cdot 10^6$	$6.32 \cdot 10^5$	$8.54 \cdot 10^5$
	max	$4.54 \cdot 10^4$	$2.42 \cdot 10^4$	$2.31 \cdot 10^4$	$4.95 \cdot 10^6$	$6.60 \cdot 10^5$	$9.30 \cdot 10^5$
Aut. guid. vehic.	min	$4.86 \cdot 10^3$	$4.90 \cdot 10^3$	$4.94 \cdot 10^3$	$9.03 \cdot 10^5$	$8.76 \cdot 10^5$	$9.45 \cdot 10^5$
	mean	$6.05 \cdot 10^3$	$5.19 \cdot 10^3$	$5.16 \cdot 10^3$	$1.45 \cdot 10^6$	$1.14 \cdot 10^6$	$1.11 \cdot 10^6$
	max	$1.61 \cdot 10^4$	$5.33 \cdot 10^3$	$5.57 \cdot 10^3$	$5.05 \cdot 10^6$	$1.24 \cdot 10^6$	$1.28 \cdot 10^6$
Ball sort. sys.	min	$2.02 \cdot 10^4$	$2.09 \cdot 10^4$	$2.11 \cdot 10^4$	$8.07 \cdot 10^6$	$9.33 \cdot 10^6$	$8.05 \cdot 10^6$
	mean	$2.44 \cdot 10^4$	$2.31 \cdot 10^4$	$2.30 \cdot 10^4$	$2.57 \cdot 10^7$	$1.39 \cdot 10^7$	$9.50 \cdot 10^6$

Table 3 continued

Name		Peak used BDD nodes			BDD operation count		
		FORCE+SW	DCSH	DCSH+ FORCE +SW	FORCE +SW	DCSH	DCSH+FORCE +SW
	max	$5.21 \cdot 10^4$	$2.57 \cdot 10^4$	$2.59 \cdot 10^4$	$2.28 \cdot 10^8$	$1.72 \cdot 10^7$	$1.31 \cdot 10^7$
Theme park veh.	min	$7.69 \cdot 10^3$	$8.79 \cdot 10^3$	$9.78 \cdot 10^3$	$1.55 \cdot 10^5$	$1.57 \cdot 10^5$	$1.57 \cdot 10^5$
	mean	$1.25 \cdot 10^4$	$1.08 \cdot 10^4$	$1.20 \cdot 10^4$	$2.00 \cdot 10^5$	$2.15 \cdot 10^5$	$1.92 \cdot 10^5$
	max	$1.65 \cdot 10^4$	$1.18 \cdot 10^4$	$1.34 \cdot 10^4$	$3.96 \cdot 10^5$	$2.45 \cdot 10^5$	$2.55 \cdot 10^5$
Cluster tool	min	$1.17 \cdot 10^4$	$1.18 \cdot 10^4$	$1.18 \cdot 10^4$	$2.26 \cdot 10^6$	$2.31 \cdot 10^6$	$2.22 \cdot 10^6$
	mean	$1.58 \cdot 10^4$	$1.18 \cdot 10^4$	$1.18 \cdot 10^4$	$6.88 \cdot 10^6$	$2.54 \cdot 10^6$	$2.40 \cdot 10^6$
	max	$4.18 \cdot 10^4$	$1.19 \cdot 10^4$	$1.18 \cdot 10^4$	$2.31 \cdot 10^7$	$2.87 \cdot 10^6$	$2.69 \cdot 10^6$
Prod. cell	min	$2.59 \cdot 10^4$	$3.00 \cdot 10^4$	$2.78 \cdot 10^4$	$5.05 \cdot 10^7$	$1.11 \cdot 10^8$	$6.55 \cdot 10^7$
	mean	$5.45 \cdot 10^4$	$4.99 \cdot 10^4$	$3.73 \cdot 10^4$	$2.25 \cdot 10^8$	$1.88 \cdot 10^8$	$1.29 \cdot 10^8$
	max	$8.75 \cdot 10^5$	$1.40 \cdot 10^5$	$1.23 \cdot 10^5$	$5.31 \cdot 10^9$	$5.50 \cdot 10^8$	$5.09 \cdot 10^8$
Modif. CMT	min	$1.83 \cdot 10^4$	$4.82 \cdot 10^5$	$4.80 \cdot 10^5$	$8.91 \cdot 10^6$	$3.33 \cdot 10^8$	$3.33 \cdot 10^8$
	mean	$2.99 \cdot 10^5$	$4.96 \cdot 10^5$	$4.96 \cdot 10^5$	$1.95 \cdot 10^8$	$3.55 \cdot 10^8$	$3.54 \cdot 10^8$
	max	$9.22 \cdot 10^5$	$5.11 \cdot 10^5$	$5.07 \cdot 10^5$	$5.28 \cdot 10^8$	$3.66 \cdot 10^8$	$3.65 \cdot 10^8$
Adv. driv. asst.	min	$5.77 \cdot 10^3$	$6.29 \cdot 10^3$	$5.84 \cdot 10^3$	$1.80 \cdot 10^5$	$1.77 \cdot 10^5$	$1.77 \cdot 10^5$
	mean	$7.22 \cdot 10^3$	$6.67 \cdot 10^3$	$6.80 \cdot 10^3$	$2.34 \cdot 10^5$	$2.03 \cdot 10^5$	$2.15 \cdot 10^5$
	max	$9.38 \cdot 10^3$	$7.10 \cdot 10^3$	$8.10 \cdot 10^3$	$3.67 \cdot 10^5$	$2.39 \cdot 10^5$	$3.08 \cdot 10^5$
CMT	min	$2.20 \cdot 10^4$	$1.81 \cdot 10^6$	$1.81 \cdot 10^6$	$3.12 \cdot 10^6$	$1.69 \cdot 10^7$	$1.69 \cdot 10^7$
	mean	$7.42 \cdot 10^5$	$1.83 \cdot 10^6$	$1.83 \cdot 10^6$	$1.44 \cdot 10^7$	$1.78 \cdot 10^7$	$1.78 \cdot 10^7$
	max	$2.09 \cdot 10^6$	$1.85 \cdot 10^6$	$1.85 \cdot 10^6$	$2.63 \cdot 10^7$	$1.89 \cdot 10^7$	$1.88 \cdot 10^7$

Table 3 continued

Name		Peak used BDD nodes			BDD operation count		
		FORCE+SW	DCSH	DCSH+ FORCE +SW	FORCE +SW	DCSH	DCSH+FORCE +SW
Lith. mach. init.	min	$3.04 \cdot 10^4$	$3.01 \cdot 10^4$	$3.01 \cdot 10^4$	$9.11 \cdot 10^6$	$9.40 \cdot 10^6$	$9.40 \cdot 10^6$
	mean	$9.01 \cdot 10^4$	$8.06 \cdot 10^4$	$8.06 \cdot 10^4$	$6.88 \cdot 10^7$	$5.05 \cdot 10^7$	$5.05 \cdot 10^7$
	max	$7.89 \cdot 10^5$	$1.73 \cdot 10^5$	$1.73 \cdot 10^5$	$5.92 \cdot 10^8$	$1.42 \cdot 10^8$	$1.42 \cdot 10^8$
Bridge	min	$5.92 \cdot 10^4$	$7.22 \cdot 10^4$	$5.99 \cdot 10^4$	$4.85 \cdot 10^6$	$9.44 \cdot 10^6$	$4.88 \cdot 10^6$
	mean	$8.88 \cdot 10^4$	$3.72 \cdot 10^5$	$7.80 \cdot 10^4$	$7.63 \cdot 10^6$	$1.80 \cdot 10^7$	$6.05 \cdot 10^6$
	max	$1.06 \cdot 10^6$	$2.13 \cdot 10^6$	$1.24 \cdot 10^5$	$2.07 \cdot 10^8$	$6.09 \cdot 10^7$	$9.73 \cdot 10^6$
FESTO	min	$3.44 \cdot 10^4$	$3.48 \cdot 10^4$	$3.48 \cdot 10^4$	$2.79 \cdot 10^6$	$2.97 \cdot 10^6$	$2.88 \cdot 10^6$
	mean	$3.65 \cdot 10^4$	$3.54 \cdot 10^4$	$3.52 \cdot 10^4$	$3.45 \cdot 10^6$	$3.15 \cdot 10^6$	$2.96 \cdot 10^6$
	max	$8.34 \cdot 10^4$	$3.62 \cdot 10^4$	$3.59 \cdot 10^4$	$9.21 \cdot 10^6$	$3.37 \cdot 10^6$	$3.15 \cdot 10^6$
Lock	min	$7.40 \cdot 10^4$	$1.26 \cdot 10^5$	$7.37 \cdot 10^4$	$1.41 \cdot 10^7$	$3.45 \cdot 10^7$	$1.35 \cdot 10^7$
	mean	$1.19 \cdot 10^5$	$1.69 \cdot 10^5$	$1.11 \cdot 10^5$	$6.04 \cdot 10^7$	$2.70 \cdot 10^8$	$4.23 \cdot 10^7$
	max	$6.74 \cdot 10^5$	$3.84 \cdot 10^5$	$1.61 \cdot 10^5$	$6.18 \cdot 10^8$	$7.48 \cdot 10^8$	$1.93 \cdot 10^8$

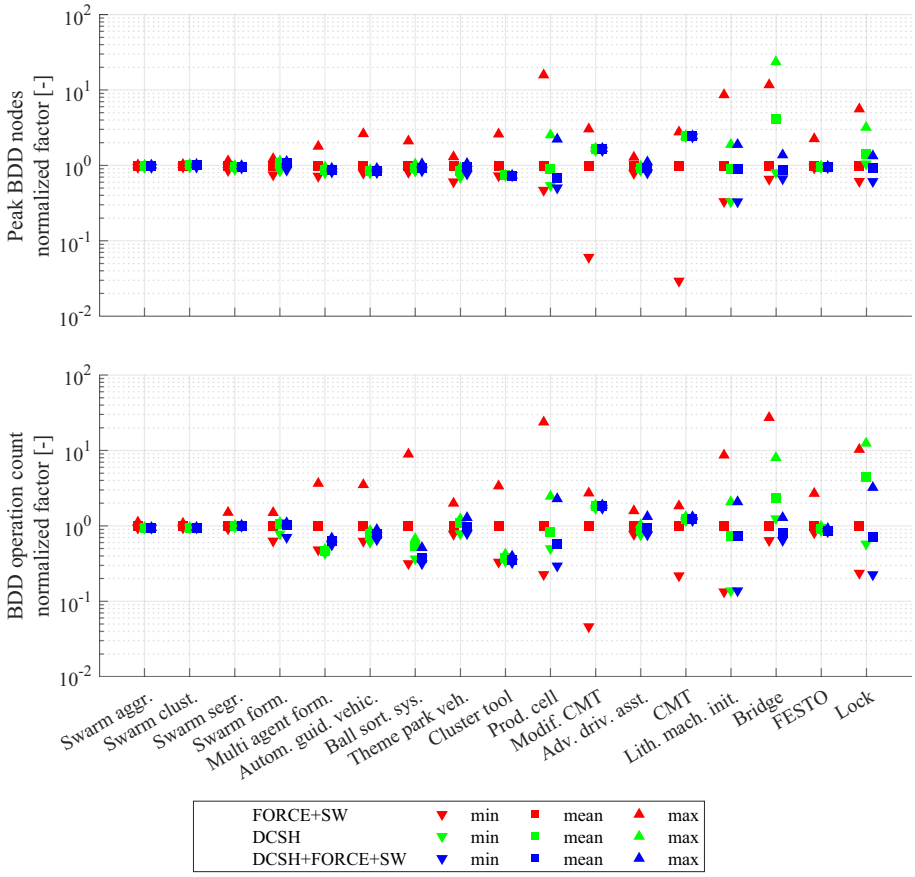


Fig. 10 Minimum, mean, and maximum computational effort for variable ordering heuristics, normalized to mean computational effort for FORCE+SW

5 Edge order

As shown in Section 3.4, next to the variable order also the edge order has a significant impact on the computational effort of synthesis. In the current implementation of CIF, there are six options to set the edge order:

1. model: the order in which each edge appears when reading the model top-to-bottom;
2. reverse-model: the reverse of 'model';
3. sorted: alphabetical sorting of the edges by their event label;
4. reverse-sorted: the reverse of 'sorted';
5. random: a random ordering (optionally with a seed);
6. a manually specified order.

In this section, we compare the efficiency of the first five options. There are perhaps more interesting approaches to ordering the edges, e.g., such as presented in Vahidi et al. (2006) and Fei et al. (2014). These approaches however do not satisfy our self-imposed restriction mentioned in Section 1 to investigate static (not on-the-fly) optimization. Yet, however simple the static edge ordering heuristics may be, we will see in the following they can still produce

good results, and investigating which option to use is worthwhile, also in order to come up with more meticulous heuristics in the future.

For each model in Table 1, synthesis is performed using model, reverse-model, sorted, and reverse-sorted edge order. Also for 100 random edge orders synthesis is performed. Following Section 4, DCSH+FORCE+SW is applied as variable ordering heuristic in these experiments, using the variable relations as introduced in Section 4.2. Besides applying DCSH+FORCE+SW, setting the edge order, and turning on BDD measurements, other settings in CIF are kept default in these experiments. The results of these experiments are shown in Table 4. For the random edge orders, the mean of the 100 measurements is shown. The same results are shown graphically in Fig. 11. For model, reverse-model, sorted, and reverse-sorted edge order, computational effort is displayed, normalized to the mean of using the 100 random edge orders. E.g., a value of $2 \cdot 10^{-1}$ indicates a 80% reduction in effort compared to the random orders on average.

As was concluded in Section 3.4, the edge order has a smaller influence on the peak used BDD nodes compared to BDD operation count. For most models the peak used BDD nodes are similar for the different edge orders. However, for the Bridge and Waterway lock models, using any edge order option other than random considerably reduces the peak memory usage, with reductions up to 90%, compared to the mean of using random edge orders, for both models. Overall, for peak used BDD nodes the average reduction by using any edge order option other than random is 10%.

The edge order has a larger influence on the BDD operation count. Again, for the Bridge and Waterway lock models the highest impacts are found, with a reduction up to 99% for the Waterway lock model compared to the mean of using random edge orders. Overall, for BDD operation count the average reduction by using any edge order option other than random is 22%.

Over all models, the best average reduction for BDD operation count is found using reverse-model, with a average reduction of 29%. For model, sorted, and reverse-sorted these average reductions are 16%, 22%, and 21% respectively. Also, reverse-model is the only heuristic that performed the same or better than the averaged random edge orders for every model. Therefore, usage of the reverse-model edge order is advisable over the other edge ordering heuristics available in CIF.

We give a possible reason why model, reverse-model, sorted, and reverse-sorted generally perform better than the random edge orders. In all models, the model is specified by a network of automata. If an event is locally specified within an automaton, its name gets prefixed with the automaton name (to avoid duplicate names). As a result for these ordering heuristics, edges that are used in the same automata are placed next to each other in the edge order. It is likely that edges specified in the same automaton, address similar variables in their guard and update expressions. Therefore, calculations on the parts of the BDD representing these variables are performed in close succession to each other. This means there is a higher likelihood of comparable calculations still being in the cache if we perform calculations on the same variables back-to-back. We refer back to Section 3.2: when a previous calculation is found in the cache, that result is used, no computations on the BDD need to be performed, and the BDD operation count is not incremented. So, iterating over edges in an order such that similar calculations are performed back-to-back improves the efficiency of the cache mechanism, therefore speeding up synthesis. This reasoning also may explain why the heuristics work particularly well for the Bridge and Waterway Lock model. In these models, the automata relevant to the same module are clustered into groups. These group names are also prefixed to the automaton name for synthesis. As a consequence, automata that have high interaction

Table 4 Experimental results event ordering heuristics

Name	Peak used BDD nodes					BDD operation count				
	Random	Model	Reverse model	Sorted	Reverse sorted	Random	Model	Reverse model	Sorted	Reverse sorted
Swarm aggr.	$1.69 \cdot 10^2$	$1.69 \cdot 10^2$	$1.69 \cdot 10^2$	$1.69 \cdot 10^2$	$1.69 \cdot 10^2$	$4.60 \cdot 10^2$	$4.60 \cdot 10^2$	$4.60 \cdot 10^2$	$4.60 \cdot 10^2$	$4.60 \cdot 10^2$
Swarm clust.	$5.98 \cdot 10^2$	$5.98 \cdot 10^2$	$5.98 \cdot 10^2$	$5.98 \cdot 10^2$	$5.98 \cdot 10^2$	$2.09 \cdot 10^3$	$2.09 \cdot 10^3$	$2.09 \cdot 10^3$	$2.09 \cdot 10^3$	$2.09 \cdot 10^3$
Swarm segr.	$8.89 \cdot 10^2$	$8.89 \cdot 10^2$	$8.89 \cdot 10^2$	$8.89 \cdot 10^2$	$8.89 \cdot 10^2$	$1.59 \cdot 10^4$	$1.42 \cdot 10^4$	$1.49 \cdot 10^4$	$1.59 \cdot 10^4$	$1.34 \cdot 10^4$
Swarm form.	$3.41 \cdot 10^3$	$3.41 \cdot 10^3$	$3.41 \cdot 10^3$	$3.41 \cdot 10^3$	$3.41 \cdot 10^3$	$2.79 \cdot 10^4$	$2.55 \cdot 10^4$	$2.73 \cdot 10^4$	$2.53 \cdot 10^4$	$2.80 \cdot 10^4$
Multi agent form.	$2.25 \cdot 10^4$	$2.25 \cdot 10^4$	$2.25 \cdot 10^4$	$2.25 \cdot 10^4$	$2.25 \cdot 10^4$	$7.53 \cdot 10^5$	$8.65 \cdot 10^5$	$6.36 \cdot 10^5$	$8.65 \cdot 10^5$	$6.36 \cdot 10^5$
Aut. guid. vehic.	$5.20 \cdot 10^3$	$5.21 \cdot 10^3$	$5.30 \cdot 10^3$	$5.21 \cdot 10^3$	$5.27 \cdot 10^3$	$7.93 \cdot 10^5$	$1.02 \cdot 10^6$	$4.51 \cdot 10^5$	$8.32 \cdot 10^5$	$7.09 \cdot 10^5$
Ball sort. sys.	$2.59 \cdot 10^4$	$2.59 \cdot 10^4$	$2.59 \cdot 10^4$	$2.59 \cdot 10^4$	$2.59 \cdot 10^4$	$1.30 \cdot 10^7$	$8.47 \cdot 10^6$	$3.89 \cdot 10^6$	$7.00 \cdot 10^6$	$8.27 \cdot 10^6$
Theme park veh.	$1.27 \cdot 10^4$	$1.26 \cdot 10^4$	$1.26 \cdot 10^4$	$1.26 \cdot 10^4$	$1.26 \cdot 10^4$	$5.50 \cdot 10^5$	$1.94 \cdot 10^5$	$2.16 \cdot 10^5$	$2.08 \cdot 10^5$	$2.05 \cdot 10^5$
Cluster tool	$1.18 \cdot 10^4$	$1.18 \cdot 10^4$	$1.18 \cdot 10^4$	$1.18 \cdot 10^4$	$1.18 \cdot 10^4$	$3.52 \cdot 10^6$	$2.41 \cdot 10^6$	$3.28 \cdot 10^6$	$2.44 \cdot 10^6$	$3.27 \cdot 10^6$
Prod. cell	$7.03 \cdot 10^4$	$7.03 \cdot 10^4$	$7.03 \cdot 10^4$	$7.03 \cdot 10^4$	$7.03 \cdot 10^4$	$2.07 \cdot 10^8$	$2.70 \cdot 10^8$	$1.03 \cdot 10^8$	$1.38 \cdot 10^8$	$2.47 \cdot 10^8$
Modif. CMT	$5.06 \cdot 10^5$	$4.93 \cdot 10^5$	$5.68 \cdot 10^5$	$4.99 \cdot 10^5$	$5.54 \cdot 10^5$	$4.04 \cdot 10^8$	$3.65 \cdot 10^8$	$3.92 \cdot 10^8$	$3.63 \cdot 10^8$	$4.05 \cdot 10^8$
Adv. driv. asst.	$6.20 \cdot 10^3$	$6.17 \cdot 10^3$	$5.96 \cdot 10^3$	$6.07 \cdot 10^3$	$5.98 \cdot 10^3$	$2.05 \cdot 10^5$	$1.84 \cdot 10^5$	$1.59 \cdot 10^5$	$1.71 \cdot 10^5$	$1.57 \cdot 10^5$
CMT	$1.82 \cdot 10^6$	$1.82 \cdot 10^6$	$1.82 \cdot 10^6$	$1.82 \cdot 10^6$	$1.82 \cdot 10^6$	$1.81 \cdot 10^7$	$1.81 \cdot 10^7$	$1.67 \cdot 10^7$	$1.70 \cdot 10^7$	$1.95 \cdot 10^7$
Lith. mach. init.	$4.63 \cdot 10^4$	$4.63 \cdot 10^4$	$4.63 \cdot 10^4$	$4.63 \cdot 10^4$	$4.63 \cdot 10^4$	$2.15 \cdot 10^7$	$2.51 \cdot 10^7$	$1.68 \cdot 10^7$	$2.46 \cdot 10^7$	$1.72 \cdot 10^7$
Bridge	$7.06 \cdot 10^5$	$7.20 \cdot 10^4$	$1.71 \cdot 10^5$	$2.05 \cdot 10^5$	$1.12 \cdot 10^5$	$2.02 \cdot 10^8$	$5.55 \cdot 10^6$	$4.42 \cdot 10^7$	$2.14 \cdot 10^7$	$1.61 \cdot 10^7$
FESTO	$3.49 \cdot 10^4$	$3.49 \cdot 10^4$	$3.49 \cdot 10^4$	$3.49 \cdot 10^4$	$3.49 \cdot 10^4$	$2.95 \cdot 10^6$	$2.96 \cdot 10^6$	$2.74 \cdot 10^6$	$2.92 \cdot 10^6$	$2.83 \cdot 10^6$
Lock	$7.58 \cdot 10^5$	$7.66 \cdot 10^4$	$8.58 \cdot 10^4$	$8.66 \cdot 10^4$	$7.56 \cdot 10^4$	$1.10 \cdot 10^9$	$1.58 \cdot 10^7$	$5.56 \cdot 10^7$	$3.33 \cdot 10^7$	$1.67 \cdot 10^7$

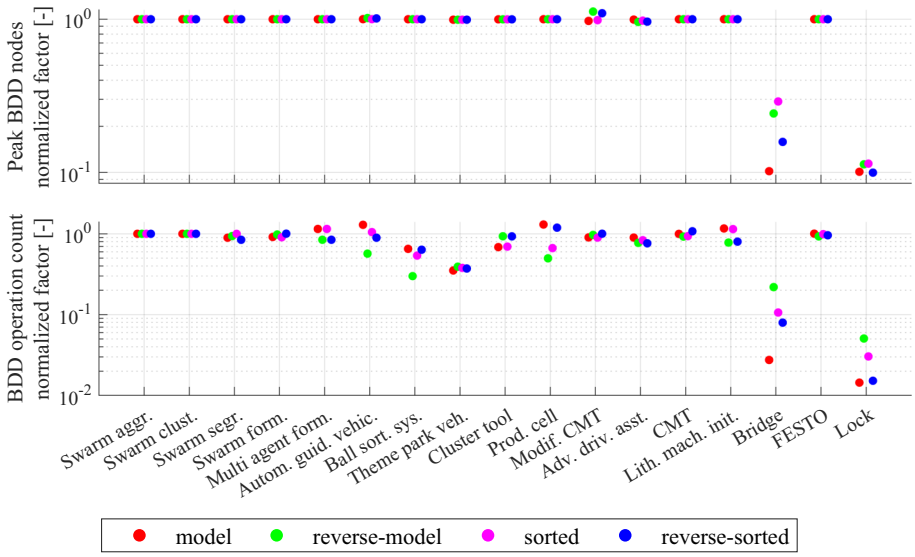


Fig. 11 Computational effort reduction factor for edge order heuristics normalized to random

are both in close proximity inside the model, as well as when sorting alphabetically. So for the edge ordering heuristics, related edges are also placed close to each other.

Also, we can reason why reverse-model generally has the best results. People generally write down the automata in a way that logically follows the behavior from top-to-bottom. For synthesis, the main computations are calculating the nonblocking and bad state predicate through backward reachability searches, i.e., the behavior is followed backwards (from the nonblocking or bad states). Evaluating an edge that does not lead to a currently found nonblocking/bad state costs computational effort, but does not aid in further construction of the nonblocking/bad state predicate. As a result, states are more efficiently found by evaluating edges in the reverse order of how they occur in the behavior, for which practically reverse-model is a good approximation.

6 Efficiently enforcing requirements

Central to supervisory control theory, is the specification of behavioral requirements, and enforcing those through supervisor synthesis. In CIF, requirements can be specified in three ways:

1. *requirement EFA*: an EFA prescribing allowed behavior w.r.t a subset of the plant's events. These are converted to an LFA prior to synthesis as described for plant EFA in Section 2.1.
2. *state exclusion expression*: a predicate defining a condition that needs to hold in every state of the controlled system.
3. *state-event exclusion expression*: a predicate defining a condition that needs to hold for a particular event to occur. We use the notation $\sigma \Rightarrow J$ for the state-event exclusion expression that expresses that event σ may only occur when predicate J holds.

The latter two requirement types are discussed in Markovski et al. (2010). Even though we define the safe states or states from which events are allowed to occur, we call these 'exclusion' requirements because they result in a restriction on the plant behavior.

Conversion of the requirements to their predicate-based counterparts is relatively straightforward, and specification of requirements, along with this conversion, is exemplified in Example 3.

Example 3 *We consider two traffic lights, each regulating traffic for their road at a two-way intersection. The plant behavior can be modeled by two automata, given in* Fig. 12. *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 EFA, given in* Fig. 13. *Alternatively, the requirement specification can be given by a state exclusion expression (i.e., this is the syntax a modeler would use in CIF):*

$$not(LightA.Green\ and\ LightB.Green),$$

which directly relates to a state exclusion predicate:

$$\neg(l_A = Green \wedge l_B = Green),$$

where, e.g., l_A is the location pointer variable for LightA that can take values Green and Red.

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

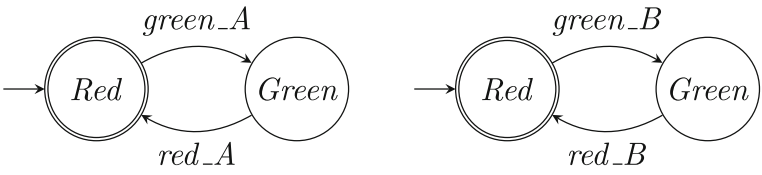
$$green\_A\ needs\ LightB.Red$$
$$green\_B\ needs\ LightA.Red,$$

these expressions, written in CIF syntax, can be directly converted to state-event exclusion predicates:

$$green\_A \Rightarrow l_B = Red$$
$$green\_B \Rightarrow l_A = Red$$

We will refer to the set of all state exclusion predicates as *SX*, and to the set of all state-event exclusion predicates as *EX*. For simplicity, we will consider specifications that do not contain any requirement EFA. Enforcing the requirements expressed by automata in (symbolic) supervisor synthesis is well known (Ramadge and Wonham 1987; Flordal et al. 2007; Ouedraogo et al. 2011; Cassandras and Lafortune 2021).

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



(a) LightA. (b) LightB.

Fig. 12 Traffic lights plant automata

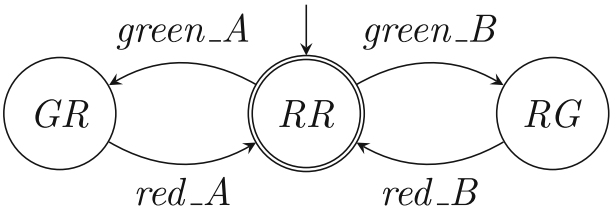


Fig. 13 Traffic lights requirement automaton

behavior is achieved. Notably, the usage of state exclusion expressions would lead to syntheses that require high computational effort. Consequently, this type of requirement specification was sometimes avoided when modeling larger systems. This can be observed in, e.g., the Waterway lock model from Reijnen et al. (2017).

For the purpose of modeling ease and model clarity, in a number of cases it might be useful to use state exclusion expressions. For the traffic lights in Example 3, the state exclusion expression is arguably the most straightforward formalization of the informal requirement specification. Ideally, the usage of state exclusion expressions would not be penalized by a higher computational effort in synthesis. This introduces the problem statement discussed in this section: How can state exclusion requirements be enforced (more) efficiently in symbolic supervisor synthesis?

We discuss the current way requirements are enforced in CIF in Section 6.1. Then we present our modified approach in Section 6.2. Experiments comparing both approaches are presented in Section 6.3.

6.1 Current application of requirements

We first introduce the current way requirements are enforced during synthesis in CIF (before the new algorithm we discuss in Section 6.2 was implemented).

In the synthesis algorithm (Algorithm 1), first requirements are applied by Algorithm 4. In this algorithm, a safe state predicate P is computed by first taking the conjunction of all state exclusion predicates. This predicate returns true only for states for which all state exclusion predicates hold. Note that the empty conjunction is assumed true. Next, the state-event exclusion predicates are enforced by computing a safe state predicate and safe edges in Algorithm 5. When these requirements consider controllable events, the guard of each edge labeled by that event can simply be strengthened by taking the conjunction with the predicate, so that the event only occurs when the predicate holds. This is not possible for uncontrollable events, because the supervisor is not able to disallow uncontrollable events from occurring when they can occur in the plant. In that case, the safe state predicate is modified to exclude states from which the event can take place (i.e., for some edge (σ, g, u), labeled by the same event σ that the state-event exclusion predicates addresses, g evaluates to true), but the state-event exclusion predicate does not hold (i.e., J evaluates to false). The predicate $g \implies J$ specifies the states where the state-event exclusion requirement is adhered to. Finally, $X_0 \wedge P$ restricts the initial state predicate to the safe part[5].

In the following we show that, when supervisor synthesis is performed using Algorithm 4 to apply the requirements, supervisor synthesis (Algorithm 1) computes the correct result, i.e.,

[5] Strictly speaking, calculating the initial state predicate here is not necessary, and only needs to be performed at the end of synthesis (line 11 Algorithm 1). We already calculate $X_{0,S}$ here to simplify our proofs.

Algorithm 4 `applyRequirements`.

Input: Mutual state exclusion predicates SX, state-event exclusion predicates EX, edges E, initial state predicate X_0
Output: Safe state predicate P, safe edges E, safe initial state predicate X_0
1: $P = \bigwedge_{I \in SX} I$
2: $(P, E) =$ `applyEventRequirements`(P, E, EX)
3: $X_0 = X_0 \wedge P$

Algorithm 5 `applyEventRequirements`.

Input: State predicate P, edges E, state-event exclusion predicates EX
Output: Safe state predicate P, safe edges E
1: **for all** $(\sigma, g, u) \in E,\ (\sigma \Rightarrow J) \in EX$
2: **if** $\sigma \in \Sigma_c$
3: $g = g \wedge J$
4: **else**
5: $P = P \wedge (g \implies J)$
6: **end if**
7: **end for**

the maximally permissive, safe, controllable, and nonblocking supervisor. Our explanation is structured as follows: We first define a *safe state* and *safe LFA* in Definition 1. We then define when an LFA is *minimally restricted* in Definition 3, i.e., no more behavior is removed from the LFA than strictly necessary so that the requirements are satisfied. We show that after performing `applyRequirements`, we can induce an LFA that is both safe (Lemma 1) and minimally restricted (Lemma 2) with respect to the requirements. Since Algorithm 1 follows the same structure as the synthesis algorithm in Ouedraogo et al. (2011), it is known that after performing `applyRequirements` in line 1 of `SS`, the remaining lines (2-11) compute the maximally permissive, controllable, and nonblocking supervisor (Theorem 3 in Ouedraogo et al. (2011)), of the minimally restricted safe LFA. Therefore, we show in Theorem 1 that Algorithm 1 computes the maximally permissive, safe, controllable, and nonblocking supervisor.

Given a set of symbols X, let $\Phi(X)$ be the set of functions $\phi(X)$, where a function $\phi(X)$ assigns a value to each variable $x \in X$, in the domain of x. We write a function $\phi(X)$ as a predicate $\bigwedge_{x \in X} x = \phi(x)$. E.g., let's say we have variables $X = \{s, t\}$, where the domain of s is $\{1, 2\}$ and the domain of t is $\{3, 4\}$, then: $\Phi(X) = \{(s{=}1 \wedge t{=}3), (s{=}1 \wedge t{=}4), (s{=}2 \wedge t{=}3), (s{=}2 \wedge t{=}4)\}$. Given an LFA A_L with symbols X, we refer to a function $\phi(X)$ as a *state* of A_L. We may write ϕ and Φ to refer to $\phi(X)$ and $\Phi(X)$ respectively. For a predicate $P(X)$ we may write $P(\phi)$ to denote the valuation of P for state ϕ.

We say there is a transition from $\phi \in \Phi$ to $\phi' \in \Phi$, if there is some edge (σ, g, u) for which $g(\phi) \wedge u(\phi, \phi')$. We say this transition is controllable or uncontrollable, when respectively $\sigma \in \Sigma_c$ or $\sigma \in \Sigma_u$.

Definition 1 *Given LFA $A_L = (X, \Sigma, E, X_0, X_m)$, and requirements SX and EX, then a state $\phi \in \Phi$ is safe when:*

- $\forall I \in SX : I(\phi)$, *and*
- $\forall (\sigma, g, u) \in E, (\sigma \Rightarrow J) \in EX : g(\phi) \implies J(\phi)$.

We call LFA A_L safe if all its reachable states are safe. Non-safe states or automata are called unsafe.

Definition 2 *Given LFA* $A_L = (X, \Sigma, E, X_0, X_m)$*, then restricted LFA* $A_{L,R}$*, with respect to predicate* N*, edges* E_S*, and safe initial states* $X_{0,S}$*, is defined as* $A_{L,R} = (X, \Sigma, E_{S \triangleleft N}, X_{0,S}, X_m)$*, where* $E_{S \triangleleft N} = \{(\sigma, g(X) \wedge \exists_{X^+}[N(X^+) \wedge u(X, X^+)], u) | (\sigma, g, u) \in E_S\}$.

Note that in Definition 2 the guards are restricted in the same manner as in line 9 of Algorithm 1.

Definition 3 *Given LFA* $A_L = (X, \Sigma, E, X_0, X_m)$*, and requirements SX and EX, then after performing* $(N, E_S, X_{0,S}) = \texttt{applyRequirements}(SX, EX, E, X_0)$*, restricted automaton* $A_{L,R}$ *(w.r.t.* $A_L, N, E_S, X_{0,S}$*) is minimally restricted w.r.t.* A_L*, SX, and EX when:*

- *all initial states in* $\{\phi \in \Phi | X_0(\phi) \wedge \neg X_{0,S}(\phi)\}$ *are unsafe, and*
- *all restricted transitions lead to an unsafe state, or they are disallowed by some state-event exclusion predicate:* $\forall \phi, \phi' \in \Phi : (\exists(\sigma, g, u) \in E : g(\phi) \wedge u(\phi, \phi') \wedge \nexists(\sigma, g_S, u_S) \in E_S : g_S(\phi) \wedge u_S(\phi, \phi')) \implies$ *(state* ϕ' *is unsafe or* $\exists(\sigma \Rightarrow J) \in EX : \neg J(\phi)$*).*

I.e. only unsafe initial states are removed, and all transitions that are removed directly lead to an unsafe state or are disallowed by some state-event exclusion predicate.

Lemma 1 *Given LFA* $A_L = (X, \Sigma, E, X_0, X_m)$*, and requirements SX and EX, then after performing* $(N, E_S, X_{0,S}) = \texttt{applyRequirements}(SX, EX, E, X_0)$*, restricted automaton* $A_{L,R} = (X, \Sigma, E_{S \triangleleft N}, X_{0,S}, X_m)$ *(w.r.t.* $A_L, N, E_S, X_{0,S}$*) is safe w.r.t.* A_L*, SX, and EX.*

Proof For all controllable transitions, it directly follows from line 3 in Algorithm 5 that $\forall \phi \in \Phi, (\sigma, g, u) \in E_S, (\sigma \Rightarrow J) \in EX$, with$\sigma \in \Sigma_c : g(\phi) \implies J(\phi)$.

It directly follows from line 1 in Algorithm 4 that $\forall \phi \in \Phi, I \in SX : N(\phi) \implies I(\phi)$.

For all uncontrollable transitions, it directly follows from line 5 in Algorithm 5 that $\forall \phi \in \Phi, (\sigma, g, u) \in E_S, (\sigma \Rightarrow J) \in EX$, with $\sigma \in \Sigma_c : N(\phi) \implies (g(\phi) \implies J(\phi))$. Therefore, all states in $\{\phi \in \Phi | N(\phi)\}$ are safe.

From line 3 in Algorithm 4 we can conclude that all initial states $\{\phi \in \Phi | X_{0,S}(\phi)\}$ are safe. From the definition of $A_{L,R}$, and specifically $E_{S \triangleleft N}$, we can conclude that only transitions to safe states are possible. Therefore, $A_{L,R}$ is safe. □

Lemma 2 *Given LFA* $A_L = (X, \Sigma, E, X_0, X_m)$*, and requirements SX and EX, then after performing* $(N, E_S, X_{0,S}) = \texttt{applyRequirements}(SX, EX, E, X_0)$*, restricted automaton* $A_{L,R} = (X, \Sigma, E_{S \triangleleft N}, X_{0,S}, X_m)$ *(w.r.t.* $A_L, N, E_S, X_{0,S}$*) is minimally restricted w.r.t.* A_L*, SX, and EX.*

Proof For all controllable transitions, it follows from line 3 in Algorithm 5 that $\forall \phi, \phi' \in \Phi : (\exists(\sigma, g, u) \in E : \sigma \in \Sigma_c \wedge g(\phi) \wedge u(\phi, \phi') \wedge (\nexists(\sigma \Rightarrow J) \in EX : \neg J(\phi)) \implies \exists(\sigma, g_S, u_S) \in E_S : g_S(\phi) \wedge u_S(\phi, \phi')$. I.e., guards of controllable transitions in E_S are not restricted from E when there is no state-event exclusion predicate that does not disallow them.

It directly follows from line 1 in Algorithm 4 that $\forall \phi \in \Phi, I \in SX : \neg I(\phi) \implies \neg N(\phi)$.

For all uncontrollable transitions, it directly follows from line 5 in Algorithm 5 that $\forall \phi \in \Phi, (\sigma, g, u) \in \{(\sigma, g, u) \in E_S | \sigma \in \Sigma_u\}, (\sigma \Rightarrow J) \in EX : \neg(g(\phi) \implies J(\phi)) \implies \neg N(\phi)$. Therefore, all states in $\{\phi \in \Phi | \neg N(\phi)\}$ are unsafe.

From the definition of $A_{L,R}$, and specifically $E_{S \triangleleft N}$, we can conclude that only transitions in E_S to unsafe states are restricted.

Also, since all states $\{\phi \in \Phi | \neg N(\phi)\}$ are unsafe, it follows that all states in $\{\phi \in \Phi | X_0(\phi) \wedge \neg X_{0,S}(\phi)\}$ are unsafe. □

Theorem 1 *The supervisor obtained by Algorithm 1 is a maximally permissive, safe, controllable, and nonblocking supervisor for automaton A_L and requirements SX and EX.*

Proof Note that during the repeat-until loop in Algorithm 1, for all states $\phi \in \Phi$, if at the start of the loop $\neg N(\phi)$, then because BRS is restricted by N, after line 4 $\neg N(\phi)$ still holds. Also after lines 5 and 6 $\neg N(\phi)$ still holds, because BRS has $\neg N(\phi)$ as a start predicate, and only the disjunction with other predicates is taken when computing B, except for conjunction with *true*. So after line 5, we know that $B(\phi)$ holds, so after line 6 $\neg N(\phi)$ still holds. In other words, as shown in Lemma 2, all states $\phi \in \Phi$ for which $\neg N(\phi)$ after line 1 of Algorithm 1 are unsafe, and these will remain unsafe (/bad/blocking) states during the fixpoint computation. Note also that in the restriction of Definition 2, the edges are restricted in the same manner as in in line 9 of Algorithm 1.

It is shown in Theorem 3 in Ouedraogo et al. (2011) that lines 2-11 compute a maximally permissive, controllable, and nonblocking supervisor. This is a maximally permissive, controllable, and nonblocking supervisor, for the LFA that is safe (Lemma 1), and minimally restricted (Lemma 2). It follows that Algorithm 1 computes a maximally permissive, safe, controllable, and nonblocking supervisor for automaton A_L and requirements *SX* and *EX*. □

6.2 Efficient application of requirements

As stated above, it has been found empirically that synthesis on models containing state exclusion expressions was inefficient. The problem is that, when there are many state exclusion expressions, the BDD describing the safe state predicate can become quite large. This predicate is the starting point for the nonblocking predicate, which is continuously updated during synthesis. It is beneficial to keep the BDD representing this predicate as small as possible, to have low computational effort for synthesis.

Because synthesis on models containing state exclusion expressions was inefficient, they are sometimes manually converted to state-event exclusion expressions. From practice it has been found that this can solve the inefficiency problem. This is because the requirements are encoded into the guards of the edges, rather than into the nonblocking predicate. Therefore, we seek our solution in the same direction: we enforce state exclusion requirements in the same manner as state-event exclusion requirements. This is done in Algorithm 6.

In Algorithm 6, for each state exclusion predicate I and each edge (σ, g, u), a predicate J is constructed that expresses the states from which the edge can be performed such that I 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, J must hold so that a safe state is reached. In line 5 it is checked whether there are any states for which this does not hold. If that is the case, the respective edge is restricted in the same way as for state-event exclusion predicates, note that lines 6-10 in Algorithm 6 are the same as lines 2-6 in Algorithm 5. So, if the edge is labeled by a controllable event, the guard is restricted by J. If the edge is labeled by an uncontrollable event, the safe state predicate is restricted so that it only describes states where if the edge can occur, a state is reached where I holds. After enforcing all state exclusion requirements for all edges, the state-event exclusion requirements are applied in the same way as in Algorithm 4, by using Algorithm 5. Finally, the initial state predicate is

modified so that all state exclusion predicates 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 restricted guards.

We show in Example 4 how the state exclusion predicate given in Example 3 is enforced by Algorithm 6.

Example 4 *This is a continuation of* Example 3*, where a traffic light system was modeled, and requirements were specified for that model. We consider the state exclusion predicate:*

$$\neg(l_A = Green \wedge l_B = Green),$$

We consider the LightA turning green. In this case there is only one edge labeled with this event, being: edge $e_{gA} = (green\_A, l_A = Red, l_A^+ = Green \wedge l_B^+ = l_B)$*. Computing* J *as in line 4 of Algorithm 6, provides the following predicate:*

$$\begin{aligned} J &= \exists_{X^+}[(l_A = Red \wedge l_A^+ = Green \wedge l_B^+ = l_B \wedge \neg(l_A^+ = Green \wedge l_B^+ = Green)] \\ &\equiv l_A = Red \wedge \neg(l_B = Green) \\ &\equiv l_A = Red \wedge l_B = Red \end{aligned}$$

Next, we compute the $(g \wedge I \implies J)$ *as in line 5:*

$$\begin{aligned} & l_A = Red \wedge \neg(l_A = Green \wedge l_B = Green) \implies l_A = Red \wedge l_B = Red \\ \equiv\ & l_A = Red \implies l_A = Red \wedge l_B = Red \\ \equiv\ & l_A = Red \implies l_B = Red \end{aligned}$$

We can see that this does not equal true, i.e., there are states from which edge e_{gA} *can be taken such that the state exclusion predicate does not hold afterwards.*

Because green_A is a controllable event, its guard g_A *is restricted as follows:*

$$\begin{aligned} g_A &= l_A = Red \wedge l_A = Red \wedge l_B = Red \\ &\equiv l_A = Red \wedge l_B = Red \end{aligned}$$

Similarly, repeating the same steps for edge e_{gB} *that models LightB turning green, would result in the following restricted guard* g_B*:*

$$g_B = l_A = Red \wedge l_B = Red$$

Algorithm 6 `applyRequirementsEfficient`.

Input: Mutual state exclusion predicates SX, state-event exclusion predicates EX, edges E, initial states X_0
Output: Safe state predicate P, safe edges E, safe initial states X_0

1: $P = true$
2: **for all** $I \in SX$
3: **for all** $(\sigma, g, u) \in E$
4: $J(X) = \exists_{X^+}[g(X) \wedge u(X, X^+) \wedge I(X^+)]$
5: **if** $(g \wedge I \implies J) \neq true$
6: **if** $\sigma \in \Sigma_c$
7: $g = g \wedge J$
8: **else**
9: $P = P \wedge (g \implies J)$
10: **end if**
11: **end if**
12: **end for**
13: **end for**
14: $(P, E) =$ `applyEventRequirements`(P, E, EX)
15: $X_0 = X_0 \wedge P \wedge \bigwedge_{I \in SX} I$

One can verify that these guards computed by Algorithm 6 restrict the behavior in the same way as the state-event exclusion predicates provided in Example 3.

In Algorithm 1, we can substitute line 1 with the following line, to apply the introduced efficient enforcement of the requirements:

1: $(N, E_S, X_{0,S}) =$`applyRequirementsEfficient`(SX, EX, E, X_0)

We will refer to Algorithm 1 with line 1 substituted as above as `SS'`.

Same as in Section 6.1, we show that `SS'` computes the maximally permissive, safe, controllable, and nonblocking supervisor in Theorem 2. We do so by first showing that the induced restricted LFA (by Definition 2) after performing `applyRequirementsEfficient` is both safe (Lemma 3) and minimally restricted (Lemma 4) with respect to the requirements.

Lemma 3 *Given LFA $A_L = (X, \Sigma, E, X_0, X_m)$, and requirements SX and EX, then after performing $(N, E_S, X_{0,S}) =$* `applyRequirementsEfficient`(SX, EX, E, X_0)*, restricted automaton $A_{L,R} = (X, \Sigma, E_{S \triangleleft N}, X_{0,S}, X_m)$ (w.r.t. $A_L, N, E_S, X_{0,S}$) is safe w.r.t. A_L, SX, and EX.*

Proof For all controllable transitions, it directly follows from line 3 in Algorithm 5 that $\forall \phi \in \Phi, (\sigma, g, u) \in E_S, (\sigma \Rightarrow J) \in EX$, with $\sigma \in \Sigma_c : g(\phi) \implies J(\phi)$. For all uncontrollable transitions, it directly follows from line 5 in Algorithm 5 that $\forall \phi \in \Phi, (\sigma, g, u) \in E_S, (\sigma \Rightarrow J) \in EX$, with $\sigma \in \Sigma_u : N(\phi) \implies (g(\phi) \implies J(\phi))$. So, in the restricted automaton the state-event exclusion predicates are satisfied for all transitions originating from a state $\phi \in \Phi$ where $N(\phi)$ holds.

For all controllable transitions, it directly follows from line 7 in Algorithm 6 that $\forall \phi, \phi' \in \Phi, (\sigma, g, u) \in E_S$, with $\sigma \in \Sigma_c, I \in SX : g(\phi) \wedge u(\phi, \phi') \wedge I(\phi) \implies g(\phi) \wedge u(\phi, \phi') \wedge I(\phi')$. For all uncontrollable transitions, it directly follows from line 9 in Algorithm 6 that $\forall \phi, \phi' \in \Phi, (\sigma, g, u) \in E_S$, with $\sigma \in \Sigma_u, I \in SX : N(\phi) \implies (g(\phi) \wedge u(\phi, \phi') \implies I(\phi'))$. So, in the restricted automaton there are no transitions to a state that does not satisfy some state exclusion predicate.

Since for all states in $\{\phi \in \Phi | N(\phi)\}$ it is implied that no uncontrollable transition can be performed that does not satisfy a state-event exclusion predicate, all states in $\{\phi \in \Phi | N(\phi) \bigwedge_{I \in SX} I(\phi)\}$ are safe. From line 15 in Algorithm 6 we can conclude that all initial states in $\{\phi \in \Phi | X_{0,S}(\phi)\}$ are safe. Since $A_{L,R}$ only has safe initial states, and can only transition to safe states, $A_{L,R}$ is safe. □

Lemma 4 *Given LFA $A_L = (X, \Sigma, E, X_0, X_m)$, and requirements SX and EX, then after performing $(N, E_S, X_{0,S}) =$* `applyRequirementsEfficient`(SX, EX, E, X_0)*, restricted automaton $A_{L,R} = (X, \Sigma, E_{S \triangleleft N}, X_{0,S}, X_m)$ (w.r.t. $A_L, N, E_S, X_{0,S}$) is minimally restricted w.r.t. A_L, SX, and EX.*

Proof For all controllable transitions, it follows from line 3 in Algorithm 5 and line 7 in Algorithm 6 that $\forall \phi, \phi' \in \Phi : (\exists (\sigma, g, u) \in E : \sigma \in \Sigma_c \wedge g(\phi) \wedge u(\phi, \phi') \wedge (\nexists (\sigma \Rightarrow J) \in EX : \neg J(\phi)) \wedge (\nexists I \in SX : \neg I(\phi'))) \implies \exists (\sigma, g_S, u_S) \in E_S : g_S(\phi) \wedge u_S(\phi, \phi')$. I.e., guards of controllable transitions in E_S are not restricted from E when there is no state-event exclusion predicate that does not disallow them, and there is no state exclusion predicate that is not satisfied in the target state.

For all uncontrollable transitions, it directly follows from line 5 in Algorithm 5 that $\forall \phi \in \Phi, (\sigma, g, u) \in \{(\sigma, g, u) \in E_S | \sigma \in \Sigma_u\}, (\sigma \Rightarrow J) \in EX : \neg(g(\phi) \implies J(\phi)) \implies \neg N(\phi)$. For all uncontrollable transitions, it directly follows from line 9 in Algorithm 6

that $\forall \phi \in \Phi, (\sigma, g, u) \in \{(\sigma, g, u) \in E_S | \sigma \in \Sigma_u\}, I \in SX : \neg(g(\phi) \wedge I(\phi) \implies g(\phi) \wedge u(\phi, \phi') \wedge I(\phi')) \implies \neg N(\phi')$. Therefore, all states in $\{\phi \in \Phi | \neg N(\phi)\}$ are unsafe. From the definition of $A_{L,R}$, and specifically $E_{S \triangleleft N}$, we can conclude that only transitions in E_S to unsafe states are restricted.

Also, since all states $\{\phi \in \Phi | \neg N(\phi)\}$ are unsafe, it follows that all states in $\{\phi \in \Phi | X_0(\phi) \wedge \neg X_{0,S}(\phi)\}$ are unsafe. □

Theorem 2 *The supervisor obtained by Algorithm 1 is a maximally permissive, safe, controllable, and nonblocking supervisor for automaton A_L and requirements SX and EX.*

Proof The same proof as in Theorem 1 applies here. □

The authors note that not necessarily the supervisor LFAs computed by `SS` and `SS'` are the same. When `applyRequirements` is used, the guards of all edges are restricted such that they can never reach an unsafe state, also if the edge originates from another unsafe state. Since it is assumed in `applyRequirementsEfficient` that unsafe states are never reached, the guards of the edges from unsafe states are not necessarily restricted such that they can never reach another unsafe state. Regardless, the behavior under control of the supervisor, that only reaches safe states, is the same.

6.3 Experiments

Here we compare the computational effort of `SS` and `SS'`. In the set of benchmark models (Table 1), there are three models that use state exclusion expressions, which are: Lithography machine initialization, Cat and mouse tower (with 3 levels, 2 cats, and 2 mice), and Modified cat and mouse tower (with 3 levels, and at most 1 cat or 1 mouse per room). As the suggested approach only influences synthesis of these models, experiments are only performed for these models. The lithography machine initialization model contains 51 state exclusion expressions. Both cat and mouse tower models contain 15 state exclusion expressions. More details on these models can be found in Thuijsman et al. (2021).

For each model, synthesis is performed using default CIF settings, other than: DCSH+FORCE+SW is applied (as a result of Section 4) using the variable relations as introduced in Section 4.2, edge order is set to reverse-model (as a result of Section 5), and BDD measurements are turned on. The results of these experiments are shown in Table 5. For all models the efficient approach (Algorithm 6) requires less computational effort than the current approach (Algorithm 4). Generally, there is a small decrease in peak used BDD nodes, which is reduced by 8% on average. The computational benefit for BDD operation count is more significant. For these models, the BDD operation counts were decreased by 64% on average. Note that

Table 5 Experimental results efficiently enforcing requirements

Name	Peak used BDD nodes		BDD operation count	
	`SS`	`SS'`	`SS`	`SS'`
Lithography machine initialization	$4.63 \cdot 10^4$	$4.17 \cdot 10^4$	$1.68 \cdot 10^7$	$5.88 \cdot 10^6$
Cat and mouse tower	$1.82 \cdot 10^6$	$1.81 \cdot 10^6$	$1.67 \cdot 10^7$	$1.09 \cdot 10^7$
Modified cat and mouse tower	$5.68 \cdot 10^5$	$4.92 \cdot 10^5$	$3.92 \cdot 10^8$	$2.83 \cdot 10^7$

the suggested method is not necessarily always more efficient than the current method. Nevertheless, these experiments suggest that using `applyRequirementsEfficient` rather than `applyRequirements` is beneficial.

To provide further discussion on the influence of `applyRequirements` and `applyRequirementsEfficient` on the computational effort, we study the evolution of the BDD during synthesis for the two methods. For Lithography machine initialization, the BDD evolution during synthesis is displayed in Fig. 14. One can validate that the peak used BDD nodes and final BDD operation count are indeed lower when `SS'` is applied instead of `SS` (and match the values in Table 5). Performing `applyRequirementsEfficient` actually requires more BDD operations than performing `applyRequirements` (i.e., only performing line 1 of `SS'` and `SS`, not yet the remainder). Respectively, these algorithms are finished after $9.5 \cdot 10^5$ and $1.8 \cdot 10^5$ BDD operations. So, `SS'` starts later on its fixpoint computation (lines 2-7 in Algorithm 1) than `SS`. However, this computation is less costly in `SS'`, because the state exclusion predicates do not appear directly in the nonblocking predicate, which is the case for `SS`. At this point, the additional computational effort that was invested when applying the predicates is "won back" (and more) by `SS'`, leading to a lower computational effort overall. The peak that is observed at the end of both syntheses in Fig. 14 is a result of restricting the guards (lines 8-10 in Algorithm 1).

The efficiency of `applyRequirementsEfficient` likely depends on the number of edges labeled by controllable/uncontrollable events in the system. When there are many edges labeled by an uncontrollable event, the state exclusion requirements are still encoded in the nonblocking predicate. Unfortunately, at the moment we do not have any more models containing state exclusion expressions to use for further experimentation. In part, this is because they were previously avoided because of their inefficient application in synthesis.

7 Conclusion

The computational effort of symbolic supervisor synthesis can be expressed using peak used BDD nodes and BDD operation count. Unlike wall clock time and peak random access

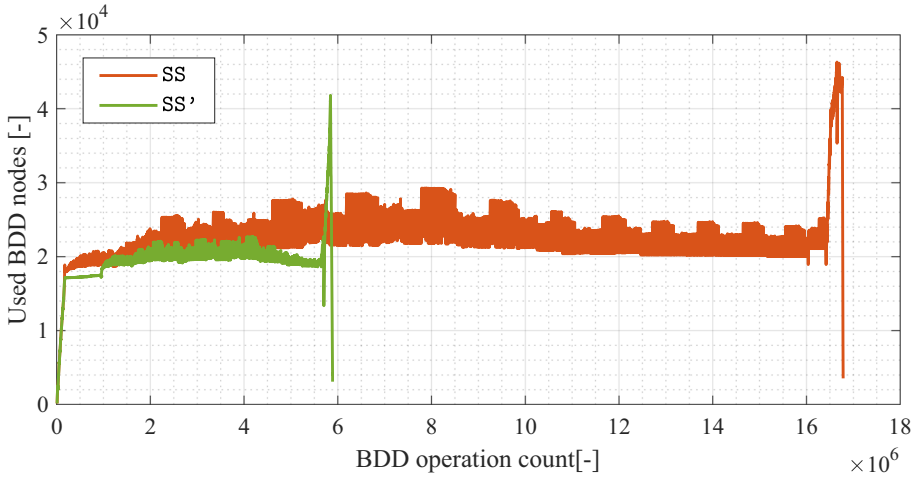


Fig. 14 BDD evolution for `SS` and `SS'` of Lithography machine initialization

memory, these BDD-based metrics are platform independent, deterministic, and include no overhead in their measurement. BDD-based metrics can be used to analyze, and improve, the efficiency of the synthesis algorithm. In this paper we showcase this approach by: introducing and analyzing DCSH, a variable ordering heuristic; analyzing several edge ordering heuristics; and introducing and analyzing an approach to efficiently enforce state exclusion requirements in synthesis. It is shown that:

1. Even though using DCSH+FORCE+SW on average requires 5% more peak used BDD nodes than FORCE+SW, it on average realizes a 14% lower BDD operation count and for 16 out of 17 models it resulted in both a lower maximal measured peak used BDD nodes and BDD operation count. Therefore, by applying DCSH+FORCE+SW as variable ordering heuristic, performing synthesis with high computational effort can be avoided, and generally low computational effort is required, relative to using just DCSH or FORCE+SW.
2. Using reverse-model edge order realizes relatively low synthesis effort, averaging 10% lower peak used BDD nodes and a 29% lower BDD operation count than using random edge orders.
3. State exclusion requirements can efficiently be enforced by restricting edge guards prior to synthesis. On average, this method reduces the peak used BDD nodes by 8% and BDD operation count by 64%, relative to the conventional method.

These methods are implemented in the ESCET toolkit, and therefore available to all those who wish to use them. Experiments like presented in this paper help in selecting what methods or settings should be used by default. For instance, starting from ESCET release v0.9, DCSH+FORCE+SW is applied by default.

From the experimental results it becomes clear that generally there are no one-size-fits-all solutions. What works for one model, does not necessarily work for another model. Unfortunately it is hard to predict when this is the case. A small change in the synthesis input, e.g., the variable order, can have a huge influence on the synthesis effort. This means that methods need to be thoroughly validated, which we do in this work. Nevertheless, these huge variances in effort also indicate how much improvement can be made. Scalability is a major factor in the industrial acceptance of supervisory control theory. This makes it worthwhile to investigate techniques like the ones discussed in this paper, to be able to keep tackling the engineering of supervisory controllers for larger and more complex systems with supervisor synthesis.

Future work

As (industrial) systems generally become more and more complex, the computational efficiency of symbolic supervisor synthesis should continuously be improved in the future. With respect to static ordering of variables or edges, more heuristics can be investigated, and their efficiency together with, or compared to, the heuristics discussed in this paper can be analyzed. Furthermore, dynamic reordering during synthesis could bring additional benefits, e.g., as considered in Panda et al. (1994) or Ranjan et al. (1995) for dynamic variable reordering and in Vahidi et al. (2006) for dynamic selection of edges. Also, other synthesis settings that directly influence the computational efficiency, such as the size of the BDD operation cache, can be evaluated and improved using BDD-based metrics. Finally, since certain methods perform well for some models, but poorly for others, it can be investigated whether these cases can be recognized prior to performing synthesis, to make a selection of methods that are likely to perform well.

Acknowledgements The authors thank all Eclipse ESCET committers and contributors for their efforts in the development of the toolkit.

Funding Research leading to these results has received funding from the EU ECSEL Joint Undertaking under grant agreement n^{o} 826452 (project Arrowhead Tools) and from the partners national programs/funding authorities.
This research is partly carried out as part of the Poka Yoka program under the responsibility of TNO-ESI in cooperation with ASML and VDL-ETG. The research activities are partly supported by the Netherlands Ministry of Economic Affairs and Climate Policy and TKI-HTSM.

Data Availability All experiments in this paper are performed using ESCET release v0.9, which is available here: https://eclipse.dev/escet/v0.9/. The models are available bundled in ESCET under "CIF Benchmarks", see https://eclipse.dev/escet/cif/examples.html on how to obtain them. The files and scripts to run the same experiments as presented in this paper are available here: https://github.com/sbthuijsman/reduce_effort.

Declarations

Conflicts of Interest The authors have no conflict of interest to declare that are relevant to this paper.

References

Akers SB (1978) Binary decision diagrams. Trans Comp 27(6):509–516. https://doi.org/10.1109/tc.1978.1675141

Aloul FA, Markov IL, Sakallah KA (2003) FORCE: a fast and easy-to-implement variable-ordering heuristic. In: Proceedings of the 13th ACM Great Lakes Symposium on VLSI. ACM Press, pp 116–11. https://doi.org/10.1145/764808.764839

Aziz A, Taşiran S, Brayton RK (1994) BDD variable ordering for interacting finite state machines. In: Proceedings of the 31st annual conference on design automation. ACM Press, pp 283–288. https://doi.org/10.1145/196244.196379

Browning T (2016) Design structure matrix extensions and innovations: a survey and new opportunities. Trans Eng Manage 63(1):27–52. https://doi.org/10.1109/tem.2015.2491283

Bryant RE (1992) Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput Surv 24(3):293–31. https://doi.org/10.1145/136035.136043

Burch JR, Clarke EM, Long DE et al (1994) Symbolic model checking for sequential circuit verification. Trans Comp-Aided Design of Integ Circ Syst 13(4):401–42. https://doi.org/10.1109/43.275352

Cabodi G, Camurati PE, Quer S (1999) Improving the efficiency of BDD-based operators by means of partitioning. Trans Comp-Aided Design of Integ Circ Syst 18(5):545–55. https://doi.org/10.1109/43.759068

Cai K, Wonham W (2014) New results on supervisor localization, with case studies. Disc Event Dyna Syst 25:203–226. https://doi.org/10.1007/s10626-014-0194-6

Cassandras CG, Lafortune S (2021) Introduction to Discrete Event Systems, 3rd edn. Springer Nature Switzerland https://doi.org/10.1007/978-3-030-72274-6

Čengić G, Åkesson K (2008) A control software development method using IEC 61499 function blocks, simulation and formal verification. In: Proceedings of the 20th IFAC World congress. Elsevier BV, pp 22–27. https://doi.org/10.3182/20080706-5-kr-1001.00003

Chaki S, Gurfinkel A (2018) BDD-based symbolic model checking. In: Handbook of model checking. Springer International Publishing, p 219–245. https://doi.org/10.1007/978-3-319-10575-8_8

Ciardo G, Siminiceanu R (2002) Using edge-valued decision diagrams for symbolic generation of shortest paths. In: Formal methods in computer-aided design. Springer Berlin Heidelberg, pp 256–273. https://doi.org/10.1007/3-540-36126-x_16

Cuthill EH, McKee J (1969) Reducing the bandwidth of sparse symmetric matrices. In: Proceedings of the 1969 24th national conference. ACM Press, pp 157–172. https://doi.org/10.1145/800195.805928

Fei Z, Miremadi S, Åkesson K et al (2013) Symbolic state-space exploration and guard generation in supervisory control theory. In: Communications in computer and information science, vol 271. Springer Berlin Heidelberg, pp 161–175. https://doi.org/10.1007/978-3-642-29966-7_11

Fei Z, Miremadi S, Åkesson K et al (2014) Efficient symbolic supervisor synthesis for extended finite automata. Trans Contr Syst Tech 22(6):2368–2375. https://doi.org/10.1109/tcst.2014.2303134

Feng L, Cai K, Wonham W (2008) A structural approach to the non-blocking supervisory control of discrete-event systems. The Int J Adv Manu Tech 41(11–12):1152–1168. https://doi.org/10.1007/s00170-008-1555-9

Flordal H, Malik R, Fabian M et al (2007) Compositional synthesis of maximally permissive supervisors using supervision equivalence. Disc Event Dyna Syst 17(4):475–504. https://doi.org/10.1007/s10626-007-0018-z

Fokkink WJ, Goorden MA, Hendriks D et al (2023) Eclipse ESCET™: the eclipse supervisory control engineering toolkit. In: Tools and algorithms for the construction and analysis of systems. Springer, p 44–52. https://doi.org/10.1007/978-3-031-30820-8_6

Forschelen STJ, van de Mortel-Fronczak JM, Su R et al (2012) Application of supervisory control theory to theme park vehicles. Discr Event Dyna Syst 22(4):511–540. https://doi.org/10.1007/s10626-012-0130-6

Goorden MA, van de Mortel-Fronczak J, Reniers MA et al (2020) Structuring multilevel discrete-event systems with dependence structure matrices. Trans Auto Contr 65(4):1625–1639. https://doi.org/10.1109/tac.2019.2928119

Knuth DE (1976) Big omicron and big omega and big theta. ACM SIGACT News 8(2):18–24. https://doi.org/10.1145/1008328.1008329

Korssen T, Dolk V, van de Mortel-Fronczak JM et al (2018) Systematic model-based design and implementation of supervisors for advanced driver assistance systems. Trans Intel Trans Syst 19(2):533–544. https://doi.org/10.1109/tits.2017.2776354

Lee CY (1959) Representation of switching circuits by binary-decision programs. The Bell Syst Tech J 38(4):985–999. https://doi.org/10.1002/j.1538-7305.1959.tb01585.x

Loose R, van der Sanden BJ, Reniers MA et al (2018) Component-wise supervisory controller synthesis in a client/server architecture. In: Proceedings of the 14th IFAC workshop on discrete event systems. Elsevier BV, pp 381–387. https://doi.org/10.1016/j.ifacol.2018.06.329

Lopes YK, Trenkwalder SM, Leal AB et al (2016) Supervisory control theory applied to swarm robotics. Swarm Intell 10(1):65–97. https://doi.org/10.1007/s11721-016-0119-0

Lousberg SAJ, Thuijsman SB, Reniers MA (2020) DSM-based variable ordering heuristic for reduced computational effort of symbolic supervisor synthesis. In: Proceedings of the 15th IFAC workshop on discrete event systems. Elsevier BV, pp 429–436. https://doi.org/10.1016/j.ifacol.2021.04.058

Ma C, Wonham W (2006) Nonblocking supervisory control of state tree structures. Transactions on Automatic Control 51(5):782–793. https://doi.org/10.1109/tac.2006.875030

Ma C, Wonham W (2008) STSLib and its application to two benchmarks. In: Proceedings of the 9th international workshop on discrete event systems. IEEE, pp 119–124. https://doi.org/10.1109/wodes.2008.4605932

Malik R, Åkesson K, Flordal H et al (2017) Supremica–an efficient tool for large-scale discrete event systems. In: Proceedings of the 20th IFAC world congress. Elsevier BV, pp 5794–5799. https://doi.org/10.1016/j.ifacol.2017.08.427

Markovski J, van Beek DA, Theunissen RJM et al (2010) A state-based framework for supervisory control synthesis and verification. In: Proceedings of the 49th IEEE conference on decision and control. IEEE, pp 3481–3486. https://doi.org/10.1109/cdc.2010.5717095

Meijer J, van de Pol JC (2016) Bandwidth and wavefront reduction for static variable ordering in symbolic reachability analysis. In: Proceedings of the 8th NASA formal methods symposium. Springer International Publishing, p 255–271. https://doi.org/10.1007/978-3-319-40648-0_20

Meinel C, Theobald T (1998) Algorithms and Data Structures in VLSI Design. Springer, Berlin Heidelberg. https://doi.org/10.1007/978-3-642-58940-9

Minato S (1996) Binary Decision Diagrams and Applications for VLSI CAD. Springer, US. https://doi.org/10.1007/978-1-4613-1303-8

Minato S (2001) Zero-suppressed BDDs and their applications. Int J Softw Tools Technol Transfer 3(2):156–170. https://doi.org/10.1007/s100090100038

Miremadi S, Lennartson B, Åkesson K (2012) A BDD-based approach for modeling plant and supervisor by extended finite automata. Trans Control Syst Technol 20(6):1421–1435. https://doi.org/10.1109/tcst.2011.2167150

Miremadi S, Lennartson B (2016) Symbolic on-the-fly synthesis in supervisory control theory. Trans Control Syst Technol 24(5):1705–1716. https://doi.org/10.1109/tcst.2015.2508978

Montgomery DC, Runger GC (2018) Applied Statistics and Probability for Engineers, 7th edn. John Wiley & Sons, Inc., https://www.wiley.com/en-us/Applied+Statistics+and+Probability+for+Engineers%2C+7th+Edition-p-9781119400363

Nadales Agut D, Reniers M (2011) Linearization of CIF through SOS. Electron Proceed Theoretical Comp Sci 64:74–88. https://doi.org/10.4204/eptcs.64.6

Ouedraogo L, Kumar R, Malik R et al (2011) Nonblocking and safe control of discrete-event systems modeled as extended finite automata. Trans Automat Sci Eng 8(3):560–569. https://doi.org/10.1109/tase.2011.2124457

Panda S, Somenzi F, Plessier BF (1994) Symmetry detection and dynamic variable ordering of decision diagrams. In: Proceedings of the 1994 IEEE/ACM international conference on computer-aided design. IEEE, p 628–631. https://doi.org/10.5555/191326.191598

Ramadge PJ, Wonham W (1987) Supervisory control of a class of discrete event processes. SIAM J Control Optim 25(1):206–230. https://doi.org/10.1137/0325013

Ramadge PJ, Wonham W (1989) The control of discrete event systems. Proc IEEE 77(1):81–98. https://doi.org/10.1109/5.21072

Ranjan RK, Aziz A, Brayton RK et al (1995) Efficient BDD algorithms for FSM synthesis and verification. In: International workshop on logic and synthesis, https://is.ifmo.ru/research/_efficient_bdd_algorithms_for_fsm_synthesis_and_verification.pdf

Reijnen FFH, Goorden MA, van de Mortel-Fronczak JM et al (2017) Supervisory control synthesis for a waterway lock. In: Proceedings of the 2017 IEEE conference on control technology and applications. IEEE, pp 1562–156. https://doi.org/10.1109/ccta.2017.8062679

Reijnen FFH, Goorden MA, van de Mortel-Fronczak JM et al (2018a) Application of dependency structure matrices and multilevel synthesis to a production line. In: Proceedings of the 2018 IEEE conference on control technology and applications. IEEE, pp 458–464. https://doi.org/10.1109/ccta.2018.8511449

Reijnen FFH, Reniers MA, van de Mortel-Fronczak JM et al (2018b) Structured synthesis of fault-tolerant supervisory controllers. In: Proceedings 10th IFAC symposium on fault detection, Supervision and Safety for Technical Processes. Elsevier BV, pp 894–901. https://doi.org/10.1016/j.ifacol.2018.09.681

Reijnen FFH, Goorden MA, van de Mortel-Fronczak JM et al (2020) Modeling for supervisor synthesis – a lock-bridge combination case study. Disc Event Dyna Syst 30(3):499–532. https://doi.org/10.1007/s10626-020-00314-0

Reniers MA, van de Mortel-Fronczak JM (2018) An engineering perspective on model-based design of supervisors. Proceedings of the 14th IFAC Workshop on Discrete Event Systems 51(7):257–264. https://doi.org/10.1016/j.ifacol.2018.06.310

Siminiceanu R, Ciardo G (2006) New metrics for static variable ordering in decision diagrams. In: Tools and algorithms for the construction and analysis of systems. Springer Berlin Heidelberg, pp 90–104. https://doi.org/10.1007/11691372_6

Sköldstam M, Åkesson K, Fabian M (2007) Modeling of discrete event systems using finite automata with variables. In: Proceedings of the 46th IEEE conference on decision and control. IEEE, pp 3387–3392. https://doi.org/10.1109/cdc.2007.4434894

Sloan SW (1989) A FORTRAN program for profile and wavefront reduction. Int J Numer Meth Eng 28(11):2651–2679. https://doi.org/10.1002/nme.1620281111

Somenzi F (1999) Binary decision diagrams. In: The VLSI handbook. CRC Press, pp 680–694. https://doi.org/10.1201/9781420049671-29

Song R, Leduc RJ (2006) Symbolic synthesis and verification of hierarchical interface-based supervisory control. In: Proceedings of the 8th IFAC workshop on discrete event systems. IEEE, pp 419–426. https://doi.org/10.1109/wodes.2006.382510

Su R, van Schuppen JH, Rooda JE (2010) Aggregative synthesis of distributed supervisors based on automaton abstraction. Trans Automat Control 55(7):1627–164. https://doi.org/10.1109/tac.2010.2042342

Theunissen RJM, Petreczky M, Schiffelers RRH et al (2014) Application of supervisory control synthesis to a patient support table of a magnetic resonance imaging scanner. Trans Automat Sci Eng 11(1):20–32. https://doi.org/10.1109/tase.2013.2279692

Thuijsman SB, Hendriks D, Theunissen RJM et al (2019) Computational effort of BDD-based supervisor synthesis of extended finite automata. In: Proceedings of the IEEE 15th international conference on automation science and engineering. IEEE, pp 486–493. https://doi.org/10.1109/coase.2019.8843327

Thuijsman SB, Reniers MA, Hendriks D (2021) Efficiently enforcing mutual state exclusion requirements in symbolic supervisor synthesis. In: Proceedings of the IEEE 17th international conference on automation science and engineering. IEEE, pp 777–783. https://doi.org/10.1109/case49439.2021.9551593

Vahidi A, Fabian M, Lennartson B (2006) Efficient supervisory synthesis of large systems. Control Eng Pract 14(10):1157–1167. https://doi.org/10.1016/j.conengprac.2006.02.013

Vos Z (2020) Efficient supervisor synthesis for feature models. Master's thesis, Eindhoven University of Technology, https://research.tue.nl/en/studentTheses/initialization-and-termination-of-flexible-manufacturing-systems

van Beek DA, Fokkink WJ, Hendriks D et al (2014) CIF 3: model-based engineering of supervisory controllers. In: Tools and algorithms for the construction and analysis of systems. Springer Berlin Heidelberg, p 575–580. https://doi.org/10.1007/978-3-642-54862-8_48

Wonham W, Cai K, Rudie K (2018) Supervisory control of discrete-event systems: a brief history. Annu Rev Control 45:250–256. https://doi.org/10.1016/j.arcontrol.2018.03.002

Wonham W, Cai K (2019) Supervisory Control of Discrete-Event Systems. Springer Int Publish. https://doi.org/10.1007/978-3-319-77452-7

Ziller R, Schneider K (2003) Reducing complexity of supervisor synthesis. Proceedings of the 2nd IFAC Conference on Control Systems Design pp 183–191. https://doi.org/10.1016/s1474-6670(17)34666-9

Sander Thuijsman has received his BSc, MSc, and PhD degree at the Mechanical Engineering department at Eindhoven University of Technology. His research interests are discrete-event systems, supervisory controller synthesis, system configuration and evolution, and computational efficiency. Currently he is bringing model-driven engineering principles into practice at Cordis. The Cordis SUITE is a low-code cross-platform solution specialized in creating or replacing machine-control software, based on graphical engineering. (photograph by Angeline Swinkels).

Dennis Hendriks is a Senior Research Fellow at TNO-ESI, a Dutch applied research center. He also has a part-time position at the department of Software Science at the Radboud University in the Netherlands. He works with both industry and academia, bringing them together to address the complexity challenges of the high-tech industry. In his applied research, he makes academic formal methods ready for industrial use. His work focuses mostly on Synthesis-Based Engineering of supervisory controllers.

Michel Reniers is currently an Associate Professor in model-based engineering of supervisory control at the Department of Mechanical Engineering at TU/e. He received both his MSc and PhD degrees in computer science form Eindhoven University of Technology in 1995 and 1999 respectively. Previously he was a postdoc at CWI (National research institute for mathematics and computer science in the Netherlands), and an Assistant Professor at both the Department of Mathematics and Computer Science and the Department of Mechanical Engineering of Eindhoven University of Technology. Dr. Reniers is coauthor of the book "Process algebra : equational theories of communicating processes" published by Cambridge University Press in 2010. He published over 200 journal and conference papers. His research portfolio ranges from model-based systems engineering and model-based validation and testing to novel approaches for supervisory control synthesis and discrete-event systems. Applications of this work are mostly in the areas of cyber-physical systems and manufacturing systems.

Dr. Reniers is currently acting as an Associate Editor for "Automata", an Associate Editor for "Discrete Event Dynamic Systems" and an Associate Editor for the "IEEE Open Journal of Control Systems". He received the best paper award of ACSD 2018 and in 2022 the "Norbert Giambiasi Award for Conceptual Modeling Excellence". (photograph by Angeline Swinkels)