Symbolic Reduction of Guards in Supervisory Control Using Genetic Algorithms

S. Miremadi and A. Voronov
Automation Research Group, Department of Signals and Systems
Chalmers University of Technology
SE-412 96 Gothenburg, Sweden
{miremads, voronov}@chalmers.se

***Abstract*—In the supervisory control theory, a supervisor is generated based on given plant and specification models. The supervisor restricts the plant in order to fulfill the specifications. A problem that is typically encountered in industrial applications is that the resulting supervisor is not easily comprehensible for the users. To tackle this problem, we previously introduced an efficient method to characterize a supervisor by logic conditions, referred to as *guards*, generated from the models. The guards express under which conditions an event is allowed to occur to fulfill the specifications. By exploiting the structure of the given models, some techniques were introduced to simplify the guards and make them tractable. In order to be able to handle complex systems efficiently, the models are symbolically represented by *binary decision diagrams* and thus all computations are performed symbolically as well. As a consequence, the size of the guards becomes very sensitive to the variable ordering in the BDDs. In this paper, by using genetic algorithms, we aim to find the optimal variable ordering for the BDDs that yields the most compact guard. The approach has been implemented in a supervisory control tool and applied to an academic example example.**

***Index Terms*—Supervisory control theory, deterministic finite automata, symbolic representation, binary decision diagrams, propositional formula, genetic algorithms.**

I. Introduction

When designing control functions for discrete event systems, a model-based approach may be used to conveniently understand the system's behavior, easily apply different modifications, and decrease the testing and debugging time. A well known example of such a model-based approach is, supervisory control theory (SCT) [1]. Having a plant (the system to be controlled) and a specification, SCT automatically synthesizes a control function, called *supervisor*, that restricts the conduct of the plant to ensure that the system never violates the given specification. SCT has various applications in different areas such as automated manufacturing and embedded systems, e.g., [2]–[4].

Generally, a supervisor is a function that, given a set of events, restricts the plant to execute some events towards the specification. A typical issue is how to realize such a control function efficiently and represent it lucidly for the users. A standard approach is to first synthesize the supervisor and then explicitly represent all the states that are allowed to be reached in the closed-loop system. However, such an approach has some drawbacks. For instance, a supervisor with a huge number of states may require more memory than available. Furthermore, from a user perspective, such supervisors are not tractable. More specifically, the users retrieve the final supervisor as a black box, without clearly understanding why some events become disabled after synthesis. Finally, for complex systems, exploring all reachable states while synthesizing the supervisor is computationally expensive, due to the state-space explosion problem.

An alternative approach is to represent the supervisor symbolically using binary decision diagrams (BDDs) [5]; useful data structures (directed acyclic graphs) for representing Boolean functions. BDDs can be used to compactly and effectively represent a huge state space [2], [6], [7]. Even if the number of states is large, the number of nodes in its corresponding BDD can be relatively manageable. In [6] a supervisor is synthesized in a few minutes for a transfer line example with more than 10^{200} states. This is possible due to a special partitioning of the involved BDDs. Nevertheless, since this approach reformulates and encodes the system's original model, it is cumbersome for the users to understand the resulting supervisor. It is more convenient and natural to represent the supervisor in a form similar to the models that were fed into the synthesis initially.

By considering the theoretical description of a supervisor, it can be represented as a function that restricts the execution of events in the plant in order to satisfy the specification. These restrictions can be expressed as logic conditions in form of propositional formulae extracted from the states of the supervisor. We refer to such logic conditions as *guards*. Intuitively, guards can be comprehended easily. However, in some cases the guards could become large and intractable. To tackle this problem, it is possible to minimize the guard expressions using standard minimization methods of Boolean functions such as Quine-McCluskey [8]; but such methods are inefficient for large systems, where a more attractive approach is to perform an approximate minimization in a symbolic manner using BDDs.

There are a number of papers which have tackled the above-mentioned issues. In [9], an implementation of decentralized supervisory control is presented. This is performed "by embedding the control map in the plant's local Finite State Machines and employing private sets of Boolean variables to encode the control information for each component supervisor" [9]. Although this process will assist the simplicity and clearness of the supervisors, the main focus is to solve the problem of decentralized communicating controllers and not much attention is paid on how to reduce the final Boolean formulae

for more complex systems.

Another class of approaches for supervisory synthesis, based on linear algebraic representation of Petri net models of the plants, has been presented in [10]–[13]. In these methods, the specifications are added to the plants in the form of linear predicates, which can be considered as constraint conditions. The resulting controller can also be formulated in a similar way as suggested in this paper. However, each approach has some restrictions. The non-blocking problem is not considered in [10]. In addition, in order to employ this approach, the system should satisfy a particular structural condition: the uncontrollable subnet extracted from the Petri net model must be loop free. In [11] the liveness problem is considered but only for controlled marked graphs. The approach proposed in [12] is applicable if the supervisory net has a convex reachability set. The focus is mainly on efficient automatic verification. In [13] the request for a minimally restrictive supervisor is abandoned, in favor of a more easily computed but also more restrictive control function.

In [14], the supervisor is represented as a set of control functions expressed by BDDs, which is relatively close to our approach. Each control function is connected to an event, which specifies when the event is allowed to be executed in order to satisfy the specifications. The system is modeled by hierarchal models called state tree structures. However, as stated earlier, for users not familiar with BDDs, having the supervisor as a number of BDDs would be hard to understand. Even for users familiar with BDDs, the tree-structured models typically become comprehensible for systems where the plants and specifications consist of a limited number of states. Otherwise, it would be complicated for the users to relate the BDD-variables to the state trees. Furthermore, it is possible to obtain simpler conditions in terms of guards by utilizing the structures of the original models. Besides, the major focus in [14] is to design a nonblocking supervisor for huge systems, rather than generating comprehensible guards added to the original automata for characterizing the supervisor.

Andersson et al. in [4] propose an algorithm for manufacturing cell controllers to extract the relations between the desired operations in the cell from the supervisor. The main advantage of these relations is to give an easy-to-read representation of the control function, and make the method usable in an industrial setting. However, their approach can merely be applied to models with a special structure and the method is not suitable for large systems. The problem formulation tackled in our paper is inspired from [4].

Previously, in [15] we presented an approach to characterize a supervisor by a set of reduced and tractable guards, which are generated based on states of the original plant and specification models. The guards can then be attached to the original models and restrict the plants' behavior. To obtain more simplified guards, a set of don't-care states were exploited and some heuristic techniques were applied, which were shown to be crucial in the reduction of the generated guards. To tackle large and complicated problems, all computations are based on BDDs. The work in [15] has some main features. The final representation, i.e., the guards, will preserve all the properties of the supervisor. The method is applicable to any system and no structural conditions are required. The supervisor is represented by a set of guards, that are understandable for the users, rather than other data structures such as BDDs. In addition, the guards can be easily implemented in a programmable logic controller.

This work is a continuation of [15]. In [15] since the guards are directly generated from the BDDs, the size of the guards is significantly sensitive to the variables ordering of the BDDs. In this paper, by using genetic algorithms, we aim to find the optimal variable ordering for the BDDs that yields the most compact guard. Usually, more compact guards are more comprehensible for the users. We have applied the method to an example representing a real car manufacturing cell and compared the results with the results of [15].

This paper is organized as follows: Section II provides some preliminaries including extended finite automata and binary decision diagrams, which are the fundamental concepts in our work. Supervisory control theory is briefly described in Section III. In Section IV, we show how the guards are generated and how they can be used to represent the supervisor modularly by the original EFAs. Section V explains how the computed guards can be reduced using genetic algorithms. The whole process is applied to a case study in Section VI. Finally, Section VII provides some conclusions and suggestions for future work.

II. Preliminaries

This section provides some preliminaries that are used throughout this paper.

A. Extended Finite Automata

The modeling formalism used in this work is *Extended Finite Automaton (EFA)* [16], which is an augmentation of an ordinary finite automaton with discrete variables.

Definition II.1 (Extended Finite Automaton)**.**
An extended finite-state automaton E is a 6-tuple

$$\mathrm{E} = (L, D, \Sigma, \rightarrow, L^0, D^0, L^m, D^m),$$

where

- L is a finite set of locations,
- $D = D_1 \times \ldots \times D_n$ is the domain of n variables $\mathcal{V} = \{v_1, \ldots, v_n\}$, where $D_i \subseteq \mathbb{Z}$,
- Σ is a nonempty finite set of events,
- $\rightarrow \subseteq L \times \Sigma \times \mathcal{G} \times \mathcal{A} \times L$ is the transition relation,
- $L^0 \subseteq L$ is the set of initial locations,
- $D^0 = D_1^0 \times \ldots \times D_n^0$ is the set of initial values of the variables,
- $L^m \subseteq L$ is the set of marked locations that are desired to be reached, and
- D^m is the marked valuations of the variables,

where $\mathcal{G}$ and $\mathcal{A}$ is sets of constraining expressions, called *guards*, and updating functions, called *actions*, respectively.

The guards and actions are associated to the transitions of the automaton. A transition in an EFA is executed if and only if its corresponding event occurs and its corresponding guard

becomes satisfied, which may follow by updates of a set of variables.

In the EFA framework, an arithmetic expression φ is formed according to the grammar

$$\varphi ::= \omega \mid v \mid (\varphi) \mid \varphi + \varphi \mid \varphi - \varphi \mid \varphi * \varphi \mid \varphi / \varphi \mid \varphi \% \varphi,$$

where $v \in \mathcal{V}$ and $\omega \in \bigcup_{i=1}^{n} D_i$. We denote an *evaluation* by $\mu \in D$, meaning that a value is assigned to each variable.

A guard $g \in \mathcal{G}$ is a propositional expression formed according to the grammar

$$\begin{aligned} g \ ::= \ & \varphi < \varphi \mid \varphi \leq \varphi \mid \varphi > \varphi \mid \varphi \geq \varphi \mid \varphi == \varphi \mid \\ & (g) \mid g \wedge g \mid g \vee g \mid \top \mid \bot, \end{aligned}$$

where $\top$ and $\bot$ represent boolean logic `true` and `false`, respectively. All nonzero values are considered as $\top$. The semantics of a guard g is specified by a *satisfaction relation* $\models$, the evaluation μ for which of guard g is $\top$; written $\mu \models g$.

An action $\mathbf{a} \in \mathcal{A}$ is an n-tuple of functions $(a_1, \ldots, a_n)$, updating variables. An action function $a_i : D \to D_i$ is formed as $v_i := \varphi$. For brevity, we use the following notation:

$$\mathbf{a}(\mu) \triangleq (a_1(\mu), \ldots, a_n(\mu)).$$

An action function a_i that does not update variable v_i is denoted by ξ. The symbol Ξ is used to denote an n-tuple $(\xi, \xi, \ldots, \xi)$, indicating that no variable is updated. The semantics of an action function can also be represented by a relation,

$$\mathsf{SAT}\mathcal{A}(\mathbf{a}) \triangleq \{(\mu, \acute{\mu}) \mid \acute{\mu} = \mathbf{a}(\mu)\}. \tag{1}$$

For modeling purposes, it is often easier to have a modular representation, specially for complex systems. Then, to have a monolithic model of the system we need to compose the multiple EFAs in the model. The composition is performed by the *full synchronous composition* operator, denoted by $\|$, described in [16].

The semantics of an EFA can be represented by its corresponding *transition system* that is based on the states, i.e., locations and variable values, of the model. In the following definition we use the *SOS-notation* (Structured Operational Semantics). The notation $\frac{\text{premise}}{\text{conclusion}}$ should be read as: if the proposition above the "solid line" (premise) holds, then the proposition under the fraction bar (conclusion) holds as well.

Definition II.2 (Transition System)**.**
Let E $= (L, D, \Sigma, \to, L^0, D^0, L^m, D^m)$ be an EFA. Its corresponding explicit state transition relation, denoted by $TS(\mathrm{E}) = (Q, \Sigma, \mapsto, Q^0, Q^m)$, is a 5-tuple where

- $Q = L \times D$, is the finite set of states,
- Σ, is the set of events,
- $\mapsto \subseteq Q \times \Sigma \times Q$, is the explicit transition relation defined by the following rule:

$$\frac{(l, \sigma, g, \mathbf{a}, \acute{l}) \in \to \ \wedge \ \mu \models g}{\big((l, \mu), \sigma, (\acute{l}, \mathbf{a}(\mu))\big) \in \mapsto}, \tag{2}$$

- $Q^0 = L^0 \times D^0$, is the set of initial states,
- $Q^m = L^m \times D^m$, is set of marked states.

B. Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are powerful data structures for representing Boolean functions. For large systems where the number of states grows exponentially, BDDs can improve the efficiency of set and Boolean operations performed on the state sets [6], [17]–[19].

Given a set of m Boolean variables $\mathcal{B}$, a Boolean function $f \colon \mathbb{B}^m \to \mathbb{B}$ ($\mathbb{B}$ is the set of Boolean values, i.e., 0 and 1) can be expressed using Shannon's decomposition [20]. This decomposition can be expressed by a directed acyclic graph, called a BDD, which consists of two types of nodes: *decision nodes* and *terminal nodes*. A terminal node can either be *0-terminal* or *1-terminal*. Each decision node is labeled by a Boolean variable and has two edges to its *low-child* and *high-child*, corresponding to assigning 0 and 1 to the variable, respectively. The *size* of a BDD, denoted as $|\mathbf{B}|$, refers to the number of decision nodes.

The power of BDDs lies in their simplicity and efficiency to perform binary operations. The time complexity of a binary operator between two BDDs $\mathbf{B}_1$ and $\mathbf{B}_2$ is $O(|\mathbf{B}_1| \cdot |\mathbf{B}_2|)$. However, the quantification operators have exponential time complexity. For a more elaborate and verbose exposition of BDDs and the implementation of different operators, refer to [21], [22].

In a BDD graph, a variable b_1 has a lower (higher) *order* than variable b_2 if b_1 is closer (further) to the root and is denoted by $b_1 \prec b_2$ ($b_2 \prec b_1$). The variable ordering will impact the size of the BDD, however, finding an optimal variable ordering of a BDD is an NP-complete problem [23].

III. Supervisory Control Theory

Supervisory Control Theory (SCT) [1], [24] is a general theory to automatically synthesize a control function, referred to as *supervisor*, based on a given plant and specification. A specification describes the allowed and inhibited behaviors. The supervisor restricts the conduct of plant to guarantee that the system never violates the given specification. However, it is often desired, and also in our work, that the supervisor restricts the plant as least as possible, referred to as *optimal* or *minimally restrictive* supervisor. This gives the developers several alternatives to implement the controller and performing further analysis such as time or energy optimization.

In the context of SCT, the behavior of a system is usually represented by its language, i.e. the sets of strings that the system may generate. Conventionally, automata has been used as the modeling formalism to generate the language. In this work, the problems are modeled by EFAs, and the result, i.e., the supervisor, is represented by EFAs as well. As mentioned earlier, we assume that the systems are deterministic, in the sense that at each moment, the supervisor knows the current and next state of the system. Therefore, we only consider deterministic EFAs.

A plant P can be described by the synchronization of a number of sub-plants $P = P_1 \| P_2 \| \ldots \| P_l$, and similarly for a specification $Sp = Sp_1 \| Sp_2 \| \ldots \| Sp_m$. There are different ways of computing a supervisor such as monolithic [1], modular [25], and compositional [26] synthesis. In our approach we

apply monolithic synthesis, which is performing fixed-point computations on the single composed automaton $S_0 = P \| Sp$. After the synthesis procedure, some *blocking* states may be identified, which are the states from where no marked state can be reached. In SCT, the events can be divided into two disjoint subsets: *controllable events*, that can be prevented from executing by the supervisor; and *uncontrollable events*, which cannot be influenced by the supervisor [1], [24]. In addition to the blocking states, the synthesis procedure may also identify some *uncontrollable* states, which are states from where the plant executes an uncontrollable event that violates the specification. By removing the blocking or controllable states from S_0, a *nonblocking* or *controllable* supervisor is obtained, respectively. For a more formal description of SCT refer to [1], [24].

IV. Representation of the Supervisor as EFAs

The last step is to compute the supervisor represented as EFAs. This computation is performed in three steps:

1) Compute a BDD representing the safe states.
2) Transform the computed BDD to guard expressions.
3) Attach the guards to the original EFAs.

Initially, the set of the safe states is computed by fixed point computations based on the synthesis algorithm described in [6].

In stage 2, based on Q_{sup}, we create two sets of states [15]:

- $Q^{\sigma}_{\mathbf{a}}$: The set of states in the supervisor where the execution of σ is defined for the supervisor.
- $Q^{\sigma}_{\mathbf{f}}$: The set of states in the supervisor where the execution of σ is defined for S_0, but not for the supervisor.

By utilizing $Q^{\sigma}_{\mathbf{a}}$ and $Q^{\sigma}_{\mathbf{f}}$ a guard expression $\mathcal{G}^{\sigma}(\langle q^{E_1}, q^{E_2}, \ldots, q^{E_n}\rangle)$ is generated for each controllable event $\sigma \in \Sigma^{S_0}_c$:

$$\mathcal{G}^{\sigma}(\langle q^{E_1}, q^{E_2}, \ldots, q^{E_n}\rangle) = \begin{cases} \text{true} & \text{if } (\langle q^{E_1}, q^{E_2}, \ldots, q^{E_n}\rangle) \in Q^{\sigma}_{\mathbf{a}} \\ \text{false} & \text{if } (\langle q^{E_1}, q^{E_2}, \ldots, q^{E_n}\rangle) \in Q^{\sigma}_{\mathbf{f}} \\ \text{don't}-\text{care} & \text{otherwise} \end{cases}$$

where q^{E_i} represents the current state of EFA E_i. $\mathcal{G}^{\sigma}(\langle q^{E_1}, q^{E_2}, \ldots, q^{E_n}\rangle)$ evaluates to $\texttt{true}$ if σ is allowed to be executed from the state $\langle q^{E_1}, q^{E_2}, \ldots, q^{E_n}\rangle$. The size of a guard $\mathcal{G}$, denoted by $|\mathcal{G}|$, is defined by the number of atomic equality and nonequality terms in the guard expression.

A. Guard Generation

The guards are computed in the following consequent steps:

- Compute the corresponding BDDs for $Q^{\sigma}_{\mathbf{a}}$ and $Q^{\sigma}_{\mathbf{f}}$.
- Convert the BDDs to integer decision diagrams.
- Convert the integer decision diagrams to guards.

First, the corresponding BDDs for the state sets are computed. Next, the BDDs are converted to their corresponding *integer decision diagrams* (IDDs) [7], which will be used to generate the guards in the last step. An IDD is an extension to a BDD where the number of terminals is arbitrary and the domain of the variables in the graph is an arbitrary set of integers. For our purpose, we use an IDD with two terminals, 0-terminal and 1-terminal.

To represent a state $\langle q^{A_1}, q^{A_2}, \ldots, q^{A_n}\rangle$ in the closed-loop automaton $A_1 \parallel \ldots \parallel A_n$, each IDD-variable is associated to an automaton A_i that has Q^{A_i} as its domain. This domain can be mapped to an integer that is represented as an IDD. In other words, each outgoing edge from node A_i represents a state in A_i. Hence the maximum number of edges from a node A_i is $|Q^A_i|$. As for BDDs the number of edges and nodes for an IDD can also be reduced. For simplicity, we use the names of the states on the IDD-edges rather than integers in the sequel.

Using IDDs to generate guards has some advantages in comparison to BDDs: 1) they make it easier to handle and manipulate propositional formulae; 2) they exploit some of the common subexpressions in a guard yielding a more factorized and smaller formula; 3) they depict a more understandable model of the state set, since the nodes and edges represent names of the automata and states, respectively. On the other side different manipulations can be carried out more efficiently on BDDs compared to IDDs. The procedure of converting a BDD to an IDD is presented in [15].

The last step of obtaining the guard is to convert the IDDs to propositional formulae. For a given IDD, a top-down depth first search is used to traverse the graph and generate its corresponding propositional formula. The algorithm starts from the root and visits the nodes whilst generating the expression and ends at the 1-terminal. For each node in the IDD, the corresponding expressions of the edges belonging to the same level (the children of that node) are logically disjuncted and if the edges belong to different levels they are logically conjuncted. Hence, the propositional formula for the IDD in Fig. 1 is

$$r \wedge ((p_1 \wedge S_1) \vee (p_2 \wedge S_2)),$$

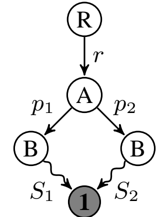


Fig. 1: Recursive representation of an IDD.

where p_i is the corresponding expression of the edge that lead to one of A's children and S_i is the corresponding expression from the node to the 1-terminal, that is recursively computed. A pseudo-algorithm of this process is presented in [15].

B. Guard Attachment

Since $q^{E_i} \in L^{E_i} \times V$, the generated guard will be a combination of $\ell^{E_i} = \ell^{E_i}_i$ (or $\ell^{E_i} \neq \ell^{E_i}_i$) and $v^i = v^i_j$ (or $v^i \neq v^i_j$) expressions. Each variable ℓ^{E_i} holds the current location of EFA E_i. However, since they are not defined in the model, they should be declared and added to the set of variables in the model. Thus, the variable v is extended to $v^+ = (v^1, \ldots, v^n, \ell^{E_1}, \ldots, \ell^{E_N})$. Hence, the transition function of each automaton E_i is extended as follows:

$$\rightarrow^+_{E_i} = \{\ell^{E_i} \xrightarrow{\sigma}_{g/a^+} \acute{\ell}^{E_i} \mid \forall \ell^{E_i} \xrightarrow{\sigma}_{g/a} \acute{\ell}^{E_i} \in \rightarrow_{E_i},$$
$$a^+(v^+) = (a^1(v), \ldots, a^n(v), \ell^{E_1}, \ldots, \acute{\ell}^{E_i}, \ldots, \ell^{E_N}).$$

Nevertheless, this extension can be performed implicitly so that it becomes transparent to the user. Finally, for each EFA E_i in the model, each generated guard $\mathcal{G}^\sigma$ is conjuncted with the guards in $\rightarrow^+_{E_i}$ that include event σ; forming a new EFA E_i^{sup} where

$$\begin{aligned}\rightarrow_{E_i^{sup}} = \{\ell \xrightarrow{\sigma}_{g^+/a^+} \acute{\ell} \mid \\ \forall \ell \xrightarrow{\sigma}_{g/a^+} \acute{\ell} \in \rightarrow^+_{E_i}, g^+ = g \wedge \mathcal{G}^\sigma\}.\end{aligned}$$

Consequently, the supervisor can be represented in a modular manner, deducing that $E_1^{sup} \parallel \ldots \parallel E_N^{sup}$ satisfies the specification without any forbidden states.

V. Guard Reduction using Genetic Algorithms

A *genetic algorithm* (GA) is a search heuristic that mimics the process of natural evolution. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover. In a genetic algorithm, a population of strings (called *chromosomes*), which encode candidate *solutions* (called *individuals*) to an optimization problem, evolves toward better solutions. The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm.

In this paper, a GA is used to optimize the size of generated guards by changing the variables ordering of the underlying BDD.

A. Representation

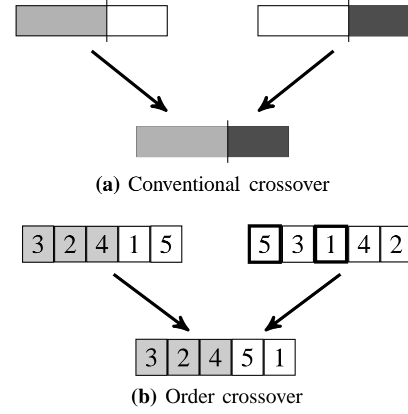
Traditionally, GA works on binary strings of 0's and 1's. However, such encoding require a special repair operation to avoid creation of invalid solutions. Another encoding was introduced for solving Traveling Salesmen Problem [27] and later used for minimization of BDDs [28], which represents a variable ordering as an integer string of length n, where n is the number of variables in a BDD, and each integer appears in the string once.

B. Initialization

The population is initialized by generating random individs. Each individ is generated by randomly permutating chromosomes in initially sorted individ. If a new individ already exists in the population, it is discarded. Individs are added until population size reaches a predefined size.

C. Selection

The selection of the individuals for mating pool is performed by roulette wheel selection, where each individ is chosen with a prbability proportional to its fitness. As a fitness measure of an individ, the size of the guard generated using the variable ordering encoded by the individ was used. The computation of the guard is performed based on the procedure mentioned earlier in the paper, described in more detail in [15]. Additionally, some of the best individs of the old generation are also included in the new generation, to ensure that the best element is never lost.



(a) Conventional crossover

(b) Order crossover

Fig. 2: Crossover operation.

D. Crossover

For each new solution to be produced, a pair of "parent" solutions is selected for breeding from the mating pool selected previously. Two parents are combined with each other using *crossover* operation to produce a "child". New parents are selected for each new child, and the process continues until a new population of solutions of appropriate size is generated. In a traditional implementation of the crossover operation, a random cut point is selected, and the chromosome of the first parent is taken up to the cut point, and chromosome of the second parent is taken from cut point to the end. This however would produce invalid variable orderings. Instead, we use a crossover illustrated in Fig. 2 [28], [29], where genes of the chromosome of the first parent are taken up to the cut point, while from the second parent all other missing genes are taken in the order they appear. This preserves relative order of some of the variables of both parents, and always generates valid solutions.

E. Mutation

Mutation helps diversifying solutions and escaping local minima. We implemented swapping of two genes in an individual as a mutation operation.

F. Termination

We terminated the algorithm after a predefined number of iterations, or when no better individuals were produced after several consecutive iterations.

VI. Academic Example

The guard generation procedure discussed in the previous sections will be applied to the classic Dining Philosophers

problem, which was designed by Dijkstra to illustrate the problem of deadlock in parallel processes with shared resources [34]. The popularity of this model for explaining deadlock is probably due to its very illustrative way of explaining "circular wait" and "starvation". The original problem consists of 5 philosophers sitting around a table. The philosophers either think or eat. In order to eat, each philosopher will need two forks, which are placed to the left and right of each philosopher. As the forks are shared between the philosophers (there are only n forks on the table), a circular wait situation may arise. There are in fact two deadlocks in this system: when each philosopher have taken his left fork and is waiting for the right fork, and vice versa. What make this particularly interesting is that all processes participate in the deadlock situation.

The program is implemented in JAVA programming language using Supremica libraries [30], [31], which uses *JavaBDD* [32] as the BDD package. The example was conducted on a standard PC (Intel Core 2 Quad CPU @ 2.4 GHz and 3GB RAM) running Windows XP.

The model consists of an automaton (plant) for each philosopher and an automaton (specification) for each fork. Automata $Philo:1$ and $Fork:1$ are shown in figures 3 and 4. Each $Philo:i$ automaton consists of the following states: $think$, lu, $ready$, eat, where lu means that the philosopher has lifted up the left work. The interpretations for the other states are straightforward. The events for philosopher i are:

take$i$:j: Philosopher i takes (lifts) fork j.

start_eating$i$: Philosopher i starts eating.

put$i$:j: Philosopher i puts down fork j.

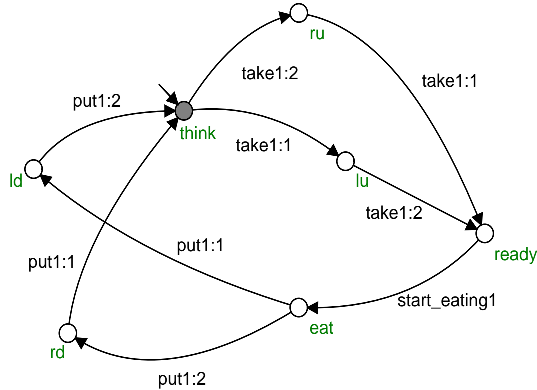


Fig. 3: Plant model for Philo:1.

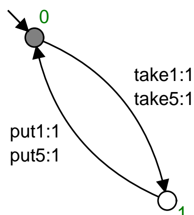


Fig. 4: Specification model for Fork:1.

The closed-loop automaton consists of 1973 reachable states where 2 of those are blocking. After the synthesis, the nonblocking supervisor consists of 1971 states and 7985 transitions. Table I compares the sizes of the guards between the implementation in [15] and the implementation including GA. The events that are always enabled by the supervisor (when the guard becomes true) are not included in the table. The supervisory synthesis, which is merely computed once for all events, was performed by BDD operations and was completed in less than a second. The computation time for generating the guards for all events with and without considering GA was 6 minutes and 1 second, respectively. It can be observed that it takes much longer time for the GA-based implementation. The reason is the time consuming operation for changing the variable ordering of the BDDs during the computation of the fitness value.

With respect to the sizes of the guards, we can see a clear reduction when considering GA. As an example, following shows the guard for event *take4:5*,

$$\begin{aligned} F1 \neq 1 \vee & P5 \neq ru \vee (P1 \neq ru \wedge P1 \neq eat) \\ & \vee P4 \neq think \vee (P2 \neq rd \wedge P2 \neq ru) \\ & \vee F4 \neq 1 \vee P3 = eat \vee P3 = ready \vee P3 = ld \end{aligned} \tag{3}$$

$$\begin{aligned} P3 \neq ru \vee P2 \neq ru \vee P5 \neq ru \\ \vee (P1 \neq eat \wedge P1 \neq ready \wedge P1 \neq ru) \end{aligned} \tag{4}$$

where (3) and (4) are the guards before and after applying GA. Pi and Fi are variables representing the current location of philosopher i and fork i, respectively.

VII. Conclusions and Future Works

In this paper, we introduced an evolutionary-based method to find the optimal variable ordering for the BDDs that will yield small guards. The approach can be divided into two components: guard generation and genetic algorithms. The algorithm starts by an initial population, where the individuals represent different variable orderings. The fitness value for each chromosome will be computed by generating its corresponding guard based on the algorithm in [15], briefly described in this paper. The remaining parts of the approach follows the regular steps of genetic algorithms, i.e., selection, crossover and mutation. The algorithm terminates when no

TABLE I: The size of the generated guards for the dining philosophers example.

Event	Without GA	With GA
take3:3	9	4
take4:5	11	6
take3:4	9	5
take4:4	9	4
take5:5	11	5
take5:1	11	6
take2:2	11	4
take2:3	10	4
take1:2	10	4
take1:1	12	5

better individuals were produced after several consecutive iterations.

Finally, the guards can be added to the original models by creating EFAs with a modular structure. This step often makes it easier and more memory efficient to implement the supervisor in a controller. The approach was applied to an academic example, the dining philosophers problem.

As a future work we desire to improve the procedure of computing the fitness value, which is due to the time consuming operation of changing the variable ordering of the BDDs. This is currently the bottleneck of the algorithm.

REFERENCES

[1] P. Ramadge and W. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81–98, 1989.

[2] S. Balemi, G. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. Franklin, "Supervisory control of a rapid thermal multiprocessor," *Automatic Control, IEEE Transactions on*, vol. 38, no. 7, pp. 1040–1059, 1993.

[3] L. Feng, W. Wonham, and P. S. Thiagarajan, "Designing communicating transaction processes by supervisory control theory," *Form. Methods Syst. Des.*, vol. 30, no. 2, pp. 117–141, 2007.

[4] K. Andersson, J. Richardsson, B. Lennartson, and M. Fabian, "Coordination of Operations by Relation Extraction for Manufacturing Cell Controllers," *IEEE Transactions on Control Systems Technology*, vol. 18, no. 2, pp. 414–429, Mar. 2010.

[5] S. B. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. 27, pp. 509–516, Jun. 1978.

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

[7] J. Gunnarsson, "Symbolic Methods and Tools for Discrete Event Dynamic Systems," Ph.D. dissertation, Electrical Engineering, Linköping University, Linköping, Sweden, 1997.

[8] E. J. McCluskey, "Minimization of Boolean functions," *Bell System Technical Journal*, vol. 35, no. 5, pp. 1417–1444, 1956.

[9] A. Mannani, Y. Yang, and P. Gohari, "Distributed extended finite-state machines: communication and control," in *Proceedings of the 8th international Workshop on Discrete Event Systems, WODES'06*, 2006, pp. 161–167.

[10] Y. Li and W. Wonham, "Control of vector discrete-event systems. II. Controller synthesis," *IEEE Transactions on Automatic Control*, vol. 39, no. 3, pp. 512–531, Mar. 1994.

[11] L. E. Holloway and B. H. Krogh, "On closed-loop liveness of discrete event systems under maximally permissive control," *IEEE Transactions on Automatic Control*, vol. 37, no. 5, pp. 692–697, 1992.

[12] A. Giua and F. DiCesare, "Blocking and controllability of Petri nets in supervisory control," *IEEE Transactions on Automatic Control*, vol. 39, no. 4, pp. 818–823, Apr. 1994.

[13] K. Yamalidou, J. O. Moody, M. D. Lemmon, and P. J. Antsaklis, "Feedback control of Petri nets based on place invariats," *Automatica*, vol. 32, no. 1, pp. 15–28, 1996.

[14] C. Ma and W. Wonham, "Nonblocking supervisory control of state tree structures," *IEEE Transactions on Automatic Control*, vol. 51, no. 5, pp. 782–793, May 2006.

[15] S. Miremadi, K. Å kesson, and B. Lennartson, "Symbolic computation of reduced guards in supervisory control," *IEEE Transactions on Automation Science and Engineering*, vol. 8, no. 4, pp. 754–765, 2011.

[16] M. Sköldstam, K. Åkesson, and M. Fabian, "Modeling of discrete event systems using finite automata with variables," *Decision and Control, 2007 46th IEEE Conference on*, pp. 3387–3392, 2007.

[17] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," in *Proceedings of the Fifth Annual IEEE Symposium on e Logic in Computer Science, 1990.*, Jun. 1990, pp. 428–439.

[18] S. Miremadi, K. Åkesson, M. Fabian, A. Vahidi, and B. Lennartson, "Solving two supervisory control benchmark problems using Supremica," in *9th International Workshop on Discrete Event Systems, 2008, WODES 08.*, May 2008, pp. 131–136.

[19] C. Ma and W. Wonham, "STSLib and its application to two benchmarks," in *9th International Workshop on Discrete Event Systems, 2008, WODES'08.*, May 2008, pp. 119–124.

[20] C. E. Shannon, "A Mathematical Theory of Communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423,625–656, 1948.

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

[22] H. Andersen, "An introduction to binary decision diagrams," Department of Information Technology, Technical University of Denmark, Tech. Rep., 1999.

[23] B. Bollig and I. Wegener, "Improving the Variable Ordering of OBDDs Is NP-Complete," *IEEE Trans. Comput.*, vol. 45, no. 9, pp. 993–1002, 1996.

[24] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 2nd ed. Springer, 2008.

[25] W. Wonham and P. Ramadge, "Modular supervisory control of discrete-event systems," *Mathematics of Control Signals and Systems*, vol. 1, no. 1, pp. 13–30, 1988.

[26] H. Flordal, R. Malik, M. Fabian, and K. Åkesson, "Compositional Synthesis of Maximally Permissive Supervisors Using Supervision Equivalence," *Discrete Event Dynamic Systems*, vol. 17, no. 4, pp. 475–504, Aug. 2007.

[27] L. D. Whitley, T. Starkweather, and D. Fuquay, "Scheduling problems and traveling salesmen: The genetic edge recombination operator," in *Proceedings of the 3rd International Conference on Genetic Algorithms*, 1989, pp. 133–140.

[28] R. Drechsler, "Genetic algorithm for variable ordering of OBDDs," in *IEE Proceedings of Computers and Digital Techniques*, 1996, pp. 364–368.

[29] D. Goldberg and R. Lingle, "Alleles, loci, and the traveling salesman problem," in *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, Pittsburgh, PA, USA, 1985, pp. 156–159.

[30] K. Åkesson, M. Fabian, H. Flordal, and R. Malik, "Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems," in *Proceedings of the 8th international Workshop on Discrete Event Systems, WODES'06*, Ann Arbor, MI, USA, 2006, pp. 384–385.

[31] K. Åkesson, M. Fabian, H. Flordal, and A. Vahidi, "Supremica—a Tool for Verification and Synthesis of Discrete Event Supervisors," in *11th Mediterranean Conference on Control and Automation*, Rhodos, Greece, 2003.

[32] "JavaBDD." [Online]. Available: http://javabdd.sourceforge.net