Overview and Performance Evaluation of Supervisory Controller Synthesis with Eclipse ESCET v4.0

Dennis Hendriks[a,b,*,1], Michel Reniers[c], Wan Fokkink[c,d] and Wytse Oortwijn[a,1]

[a]*TNO-ESI, High Tech Campus 25, 5656 AE, Eindhoven, The Netherlands*

[b]*Radboud University, Toernooiveld 212, 6525 EC, Nijmegen, The Netherlands*

[c]*Eindhoven University of Technology, Groene Loper 5, 5612 AE, Eindhoven, The Netherlands*

[d]*Vrije Universiteit Amsterdam, De Boelelaan 1111, 1081 HV, Amsterdam, The Netherlands*

ARTICLE INFO

Keywords:
Supervisory controller synthesis
Eclipse ESCET
Benchmark models
Performance evaluation

ABSTRACT

Supervisory controllers control cyber-physical systems to ensure their correct and safe operation. Synthesis-based engineering (SBE) is an approach to largely automate their design and implementation. SBE combines model-based engineering with computer-aided design, allowing engineers to focus on 'what' the system should do (the requirements) rather than 'how' it should do it (design and implementation). A main ingredient of SBE is supervisory controller synthesis, which automatically generates correct-by-construction supervisory controllers. In the Eclipse Supervisory Control Engineering Toolkit (ESCET™) open-source project, a community of users, researchers and tool vendors jointly develop a toolkit to support the entire SBE process, particularly through the CIF modeling language and tools. In this paper, we first provide a complete description (up to a certain level of detail) of CIF's symbolic synthesis algorithm, and thereby include aspects that are often omitted in the literature, but are of great practical relevance, such as the prevention of runtime errors, handling different types of requirements, and supporting input variables (to connect to external inputs). This description assists researchers that wish to improve the algorithm. Secondly, we introduce and describe CIF's set of benchmark models, a collection of 23 industrial and academic models of various sizes and complexities, that are freely available, allowing researchers to benchmark synthesis algorithms and their improvements. Thirdly, we describe recent improvements between ESCET versions v0.8 (December 2022) and v4.0 (June 2024) that affect synthesis performance, evaluate them on our set of benchmark models, and show the current practical synthesis performance of CIF. Fourthly, we briefly look at multi-level synthesis, a non-monolithic synthesis approach, evaluate its gains, and show that while it can help to further improve synthesis performance, further performance improvements are still needed to synthesize complex models.

1. Introduction

A supervisory controller, supervisor for short, coordinates the behavior of a cyber-physical system according to discrete-event observations of the system's behavior. Based on such observations, the supervisor decides which events the system can safely perform and which events must be disabled, because they would lead to violations of requirements or to a blocking state. Engineering of supervisors is a challenging task, due to the high complexity of real-life discrete-event systems [3, 20, 21, 56].

Synthesis-based engineering (SBE) is an engineering approach to design and implement supervisors that combines model-based engineering with computer-aided design, to produce correct-by-construction controllers, by automating the engineering process as much as possible. A main ingredient of SBE is supervisory controller synthesis [47], a theory and technique for automatically deriving a model of a supervisor from discrete-event models of the uncontrolled system behavior and the system's requirements, such as functional or safety-related requirements that intend to rule out all undesired behavior. This allows engineers to focus on *what* the system should do (the requirements) rather than *how* the controller should do it (the design and implementation).

The Eclipse Supervisory Control Engineering Toolkit (ESCET™, pronounced *èsèt*) project[2] [19], provides a model-based approach and toolkit for the development of supervisors. It targets the entire model-based engineering process for the development of supervisors, including modeling, synthesis, simulation-based validation and visualization, formal verification, real-time testing and code generation. The ESCET toolkit contains multiple modeling languages and associated tools. In this paper we consider only one of them, CIF[3] [4], which features an automata-based modeling language for convenient specification of large-scale systems, and tools that support the entire SBE process.

The ESCET project, an Eclipse Foundation open-source project since 2020, builds upon decades of research and

---

*Corresponding author

✉ dennis.hendriks@tno.nl / dennis.hendriks@ru.nl (D. Hendriks); m.a.reniers@tue.nl (M. Reniers); w.j.fokkink@vu.nl (W. Fokkink); wytse.oortwijn@tno.nl (W. Oortwijn)

ORCID(s): 0000-0002-9886-7918 (D. Hendriks); 0000-0002-9283-4074 (M. Reniers); 0000-0001-7443-8978 (W. Fokkink); 0000-0002-5244-2519 (W. Oortwijn)

[2]See `https://eclipse.dev/escet`. 'Eclipse', 'Eclipse ESCET' and 'ESCET' are trademarks of Eclipse Foundation, Inc.

[3]See `https://eclipse.dev/escet/cif`.

---

tool development at the Eindhoven University of Technology. Vital for the evolvement from an academic into an industrially applicable toolkit are the various years-long research collaborations with industry, such as with Rijkswaterstaat [21, 49, 51], ASML [56, 65, 68] and Philips [61, 62]. Rijkswaterstaat, part of the Dutch Ministry of Infrastructure and Water Management, is responsible for infrastructure in the Netherlands, including roads, bridges, tunnels and waterway locks. ASML is an innovation leader in the semiconductor industry, providing chipmakers with all they need to mass produce patterns on silicon through lithography. Philips is a leading company in healthcare technology, and provides among others medical equipment such as MRI and X-ray machines. The quality of supervisory control software for such systems impacts their availability, reliability and safety. Synthesis-based engineering allows for automation, modularization and standardization, increasing quality and evolvability, and decreasing life-cycle costs [3, 62].

Supported by the Eclipse Foundation's principles of transparency, openness, meritocracy and vendor-neutrality, the ESCET project aims to be an open environment and a growing community. It allows interested parties, such as academic and applied research institutes, industrial partners and tool vendors, to collaborate on and profit from further tool development for the model-based construction of supervisors. Furthermore, the project's open nature allows any vendor to develop commercial tool support.

This paper has four main contributions:

1. We provide a complete description (up to a certain level of detail) of CIF's symbolic synthesis algorithm, and thereby include aspects that are often omitted in the literature, but are of great practical relevance, such as the prevention of runtime errors, handling different types of requirements, and supporting input variables (to connect to external inputs). This description assists researchers that wish to improve the algorithm.

2. We introduce and describe CIF's set of benchmark models, a collection of 23 industrial and academic models of various sizes and complexities, that are freely available, allowing researchers to benchmark synthesis algorithms and their improvements.

3. We describe recent improvements between ESCET versions v0.8 (December 2022) and v4.0 (June 2024) that affect synthesis performance, evaluate them on our set of benchmark models, and show the current practical synthesis performance of CIF.

4. We briefly look at multi-level synthesis, a non-monolithic synthesis approach, evaluate its gains, and show that while it can help to further improve synthesis performance, further performance improvements are still needed to synthesize complex models.

The artifact accompanying this paper includes the models and scripts that can be used to reproduce various results from the paper [26].

We provide background information on supervisory controller synthesis and the SBE process in Section 2, before

we introduce CIF's symbolic synthesis algorithm in Section 3. We then describe the recent improvements in Section 4, introduce the benchmarks in Section 5, and experimentally evaluate the improvements on these benchmarks in Section 6. We further look at non-monolithic synthesis, and multi-level synthesis in particular, in Section 7, before we conclude in Section 8.

2. Background

2.1. Supervisory controller synthesis

Figure 1 depicts the general system structure for supervisory control. A cyber-physical system consists of mechanical components to be controlled, such as robot arms, conveyor belts, and so on. Actuators drive their operation, while sensors indicate their status. Resource control provides low-level control, often offering more abstract actuator and sensor signals for higher levels of control to use. Supervisors ensure actuator signals that would violate requirements are disabled. Large systems may be divided into (layers of) subsystems, and supervisors can be present at each level, coordinating lower-level subsystems (only a single layer is depicted). A (sub)system is often controlled by a human operator through a graphical user interface, or part of a larger system to which it is connected by an interface. The part of the system that is controlled by the supervisor, so the environment of the supervisor, is called the *plant* or *uncontrolled system*. In the figure it is the part of the system enclosed within the dashed line. This may include higher layers, as for instance warning lights of the GUI may need to be controlled. The uncontrolled system and supervisor are together called the *controlled system*.

Supervisory controller synthesis [47, 72] automatically generates a correct-by-construction supervisor model for
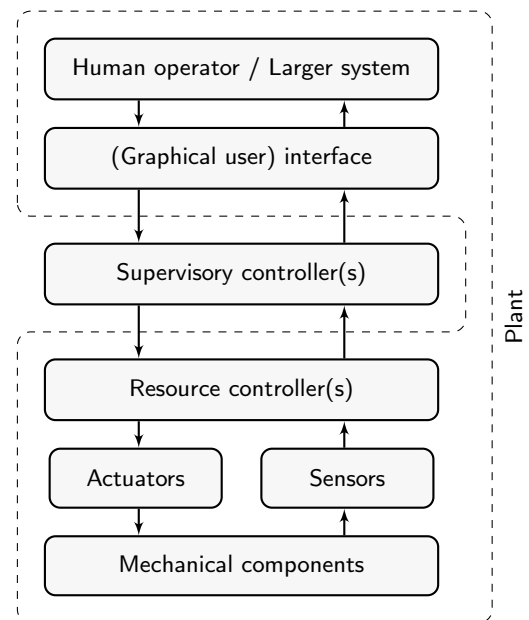
Figure 1: Structure of supervisory control. Part enclosed with dashed line is the 'plant'. Adapted from a similar figure in [19].

a discrete-event system, given precise descriptions of the behavior of the plant components as well as the (safety) requirements for the desired system behavior. These can be specified conveniently as extended finite automata (EFAs), i.e., automata that model the states of a system and the transitions between these states, extended with variables, guards and updates that allow to model states as data, and invariants that restrict the reachable or allowed states [38, 58].

Synthesis considers the parallel composition of the plant automata, specifying which behavior is physically possible, together with the requirement automata, specifying which behavior is allowed. That is, these automata synchronize on shared events, meaning these events must be executed simultaneously by all participating automata. If an event is missing in the local state of any plant automaton, or is restricted by a plant invariant, it is absent from the overall system state and is considered physically impossible. If, on the other hand, an event is missing only in the states of requirement automata, or is restricted by a requirement invariant, it is physically possible but must be disabled by the synthesized supervisor to ensure *safety*.

Controllable events (such as output signals to actuators) can be prevented by a supervisor, but uncontrollable events (such as input signals from sensors) cannot. To ensure *controllability*, if an uncontrollable event must be prevented, the supervisor makes the system state where it occurs unreachable by disabling all controllable events leading to it. Moreover, if an uncontrollable event leads to such a state, the origin state of this event must be made unreachable too.

If safety of, for instance, a drawbridge is ensured by forcing it to remain raised forever, it is useless for road traffic. Therefore states of the plant and requirement EFAs can be marked, for instance states where the bridge deck is lowered, the barriers are open, and the signals are green. A marked state in the parallel composition means all individual plant components are in a marked local state, in this case allowing traffic to proceed over the bridge. The supervisor must guarantee that the plant can always reach a marked state, by disabling (events leading to) states that violate this property. Only then is the controlled system *nonblocking*.

Supervisory controller synthesis ensures *safety*, *controllability* and *nonblockingness* of a system with respect to its requirements, accounting for all possible behavior, also disabling events that lead to problems such as blocking behavior or requirement violations much later in the system's execution. It does so by restricting as little behavior as possible, thus ensuring *maximal permissiveness*.

The use of EFAs also allows to extract and represent the synthesized supervisor compactly and intuitively [41, 42]. CIF represents supervisor models as the collection of the provided plant and requirement models together with the addition of a single supervisor EFA. This supervisor EFA has per controllable event an additional guard, which represents the extra restrictions that the supervisor imposes in order to ensure safety, controllability and nonblockingness.

2.2. Synthesis-based engineering process

The CIF language and tools, as part of the ESCET toolkit, together enable a synthesis-based engineering approach and support its entire process. The CIF language is a declarative modeling language for the specification of discrete-event, timed, and hybrid systems as a collection of synchronizing automata. It can be used to specify various different models during the entire development process, including plants and requirements models as well as simulation models, as it has a rich set of concepts. This prevents having to use multiple languages during the engineering process.
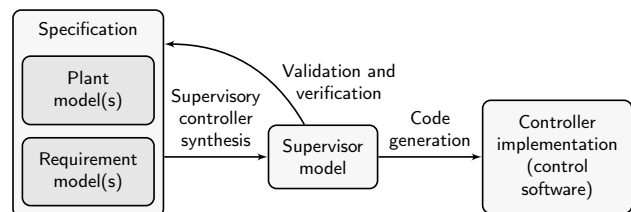
The CIF tools support the various steps of the synthesis-based development process, including among others specification, supervisory controller synthesis, simulation-based validation and visualization, verification, real-time testing, and code generation. The toolkit has a strong focus on industrial application, with, e.g., modeling convenience, efficient algorithms, and checks for common mistakes, as well as extensive documentation, tutorials and examples. Some CIF tools only support a subset of the CIF language concepts.

Next to the ESCET toolkit, other supervisory controller synthesis tools include DESTool [43], DESUMA [54], Supremica [36] and TCT [17]. A comparison between these tools is provided in [53].

We briefly introduce CIF's synthesis-based engineering process and associated tools[4], as shown in simplified form in Figure 2. It starts with a model-based specification, consisting of plant and requirement models, modeled as EFAs with invariants. To these models, supervisory controller synthesis is applied, resulting in a model of the supervisor. The ESCET toolkit supports synthesis both with its own synthesis tools, and by a transformation to Supremica, for a subset of CIF models.

Synthesis ensures that all specified requirements are satisfied by the synthesized supervisor, and thus the requirements are correctly enforced. Validation is supported by CIF's automated or interactive simulation and visualization. It helps to determine whether the specified plants and requirements, and by extension the supervisor, achieve the desired system behavior. It allows to check that the plants properly represent the possible behavior of the system as understood by the engineers, and that the specified requirements are the intended requirements and lead to the desired behavior. Verification,

---

[4]See for more detailed information the CIF website: `https://eclipse.dev/escet/cif/synthesis-based-engineering/` and `https://eclipse.dev/escet/cif/tools`.

Figure 2: Simplified representation of ESCET's synthesis-based engineering process.

such as model checking, supported through transformations to UPPAAL [1, 6] and mCRL2 [7, 52], can also be used to spot flaws in the plant and requirement models. It can moreover be employed to check other requirements not yet supported by synthesis, including stronger liveness guarantees, timing properties, and some additional properties that must hold for the supervisor to be properly implemented as a controller [50].

An implementation of the controller can then be obtained automatically from a model of the supervisor, by generating code for its control software. The CIF tools support code generation for multiple languages and platforms, including Java, JavaScript, C, Simulink, and PLC code (IEC standard 61131-3) for multiple vendors.

3. Symbolic supervisory controller synthesis

The 'data-based synthesis tool' is CIF's primary synthesis tool. It is based on the symbolic supervisory controller synthesis algorithm of Ouedraogo et al. [45]. It iteratively strengthens guard predicates on transitions so that forbidden states become unreachable in the controlled system. The use of a symbolic algorithm prevents that all states have to be explicitly represented, which would be infeasible for the kind of real-world systems we consider in Section 5 [10]. Symbolic synthesis therefore represents a major step forward for the industrial applicability of supervisory controller synthesis, by allowing for efficient synthesis of plants and requirements intuitively modeled as EFAs [15], as we will also show in Section 6.

In this section we provide a description of CIF's symbolic synthesis algorithm. Most parts have been described in the literature before, and we include these parts in some detail, into a single description of the synthesis algorithm. Furthermore, we also include aspects that are often omitted in the literature [45, 65], but are of great practical relevance, such as the prevention of runtime errors, handling different types of requirements, and supporting input variables (to connect to external inputs). This description can assist researchers to better understand the algorithm, such that they can improve upon it. We leave out most optional parts, that can be configured through the synthesis tool's many options.

Figure 3 shows an overview of the steps involved in CIF's symbolic synthesis tool. A user makes a model with the plants and requirements, making use of CIF's various modeling concepts, and some of the extended concepts are eliminated to core concepts, to simplify subsequent steps, as described in Section 3.1. In the model with core concepts, to further reduce the number of concepts, the requirement automata – but not the requirement invariants – are 'plantified' to plant automata and state/event exclusion requirements, as described in Section 3.2. The (partially) plantified model is linearized, to eliminate synchronization between automata, which results in a linearized CIF model, with an automata with a single location and self-loop edges, as described in Section 3.3. The linearized model is then converted to a Symbolic EFA (SEFA). We explain the symbolic encoding
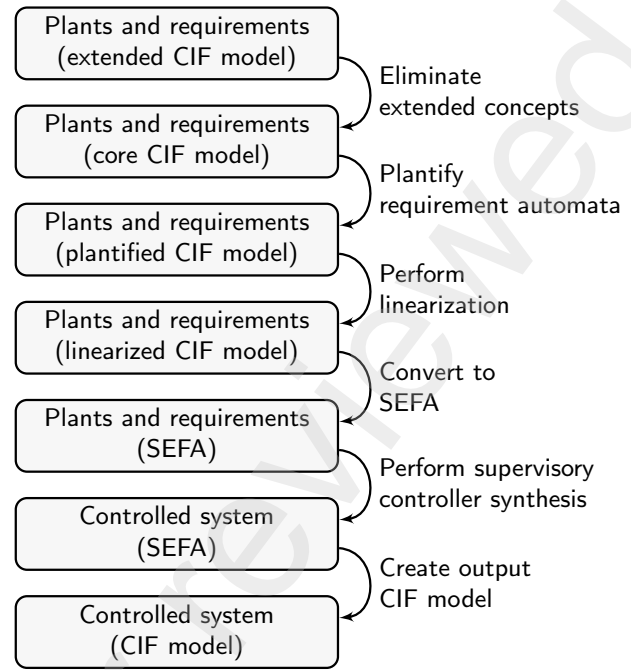
Figure 3: Overview of the steps involved in CIF's symbolic supervisory controller synthesis tool.

of predicates using Binary Decision Diagrams (BDDs) in Section 3.4, introduce SEFAs in Section 3.5, discuss the computation of SEFA transitions in Section 3.6, and detail the conversion from linearized models to SEFAs in Section 3.7. The SEFA is used to perform synthesis, which results in a SEFA of the controlled (or supervised) system, as described in Section 3.8. Finally, the output CIF model is constructed from the controlled-system SEFA, as described in Section 3.9.

3.1. CIF models

We start at the beginning, with the plants and requirements model that an engineer makes, and that forms the input to the synthesis process. We discuss what concepts are used in CIF to model these plants and requirements, omitting concepts that are not supported by synthesis.

Automata: CIF allows modeling the system as extended finite automata (EFAs). An *automaton* consists of one or more *locations*, and is always in exactly one location at any time, its *current* location. An automaton further has directed *edges* between the locations. Each edge has a *source* location and a *target* location. If an edge has the same source and target location, it is called a *self-loop* edge. Edges are labeled with *events*. A single edge labeled with multiple events is semantically equivalent to multiple edges between the same source and target location, each with one of the events.

Discrete variables: Automata may optionally declare *discrete variables*, often simply called 'variables'. Each variable has an associated finite data type. CIF has many data types, but for synthesis only booleans, integers (ranged non-negative numbers) and enumerations are supported.

States: A *state* of a CIF automaton consists of its current location and a current value for each of its variables. A state

of a CIF model consists of the combined state of each of its automata.

Guards: An edge may have a *guard* that indicates a restriction on when the edge may be taken. If no guard is given, it is implicitly *true*. A guard may refer to any variable in the model, so not only of the automaton in which it is specified. Guards are evaluated over the state before the edge is taken.

Updates: An edge may also have any number of *updates* in the form of assignments that give new values to variables. For instance, an *assignment* $x := 5$ assigns value 5 to x, and $x := x + 1$ increments the value of x by one. CIF has the 'global read, local write' principle, meaning that all automata may read the values of all variables of the system, in for instance guards and the right hand sides of updates, but only the automaton in which the variable is declared may assign it new values.

Initialization: CIF allows any number of locations of an automaton to be indicated as *initial* locations. A location may also be conditionally initial, by indicating an initialization predicate for the location, for instance depending on the initial values of variables, or the initial locations of other automata. Each variable is given one or more potential initial values. The initial locations and initial values of variables can be further restricted by initialization predicates specified in components (inside and outside automata). A state is thus initial if each automaton is in an initial location, each variable has one of its possible initial values, and the state satisfies all initialization predicates.

Marking: CIF similarly allows any number of locations of an automaton to be *marked*, and they may also be conditionally marked. Marker predicates specified in components may further restrict the marking of both locations and variables. A state is thus marked if each automaton is in a marked location and the state satisfies all marker predicates.

Transitions: *Transitions* between states are possible by taking edges of automata. If multiple automata share an event, the automata *synchronize* on that event. That is, they must all take an edge for that event, at the same time, to perform a transition to a new state. In other words, each synchronizing automaton must have an edge for that event in their current location, and the guards of those edges must hold in the current state. Then each automaton takes one edge for the event, and they all go to the target locations of their own edges. Furthermore, the assignments of all the edges are then executed to determine the new state that is reached. The right hand sides of the assignments are all evaluated in the source state of the transition, making it irrelevant in which order the assignments are executed.

Alphabets: The *alphabet* of an automaton defines the events over which the automaton synchronizes. It can be explicitly specified, and otherwise it is implicitly equal to all the events that are are used on the edges of the automaton.

Runtime errors: For a CIF variable with an integer *range* $[0, 5]$, an assignment $x := x + 1$ for variable x could lead to a *runtime error*. If $x = 5$, then after the assignment $x = 6$, which is outside the variable's range, causing a runtime error.

Synthesis in CIF has implicit requirements for all variables that they must always be within their range, and implicit requirements for all assignments that they must not result in runtime errors. Synthesis thus enforces that variables are kept within their ranges.

Invariants: CIF furthermore supports two types of *invariants*: *state invariants* and *state/event exclusion invariants*. A state invariant indicates a predicate that must always hold, in all states. State/event exclusion invariants indicate a predicate that must hold, for a certain event to be allowed or disallowed. They thus restrict the states from which the event can be taken (plant invariant) or is allowed to be taken (requirement invariant). For instance, 'e needs $x = 1$' is a state/event exclusion invariant that indicates that event e may only occur in states where $x = 1$ holds. And '$x \neq 1$ disables e' indicates that event e may not occur in states where $x \neq 1$ holds. These two state/event exclusion invariants are equivalent, and the one can be rewritten into the other, and vice versa.

Plants: For synthesis, all automata and invariants are labeled as being part of the *plant* or being a *requirement*. *Plant EFAs* indicate the possible behavior of the uncontrolled system. Plant invariants restrict this possible behavior: *state plant invariants* indicate impossible states, and *state/event exclusion invariants* indicate that certain events are impossible in certain states. For instance, an elevator could have two limit sensors that indicate whether it is at the bottom or the top. A state plant invariant could indicate that it is impossible for both sensors to be on at the same time, as the elevator can't be both at the bottom and the top. Alternatively, a state/event exclusion invariant could indicate that it is impossible for the top sensor to go on when the elevator is at the bottom.

Requirements: Requirements indicate constraints on the possible system behavior. *Requirement EFAs* are especially useful to indicate allowed orders of events, such as the way products must flow through a system. *State requirement invariants* indicate states that may never be reached, such as two robots that are at the same position and would collide, leading to damage. *State/event exclusion requirement invariants* indicate that certain events may not be performed in certain states, such as opening a bridge when the traffic light is green and road traffic may thus still be driving over the bridge.

Input variables: CIF also features *input variables*, which represent external values. For synthesis, they must also have a finite data type, and their supported types are the same as for discrete variables. Input variables can be read globally, in guards, right hand sides of updates, initialization predicates, marker predicates, invariants, and so on, like for discrete variables. But, input variables can not be assigned a value. Instead, their value is determined by the environment of the model, and could change at any time. They could for instance be connected to input ports of the system, such as input ports of a PLC.

Extensions: CIF has various other language features that make it easier to model, such as *constants* (variables with a fixed value), *algebraic variables* (variables declaratively defined as an expression over the system's state), *groups* to

---

group together automata, *imports* to allow reusing partial specifications, parameterized *component and group definitions* that can be instantiated any number of times, and so on. These concepts can be seen as syntactic sugar, extended concepts that can be eliminated into core concepts: automaton definitions can be instantiated to automata, constants and algebraic variables can be inlined, and so on. Eliminating the extended concepts preserves the semantics of the model, while easing subsequent steps.

3.2. Plantification of requirement automata

Requirement automata are not taken into account directly during linearization, but are 'plantified' to reduce the number of concepts to consider during synthesis. They are converted to plant automata and state/event requirement invariants. Note that in this step we only partially plantify the requirements, namely only the requirement automata. The requirement invariants are not plantified (we come back to this later).

More concretely, each requirement automaton is relabeled as a plant automaton that observes the system state, without restricting it. To ensure the automaton imposes no restrictions, for each event in the requirement automaton's alphabet, for each location without edges for that event, a self-loop edge is added to ensure the event is not restricted in that location. If a location already has one or more edges for the event, a self-loop edge is added with the negation of the union of the guards of the existing edges for the event in that location. This way, the plantified requirement automaton does not restrict the behavior of the system, and can be considered as a plant automaton.

To ensure that the restrictions of the requirement automaton are still taken into account during synthesis, for each added edge a state/event exclusion invariant is added that imposes the same restriction as its guard.

For instance, consider an event e and two edges for that event in a location l of a requirement automaton, with guards g_1 and g_2. A self-loop edge is added to l with event e and guard $\neg (g_1 \vee g_2)$. Furthermore, a state/event exclusion requirement '$\neg (g_1 \vee g_2)$ disables e' is added to the model.

3.3. Linearization

The plantified CIF model may have many EFAs, which synchronize on shared events. CIF's symbolic synthesis algorithm linearizes the CIF model, eliminating the parallel composition between the different EFAs, resulting in a single automaton with one location and only self-loop edges [44].

As part of this transformation, each EFA gets a location pointer variable, a discrete variable that indicates its current location. Each edge is additionally guarded with a predicate that indicates the location pointer variable being equal to the value for its source location. Each non-self-loop edge also gets an additional update that assigns the value matching the target location to the location pointer variable.

Parallel composition is then eliminated, by for each event combining the possible edges for that event from each synchronizing EFA, in all possible combinations. For instance, if in one EFA there are three possible edges for the

event, and in another there are 4, then combined they get $3 * 4 = 12$ linearized edges.

The resulting single EFA thus has a single location and only self-loop edges. The locations are then no longer relevant, as the state of the system is then captured solely by the variables. Internally, symbolic synthesis represents the model entirely as predicates over variables.

Figure 4 shows two EFAs (4a and 4b) and their corresponding linearized EFA (4c). Locations are indicated by circles. Marked locations have a double border. Initial locations have an unconnected incoming arrow, with before it the initialization predicates and possible initial values of the variables. A conditionally marked location is indicated by a predicate after **marked**, next to the location. Marker predicates of components (outside of locations) are shown similarly. Edges are indicated by arrows from circle to circle. Edges are labeled with the event, and optionally a guard after **when** and updates after **do**.

In the linearized EFA several predicates are combined. The initialization predicate of each location is made conditional on its corresponding value of the location pointer variable, and they are combined for the various locations of the automaton through disjunction (the example does not feature such conditional initial locations). Similarly, marker predicates for the locations are combined per automaton. And invariants, which may in CIF also be written inside locations, are also made conditional on being in the locations.

After linearization, there is a single EFA, with various linearized self-loop edges, initialization predicates, marker predicates, and invariants (the example has none).

3.4. Binary decision diagrams

CIF's symbolic synthesis algorithm works on a symbolic representation of the linearized model, represented entirely as predicates over variables, which is introduced in the next section. But first, we introduce the Binary Decision Diagram (BDD) data structure, which CIF uses to symbolically represent the predicates representing (parts of) state spaces, and to perform computations on such predicates during synthesis (see Section 3.8). BDDs are not only compact in terms of storage, but also allow for efficient manipulation [39].

BDDs are rooted, directed, acyclic graphs, with decision and terminal nodes. There are two terminal nodes, one representing *true* and one representing *false*. Each decision node is labeled with a boolean variable v, and has two outgoing edges to other nodes, one edge for when the variable is *true* and one edge for when the variable is *false*. To evaluate a BDD for specific values of the variables, start at the root node, evaluate its variable, take the relevant edge, evaluate the variable of the next node that is reached, take the relevant edge, and so on, until a terminal node is reached, which indicates the resulting truth value of the predicate for the given values of the variables.

In our work, we use *reduced ordered* BDDs (ROBDDs), which we will simply call BDDs from now on. The variables are totally ordered using ordering relation $<$, such that for two variables $v_1 < v_2$, variable v_1 is placed closer to the root than
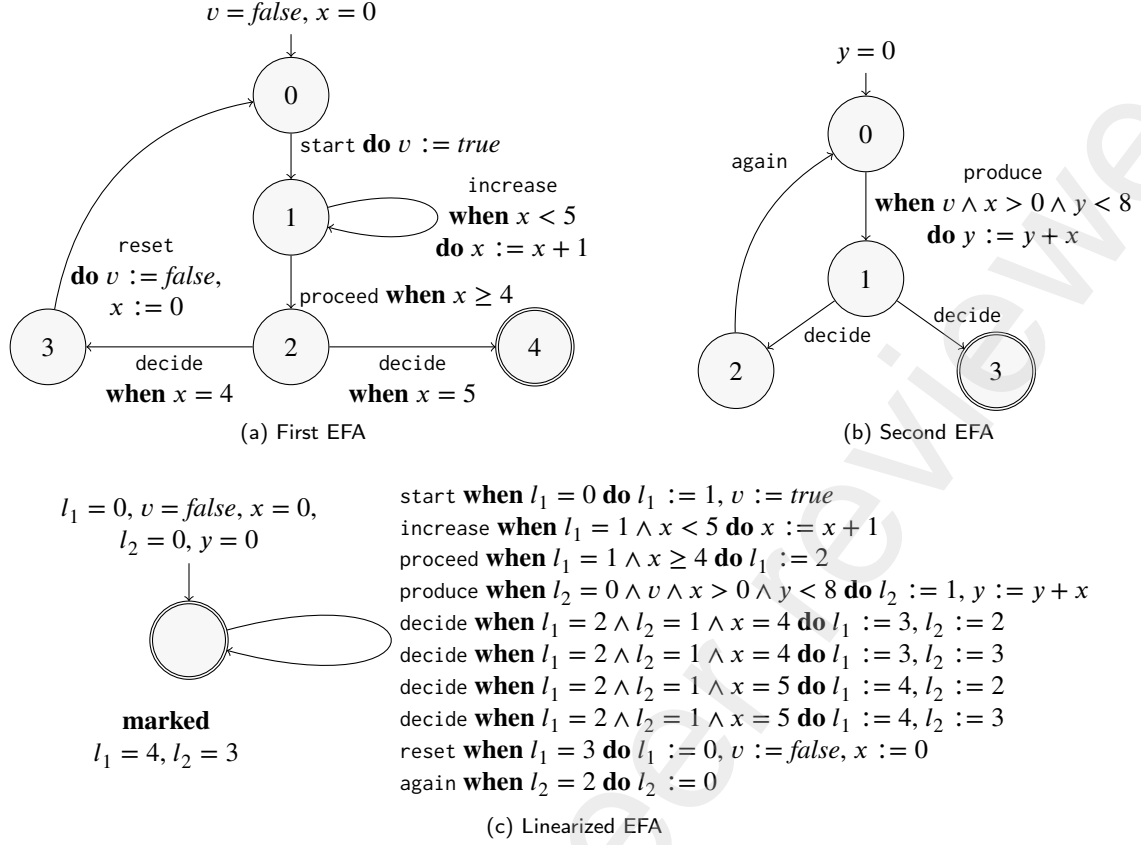
**Figure 4:** Example of two EFAs and their linearized form.

v_2 in BDDs. And the BDDs are reduced, meaning they are always represented using a minimal number of nodes (with respect to a variable order), reusing existing nodes (and their subtrees towards the terminal nodes) whenever possible. With these two restrictions, a BDD always has a unique minimal representation, its canonical representation.

Figure 5 shows two BDDs for the same predicate over the same variables, but with two different variable orders. Decision nodes are depicted as circles, labeled with boolean variables. Terminal nodes are depicted as squares, labeled with F for *false* and T for *true*. The root node is shown at the top, edges go down and are shown as arrows. *False* edges are depicted as dotted lines, *true* edges as solid lines.

The size of a BDD is its number of decision nodes. In this case, the same predicate can thus be represented with BDDs of sizes 4 (left) and 6 (right). The variable order thus has an influence on the size of BDDs. Since operations on BDDs are defined recursively over their structure, their computational effort is predominantly determined by the sizes of the BDDs on which they operate. The variable order thus has a significant impact on both the amount of memory used to represent BDDs, and the time it takes to manipulate them.

For synthesis, CIF supports variables with boolean, ranged integer and enumeration data types. To work with BDDs, each variable is represented using one or more boolean variables, its BDD variables. For instance, a variable x with an integer type with range $[0, 5]$ can have six different values, which requires three boolean variables (x_2, x_1 and x_0), since $\lceil log_2(6) \rceil = 3$. Three variables can represent $2^3 = 8$ different values, of which in this case then only six are used. If $x_2 = false$, $x_1 = true$ and $x_0 = true$, x has value $4*0 + 2*1 + 1*1 = 3$. For each CIF variable v, also a variable v^+ is created, that allows representing the new value of the variable in update predicates (more on that in the next section). The current and new value boolean variables are kept together in the variable order, and they are interleaved.
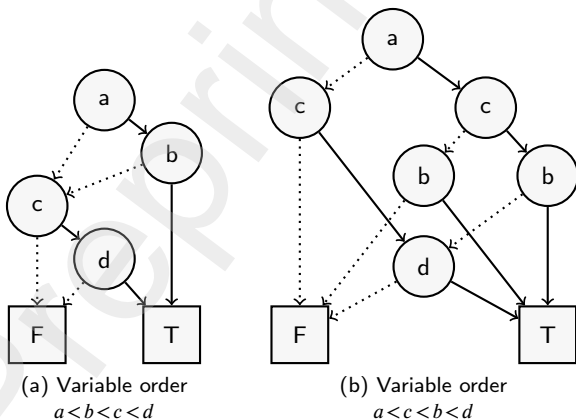
Figure 5: Two BDDs representing $(a \wedge b) \vee (c \wedge d)$ for different variable orders.

For instance, for CIF variable x, the boolean variables are in the following order: $x_0 < x_0^+ < x_1 < x_1^+ < x_2 < x_2^+$.

3.5. Symbolic EFAs

As mentioned, CIF's symbolic synthesis algorithm works on a symbolic representation of the linearized model, represented entirely as predicates over variables. We therefore introduce the concept of a *symbolic EFA*, or *SEFA*. A SEFA is a 6-tuple $(V, D, \Sigma, E, p_0, p_m)$, with V its finite set of *variables*, D for each variable its finite domain of possible values, Σ its finite set of events called the *alphabet*, E its finite set of *edges* (the *transition relations*), p_0 a predicate denoting the *initial* states, and p_m a predicate denoting the *marked* states. We may sometimes refer to a SEFA as EFA, if the difference is not relevant.

Events: Alphabet Σ is partitioned into two disjoint sets: Σ_c with the controllable events, Σ_u the uncontrollable ones.

Variables: V contains $n \geq 0$ variables. Each variable v_i, $1 \leq i \leq n$, has a value from its finite *domain* of possible data values D_i. D is a function that for each variable v_i gives it domain D_i, i.e., $D(v_i) = D_i$. The set of possible states $X = D_1 \times D_2 \times \ldots \times D_n$ then gives all possible combinations of values for all of the variables, while a single state $x \in X$ is a tuple, a *valuation* that associates with each variable v_i a single value from D_i.

Predicates: A *predicate* p is a boolean expression over variables V, such as $a \wedge \neg b$, $(c + 5) > (d \bmod 4)$, and so on. Given a valuation $x \in X$, the variables $v \in V$ in a predicate p can be replaced by their values in x, which we write as $p(x)$. The resulting predicate is a boolean expression without variables, and thus its single truth value can be computed. If it is *true*, state x is part of the set of states represented by p, while if it is *false*, it is not part of the set of states. A predicate p thus symbolically represents a set of states $\{x \in X \mid p(x)\}$. p_0 is the predicate that indicates the states in which the system may initially start, and p_m is the predicate that indicates which states are marked.

Edges: Each edge $(g, r, \sigma, u) \in E$ consists of a guard predicate g, an error predicate r, an event $\sigma \in \Sigma$, and an update u. Sets $E_c, E_u \subseteq E$ represent a partitioning of E based on the events of the edges, such that E_c contains all edges for controllable events in Σ_c, and E_u contains all edges for uncontrollable events in Σ_u. Guard predicate g indicates for which states the edge is enabled, and thus a transition is possible (typically to another state).

Updates: We introduce for each variable $v \in V$ its counterpart $v^+ \in V^+$, with $V^+ = \{v^+ \mid v \in V\}$, representing the new values of variables after a transition. We furthermore define $V^* = V \cup V^+$, the set of all old and new state variables. Update predicate u over variables V^* then indicates how the variables can be updated by taking the edge, as it indicates which old and new value combinations are possible for source and target states of transitions for the edge. For instance, an assignment $a := a + 1$, which increments a by one, can be represented as update predicate $a^+ = a + 1$, with $a^+ \in V^+$ and $a \in V$. It allows all old and new value combinations

where the new value of variable a is one higher than its old value.

Partial transition relations: In the literature on supervisor synthesis, it is not uncommon to add for each variable x on an edge that is not assigned a value, an update predicate $x^+ = x$ to u, to ensure the variable remains unchanged [65, 66]. We use partial transition relations, and thus do not need these extra restrictions, which keeps the update relations smaller. We come back to this in Section 3.6.

Error predicates: Error predicate r indicates for which states the update u goes outside the BDD-representable range of the variable. For an integer variable x with a BDD-representable range $[x_{min}, x_{max}]$ and assignment $x := e$, with e some expression, r is $e < x_{min} \vee e > x_{max}$. Note that for technical reasons, r only keeps variables within their BDD-representable ranges. r does not yet ensure that variables remain within their original CIF variable ranges; we come back to that later in Section 3.7.

For instance, consider a variable y with CIF variable range $[0, 5]$, and update $y^+ = y + 1$. y is represented using 3 BDD variables, with BDD-representable range $[0, 7]$. r is then $y + 1 < 0 \vee y + 1 > 7$, since after the update $y + 1$ is the new value of y and if that is smaller than 0 or larger than 7, then it can not be represented using the 3 BDD variables. In this particular case, y has BDD-representable range $[0, 7]$, so $y + 1$ is in range $[1, 8]$, which means $y + 1 < 0$ is *false*, and thus only $y + 1 > 7$ is relevant. Error predicate r thus keeps y within BDD-representable range $[0, 7]$, but not yet within CIF range $[0, 5]$; we come back to that later in Section 3.7.

Auxiliary definitions: We also introduce function $vars(p)$ that given a predicate p over V or V^* gives the variables in V that occur in p. Similarly, $vars^+(p)$ over V^+ or V^* gives the variables in V^+ that occur in p. Finally, given a predicate p over variables from V, a predicate $p[V := V^+]$ over variables from V^+ is obtained by replacing each variable $v \in V$ by its counterpart $v^+ \in V^+$.

As an example, the SEFA for the linearized EFA of Figure 4c is:

$$V = \{l_1, v, x, l_2, y\}$$
$$D = l_1 \to \mathbb{N}_{0..4}, v \to \mathbb{B}, x \to \mathbb{N}_{0..5}, l_2 \to \mathbb{N}_{0..3}, y \to \mathbb{N}_{0..12}$$
$$\Sigma = \{\text{start}, \text{increase}, \text{proceed}, \text{decide}, \text{reset}, \text{produce},$$
$$\text{again}\}$$
$$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$$
$$e_1 = (l_1 = 0, \textit{false}, \text{start}, l_1^+ = 1 \wedge v^+)$$
$$e_2 = (l_1 = 1 \wedge x < 5, x + 1 > 7, \text{increase}, x^+ = x + 1)$$
$$e_3 = (l_1 = 1 \wedge x \geq 4, \textit{false}, \text{proceed}, l_1^+ = 2)$$
$$e_4 = (l_2 = 0 \wedge v \wedge x > 0 \wedge y < 8, y + x > 15, \text{produce},$$
$$l_2^+ = 1 \wedge y^+ = y + x)$$
$$e_5 = (l_1 = 2 \wedge l_2 = 1 \wedge x = 4, \textit{false}, \text{decide},$$
$$l_1^+ = 3 \wedge l_2^+ = 2)$$
$$e_6 = (l_1 = 2 \wedge l_2 = 1 \wedge x = 4, \textit{false}, \text{decide},$$
$$l_1^+ = 3 \wedge l_2^+ = 3)$$

$$e_7 = (l_1 = 2 \wedge l_2 = 1 \wedge x = 5, \; false, \; \texttt{decide},$$
$$l_1^+ = 4 \wedge l_2^+ = 2)$$

$$e_8 = (l_1 = 2 \wedge l_2 = 1 \wedge x = 5, \; false, \; \texttt{decide},$$
$$l_1^+ = 4 \wedge l_2^+ = 3)$$

$$e_9 = (l_1 = 3, \; false, \; \texttt{reset}, \; l_1^+ = 0 \wedge \neg v^+ \wedge x^+ = 0)$$

$$e_{10} = (l_2 = 2, \; false, \; \texttt{again}, \; l_2^+ = 0)$$

$$p_0 = l_1 = 0 \wedge \neg v \wedge x = 0 \wedge l_2 = 0 \wedge y = 0$$

$$p_m = l_1 = 4 \wedge l_2 = 3$$

Here, $\mathbb{N}_{a..b}$ represents the set of natural numbers in range $[a, b]$, and $\mathbb{B}$ represents the boolean values: $\{false, true\}$. Variables l_0, x, l_2 and y have domains $\mathbb{N}_{0..4}$, $\mathbb{N}_{0..5}$, $\mathbb{N}_{0..3}$ and $\mathbb{N}_{0..12}$, and BDD-representable ranges $[0, 7]$, $[0, 7]$, $[0, 3]$ and $[0, 15]$, respectively. The upper bounds 7 and 15 come back in the error predicates. Note that we simplified $v^+ = true$ to v^+, $v^+ = false$ to $\neg v^+$ and $v = false$ to $\neg v$. And we omitted parts of error predicates that are trivially $false$.

3.6. Computing SEFA transitions

From a set of source states, we may for a SEFA edge compute the possible transitions to target states, and vice versa. Computing transitions is also called 'applying edges', 'taking edges', or 'computing the (pre-)image', and is an essential ingredient in synthesis. To apply edges efficiently, we use two BDD operations: *relnext* for forward application and *relprev* for backward application [67].

relnext(e, p) computes, given an edge $e = (g, r, \sigma, u)$ and a predicate p over V, the predicate p' over V of states that can be reached from states in p by taking edge e. Predicate p indicates the source states. To take the edge, guard g must hold, error predicate r must not hold, and update u then indicates the possible target states. Thus, $p \wedge g \wedge \neg r \wedge u$ indicates the possible combinations of source and target states according to edge e. We precompute transition relation $t = g \wedge \neg r \wedge u$ for each edge, to avoid repeatedly computing it. *relnext* then computes $(\exists_{vars(t)}(p \wedge t))[V^+ := V]$.

relnext thus first computes $p \wedge t$. Then, to get only the possible target states as a predicate over V^+, we take the existential quantification [8] over V. That is, predicate $p \wedge t$ is a predicate over V^*, and the existential quantification gives us a predicate over V^+, for any valuation of the variables in V that satisfies $p \wedge t$. We perform the quantification over a limited set of old states variables $vars(t) \subseteq V$, that contains only the old state variants of the variables that occur in t. By not quantifying over all old state variables V, but only the relevant ones, allowing us to use a partial transition relation [9], we achieve better performance. It reduces the amount of quantification needed. And it also avoids having to add $x^+ = x$ relations to u that prevent unassigned variables from being unconstrained and thus getting any value. This leads to smaller predicates and thus less work in computations.

After applying the edge, the target states should become the current states. We therefore want to have a predicate over variables V, rather than over variables V^+. Thus, *relnext* performs substitution $[V^+ := V]$ to obtain predicate p' over V, of states that can be reached from p by taking edge e.

BDD operation *relprev*(e, p) is similar. It computes for edge e and predicate p over V, the predicate p' over V of states from which by taking edge e the states of p can be reached. That is, *relprev*(e, p) computes $\exists_{vars^+(t)}(t \wedge p[V := V^+])$, with $vars^+(t) \subseteq V^+$ the relevant new state variables.

By combining various operations into a single BDD operation, and by precomputing t, BDD operations *relnext* and *relprev* can efficiently apply SEFA edges.

3.7. Linearized model to SEFA

The symbolic supervisory controller synthesis algorithm requires as input a single plant SEFA $P = (V, D, \Sigma, E, p_0, p_m)$, and a predicate p_f indicating states that are forbidden (e.g., by the requirements). We now consider how this input can be derived from the linearized CIF model, with its discrete and input variables, linearized self-loop edges, initialization predicates, marker predicates and invariants.

Basic SEFA: The discrete variables, including the added location pointer variables, and the input variables, determine V and D. The BDD variables corresponding to V are ordered (see Section 4.1). The events used on the linearized edges determine alphabet Σ. The linearized edges, with their events, guards and updates are transformed to SEFA edges E. To ensure that error predicates do not need to be considered explicitly in cases where we only use the guard and not the entire transition relation, we modify for each edge $e = (g, r, \sigma, u)$ its guard g to $g \wedge \neg r$. The assignments of the linearized edges become updates u of the edges, as described in Section 3.5. The initialization predicates are combined through conjunction to form p_0. The marker predicates are combined through conjunction to form p_m.

Input variables: Input variables can initially have any value, unless their initial values are constrained through initialization predicates. They should be allowed to change to any other value at any time. We therefore introduce, for each input variable i, a unique uncontrollable event e_i and add it to Σ. And we also add a SEFA edge for it, with guard $true$, error $false$, event e_i, and update $i^+ \neq i$, that allows i to change to any other value than the one it currently has.

Integer variable ranges: An integer-typed discrete or input variable with a CIF range $[0, 4]$ allows 5 different values, but as we saw in Section 3.4, it is represented using 3 BDD variables that can represent $2^3 = 8$ different values (BDD-representable range $[0, 7]$). To ensure that the variable never gets the 3 extra values, we forbid all states where it has a value outside of its CIF range. That is, for every integer variable x with CIF range $[x_{min}, x_{max}]$, we add a state requirement invariant $x_{min} \leq x \wedge x \leq x_{max}$. Note that this is thus in addition to error predicates r of edges, which prevents going outside of the BDD-representable range, but does not prevent using out-of-CIF-range values that fall within the BDD-representable range. Furthermore, error predicates r of edges do not ensure that the initial state satisfies integer variable ranges.

Runtime errors for uncontrollable events: Additionally, to ensure *controllability*, for each edge with an uncontrollable event $\sigma \in \Sigma_u$, a state/event exclusion requirement invariant σ needs $g \wedge \neg r$ is added. The state/event exclusion invariant will ensure that $g \wedge r$ is added to p_f (see below), thus ensuring that all states where the uncontrollable event is enabled and would lead to a runtime error are prevented from being reached. Preventing runtime errors for controllable events is discussed later, in Section 3.8.

State plant invariants: The state plant invariants are incorporated in the initialization predicate and the guards of the edges. We first compute p_p, the full restriction imposed by all state plant invariants, by combining these predicates with conjunction. We change p_0 to $p_0 \wedge p_p$ such that the system can only start in states where p_p holds. We must then prevent going from a state where p_p holds to a state where it doesn't hold. If p_p holds before taking an edge e, then it should still hold after taking the edge. We compute from which states s, after taking edge e, we can reach states where p_p still holds, by apply the edge backwards to p_p: $s = relprev(e, p_p)$. Finally, we update guard g of e to $g \wedge s$, to ensure we only end up in states where p_p holds after taking the edge. We thus start in states where p_p holds, by adapting p_0, and stay in states where p_p holds, by adapting the guards of all edges. Note that we apply the optimization of Thuijsman et al. [65], who first check whether s is already implied by $g \wedge p_p$, and forgo updating g if that check holds.

State requirement invariants: The state requirement invariants indicate conditions that must always hold in every state. Their inverse conditions indicate states that may never be reached. We first compute the full restriction p_r imposed by all state requirement invariants, by combining their predicates with conjunction. We then initialize p_f, the forbidden states predicate, to $p_f = \neg p_r$.

State/event exclusion plant invariants: The state/event exclusion plant invariants are incorporated into the edges. Each `disables` state/event exclusion plant invariant is transformed to a corresponding `needs` invariant. Each 'σ needs p' state/event exclusion plant invariant is applied to each edge (g, r, σ, u) by adapting g to $g \wedge p$.

State/event exclusion requirement invariants: The state/event exclusion requirement invariants are mostly considered in the same way. The `disables` invariants become `needs` invariants. The guards of edges with controllable events are restricted as for state/event exclusion plant invariants. Edges with uncontrollable events are handled differently, as synthesis may not restrict the guards of uncontrollable edges, to ensure *controllability* of the resulting controlled system. For each state/event exclusion requirement invariant 'σ needs p', and each edge (g, r, σ, u) with $\sigma \in \Sigma_u$, we compute the states $b = g \wedge \neg p$, where guard g holds, but the invariant doesn't hold ($\neg p$). Such states b would allow the uncontrollable event, but do not satisfy the requirement, and since a supervisor can't prevent uncontrollable event transitions, states where b holds should never be reached. Therefore, we update p_f to $p_f \vee b$.

3.8. Symbolic supervisory controller synthesis algorithm

With these definitions, we may define the symbolic supervisory controller synthesis algorithm, see Algorithm 1. The algorithm takes a plant SEFA P as input, as well as a set of states p_f that are forbidden (e.g., by the requirements).

The algorithm starts (line 1) by initializing the controlled-system states predicate C to be those states that are not forbidden, to ensure only safe states are part of the controlled-system behavior.

It then repeatedly computes non-blocking, controllable and reachable states until a fixed point is reached (lines 2–8). For each iteration, it first preserves the current controlled-system states predicate C in C' (line 3). It then computes the non-blocking states from the marked states p_m using a backward reachability search on all edges E, restricted to stay within the current controlled-system states C, using $BRS(p_m, E, C)$ from Algorithm 2; and it updates the controlled-system states C to only those non-blocking states (line 4).

Subsequently, it performs a second backward reachability search on the inverse of the controlled-system (good) states C (line 5). By pushing these bad states backwards through edges labeled with uncontrollable events, more bad states are computed (B). No additional restrictions are imposed on this reachability computation (*true*). This second reachability computation ensures that states with a transition for an uncontrollable event to a bad state are in turn labeled bad, and thus ensures that the controlled system is controllable. It also prevents runtime errors for edges with uncontrollable events, through r of each edge, and through range restrictions of p_f being in C. The bad states are inverted and become the updated controlled-system states (line 6).

Optionally, an additional forward reachability search can be performed from the initial states p_0 (line 7). Using a forward reachability search, the controlled-system states in C are also all reachable. By using forward reachability the result of synthesis is generally more intuitive for engineers,

---

Algorithm 1 Symbolic Supervisory Controller Synthesis (*SSCS*)

Input: Plant SEFA $P = (V, D, \Sigma, E, p_0, p_m)$ and forbidden states p_f over V.
Output: Controlled-system SEFA S.
1: $C \leftarrow \neg p_f$
2: **repeat**
3:     $C' \leftarrow C$
4:     $C \leftarrow BRS(p_m, E, C)$
5:     $B \leftarrow BRS(\neg C, E_u, true)$
6:     $C \leftarrow \neg B$
7:     $C \leftarrow FRS(p_0, E, C)$ ▷ Optional step.
8: **until** $C = C'$
9: **for all** $e \in E$ with $\sigma \in \Sigma_c$ **do**
10:     $g \leftarrow g \wedge relprev(e, C)$
11: **end for**
12: $S \leftarrow (V, D, \Sigma, E, p_0 \wedge C, p_m \wedge C)$

---

as control conditions do not restrict unreachable states. Forward reachability may also positively or negatively impact synthesis performance, as it may lead to smaller or larger predicates for C, which affects the effort required for subsequent reachability computations. The forward reachability search is performed from the initial states, using all the edges, and is restricted to the current controlled-system states.

The loop with the various reachability searches is repeated until a fixed point is reached ($C = C'$) (line 8). At this point, C contains the controlled-system states, namely all safe, controllable, non-blocking (and, optionally, reachable) states. Note that the implementation in ESCET has an optimization to prevent unnecessary fixed-point computations, and stops earlier if possible. This is akin to what is described later in Section 4.5 for applying edges, but then for applying multiple fixed-point computations. Furthermore, the implementation in ESCET checks after each reachability computation whether the combination of the uncontrolled system initialization predicate and the current controlled-system predicate still allow for initialization in the controlled system. If not, then the algorithm can also stop earlier.

Finally, the controlled system, or supervised system, S is determined (lines 9–12). First, the guards of the edges with controllable events are restricted to ensure that the controlled system stays within the controlled-system states C (lines 9–11). Each edge gets a strengthened guard by combining the edge's existing guard with those conditions under which only states in the controlled-system behavior can be reached by considering the update of the edge (line 10). This also ensures that runtime errors are prevented for edges with controllable events, similar to preventing it for uncontrollable events before. Finally, the controlled-system SEFA is constructed, using the updated edges E, and by restricting the initial and marked states to controlled-system states C (line 12).

The algorithm performs backward reachability searches using the *BRS* algorithm, see Algorithm 2. As input, the algorithm gets a start predicate P, edges E and a restriction predicate R. It performs a fixed-point computation in the loop (lines 1–4). In each iteration, it preserves the current predicate P in P' (line 2). It then updates P, adding all states from which states in P can be reached by edges from E, restricting the result to R (line 3). BDD operation

---

Algorithm 2 Backward Reachability Search (*BRS*)

Input: Start predicate P, edges E, and restriction predicate R.
Output: The coreachable states P', that is, those states that can reach states in P via edges in E, restricted to states in R.

1: **repeat**
2:     $P' \leftarrow P$
3:     **for all** $e \in E$ **do**
4:         $P \leftarrow P \lor relprevIntersection(e, P, R)$
5:     **end for**
6: **until** $P = P'$

---

relprevIntersection(e, P, R) implements *relprev*(e, P) $\land$ R as a single BDD operation, for better performance. The edges are repeatedly applied until a fixed point is reached ($P = P'$), when P contains the coreachable states (line 4). The implementation in ESCET is actually a bit smarter in detecting a fixed point, see Section 4.5.

The *FRS* algorithm to perform a forward reachability search is similar, but applies the edges forward using *relnextIntersection*, rather than backward using *relprevIntersection*.

3.9. Supervised system CIF model creation

The last step that the synthesis tool performs is to create the output CIF model that represents the controlled system. It does not directly convert the controlled-system SEFA to a single CIF EFA as output of the synthesis algorithm. Instead, a supervisor EFA is derived from it, which is put in parallel to the original plant model, and restricts the plant behavior.

If the controlled-system SEFA has no initial state ($p_0 = false$), then synthesis completes by indicating to the user that the result is an empty supervisor and no CIF model is produced. Otherwise, the plantified CIF model (see Figure 3) is used as a starting point to construct the output CIF model. All plantified requirement automata are labeled as supervisor automata, to make it clear that they track additional plant states for the supervisor. Subsequently, all requirement invariants are removed, as they will be enforced by the supervisor. And the initialization predicate p_0 of the controlled-system SEFA is added as an initialization predicate to the output model.

Finally, an extra supervisor automaton is added to complete the controlled-system CIF model. Its alphabet is equal to the controllable events Σ_c of the controlled-system SEFA. It has a single location that is initial and marked. For each (controllable) event in its alphabet, it has a self-loop edge without updates. Each edge has a guard that indicates the condition under which the controllable event is enabled in the controlled system, i.e., the disjunction of the guards of the edges for that event in the controlled-system SEFA.

The supervisor guards may become quite complex predicates. To make it easier to understand the supervisor, the supervisor guard predicates are simplified, akin to the work of Miremadi et al. [41, 42]. That is, we simplify the guards predicates to only those conditions that must be additionally enforced on top of the plant behavior and restrictions already enforced by the requirements. The extra restrictions are obtained by simplifying the controlled system guards under the assumption of the plant guards, plant invariants, requirements and the controlled-system behavior. Similarly, the initialization predicate of the controlled-system model is simplified to only those additional restrictions not already expressed by the uncontrolled-system initialization and the plant state invariants.

If synthesis did not need to restrict any further behavior, the resulting controlled-system CIF model has an additional controlled system initialization predicate *true* and the supervisor automaton also has only *true* guards. If synthesis does impose additional restrictions, for instance to prevent runtime

errors, or ensure non-blockingness or controllability, the resulting model will have a non-*true* initialization predicate and/or supervisor guards.

Simplification is performed by default, but can be disabled. If it is enabled, the supervisor no longer enforces the requirements. The requirement invariants are then preserved, rather than removed, and are relabeled as supervisor invariants.

4. Technical improvements

We describe the improvements that were made to CIF's symbolic synthesis tool between ESCET releases v0.8 (December 2022) and v4.0 (June 2024), but only as far as they affect synthesis performance. We forgo improvements that were subsequently made obsolete by improvements in newer releases. Concretely, this covers the improvements released as part of ESCET releases v0.9, v0.10, v1.0, v2.0, v3.0 and v4.0. As the ESCET project does quarterly releases, these six releases span one and a half years of ESCET tool development, although some improvements were in preparation already before the v0.8 release. The various improvements likely differ substantially in the impact that they have on the synthesis performance, as they are not all equality profound. We do explain all of them, given that in Section 6 we will experimentally show the cumulative performance benefits that they bring.

The following improvements are discussed in this section:

1. DCSH variable ordering for BDDs (Section 4.1)
2. Linearized variable ordering relations for BDDs (Section 4.2)
3. Per-event edge granularity (Section 4.3)
4. Efficient edge application (Section 4.4)
5. Early fixed-point detection (Section 4.5)
6. Invariants: improved state plant invariants enforcement (Section 4.6)

Figure 6 positions these improvements, on relevant part of the same overview presented earlier in Figure 3, as numbers in parentheses, and with an extra dotted/blue arrow.

4.1. DCSH variable ordering for BDDs

As mentioned in Section 3.4, the order of the boolean BDD variables is vital to the memory and running time characteristics of BDD representations and manipulations, as used by CIF's symbolic synthesis algorithm [63]. Heuristic variable ordering algorithms that exploit the inherent structure of the system modeled as EFAs are able to significantly reduce the synthesis effort [34], especially for larger inputs, making synthesis applicable to more complex systems.

In CIF, the variable ordering algorithms order the CIF variables, and the order of the corresponding BDD variables is derived from that. The BDD variables of a single CIF variable – both the current and new state ones – remain together, and are kept interleaved, as explained in Section 3.4.
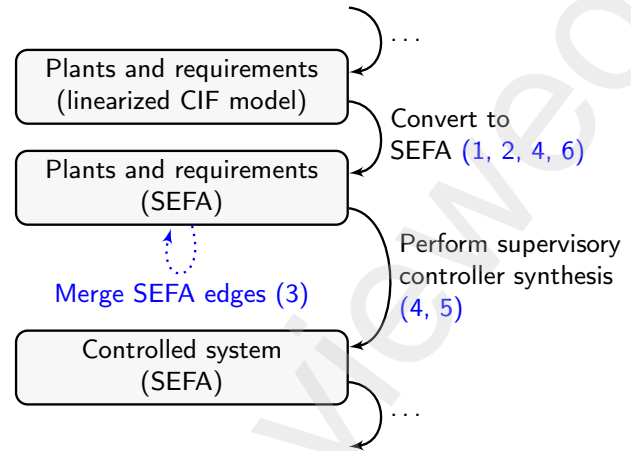
Figure 6: Synthesis performance improvements, positioned on the synthesis tool steps (only relevant part is shown).

The DSM-based Cuthill-McKee / Sloan variable ordering Heuristic (DCSH) is such a heuristic variable ordering algorithm, for which Lousberg et al. experimentally showed a significant improvement on seven out of eleven of their case studies [34]. The algorithm uses a Design Structure Matrix (DSM) [14], a symmetric matrix in which relations between the variables of the linearized CIF model (see Figure 3) are encoded. The rows and columns of the symmetric matrix are both labeled with the variables, and they are then reordered, to place closely related variables together, closer to the diagonal of the matrix. DCSH uses two ordering heuristics, which operate on nodes of the undirected adjacency-graph equivalent of the matrix (each node represents a variable of the CIF model). The two ordering algorithms that are used are the Sloan profile/wavefront-reducing node ordering heuristic [59], and a weighted version of the Cuthill-McKee bandwidth-reducing node ordering heuristic [13]. For the orders produced by both algorithms, as well as the reverse of these two orders, DCSH then computes the Weighted Event Span (WES), a simple-to-compute metric that correlates to the computational effort of symbolic model checking [40], which involves similar computations as symbolic supervisor synthesis. From the four variable orders (from the two algorithms, and those orders in reverse) the one with the best WES is selected as the result of the DCSH algorithm.

Compared to the DCSH algorithm from [34], the CIF implementation can moreover deal with multiple unconnected partitions within the adjacency graph, by ordering each of them separately. The partitions are sorted in descending order based on their size, such that the variables corresponding to unconnected nodes (single-variable partitions) still appear at the end of the resulting variable order, as in [34].

An example of the effects of applying DCSH is shown in Figure 7. It shows both the DSM before applying DCSH (7a) and the one after applying DCSH (7b). From top to bottom, and left to right, the variables are shown, either in
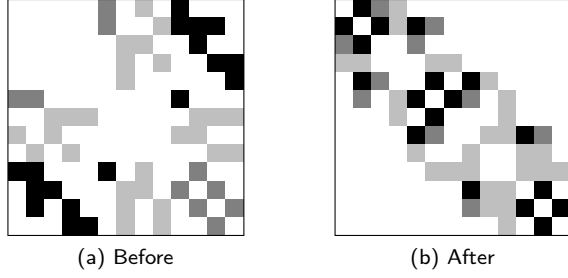
(a) Before   (b) After

Figure 7: Example of a DSM, before and after applying DCSH.

Figure 8: Example DSM with linearized variable relations derived from linearized edges.

the original model order (before ordering) or in the computed order (after ordering). The shading of the cell indicates the weight between the variables (the darker the cell, the larger the weight).

ESCET v0.8 uses only the FORCE algorithm [2], followed by the classic sliding window algorithm, to automatically order the variables. ESCET v4.0 first uses the DCSH algorithm, before still applying the FORCE and sliding window algorithms. Lousberg et al. have shown that applying FORCE after DCSH produces better results than only applying one of them [34]. The sliding window algorithm further locally optimizes the variable order.

4.2. Linearized variable ordering relations for BDDs

Heuristic variable ordering algorithms for BDDs, like DCSH, require as input some information about the relations between the variables that are to be ordered, for instance in the form of a DSM or an adjacency graph. The variable relations are also determined using heuristics, and using different variable relation heuristics may lead to different variable orders produced by variable ordering algorithms, which affects synthesis performance. Synthesis performance can therefore be improved by choosing a variable relations heuristic in such a way that it fits well with the reachability computations (see Algorithm 2), a key ingredient of the symbolic synthesis algorithm (see Algorithm 1), and especially with the transition relations on which these reachability computations operate.

The symbolic synthesis algorithm in the ESCET toolkit linearizes the CIF model to eliminate the parallel composition between the different EFAs, resulting in a single EFA with a single location and only self-loop edges, which is then encoded as a SEFA (as previously explained in Section 3). Since the transition relations – the SEFA edges – are directly derived from the linearized edges, it is beneficial to also derive the variable relations from the linearized edges.

Lousberg et al. [34] propose to count the number of times that two variables occur together in the different linearized edges, and use that 'weight' as the relation between two variables. That is, in the DSM the cells are filled with these weights, and in the adjacency graph the edges are labeled with the weights. By using this particular variable relation, variable ordering algorithms places all variables that occur

more frequently together in transition relations more closely together in the variable order. This allows the transition relations to be more efficiently applied during synthesis.

For the linearized edges of Figure 4c, the variables that occur in the linearized edges are:

$$e_1 : l_1, v \qquad\qquad e_6 : l_1, l_2, x$$
$$e_2 : l_1, x \qquad\qquad e_7 : l_1, l_2, x$$
$$e_3 : l_1, x \qquad\qquad e_8 : l_1, l_2, x$$
$$e_4 : l_2, v, x, y \qquad e_9 : l_1, v, x$$
$$e_5 : l_1, l_2, x \qquad\qquad e_{10} : l_2$$

The corresponding linearized variable relations are shown in Figure 8 as a DSM. For instance, variables l_1 and x occur together in 7 linearized edges. Self-relations of variables are excluded, and thus the top-left to bottom-right diagonal always only contains entries with value zero.

ESCET v0.8 does not have these linearized variable ordering relations, but instead used more complex custom relations. ESCET v4.0 uses the linearized variable ordering relations for the FORCE and sliding window algorithms, and the legacy relations from v0.8 for the DCSH algorithm. This combination was found to give the best results.

4.3. Per-event edge granularity

The transition relations that are used during synthesis are derived from the linearized edges. It is possible to use each linearized edge as a separate transition relation (as shown in Section 3.5), applying them one by one during reachability computations (see Algorithm 2). SEFA edges and transition relations then match one on one. It is also possible to combine multiple transition relations into a single transition relation [8]. Two SEFA edges $(g_1, r_1, \sigma, u_1), (g_2, r_2, \sigma, u_2) \in E$ can be combined to a single edge (g, r, σ, u) as follows:

$$g = g_1 \vee g_2$$
$$r = (g_1 \wedge r_1) \vee (g_2 \wedge r_2)$$
$$u = (g_1 \wedge u_1') \vee (g_2 \wedge u_2')$$

Update u_1' is obtained from u_1 by adding $x^+ = x$ for each variable x that is assigned in update u_2 but not in u_1, and similarly update u_2' is obtained from u_2.

As an example, consider the following two SEFA edges, and their combination:

Edge 1: $(x \leq 4, true, e, y^+ = y + 1)$

Edge 2: $(x \geq 4, true, e, z^+ = z + 1)$

Combined: $(true, true, e,$

$\quad (x \leq 4 \wedge y^+ = y + 1 \wedge z^+ = z) \vee$

$\quad (x \geq 4 \wedge y^+ = y \wedge z^+ = z + 1)$

$)$

Combining similar transition relations leads to fewer transition relations that need to be applied during reachability computations. The combined transition relation may be simpler than the original ones, as in the example above the guards $x \leq 4$ and $x \geq 4$ get combined to $true$. This kind of reduction is not uncommon, but there are also cases where the combination is more complex than the original guards. Similarly, update predicates may also get reduced, or become more complex. In the example above, they get more complex, as they mostly concern different variables (one increments y, the other increments z). Combining edges is often a tradeoff between the number of transition relations and their size.

In ESCET v0.8, CIF's symbolic synthesis tool uses the per-linearized-edge transition relations. In ESCET v4.0, it combines all linearized transition relations, per event. This level of granularity is more combined than the per-linearized-edge transition relations, but less so than combining all transitions relations for all events into a single transition relation (which is not allowed by our definition). Using a single transition relation for all events often leads to much larger transition relations and intermediate results during synthesis, compared to using per-event transition relations, as shown by Fei et al. [15].

Per-event transition relation granularity makes sense in our case, since linearized edges with the same event will often (at least in part) have similar guards and updates, as they are constructed by combining the edges of the original automata from the CIF model in all different combinations. We saw an example earlier in Figure 4c, where the four linearized edges for event decide have similar guards and updates. Using per-event transition relation granularity, in practice, the number of transition relations is reduced, while their size does not increase too much.

4.4. Efficient edge application

Since applying edges is a key operation in synthesis, applying edges more efficiently can significantly improve the synthesis performance. This can among others be achieved by removing redundant operations and by using compounded operations that combine multiple separate operations.

In ESCET v4.0, the synthesis algorithm uses the *relnext(Intersection)* and *relprev(Intersection)* BDD operations to efficiently compute SEFA transitions by applying SEFA edges (see Section 3.6). Remember that *relnextIntersection*(e, p, R) computes for an edge $e = (g, r, \sigma, u)$, predicate p and restriction R, the predicate

$(\exists_{vars(t)}(p \wedge t))[V^+ := V] \wedge R$, with $t = g \wedge \neg r \wedge u$ being precomputed.

In ESCET v0.8, applying edges is less efficient. It does not use *relnextIntersection*. It computes the predicate $(\exists_V (p \wedge g \wedge \neg r \wedge u))[V^+ := V] \wedge R$, with $g \wedge u \wedge R$ being precomputed. These differences have several performance implications.

First of all, transition relation t does not change during synthesis, and thus needs to be computed only one. Contrarily, $g \wedge u \wedge R$ contains R, which does change during synthesis; in Algorithm 1, controlled-behavior predicate C is passed for R, which is constantly updated during the algorithm. In v0.8, the precomputation is therefore done for each reachability computation, rather than once before all reachability computations, leading to more computations.

Secondly, in v4.0, the transition relation includes error predicate r. r is then applied together with guard g and update u. In v0.8, r is applied using a separate conjunction operation. This is less efficient, as it uses more separate operations that each have to go through the full BDDs and modify them.

Thirdly, in v0.8, the existential quantification and replacement operation are applied as separate BDD operations. In v4.0, this is all part of the compounded operations *relnext* and *relprev*, which are more efficient.

And fourthly, v4.0 uses partial transition relations, for which a reduced existential quantification over a more limited set of variables can be used. v0.8 uses full transition relations, with $x^+ = x$ in updates u for each variable x that is not assigned on the edge, leading to larger predicates and more work. v0.8 further uses an existential quantification over full variable set V, which also requires more work.

4.5. Early fixed-point detection

Symbolic supervisory controller synthesis involves several fixed-point computations (see Algorithm 1), where edges (transition relations) are repeatedly applied to a certain predicate representing a part of the state space, and this is repeated until the state space does not grow any further (see Algorithm 2). If after applying certain edges it becomes clear that no more states will be reached, the reachability computation is completed (line 4 in Algorithm 2).

A naive approach for computing reachability, as shown in Algorithm 2, is to apply each edge once, and subsequently apply all of them again if any of them led to reaching more states, and to keep repeating this for as long as more states are reached. If in a certain iteration of applying the edges no new states are reached, the reachability computation terminates.

An example is shown in Figure 9a. There are six edges, that are applied one by one, each iteration. In the first iteration, four edges reach new states (indicated by '✓'), and two do not (indicated by '✗'). Since at least one edge reached new states, a second iteration is needed. In that second iteration, three edges reach new states, and thus a third iteration is needed. In that third iteration, none of the edges reach new states, and thus the reachability computation has reached a fixed point.

Assuming there are n edges that are ordered in some way, then it is also possible to stop earlier in the last iteration. For instance, in the example, in the second iteration of applying

Edge	Iteration			Edge	Iteration		
	1	2	3		1	2	3
e_1	✓	×	×	e_1	✓	×	×
e_2	×	✓	×	e_2	×	✓	×
e_3	✓	✓	×	e_3	✓	✓	×
e_4	×	✓	×	e_4	×	✓	×
e_5	✓	×	×	e_5	✓	×	–
e_6	✓	×	×	e_6	✓	×	–
(a) Naive approach				(b) Stop earlier			

Figure 9: Example of concluding earlier that a reachability fixed point is reached.

the edges, more states are reached by applying e_2, e_3 and e_4, but not by e_5 and e_6 afterwards. Once all n edges have been applied once more, after the last edge that reached new states, the algorithm could already stop. That is, in the third iteration, there is no need to apply the last two edges (indicated by '–'), as they would be applied the same way to the same predicate as in the previous iteration, and would thus again not reach any new states. This is shown in Figure 9b.

In this example, by stopping earlier, two edges do not need to be applied. In general, the number of edges that does not need to be applied in the last iteration may vary within the range [1,n].

ESCET v0.8 uses Algorithm 2, while ESCET v4.0 uses the optimized version described in this section.

4.6. Invariants: improved state plant invariants enforcement

In CIF, the behavior of plant automata can be restricted using plant invariants (see Section 3.1). They are taken into account during synthesis by restricting the guards of edges (see Section 3.7). In ESCET v4.0, the approach of Thuijsman et al. [65] is used to optimize that, by forgoing updating the edge guard if it is not required.

In ESCET v0.8, a different approach is used to prevent the guards from being needlessly updated. Instead of checking whether s is already implied by $g \wedge p_p$, it simplifies s with respect to p_p to produce s', and then always adds s' to the guard by changing g to $g \wedge s'$. The simplification algorithm is based on Coudert and Madre's 'restrict' function [12].

The simplification approach of Thuijsman et al. [65] is superior, as it computes with certainty whether an additional restriction is needed, while the simplification is based on best-effort heuristics, and therefore does not offer the same guarantees.

5. Benchmark models

To be able to perform performance measurements, suitable benchmark models are needed. CIF ships with a set of benchmark models. These benchmarks are shared under an open license, the MIT license, and can thus also be used by other researchers, to benchmark synthesis algorithms and try out and evaluate any improvements made to them. The benchmarks consist of a mix of industrial and academic models, that differ in size and complexity, making them suitable for performance evaluations.

In this section, we introduce and describe CIF's set of benchmark models, and provide various metrics for them. The artifact that accompanies this paper includes both the models and the scripts to produce the metrics [26]. The benchmark models themselves are not new, and are thus not a contribution of this paper.

5.1. Overview of benchmark models

Table 1 shows an overview of the benchmark models shipped with Eclipse ESCET v4.0, and that we use for our experiments in Section 6. For each benchmark, the table lists the model number, the short and full names of the model, references to publications with further information (if available), the size of the uncontrolled system state space ('US', in number of states), and the size of the controlled system state space ('CS', in number of states). Although the state space sizes do not translate one-to-one to the sizes of the BDD representations of the models, nor to the complexity of the computations performed during synthesis, they still provide some indication of the complexity of the models. For some benchmarks, the number of states in the controlled system is larger than that of the uncontrolled system. While the requirements restrict the behavior of the uncontrolled system, in such cases the requirements themselves also introduce additional state to monitor the plants.

5.2. Benchmark descriptions

We provide a short description of each of the 23 benchmark models.

1) The *advanced driver assistance system* (ADAS) concerns a safe controller for an adaptive cruise control system for the Toyota Prius Executive, which is controlled via a Human Machine Interface (HMI). The synthesized controller was tested in multiple ways, including by driving around in real-world traffic [30].

2) The *automated guided vehicles* (AGVs) serve a manufacturing work cell that produces parts. The five AGVs travel on fixed circular routes, alternately loading and unloading. The AGVs move through shared zones, and no two AGVs may be in such a zone at the same time, to avoid collisions [71].

3+4) The *body comfort system* (BCS) product line is a frequently used benchmark in product line engineering (PLE). The BCS product line originates from the automotive industry. There is a *dynamic* version of the model (benchmark 3), which allows the system to be reconfigured during execution, and a *static* version of the model (benchmark 4), which does not allow the system to be reconfigured during execution. Synthesis will ensure that all safety requirements are satisfied, regardless of the feature configuration [5, 64].

5) The Algera *bridge* over the river Hollandse IJssel connects two cities in the Netherlands. It is a movable bridge that consists of two vehicle lanes, a lane for cyclists, and a lane for pedestrians, as well as various stop signs, traffic barriers, and so on. A fault-tolerant controller is synthesized to ensure the safe operation of the bridge, even in case of for instance faulty sensors or broken actuator connections [51].

Table 1
Overview of the benchmark models.

Nr	Short name	Full name	Refs	Size (US)	Size (CS)
1	adas	Advanced driver assistance system	[30]	$3.40 \cdot 10^{09}$	$2.04 \cdot 10^{10}$
2	agv	Automated guided vehicles	[71]	$3.07 \cdot 10^{03}$	$4.41 \cdot 10^{03}$
3	bcs-dynamic	Body comfort system (dynamic)	[5, 64]	$6.16 \cdot 10^{20}$	$4.23 \cdot 10^{20}$
4	bcs-static	Body comfort system (static)	[5, 64]	$3.18 \cdot 10^{14}$	$1.97 \cdot 10^{14}$
5	bridge	Bridge	[51]	$5.92 \cdot 10^{33}$	$9.55 \cdot 10^{28}$
6	cmt-v1	Cat and mouse tower (v1, $n=3$, $k=2$)	[35, 65]	$1.44 \cdot 10^{04}$	$8.61 \cdot 10^{03}$
7	cmt-v2	Cat and mouse tower (v2, $n=3$, $k=1$)	[35, 65]	$1.07 \cdot 10^{09}$	$3.03 \cdot 10^{05}$
8	cluster-tool	Cluster tool	[60]	$7.50 \cdot 10^{06}$	$2.70 \cdot 10^{06}$
9	dining-phils	Dining philosophers	[28]	$2.43 \cdot 10^{02}$	$2.41 \cdot 10^{02}$
10	festo	FESTO production line	[48]	$1.47 \cdot 10^{26}$	$2.22 \cdot 10^{25}$
11	litho-init	Lithography machine initialization	[65, 68]	$7.21 \cdot 10^{09}$	$2.46 \cdot 10^{09}$
12	mri-pss-event	MRI scanner PSS (event-based)	[61, 62]	$3.60 \cdot 10^{03}$	$3.09 \cdot 10^{04}$
13	mri-pss-state	MRI scanner PSS (state-based)	[61]	$1.44 \cdot 10^{04}$	$2.79 \cdot 10^{04}$
14	multi-agent-form	Multi-agent formation	[11]	$1.00 \cdot 10^{03}$	$3.04 \cdot 10^{02}$
15	prod-cell	Production cell	[16]	$3.76 \cdot 10^{08}$	$1.15 \cdot 10^{08}$
16	robo-swarm-aggr	Robotic swarm aggregation	[33]	$1.00 \cdot 10^{00}$	$9.00 \cdot 10^{00}$
17	robo-swarm-clus	Robotic swarm clustering	[33]	$1.00 \cdot 10^{00}$	$1.60 \cdot 10^{01}$
18	robo-swarm-form	Robotic swarm formation	[33]	$8.00 \cdot 10^{01}$	$1.38 \cdot 10^{02}$
19	robo-swarm-segr	Robotic swarm segregation	[33]	$6.40 \cdot 10^{01}$	$1.28 \cdot 10^{02}$
20	sudoku	Sudoku		$1.53 \cdot 10^{11}$	$1.58 \cdot 10^{07}$
21	theme-park	Theme park vehicles	[22]	$2.95 \cdot 10^{05}$	$6.69 \cdot 10^{06}$
22	wafer-scanner-n1	Wafer scanner ($n=1$)	[56]	$5.30 \cdot 10^{05}$	$5.24 \cdot 10^{04}$
23	waterway-lock	Waterway lock	[49]	$5.96 \cdot 10^{32}$	$5.87 \cdot 10^{24}$

6+7) A *cat and mouse tower* (CMT) is a tower that has n levels, with 5 rooms at each level. Cats and mice may move between connected rooms, either at the same level or to the level above or below. The controller must ensure that a cat and a mouse are never in the same room at the same time. Most moves can be prevented by the supervisor, but not all of them. In the original version of the model (*version 1*, benchmark 6), k cats start in room 1 at level 1, and k mice in room 5 of level n. In the adapted version of the model (*version 2*, benchmark 7), initially all rooms are empty, cats and mice may enter and leave the tower (cats via room 1 at level 1, and mice via room 5 at level n), and at most k cats may be in each room, and similarly at most k mice may be in each room [35, 65].

8) The *cluster tool* is an integrated manufacturing system used for wafer processing. Wafers enter via the entering load lock and exit via the exiting load lock. They can be processed in nine chambers. Four transportation robots transport the wafers. Three one-slot buffers are used for intermediate storage. Wafers follow pre-specified routing sequences. The synthesized controller guarantees continuous wafer processing without blocking situations [60].

9) The *dining philosophers* model concerns a simplified version of the traditional dining philosophers problem. Five philosophers dine at a table, and each has its own plate. Between the plates are forks. Each philosopher requires two forks to eat the meal. No two philosophers can take the same fork at the same time [28].

10) The *FESTO production line* consists of six connected stations. Products enter via the distributing station. They proceed to the handling station, which buffers products, before they go to the testing station. The testing station measures the height of the products, rejects faulty products, and lets correct products through to the buffering station. After the buffering station, products are processed by drilling a hole in them. Finally, products arrive at the storage station. The supervisor ensures safe, correct and efficient operation of the product line [48].

11) The *lithography machine initialization* concerns a lithography system with multiple sub-systems, that need to be initialized in the correct order. Each sub-system is initialized from its unknown starting state to its initialized state. A sub-system can be initialized via one or more phases, by transitioning from a phase to a previous or next one. The synthesized supervisor takes into account the dependencies between the system, constraining the initialization order. If the synthesized supervisor is not empty, a successful initialization sequence exists [65, 68].

12+13) The *patient support system (PSS)* positions a patient in an *MRI scanner*. It consists primarily of the patient table, on which the patient lies. The table is moved into a hole in the scanner (the 'bore'). A user interface allows medical personnel to operate the patient table. The table can be positioned manually, but there is also a 'light-visor' for automatic positioning. The synthesized supervisor must ensure that the patient support system operates safely, such that the MRI scanner is not damaged, and that the patient remains safe [61, 62]. There are two versions of the model, a traditional event-based one (benchmark 12), and one with state-based invariants (benchmark 13).

14) The *multi-agent formation* problem has as goal to control a team of agents. Agents can only move clockwise around a circular route. They must assume certain geometric

formations, i.e., an equilateral triangle and an alignment curve. The team leader or remote operator decides the formation to use. The synthesized supervisor ensures the desired formations are reached [11].

15) A *production cell* consists of a stock, a feed belt, an elevating rotary table, a robot, a press, a deposit belt, and a crane. While they each execute asynchronously and independently, they synchronize for cooperation and safety [16].

16–19) Swarm robotics involves a group of wheeled robots that interact to solve relatively complex tasks cooperatively. *Swarm robotics aggregation* (benchmark 16) involves a group of robots that gather in a homogeneous environment. Each robot is equipped with a sensor to detect the presence of other robots in its line of sight. A robot that does not detect any other robots moves backwards along a circular trajectory. A robot that does detect another robot turns clockwise. *Swarm robotics clustering* (benchmark 17) involves that the robots cluster objects that are initially dispersed in the environment. Robots are equipped with sensors to detect objects and other robots in their direct line of sight, and they move based on their sensor readings. *Swarm robotics group formation* (benchmark 18) involves leader robots, green follower robots and blue follower robots. Each leader should be in equilibrium, its number of green and blue followers differing by at most one. Leaders initially stand still at a random location, while followers move randomly. Leaders send offers to followers, which followers may accept, making them stop and relay their leader's messages. *Swarm robotics segregation* (benchmark 19) also involves leader and follower robots. Each leader has a type, characterized by a color (red, green, or blue). Followers that are not in range of a leader, or only in range of a single type of leader, do not move. Followers that are in range of multiple types of leaders move randomly [33].

20) *Sudoku* is a number placement game. Within a (in this case) 4-by-4 grid, each entry needs to be assigned a number. The game is completed when the top-left, top-right, bottom-left, and bottom-right 2-by-2 grids contain the numbers 1 through 4, every row contains these numbers as well, and so does every column. Synthesis computes all the possible solutions.

21) The Multimover is a flexible *theme park vehicle*. It is used in both theme parks and museums. The vehicle is battery-operated and follows an electrical wire that is integrated in the floor. The floor also contains floor codes, that provide information about the track, influence the route and speed of the vehicle, the music it plays, and so on. An operator controls deployment of the vehicles, while 'ride control' coordinates all vehicles, starts and stops them. Sensors are used to detect physical objects and prevent collisions. The synthesized supervisor ensures safe operation of the vehicles [22].

22) The *wafer scanner* is a lithography machine used in the production process of integrated circuits. Part of the control of the system is the wafer logistics: the wafer handler takes wafers from the track, performs a number of pre-exposure steps (such as conditioning and alignment), and routes wafers to the wafer stage and back. At the stage, wafers

are put on one of the two chucks, and subsequently they are measured and exposed. Since immersion lithography is used, a wafer must always be present on the exposure chuck to avoid disruption of the film of water below the lens; dummy wafers are inserted into the system for this. The supervisor ensures a correct and safe wafer flow through the system. The model for $n = 1$ is a simplified version of the full model, with only one production wafer, besides the two dummy wafers, for at most three wafers in total in the system [56].

23) Lock III is a *waterway lock* in Tilburg, The Netherlands. The lock features gates, watertight doors used to seal off the chamber from the outside water. The lock's downstream side features paddles for emptying the chamber, while its upstream side has culverts to fill the chamber. Vessel traffic is regulated by traffic lights. In a remote control center operators monitor the lock via camera images, and issue commands via a graphical user interface (GUI) of the SCADA system, to for instance open the gates. The supervisor ensures the safe and correct operation of the lock [49].

5.3. Additional benchmark metrics

Table 2 shows the following additional metrics for each of the benchmark models:

Σ_c The number of controllable events.

Σ_u The number of uncontrollable events.

A_p The number of plant automata.

A_r The number of requirement automata.

l_p The number of plant locations, summed over the automata of A_p.

l_r The number of requirement locations, summed over the automata of A_r.

e_p The number of plant edges, summed over the locations of l_p. If a single edge is labelled with multiple events, as shorter notation for writing multiple edges, then the edge is counted as many times as there are events on the edge.

e_r The number of requirement edges, summed over the locations of l_r. Similarly counted as e_p.

g_p The number of plant guards, i.e., the number of edges in e_p that have a non-*true* guard.

g_r The number of requirement guards, i.e., the number of edges in e_r that have a non-*true* guard.

a_p The number of plant assignments, i.e., the number of assignments on edges in e_p. A single edge may have multiple assignments for the same variable, if conditional updates are used, and these are all counted separately.

a_r The number of requirement assignments, i.e., the number of assignments on edges in e_r. Similarly counted as a_p.

i_p The number of initial plant locations, i.e., the locations in l_p that could be initial locations. In CIF, a location can have initialization predicates that indicate under which condition the location is an initial location, for instance in terms of the values of variables or the initial locations of other automata. Each potential initial location is counted once.

Table 2
Additional metrics about the benchmark models.

Nr	Short name	Events		Automata										Initial.			Marking			Vars.		Invariants			
		Σ_c	Σ_u	A_p	A_r	l_p	l_r	e_p	e_r	g_p	g_r	a_p	a_r	i_p	i_r	i_c	m_p	m_r	m_c	v_n	v_v	$t_{p,s}$	$t_{r,s}$	$t_{p,e}$	$t_{r,e}$
1	adas	32	35	28	3	63	6	75	12	0	0	0	0	28	3	0	28	4	0	0	0	0	0	0	69
2	agv	10	16	5	8	26	23	26	29	0	0	0	0	5	8	0	5	23	0	0	0	0	0	0	0
3	bcs-dynamic	48	96	58	0	115	0	179	0	54	0	54	0	61	0	1	62	0	0	27	54	0	0	88	54
4	bcs-static	48	42	58	0	115	0	125	0	0	0	0	0	61	0	1	62	0	0	27	54	0	0	88	54
5	bridge	65	149	80	0	181	0	229	0	0	0	27	0	80	0	0	88	0	0	21	42	0	0	136	189
6	cmt-v1	44	6	30	0	90	0	200	0	0	0	0	0	30	0	0	30	0	0	0	0	0	15	0	0
7	cmt-v2	44	10	30	0	60	0	104	0	0	0	0	0	30	0	0	30	0	0	0	0	0	15	0	0
8	cluster-tool	32	9	18	16	58	32	85	32	0	0	0	0	18	16	0	18	16	0	0	0	0	0	0	0
9	dining-phils	15	0	10	0	30	0	45	0	0	0	0	0	10	0	0	10	0	0	0	0	0	0	0	0
10	festo	64	122	91	3	186	16	190	18	0	0	0	0	91	3	0	95	3	0	0	0	0	0	25	200
11	litho-init	68	98	16	0	212	0	324	0	0	0	0	0	46	0	0	16	0	0	0	0	0	50	0	0
12	mri-pss-event	20	19	17	17	47	46	105	112	0	0	0	0	17	17	0	41	44	0	0	0	0	0	0	0
13	mri-pss-state	13	19	12	3	32	7	47	15	0	0	0	0	18	3	0	22	5	0	0	0	3	2	8	12
14	multi-agent-form	30	2	4	13	31	26	32	29	2	0	0	0	4	13	0	28	25	1	0	0	0	0	0	0
15	prod-cell	27	48	11	19	81	55	87	61	0	0	0	0	11	19	0	11	19	0	0	0	0	0	0	0
16	robo-swarm-aggr	2	2	2	4	2	8	2	16	0	0	0	0	2	4	0	2	8	0	0	0	0	0	0	0
17	robo-swarm-clus	3	3	2	6	2	12	2	24	0	0	0	0	2	6	0	2	12	0	0	0	0	0	0	0
18	robo-swarm-form	17	11	6	6	14	22	15	38	0	0	0	0	6	6	0	10	22	0	0	0	0	0	0	0
19	robo-swarm-segr	8	8	6	3	13	8	14	17	0	0	0	0	6	3	0	6	6	0	0	0	0	0	0	0
20	sudoku	64	0	16	0	16	0	64	0	64	0	64	0	16	0	0	16	0	48	16	80	0	0	0	0
21	theme-park	22	25	17	5	36	19	47	77	0	0	0	0	17	5	0	17	5	0	0	0	0	0	0	44
22	wafer-scanner-n1	75	79	26	10	192	26	287	27	7	0	4	0	26	10	0	57	10	0	1	4	0	0	0	31
23	waterway-lock	94	170	71	0	230	0	382	0	0	0	0	0	71	0	0	73	0	0	0	0	0	0	116	198

i_r The number of initial requirement locations. Similarly counted as i_p.

i_c The number of initialization predicates in components.

m_p The number of marked plant locations. Similarly counted as i_p.

m_r The number of marked requirement locations. Similarly counted as i_r.

m_c The number of marker predicates in components. Similarly counted as i_c.

v_n The number of CIF variables.

v_v The number of values for the CIF variables combined. For each variable, the number of potential values in their CIF ranges are counted, and summed up.

$t_{p,s}$ The number of plant state invariants.

$t_{r,s}$ The number of requirement state invariants.

$t_{p,e}$ The number of plant state/event exclusion invariants.

$t_{r,e}$ The number of requirement state/event exclusion invariants.

6. Experimental evaluation of symbolic synthesis performance improvements

To show the cumulative effect of the performance improvements described in Section 4, we perform experiments. Besides the relative delta in performance due to the improvements, we also evaluate the current synthesis performance (the absolute time that synthesis takes). The artifact that accompanies this paper includes the scripts to reproduce the data of all tables and figures from this section of the paper [26].

6.1. Experimental setup

The synthesis performance improvements described in Section 4 cover the changes made to CIF's symbolic synthesis tool between ESCET releases v0.8 and v4.0. We therefore compare the synthesis performance of these two versions of the synthesis tool. This results in a comparison of the cumulative impact of the improvements, which are all enabled by default in ESCET release v4.0, and are thus available out-of-the-box. No other performance improvements are present in v4.0 compared to v0.8, except for ones that need to be explicitly enabled through options, which we do not consider.

We perform synthesis for each benchmark model from Section 5, for both versions of the toolkit. We thereby measure the execution time and memory use. Since synthesis performance is predominantly determined by the creation of BDDs and operations on BDDs, we use BDD-related time and memory metrics [63]. Such metrics are platform independent. This allows the experiments to be repeated on different hardware, regardless of the use of a simple laptop or a supercomputer. It also allows to compare the performance described in this paper with future improvements, even if they will be made years into the future.

To measure time, we count the number of BDD operations performed. Recall that BDD operations are implemented as recursive functions. We count each recursive invocation of such functions, but do not count trivial computations, namely those that can be obtained from the cache and those that require no further recursive processing.

Table 3
Results from the experiments showing the detailed effect of the performance improvements.

Nr	Short name	Time [number of operations]			Memory [maximum number of nodes]		
		v0.8	v4.0	Reduction	v0.8	v4.0	Reduction
1	adas	$2.2 \cdot 10^{05}$	$3.0 \cdot 10^{04}$	7.4	$7.0 \cdot 10^{03}$	$1.5 \cdot 10^{03}$	4.5
2	agv	$1.2 \cdot 10^{06}$	$2.3 \cdot 10^{05}$	5.4	$5.4 \cdot 10^{03}$	$2.5 \cdot 10^{03}$	2.2
3	bcs-dynamic	$2.4 \cdot 10^{07}$	$1.1 \cdot 10^{05}$	223.9	$2.7 \cdot 10^{05}$	$2.9 \cdot 10^{03}$	91.2
4	bcs-static	$6.1 \cdot 10^{08}$	$1.4 \cdot 10^{05}$	4293.6	$3.1 \cdot 10^{06}$	$5.9 \cdot 10^{03}$	518.3
5	bridge	$1.2 \cdot 10^{07}$	$1.6 \cdot 10^{06}$	7.5	$8.4 \cdot 10^{04}$	$2.4 \cdot 10^{04}$	3.5
6	cmt-v1	$3.1 \cdot 10^{06}$	$4.3 \cdot 10^{05}$	7.3	$2.2 \cdot 10^{04}$	$1.6 \cdot 10^{04}$	1.4
7	cmt-v2	$8.0 \cdot 10^{06}$	$3.3 \cdot 10^{06}$	2.4	$2.5 \cdot 10^{04}$	$1.2 \cdot 10^{04}$	2.1
8	cluster-tool	$3.3 \cdot 10^{06}$	$4.7 \cdot 10^{05}$	7.0	$1.2 \cdot 10^{04}$	$7.2 \cdot 10^{03}$	1.7
9	dining-phils	$2.2 \cdot 10^{04}$	$6.5 \cdot 10^{03}$	3.4	$1.0 \cdot 10^{03}$	$4.8 \cdot 10^{02}$	2.2
10	festo	$8.1 \cdot 10^{08}$	$4.8 \cdot 10^{05}$	1672.9	$3.6 \cdot 10^{06}$	$6.1 \cdot 10^{03}$	598.5
11	litho-init	$2.3 \cdot 10^{07}$	$5.7 \cdot 10^{06}$	4.0	$4.0 \cdot 10^{04}$	$3.7 \cdot 10^{04}$	1.1
12	mri-pss-event	$5.6 \cdot 10^{07}$	$1.4 \cdot 10^{07}$	4.1	$5.6 \cdot 10^{05}$	$7.5 \cdot 10^{04}$	7.5
13	mri-pss-state	$7.5 \cdot 10^{04}$	$2.1 \cdot 10^{04}$	3.6	$3.1 \cdot 10^{03}$	$1.0 \cdot 10^{03}$	3.0
14	multi-agent-form	$1.7 \cdot 10^{06}$	$4.0 \cdot 10^{05}$	4.3	$2.6 \cdot 10^{04}$	$2.0 \cdot 10^{04}$	1.3
15	prod-cell	$9.9 \cdot 10^{07}$	$1.3 \cdot 10^{07}$	7.4	$3.0 \cdot 10^{04}$	$1.7 \cdot 10^{04}$	1.8
16	robo-swarm-aggr	$5.5 \cdot 10^{02}$	$3.5 \cdot 10^{02}$	1.5	$1.7 \cdot 10^{02}$	$6.8 \cdot 10^{01}$	2.5
17	robo-swarm-clus	$2.4 \cdot 10^{03}$	$1.2 \cdot 10^{03}$	2.0	$5.7 \cdot 10^{02}$	$1.7 \cdot 10^{02}$	3.4
18	robo-swarm-form	$3.8 \cdot 10^{04}$	$6.8 \cdot 10^{03}$	5.6	$4.3 \cdot 10^{03}$	$6.5 \cdot 10^{02}$	6.6
19	robo-swarm-segr	$1.3 \cdot 10^{04}$	$2.6 \cdot 10^{03}$	5.1	$8.6 \cdot 10^{02}$	$2.5 \cdot 10^{02}$	3.5
20	sudoku	$1.6 \cdot 10^{07}$	$6.7 \cdot 10^{06}$	2.4	$6.1 \cdot 10^{05}$	$5.2 \cdot 10^{05}$	1.2
21	theme-park	$4.1 \cdot 10^{05}$	$2.6 \cdot 10^{04}$	15.3	$1.3 \cdot 10^{04}$	$1.2 \cdot 10^{03}$	11.6
22	wafer-scanner-n1	$1.2 \cdot 10^{09}$	$5.7 \cdot 10^{08}$	2.1	$3.6 \cdot 10^{05}$	$7.9 \cdot 10^{05}$	0.5
23	waterway-lock	$5.5 \cdot 10^{07}$	$6.8 \cdot 10^{06}$	8.1	$9.7 \cdot 10^{04}$	$1.4 \cdot 10^{04}$	7.1
Average (per column)		$1.3 \cdot 10^{08}$	$2.7 \cdot 10^{07}$	**273.7**	$3.9 \cdot 10^{05}$	$6.7 \cdot 10^{04}$	**55.5**
Median (per column)		$3.3 \cdot 10^{06}$	$4.0 \cdot 10^{05}$	**5.4**	$2.5 \cdot 10^{04}$	$6.1 \cdot 10^{03}$	**3.0**

To measure memory, we count the maximum number of BDD nodes in use, during synthesis. We do not count nodes that have been allocated but are no longer in use, waiting to be garbage collected. This gives an accurate measure of the memory usage of synthesis [63].

6.2. Results

The results of the experiments are shown in Table 3. For each benchmark, the time and memory are indicated, for each ESCET version. Additionally, the effort reduction factor is indicated. For instance, a factor of 2.0 means half the number of operations or half the number of nodes, while a factor of 0.5 means twice the number of operations or twice the number of nodes. For o_1 the number of operations for ESCET v0.8, and o_2 the number of operations for ESCET v4.0, the reduction factor is computed as $\frac{o_1}{o_2}$. Similarly, for n_1 the maximum number of nodes for ESCET v0.8, and n_2 the maximum number of nodes for ESCET v4.0, the reduction factor is computed as $\frac{n_1}{n_2}$. A more graphical overview of the same results is shown in Figure 10, where the reduction factors per benchmark are plotted using a scatter plot, using a logarithmic vertical axis.

The time (in number of BDD operations) has reduced in v4.0 compared to v0.8, for all benchmarks. On average, the time is reduced by a factor of 274, more than two orders of magnitude. The median is a reduction factor of only 5.4, since a few benchmarks with much higher reduction factors increase the average quite a bit. The maximum reduction is by a factor of 4,294, but for most models the reduction factor is less than 10. The result thus greatly varies between benchmarks, but overall it takes significantly less time to perform synthesis.

The memory usage (in maximum number of BDD nodes) is on average reduced by a factor of 56, considerably more than an order of magnitude. The median is a reduction factor of only 3.0, since here too a few benchmarks with much higher reduction factors increase the average quite a bit. The maximum reduction is by a factor of 598, but for most models the reduction factor is less than 8. One benchmark has a reduction factor less than one, and thus synthesis actually uses more memory: for the wafer scanner, the maximum number of BDD nodes has approximately doubled. Here too, the result thus greatly varies between benchmarks, but overall it takes significantly less memory to perform synthesis.

That the results differ between benchmarks is not surprising. One reason is that the effort that synthesis needs to spend simply differs for the problem being solved. Another reason is that various improvements are based on heuristics, such as the variable ordering algorithms. The heuristics are tuned to work well for most models, but there can always be models where performance is adversely affected.

CIF's synthesis algorithm can be customized through many settings. This includes configuration of the various heuristics. By experimenting with these settings, the synthesis
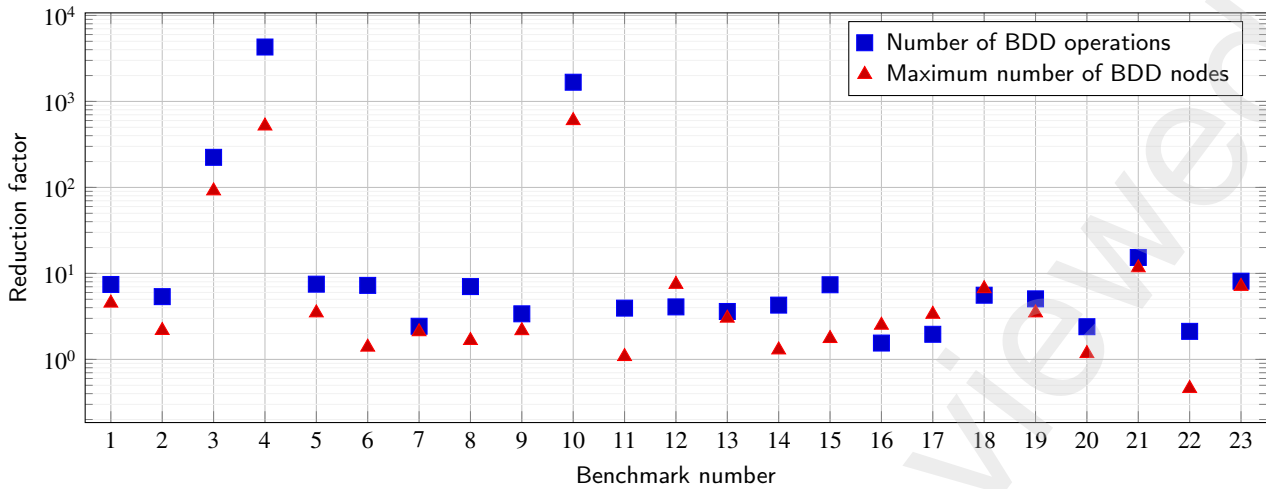
Figure 10: Results from the experiments showing a graphical overview of the effect of the performance improvements.

Table 4

Impact of further optimizing synthesis settings on the performance for the wafer scanner benchmark.

Version	Operations	Nodes
ESCET v0.8	$1.2 \cdot 10^{09}$	$3.6 \cdot 10^{05}$
ESCET v4.0 (default config.)	$5.7 \cdot 10^{08}$	$7.9 \cdot 10^{05}$
ESCET v4.0 (optimized config.)	$3.4 \cdot 10^{07}$	$8.4 \cdot 10^{04}$

performance may be further improved for specific models. For instance, for the wafer scanner benchmark, by further tweaking the variable ordering algorithms, enabling forward reachability, computing the reachable states first (before non-blocking and controllable states), and changing the order in which the edges are applied during reachability computations, we can reduce both the number of BDD operations and the maximum number of used BDD nodes, as shown in Table 4. In this case, the number of BDD operations and the maximum number of BDD nodes are both reduced by about an order of magnitude.

6.3. Current synthesis performance

So far, in this section, we showed the relative improvements to the synthesis performance, as a delta in performance between ESCET v0.8 and v4.0. To conclude this section, we now show the current synthesis performance in v4.0.

We measure the absolute amount of time, using default configuration, to synthesize the benchmarks using ESCET v4.0. This gives an indication on the current state of the efficiency of synthesis for practical application. For each benchmark, synthesis was performed ten times, and the average synthesis time in seconds is computed. Each benchmark is synthesized once before the actual measurements, to warm up the Java Virtual Machine (JVM), resulting in more stable measurements. The experiments are performed on a laptop with a 3 GHz Intel Core i7-1185G7 processor.

Table 5 shows the results. The wafer scanner benchmark takes by far the longest: a bit more than one and a half minute.

Table 5

The absolute time in seconds it takes to synthesize the benchmarks using ESCET v4.0, with default configuration.

Nr	Short name	Time	Nr	Short name	Time
1	adas	0.04	13	mri-pss-state	0.02
2	agv	0.05	14	multi-agent-form	0.08
3	bcs-dynamic	0.08	15	prod-cell	0.98
4	bcs-static	0.08	16	robo-swarm-aggr	0.01
5	bridge	0.30	17	robo-swarm-clus	0.02
6	cmt-v1	0.09	18	robo-swarm-form	0.02
7	cmt-v2	0.34	19	robo-swarm-segr	0.01
8	cluster-tool	0.07	20	sudoku	4.40
9	dining-phils	0.02	21	theme-park	0.04
10	festo	0.14	22	wafer-scanner-n1	97.03
11	litho-init	0.53	23	waterway-lock	0.73
12	mri-pss-event	1.93			

The other benchmarks can each be synthesized within just a few seconds, and most of them even within a second.

6.4. Threats to validity

There is a risk of selection bias, since the benchmarks that we use come for a large part from our own academic community (of which we are a part) and from the companies that we work with. Given the lack of other suitable benchmark sets in the literature, this can't be avoided.

Furthermore, various algorithms involve heuristics that were, to a large degree, tuned using the same benchmarks. To mitigate this risk, additional benchmarks were added later on, such as the MRI and wafer scanner ones.

There is some risk for attrition bias, as for some benchmarks, such as CMT and wafer scanner, lower parameter values were used to ensure the benchmarks could be synthesized within reasonable time in older ESCET releases. As most benchmarks do not have parameters, this significantly reduces this risk. In ESCET v4.0, most benchmarks can be

---

easily synthesized in a short time. It may thus be time to consider adding more challenging benchmarks.

Finally, there is a risk of sampling bias, since we can't guarantee that the benchmarks we've used are representative of the real-world models used in industry. We have partially mitigated this risk by including various industrial models as benchmarks.

7. Non-monolithic synthesis

Despite the performance improvements over the years, and the ones described in Section 4, scalability remains a concern for the practical application of supervisory controller synthesis. For instance, the wafer scanner benchmark model as discussed in Section 5 is used in this paper with parameter $n = 1$. To get the full system behavior, we would need to use $n = 7$. Then seven production wafers and two dummy wafers, for a total of nine wafers, can be in the system at a time. However, the synthesis time and memory usage increase significantly when we increase n. As mentioned in Section 6, the model for $n = 1$ can be synthesized on a laptop in under 2 minutes using the default settings. And with tuning to optimize the settings, we can synthesize it in about 4 seconds. We were able to also synthesize a supervisor for $n = 2$, using the same tuned settings as for $n = 1$, in about 10 minutes. For $n = 3$, again using the same tuned settings, we were able to synthesize a supervisor in a bit more than 8 hours. We have so far not been able to synthesize a supervisor for $n = 4$, even through tuning with different settings, as either synthesis runs out of memory, or is still running after 100 hours. Nonetheless, this is significantly better than what van der Sanden et al. originally reported in 2015 [56]. They were able to synthesize a supervisor for $n = 1$, but they could only go to $n = 2$ by significantly reducing the model's complexity. The artifact that accompanies this paper includes the models and scripts we used to synthesize the wafer scanner benchmark for $1 \leq n \leq 3$ [26].

While we believe further performance improvements to be possible, the question is whether they would be sufficient to resolve the seemingly exponentially increasing synthesis times for the wafer scanner benchmark. So far, we have only discussed monolithic supervisory controller synthesis, where based on the plants and the requirements a single supervisory controller is synthesized. An alternative is to use non-monolithic synthesis approaches, that synthesize multiple supervisors for parts of the system, which together ensure the correct and safe operation of the system. Various techniques have been proposed over the years, such as modular [37, 46, 73], decentralized [31, 32, 55], hierarchical [70, 74] and compositional [18] synthesis, as well as combinations of them [27, 57].

Relatively recent is multilevel synthesis [29], where the plant components and requirements are automatically grouped together into a hierarchical structure, the multilevel system, and a separate supervisor is synthesized for each group of this multilevel system. This allows to distribute the control problem over multiple cooperating supervisors, which together are significantly smaller than one monolithic supervisor. By encoding relations between plant components and requirements in a design structure matrix (DSM), and algorithmically reordering its rows and columns to place tightly coupled plant components side by side [69], a suitable multilevel structure can be obtained (semi-)automatically. Compared to monolithic synthesis, this can for certain systems substantially reduce synthesis effort [25], enabling synthesis for much larger variants of such systems.

To show how much multilevel synthesis can help in practice, we use again the set of benchmark models from Section 5. We use the in-development multilevel synthesis tool from ESCET v4.0 to generate specifications for the various groups of the multilevel system. We then use ESCET v4.0 to synthesize a supervisor for each of the groups. The artifact that accompanies this paper includes the scripts we used for this experiment [26]. Out of 23 benchmarks, currently only 11 of them are supported by CIF's multilevel synthesis tool. The other 12 benchmarks are not yet supported, because they have more than one initial state, have plant invariants, have state requirement invariants, have no requirements, or have marker predicates in components. To gauge the reduction in synthesis effort for the supported benchmarks, we compare the same BDD metrics as in Section 6, namely the number of BDD operations performed, and the maximum number of BDD nodes used. For multilevel synthesis, we sum up the number of operations for the multiple syntheses for different parts of the system, and we take the maximum for the number of nodes. The results are presented in Table 6. In the third column, the number of groups in the multilevel system is shown. Similar to Section 6, we again indicate a reduction factor.

For all benchmarks, the total number of operations and maximum number of nodes decreases or stays the same, when using multi-level synthesis. For the MRI patient support system (event-based), there is a reduction of over 42 times in the number of operations, and a reduction of 12 times in the maximum number of nodes. For some of the other benchmarks, there is also a significant reduction in effort, while for some the benefits are small. Interestingly, for the production cell, no reduction is achieved even though the multi-level system consists of 11 groups. For two benchmarks, the multilevel system has only one group, and thus multilevel synthesis effectively amounts to monolithic synthesis.

To make efficient use of this technique, the plant components and requirements can be first split as much as possible, to avoid the emergence of large groups [24]. Furthermore, 'bus' components, that have connections to many other components, lead to clustering all these components and the associated requirements in the same group. To address this challenge, 'bus' components can be treated separately [23]. Applying these improvements is left as future work.

It is also future work to evaluate other non-monolithic synthesis approaches on our benchmarks.

Table 6

Results of multilevel synthesis compared to monolithic synthesis, for supported benchmarks.

Nr	Short name	Nr. of groups	Number of operations			Maximum number of nodes		
			Monolithic	Multilevel	Reduction	Monolithic	Multilevel	Reduction
1	adas	34	$3.0 \cdot 10^{04}$	$2.1 \cdot 10^{04}$	1.4	$1.5 \cdot 10^{03}$	$1.1 \cdot 10^{03}$	1.4
2	agv	8	$2.3 \cdot 10^{05}$	$3.3 \cdot 10^{04}$	6.9	$2.5 \cdot 10^{03}$	$6.6 \cdot 10^{02}$	3.8
8	cluster-tool	1	$4.7 \cdot 10^{05}$	$4.7 \cdot 10^{05}$	1.0	$7.2 \cdot 10^{03}$	$7.2 \cdot 10^{03}$	1.0
12	mri-pss-event	7	$1.4 \cdot 10^{07}$	$3.3 \cdot 10^{05}$	42.2	$7.5 \cdot 10^{04}$	$6.2 \cdot 10^{03}$	12.1
15	prod-cell	11	$1.3 \cdot 10^{07}$	$1.3 \cdot 10^{07}$	1.0	$1.7 \cdot 10^{04}$	$1.7 \cdot 10^{04}$	1.0
16	robo-swarm-aggr	3	$3.5 \cdot 10^{02}$	$1.8 \cdot 10^{02}$	2.0	$6.8 \cdot 10^{01}$	$2.6 \cdot 10^{01}$	2.6
17	robo-swarm-clus	3	$1.2 \cdot 10^{03}$	$5.9 \cdot 10^{02}$	2.0	$1.7 \cdot 10^{02}$	$5.3 \cdot 10^{01}$	3.2
18	robo-swarm-form	9	$6.8 \cdot 10^{03}$	$5.0 \cdot 10^{03}$	1.3	$6.5 \cdot 10^{02}$	$4.4 \cdot 10^{02}$	1.5
19	robo-swarm-segr	9	$2.6 \cdot 10^{03}$	$1.8 \cdot 10^{03}$	1.4	$2.5 \cdot 10^{02}$	$1.4 \cdot 10^{02}$	1.8
21	theme-park	22	$2.6 \cdot 10^{04}$	$8.2 \cdot 10^{03}$	3.2	$1.2 \cdot 10^{03}$	$5.6 \cdot 10^{02}$	2.1
22	wafer-scanner-n1	1	$5.7 \cdot 10^{08}$	$5.7 \cdot 10^{08}$	1.0	$7.9 \cdot 10^{05}$	$7.9 \cdot 10^{05}$	1.0

8. Conclusions and future work

The Eclipse Supervisory Control Engineering Toolkit (ESCET) toolkit supports synthesis-based engineering (SBE) primarily through the CIF language and tools. In this paper, we described CIF's symbolic synthesis algorithm, including aspects that are often omitted in the literature, but are of great practical relevance, such as the prevention of runtime errors, handling different types of requirements, and supporting input variables. We also introduced and describe CIF's set of benchmark models, a collection of 23 industrial and academic models of various sizes and complexities, that are freely available. Together, they allow researchers to more easily improve the synthesis algorithm, and evaluate those improvements.

We also described recent improvements between ESCET versions v0.8 and v4.0 that affect synthesis performance, and evaluated them on our set of benchmark models. This showed synthesis performance has been significantly improved. We then showed the current practical synthesis performance of CIF, which is sufficient in most cases. However, for some models, still further improvements are needed. We evaluated multi-level synthesis, a form of non-monolithic synthesis, showing it can help to reduce the synthesis effort. But this too, is not always sufficient.

As future work, we consider evaluation of multi-level synthesis with split requirements and better handling of 'bus' components, as well as the evaluation of other non-monolithic synthesis approaches on our benchmarks.

9. Data-Availability Statement

The artifact accompanying this paper includes the models and scripts that can be used to reproduce various results from this paper [26]. The artifact is available at Zenodo under identifier doi:10.5281/zenodo.13358811 [26]. Eclipse ESCET v4.0, which is used for most of the experiments, can be downloaded at `https://eclipse.dev/escet/v4.0/download.html`. Eclipse ESCET v0.8, against which we compare v4.0, can be downloaded at `https://eclipse.dev/escet/v0.8/download.html`. The latest version of Eclipse ESCET can be downloaded at `https://eclipse.dev/escet/download.html`, but should not be used to reproduce the results from this paper.

References

[1] Agut, D.E.N., Reniers, M.A., Schiffelers, R.R.H., Jørgensen, K.Y., van Beek, D.A., 2011. A semantic-preserving transformation from the Compositional Interchange Format to UPPAAL. 18th IFAC World Congress, IFAC Proceedings Volumes 44, 12496–12502. doi:10.3182/20110828-6-IT-1002.03030.

[2] Aloul, F.A., Markov, I.L., Sakallah, K.A., 2003. Force: a fast and easy-to-implement variable-ordering heuristic, in: Proceedings of the 13th ACM Great Lakes Symposium on VLSI, Association for Computing Machinery. p. 116–119. doi:10.1145/764808.764839.

[3] Baubekova, M.M., van Eldik, K.J., van de Mortel-Fronczak, J.M., Fokkink, W.J., Rooda, J.E., 2024. SBE configurator: A model generation tool for synthesis of ship lock supervisors, in: 17th Workshop on Discrete Event Systems (WODES'24), pp. 288–293. doi:10.1016/j.ifacol.2024.07.049.

[4] van Beek, D.A., Fokkink, W.J., Hendriks, D., Hofkamp, A.T., Markovski, J., van de Mortel-Fronczak, J.M., Reniers, M.A., 2014. CIF 3: Model-based engineering of supervisory controllers, in: Proc. 20th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer. pp. 575–580. doi:10.1007/978-3-642-54862-8_48.

[5] ter Beek, M.H., Reniers, M.A., de Vink, E.P., 2016. Supervisory controller synthesis for product lines using CIF 3, in: Margaria, T., Steffen, B. (Eds.), Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I, pp. 856–873. doi:10.1007/978-3-319-47166-2_59.

[6] Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M., 2006. UPPAAL 4.0, in: Proc. 3rd Conference on the Quantitative Evaluation of Systems (QEST), IEEE. pp. 125–126. doi:10.1109/QEST.2006.59.

[7] Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A.J., Willemse, T.A.C., 2019. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability, in: Proc. 25th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer. pp. 21–39. doi:10.1007/978-3-030-17465-1_2.

[8] Burch, J., Clarke, E., Long, D., McMillan, K., Dill, D., 1994. Symbolic model checking for sequential circuit verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13, 401–424. doi:10.1109/43.275352.

[9] Burch, J.R., Clarke, E.M., Long, D.E., 1991. Symbolic model checking with partitioned transition relations, in: VLSI.

[10] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J., 1992. Symbolic model checking: 10^{20} states and beyond. Information and Computation 98, 142–170. doi:10.1016/0890-5401(92)90017-A.

[11] Cai, K., Wonham, W.M., 2015. New results on supervisor localization, with case studies. Discrete Event Dynamic Systems 25, 203–226. doi:10.1007/s10626-014-0194-6.

[12] Coudert, O., Madre, J., 1990. A unified framework for the formal verification of sequential circuits, in: 1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers. IEEE Computer Society, pp. 126–129. doi:10.1109/ICCAD.1990.129859.

[13] Cuthill, E., McKee, J., 1969. Reducing the bandwidth of sparse symmetric matrices, in: Proceedings of the 1969 24th National Conference, Association for Computing Machinery, New York, NY, USA. pp. 157–172. doi:10.1145/800195.805928.

[14] Eppinger, S.D., Browning, T.R., 2012. Design Structure Matrix Methods and Applications. MIT Press.

[15] Fei, Z., Miremadi, S., Åkesson, K., Lennartson, B., 2014. Efficient symbolic supervisor synthesis for extended finite automata. IEEE Transactions on Control Systems Technology 22, 2368–2375. doi:10.1109/TCST.2014.2303134.

[16] Feng, L., Cai, K., Wonham, W.M., 2009. A structural approach to the non-blocking supervisory control of discrete-event systems. The International Journal of Advanced Manufacturing Technology 41, 1152–1168. doi:10.1007/s00170-008-1555-9.

[17] Feng, L., Wonham, W.M., 2006. TCT: A computation tool for supervisory control synthesis, in: Proc. 8th Workshop on Discrete Event Systems (WODES), IEEE. pp. 388–389. doi:10.1109/WODES.2006.382399.

[18] Flordal, H., Malik, R., Fabian, M., Åkesson, K., 2007. Compositional synthesis of maximally permissive supervisors using supervision equivalence. Discrete Event Dynamic Systems 17, 475–504. doi:10.1007/s10626-007-0018-z.

[19] Fokkink, W.J., Goorden, M.A., Hendriks, D., van Beek, D.A., Hofkamp, A.T., Reijnen, F.F.H., et al., 2023a. Eclipse ESCET™: The Eclipse supervisory control engineering toolkit, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Springer. pp. 44–52. doi:10.1007/978-3-031-30820-8_6.

[20] Fokkink, W.J., Goorden, M.A., Hendriks, D., van de Mortel-Fronczak, J.M., Oortwijn, W., Rooda, J.E., 2023b. Synthesis-based engineering of supervisory controllers. Mikroniek 6, 22–26.

[21] Fokkink, W.J., Goorden, M.A., van de Mortel-Fronczak, J.M., Reijnen, F.F.H., Rooda, J.E., 2022. Supervisor synthesis: Bridging theory and practice. Computer 55, 48–54. doi:10.1109/MC.2021.3134934.

[22] Forschelen, S.T.J., van de Mortel-Fronczak, J.M., Su, R., Rooda, J.E., 2012. Application of supervisory control theory to theme park vehicles. Discrete Event Dynamic Systems 22, 511–540. doi:10.1007/s10626-012-0130-6.

[23] Goorden, M.A., Dingemans, C., Reniers, M.A., van de Mortel-Fronczak, J.M., Fokkink, W.J., Rooda, J.E., 2019a. Supervisory control of multilevel discrete-event systems with a bus structure, in: Proc. 17th European Control Conference (ECC), IEEE. pp. 3204–3211. doi:10.23919/ECC.2019.8795835.

[24] Goorden, M.A., van de Mortel-Fronczak, J.M., Reniers, M.A., Fokkink, W.J., Rooda, J.E., 2019b. The impact of requirement splitting on the efficiency of supervisory control synthesis, in: Proc. 24th Conference on Formal Methods for Industrial Critical Systems (FMICS), Springer. pp. 76–92. doi:10.1007/978-3-030-27008-7_5.

[25] Goorden, M.A., van de Mortel-Fronczak, J.M., Reniers, M.A., Fokkink, W.J., Rooda, J.E., 2020. Structuring multilevel discrete-event systems with dependence structure matrices. IEEE Transactions on Automatic Control 65, 1625–1639. doi:10.1109/TAC.2019.2928119.

[26] Hendriks, D., Reniers, M.A., Fokkink, W.J., Oortwijn, W., 2024. Artifact for the paper 'Overview and performance evaluation of supervisory controller synthesis with Eclipse ESCET v4.0'. doi:10.5281/zenodo.13358811.

[27] Hill, R., Tilbury, D., 2006. Modular supervisory control of discrete-event systems with abstraction and incremental hierarchical construction, in: 2006 8th International Workshop on Discrete Event Systems, pp. 399–406. doi:10.1109/WODES.2006.382507.

[28] Hoare, C.A.R., 1985. Communicating sequential processes. volume 178. Prentice-hall.

[29] Komenda, J., Masopust, T., van Schuppen, J.H., 2016. Control of an engineering-structured multilevel discrete-event system, in: Proc. 13th Workshop on Discrete Event Systems (WODES), IEEE. pp. 103–108. doi:10.1109/WODES.2016.7497833.

[30] Korssen, T., Dolk, V., van de Mortel-Fronczak, J.M., Reniers, M.A., Heemels, M., 2018. Systematic model-based design and implementation of supervisors for advanced driver assistance systems. IEEE Transactions on Intelligent Transportation Systems 19, 533–544. doi:10.1109/TITS.2017.2776354.

[31] Lee, S.H., Wong, K.C., 2002. Structural decentralised control of concurrent discrete-event systems. European Journal of Control 8, 477–491. doi:10.3166/ejc.8.477-491.

[32] Lin, F., Wonham, W., 1988. Decentralized supervisory control of discrete-event systems. Information Sciences 44, 199–224. doi:10.1016/0020-0255(88)90002-3.

[33] Lopes, Y.K., Trenkwalder, S.M., Leal, A.B., Dodd, T.J., Groß, R., 2016. Supervisory control theory applied to swarm robotics. Swarm Intelligence 10, 65–97. doi:10.1007/s11721-016-0119-0.

[34] Lousberg, S., Thuijsman, S.B., Reniers, M.A., 2020. DSM-based variable ordering heuristic for reduced computational effort of symbolic supervisor synthesis. IFAC-PapersOnLine 53, 429–436. doi:10.1016/j.ifacol.2021.04.058.

[35] Ma, C., Wonham, W.M., 2008. STSLib and its application to two benchmarks, in: 2008 9th International Workshop on Discrete Event Systems, pp. 119–124. doi:10.1109/WODES.2008.4605932.

[36] Malik, R., Åkesson, K., Flordal, H., Fabian, M., 2017. Supremica–an efficient tool for large-scale discrete event systems. IFAC-PapersOnLine 50, 5794–5799. doi:10.1016/j.ifacol.2017.08.427.

[37] Malik, R., Teixeira, M., 2016. Modular supervisor synthesis for extended finite-state machines subject to controllability, in: 2016 13th International Workshop on Discrete Event Systems (WODES), pp. 91–96. doi:10.1109/WODES.2016.7497831.

[38] Markovski, J., van Beek, D.A., Theunissen, R.J.M., Jacobs, K.G.M., Rooda, J.E., 2010. A state-based framework for supervisory control synthesis and verification, in: Proc. 49th IEEE Conference on Decision and Control (CDC), pp. 3481–3486. doi:10.1109/CDC.2010.5717095.

[39] McMillan, K.L., 1993. Symbolic Model Checking. Springer. doi:10.1007/978-1-4615-3190-6.

[40] Meijer, J., van de Pol, J.C., 2016. Bandwidth and wavefront reduction for static variable ordering in symbolic reachability analysis, in: Rayadurgam, S., Tkachuk, O. (Eds.), NASA Formal Methods, Springer International Publishing, Cham. pp. 255–271. doi:10.1007/978-3-319-40648-0_20.

[41] Miremadi, S., Åkesson, K., Lennartson, B., 2008. Extraction and representation of a supervisor using guards in extended finite automata, in: Proc. 9th Workshop on Discrete Event Systems (WODES), IEEE. pp. 193–199. doi:10.1109/WODES.2008.4605944.

[42] Miremadi, S., Akesson, K., Lennartson, B., 2011. Symbolic computation of reduced guards in supervisory control. IEEE Transactions on Automation Science and Engineering 8, 754–765. doi:10.1109/TASE.2011.2146249.

[43] Moor, T., Schmidt, K., Perk, S., 2008. libFAUDES — an open source C++ library for discrete event systems, in: Proc. 9th Workshop on Discrete Event Systems (WODES), IEEE. pp. 125–130. doi:10.1109/WODES.2008.4605933.

[44] Nadales Agut, D., Reniers, M.A., 2011. Linearization of CIF through SOS. Electronic Proceedings in Theoretical Computer Science 64, 74–88. doi:10.4204/eptcs.64.6.

[45] Ouedraogo, L., Kumar, R., Malik, R., Åkesson, K., 2011. Nonblocking and safe control of discrete-event systems modeled as extended finite automata. IEEE Transactions on Automation Science and Engineering 8, 560–569. doi:10.1109/TASE.2011.2124457.

[46] de Queiroz, M.H., Cury, J.E.R., 2000. Modular control of composed systems, in: Proceedings of the 2000 American Control Conference (ACC), pp. 4051–4055 vol.6. doi:10.1109/ACC.2000.876983.

[47] Ramadge, P.J., Wonham, W.M., 1987. Supervisory control of a class of discrete event processes. SIAM Journal on Control and Optimization 25, 206–230. doi:10.1137/0325013.

[48] Reijnen, F.F.H., Goorden, M.A., van de Mortel-Fronczak, J.M., Reniers, M.A., Rooda, J.E., 2018a. Application of dependency structure matrices and multilevel synthesis to a production line, in: 2018 IEEE Conference on Control Technology and Applications (CCTA), pp. 458–464. doi:10.1109/CCTA.2018.8511449.

[49] Reijnen, F.F.H., Goorden, M.A., van de Mortel-Fronczak, J.M., Rooda, J.E., 2017. Supervisory control synthesis for a waterway lock, in: 2017 IEEE Conference on Control Technology and Applications (CCTA), pp. 1562–1563. doi:10.1109/CCTA.2017.8062679.

[50] Reijnen, F.F.H., Hofkamp, A.T., van de Mortel-Fronczak, J.M., Reniers, M.A., Rooda, J.E., 2019. Finite response and confluence of state-based supervisory controllers, in: 2019 IEEE 15th International Conference on Automation Science and Engineering (CASE), pp. 509–516. doi:10.1109/COASE.2019.8843335.

[51] Reijnen, F.F.H., Reniers, M.A., van de Mortel-Fronczak, J.M., Rooda, J.E., 2018b. Structured synthesis of fault-tolerant supervisory controllers. 10th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes (SAFEPROCESS 2018), IFAC-PapersOnLine 51, 894–901. doi:10.1016/j.ifacol.2018.09.681.

[52] Reniers, M.A., Keiren, J.J.A., 2024. Validation of supervisory control synthesis tool CIF using model checker mCRL2, in: Proceedings CASE. Accepted for publication.

[53] Reniers, M.A., van de Mortel-Fronczak, J.M., 2018. An engineering perspective on model-based design of supervisors. IFAC-PapersOnLine 51, 257–264. doi:10.1016/j.ifacol.2018.06.310.

[54] Ricker, L., Lafortune, S., Genc, S., 2006. DESUMA: A tool integrating GIDDES and UMDES, in: Proc. 8th Workshop on Discrete Event Systems (WODES), IEEE. pp. 392–393. doi:10.1109/WODES.2006.382402.

[55] Rudie, K., Wonham, W.M., 1991. Think globally, act locally: Decentralized supervisory control, in: 1991 American Control Conference, pp. 898–903. doi:10.23919/ACC.1991.4791508.

[56] van der Sanden, L.J., Reniers, M.A., Geilen, M.C.W., Basten, A.A., Jacobs, J., Voeten, J.P.M., Schiffelers, R.R.H., 2015. Modular model-based supervisory controller design for wafer logistics in lithography machines, in: Proc. 18th Conference on Model Driven Engineering Languages and Systems (MODELS), IEEE. pp. 416–425. doi:10.1109/MODELS.2015.7338273.

[57] Schmidt, K., Reger, J., Moor, T., 2004. Hierarchical control for structural decentralized DES. 7th International Workshop on Discrete Event Systems (WODES'04), IFAC Proceedings Volumes 37, 279–284. doi:10.1016/S1474-6670(17)30759-0.

[58] Sköldstam, M., Åkesson, K., Fabian, M., 2007. Modeling of discrete event systems using finite automata with variables, in: 2007 46th IEEE Conference on Decision and Control, pp. 3387–3392. doi:10.1109/CDC.2007.4434894.

[59] Sloan, S., 1989. A FORTRAN program for profile and wavefront reduction. International Journal for Numerical Methods in Engineering 28, 2651–2679. doi:10.1016/j.ifacol.2021.04.058.

[60] Su, R., van Schuppen, J.H., Rooda, J.E., 2010. Aggregative synthesis of distributed supervisors based on automaton abstraction. IEEE Transactions on Automatic Control 55, 1627–1640. doi:10.1109/TAC.2010.2042342.

[61] Theunissen, R.J.M., 2015. Supervisory Control in Health Care Systems. Ph.D. thesis. Eindhoven University of Technology, The Netherlands.

[62] Theunissen, R.J.M., Petreczky, M., Schiffelers, R.R.H., van Beek, D.A., Rooda, J.E., 2014. Application of supervisory control synthesis to a patient support table of a magnetic resonance imaging scanner. IEEE Transactions on Automation Science and Engineering (TASE) 11, 20–32. doi:10.1109/TASE.2013.2279692.

[63] Thuijsman, S.B., Hendriks, D., Theunissen, R.J.M., Reniers, M.A., Schiffelers, R.R.H., 2019. Computational effort of BDD-based supervisor synthesis of extended finite automata, in: Proc. 15th International Conference on Automation Science and Engineering (CASE), pp. 486–493. doi:10.1109/COASE.2019.8843327.

[64] Thuijsman, S.B., Reniers, M.A., 2023. Supervisory control for dynamic feature configuration in product lines. ACM Transactions on Embedded Computing Systems doi:10.1145/3579644.

[65] Thuijsman, S.B., Reniers, M.A., Hendriks, D., 2021. Efficiently enforcing mutual state exclusion requirements in symbolic supervisor synthesis, in: 2021 IEEE 17th International Conference on Automation Science and Engineering (CASE), pp. 777–783. doi:10.1109/CASE49439.2021.9551593.

[66] Vahidi, A., Lennartson, B., Fabian, M., 2005. Efficient analysis of large discrete-event systems with binary decision diagrams, in: Proceedings of the 44th IEEE Conference on Decision and Control, pp. 2751–2756. doi:10.1109/CDC.2005.1582579.

[67] Van Dijk, T., van de Pol, J.C., 2017. Sylvan: multi-core framework for decision diagrams. International Journal on Software Tools for Technology Transfer 19, 675–696. doi:10.1007/s10009-016-0433-2.

[68] Vos, Z., 2020. Initialization and Termination of Flexible Manufacturing Systems: a formal approach. Master's thesis. Eindhoven University of Technology.

[69] Wilschut, T., Etman, L.F.P., Rooda, J.E., Adan, I.J.B.F., 2017. Multilevel flow-based markov clustering for design structure matrices. Journal of Mechanical Design 139. doi:10.1115/1.4037626.

[70] Wong, K.C., Wonham, W.M., 1996. Hierarchical control of discrete-event systems. Discrete Event Dynamic Systems 6, 241–273. doi:10.1007/BF01797154.

[71] Wonham, W.M., Cai, K., 2019. Supervisory Control of Discrete-Event Systems. Springer. doi:10.1007/978-3-319-77452-7.

[72] Wonham, W.M., Cai, K., Rudie, K., 2018. Supervisory control of discrete-event systems: A brief history. Annual Reviews in Control 45, 250–256. doi:10.1016/j.arcontrol.2018.03.002.

[73] Wonham, W.M., Ramadge, P.J., 1988. Modular supervisory control of discrete-event systems. Mathematics of Control, Signals and Systems 1, 13–30. doi:10.1007/BF02551233.

[74] Zhong, H., Wonham, W., 1990. On the consistency of hierarchical supervision in discrete-event systems. IEEE Transactions on Automatic Control 35, 1125–1134. doi:10.1109/9.58555.