# Symbolic Computation of Reduced Guards in Supervisory Control

Sajed Miremadi, Knut Åkesson, and Bengt Lennartson, *Member, IEEE*

***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 introduce an efficient method to characterize a supervisor by tractable 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. To obtain tractable guard expressions, we reduce them by exploiting the structure of the given models. In order to be able to handle complex systems efficiently, the models are symbolically represented by *binary decision diagrams* and all computations are performed on these data structures. The algorithms have been implemented in a supervisory control tool and applied to an industrially relevant example.**

***Note to Practitioners*—In today's industry, the control functions are implemented to a great extent manually, which makes it a tedious, error-prone, and time consuming process. supervisory control theory (SCT) provides a powerful framework for automatically producing safe and flexible control functions. SCT is based on state-transition models, however, industrial people are used to other representations. Specifically, the interpretation of a control function represented by a huge and cluttered state-transition model requires the maintenance personnel to have other skills than are common today. As a consequence, SCT is seldom utilized in the industry. This paper aims to facilitate the realization and manipulation of control functions that are generated based on SCT. This is performed by restricting the state-transition models by tractable logic conditions extracted from the generated control function. To be able to handle large systems efficiently, the computations are carried out on implicit representations of the state-transition models.**

***Index Terms*—Binary decision diagrams, deterministic finite automata, heuristic techniques, propositional formula, supervisory control theory (SCT), symbolic representation.**

## 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 closed-loop specification. These restrictions can be expressed as logic conditions in terms of propositional formulae generated from different sets of states related to the closed-loop model. 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];

Manuscript received March 10, 2010; revised September 17, 2010; accepted November 19, 2010. Date of publication May 10, 2011; date of current version October 05, 2011. This paper was recommended for publication by Associate Editor K. T. Seow and Editor M. Zhou upon evaluation of the reviewers' comments. This work was carried out at the Wingquist Laboratory VINN Excellence Centre within the Area of Advance—Production at Chalmers and supported by the Swedish Governmental Agency for Innovation Systems (VINNOVA). The support is gratefully acknowledged.

The authors are with the Department of Signals and Systems Automation, Chalmers University of Technology, Gothenburg, Sweden, 412 96 (e-mail: miremads@chalmers.se; knut@chalmers.se; bengt.lennartson@chalmers.se).

Digital Object Identifier 10.1109/TASE.2011.2146249

1545-5955/$26.00 © 2011 IEEE

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 abovementioned 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 nonblocking 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.* [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].

The contribution in this paper is 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, yielding a modular supervisor in form of *extended finite automata* (EFA) [15]. To be able to handle large systems, all computations are performed by BDDs. Hence, in this paper, we presuppose that the monolithic supervisor is computed and its states are represented by a BDD. Based on the BDD the guards are computed. To obtain more simplified and reduced guards, we exploit a set of don't-care states and apply some heuristic techniques, which are all performed by BDDs as well. These heuristics are shown to be crucial in the reduction of the generated guards.

Our method has some main features. The final representation, i.e., the guards, will preserve all the properties of the supervisor. So if the supervisor is nonblocking, controllable or minimally restrictive, then the guards will also represent a supervisor with those properties. 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 paper is organized as follows. Section II gives a brief overview of the synthesis procedure based on deterministic finite automata. The process of generating the guards based on a set of states is discussed in Section III. In Section IV, we explain BDDs. The procedure of generating a guard based on a BDD and applying the heuristic techniques to simplify the guards is presented in Section V. Section VI describes briefly how the guards can be used to represent the supervisor. The guard generation algorithm is applied to a real case study in Section VII. Finally, Section VIII provides some conclusions and suggestions for future work.

## II. Preliminaries

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

### A. Deterministic Finite Automata

A deterministic finite automaton (DFA) $A$ is a five-tuple $\left(Q^A, \Sigma^A, \delta^A, q^A_{init}, Q^A_m\right)$, where $Q^A$ is a finite set of states; $\Sigma^A$ is a finite set of events called the *alphabet*; $\delta^A$: $Q^A \times \Sigma^A \rightarrow Q^A$ is a partial transition *function* that describes the state transitions, where $\delta^A(q,\sigma) = \acute{q}$ means that there exists a transition labeled by event $\sigma \in \Sigma^A$ from state $q \in Q^A$, called *source-state*, to state $\acute{q} \in Q^A$, called *target-state*. $Q^A_m$ is the set of *marked* states that are desired to be reached. We write $\Gamma^A(q)$ to denote all the events that are defined from state $q \in Q^A$. Formally, $\Gamma^A(q) = \{\sigma \in \Sigma^A \mid \delta^A(q,\sigma)\ is\ defined\}$.

A sequence of events is called a *string* of events. An empty string is denoted by $\varepsilon$ and all possible strings consisting of events from $\Sigma$ is denoted by $\Sigma^*$. The domain of the transition function of an automaton can be recursively extended to strings of events by $\delta(q,\varepsilon) = q$ and $\delta(q, s\sigma) = \delta(\delta(q,s),\sigma)$, where $s \in \Sigma^*$ and $\sigma \in \Sigma$.

The composition of a number of automata is performed by the *full synchronous (parallel) composition* operator $\|$ defined in [1] and [16].

If $A$ is the full synchronous composition of $n$ automata $A_1, A_2, \ldots, A_n$, a state, $q_k^A \in Q^{A_1 \| \cdots \| A_n}$, will have the form

$$q_k^A = \left\langle q_{k_1}^{A_1}, q_{k_2}^{A_2}, \ldots, q_{k_n}^{A_n} \right\rangle.$$

We call an element of $q_k^A$ a *substate*. A substate belonging to a specific automaton $A_i$ is extracted from $q_k^A$ by $\Phi_i : \left(Q^{A_1} \times Q^{A_2} \times \cdots \times Q^{A_n}\right) \rightarrow Q^{A_i}$. For instance, $\Phi_2\left(q_k^A\right) = q_{k_2}^{A_2}$.

### B. Supervisory Control Theory

As described in the introduction, SCT [1] is a general theory to, given a plant and specification, automatically synthesize a supervisor restricting the plant towards the specification. Notice that if the plant is given as a number of subplants $P_1, \ldots, P_n$, the monolithic plant $P$ is computed by synchronizing the subplants $P = P_1 \parallel \cdots \parallel P_n$, and similarly for the specifications.

In SCT, a first candidate to the supervisor is the composed automaton $S_0 = P \parallel Sp$, where $Sp$ is the monolithic specification. After the synthesis procedure, a set of states are identified as *forbidden states* denoted by $Q_\otimes$, which should be excluded from $S_0$ in order to obtain the final supervisor. The supervisor can have different properties such as nonblocking, controllability, minimally restrictive, etc. It should be emphasized that the supervisor representation, which will be presented in form of guards, will preserve all the properties of the original supervisor. We show the synthesis procedure by Example II.1. For a more formal and detailed explanation of supervisory synthesis, see [1], [16], and [17].

*1) Example II.1:* Consider two users that will use two resources in opposite order. Fig. 1(a) and (b) depict the user (plant) models and Fig. 1(c) and (d) depict the resource (specification) models. The event $use_{R_1}^A$ means that user A uses resource $R_1$. Similarly, the other events can be interpreted. Fig. 1(e) shows the composed automation $S_0 = A \parallel B \parallel C \parallel D$. As an additional specification, we assume that state $\left\langle q_3^A, q_3^B, q_1^C, q_1^D \right\rangle$ is the marked state, i.e., the state that is desired to be reached. In $S_0$, the state $\left\langle q_2^A, q_2^B, q_2^C, q_2^D \right\rangle$ is a forbidden (blocking) state and by removing it, the supervisor is obtained.

## III. Supervisor as Guards

Recall that the supervisor influences the plant by preventing it to execute some events in its current state, in order to avoid violations of the given specification. Accordingly, for each event, there is a set of states in $S_0$, where the event is either allowed or forbidden to occur, in order to end up in a state of the supervisor. It is also possible that the execution of the event at some states does not affect the synthesis result. On the basis of the mentioned state sets, some conditional propositional formulae, referred to as *guards*, can be extracted indicating under which conditions the event can be executed without violating the specifications. The guards can assist the users to get a profound understanding of the result of the synthesis process. Our goal is to make such guards as compact and comprehensible as possible for the users. Note that all the proceeding computations focus on generating guards for a single event, which can indeed be generalized to the entire alphabet.

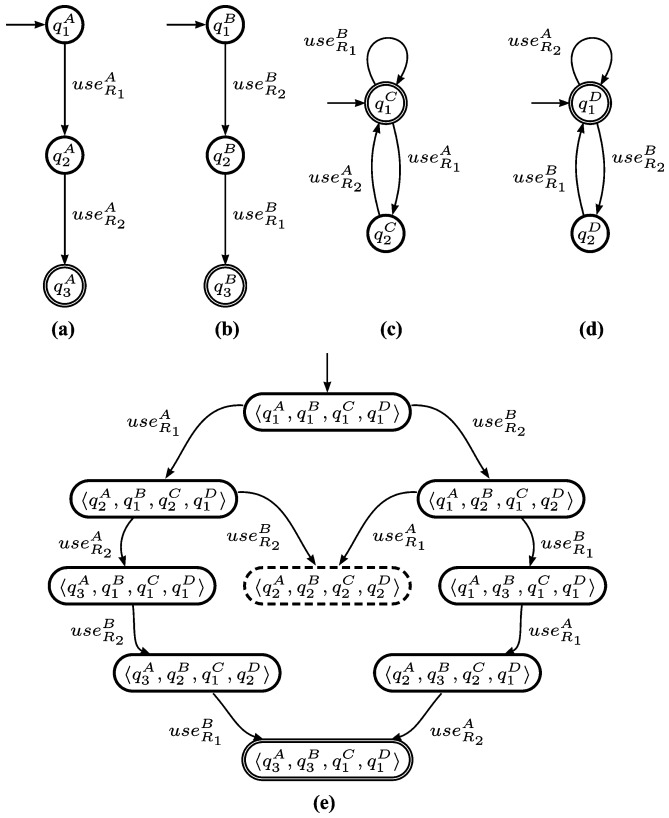


Fig. 1. Example II.1. (a) and (b) User automata A and B. (c) and (d) resource automata C and D. (e) The reachable states in the composed automaton $S_0 = A \parallel B \parallel C \parallel D$.

Before proceeding, for a number of automata $A_1 \ldots A_n$, a number of state sets are needed to be introduced, required to compute and formally represent the guards:

| | |
|---|---|
| $Q$ | $Q^{A_1} \times Q^{A_2} \times \cdots \times Q^{A_n}$; |
| $Q^\sigma$ | $\left\{q \in Q \mid \sigma \in \Gamma^{S_0}(q)\right\}$–the states where $\sigma$ is enabled; |
| $Q_{reach}$ | $\left\{q \in Q \mid \forall\, s \in \Sigma^* : \delta^{S_0}\left(q_{init}^{S_0}, s\right) = q\right\}$–the states that can be reached from the initial state; |
| $Q_{sup}$ | All the reachable states belonging to the supervisor referred to as *safe states*; |
| $Q_\otimes$ | $Q_{reach} \backslash Q_{sup}$–the states that are removed from $S_0$ during synthesis; |

where $S_0 = A_1 \parallel \cdots \parallel A_n$, $\Sigma = \Sigma^{S_0}$ and $M \backslash N = M \cap C(N)$ for two arbitrary sets $M$ and $N$. For a state set $X$, $C(X)$ denotes the complement of the set $X$ by having $Q$ as the universal set.

*1) Example III.1:* In Example II.1:

| | |
|---|---|
| $Q$ | $Q^A \times Q^B \times Q^C \times Q^D$ contains 36 states; |
| $Q^{use_{R_1}^A}$ | $\left\{\left\langle q_1^A, q_1^B, q_1^C, q_1^D \right\rangle, \left\langle q_1^A, q_2^B, q_1^C, q_2^D \right\rangle, \left\langle q_1^A, q_3^B, q_1^C, q_1^D \right\rangle, \left\langle q_1^A, q_1^B, q_1^C, q_2^D \right\rangle, \left\langle q_1^A, q_2^B, q_1^C, q_1^D \right\rangle, \left\langle q_1^A, q_3^B, q_1^C, q_2^D \right\rangle\right\}$. Note that the three last states are not reachable and are not shown in Fig. 1(e); |
| $Q_{reach}$ | All nine states shown in Fig. 1(e); |

$$Q_{sup} \qquad Q_{reach} \backslash \left\{\left\langle q_2^A, q_2^B, q_2^C, q_2^D\right\rangle\right\};$$
$$Q_{\otimes} \qquad \left\{\left\langle q_2^A, q_2^B, q_2^C, q_2^D\right\rangle\right\}.$$

### A. Basic State Sets

Concerning the states that are retained or removed after the synthesis process, the states that enable an arbitrary event $\sigma$ can be divided into three *basic state sets*.

1) The states where $\sigma$ must be enabled in order to end up in states that belong to the supervisor.
2) The states where $\sigma$ must be disabled in order to avoid ending up in $Q_{\otimes}$, i.e., the states that were removed after the synthesis process.
3) The states where enabling or disabling $\sigma$ does not make any changes in the final supervisor.

These state sets will form the basis for generating the constraining propositional formulae. In the sequel, each state set will be described formally and in more detail.

*Definition III.1 (Forbidden State Set): Forbidden state set*, $Q_{\mathbf{f}}^{\sigma}$, is the set of states in the supervisor where the execution of $\sigma$ is defined for $S_0$, but not for the supervisor

$$Q_{\mathbf{f}}^{\sigma} = \left\{q \in Q_{sup} \mid \sigma \in \Gamma^{S_0}(q) \wedge \sigma \notin \Gamma^{sup}(q)\right\}.$$

*Definition III.2 (Allowed State Set): Allowed state set* $Q_{\mathbf{a}}^{\sigma}$ is the set of states in the supervisor where the execution of $\sigma$ is defined for the supervisor

$$Q_{\mathbf{a}}^{\sigma} = \left\{q \in Q_{sup} \mid \sigma \in \Gamma^{sup}(q)\right\}.$$

In other words, for each event $\sigma \in \Sigma$, $Q_{\mathbf{a}}^{\sigma}$ represents the set of states where event $\sigma$ *must* be *allowed* to be executed in order to end up in states belonging to the supervisor (an analogous argument can be given for $Q_{\mathbf{f}}^{\sigma}$). For instance, in Example II.1, $Q_{\mathbf{a}}^{use_{R_1}^A} = \left\{\left\langle q_1^A, q_1^B, q_1^C, q_1^D\right\rangle, \left\langle q_1^A, q_3^B, q_1^C, q_1^D\right\rangle\right\}$.

In order to obtain compact and simplified guards, inspired from the Boolean minimization techniques, we determine a set of states where executing $\sigma$ will not impact the result of the synthesis and utilize these states to minimize the guards.

*Definition III.3 (Don't-Care State Set): Don't-care state set*, $Q_{\mathbf{dc}}^{\sigma}$, is the set of states where event $\sigma$ could either be enabled or disabled without having any impact on the final supervisor and is formally defined as

$$Q_{\mathbf{dc}}^{\sigma} = C\left(Q_{\mathbf{a}}^{\sigma} \cup Q_{\mathbf{f}}^{\sigma}\right).$$

From Definitions III.2 and III.1, it can be concluded that for a given event $\sigma$, the states that can impact the supervisor are only the states where $\sigma$ *must* be allowed, $Q_{\mathbf{a}}^{\sigma}$, or forbidden, $Q_{\mathbf{f}}^{\sigma}$, to occur and the remaining states can be considered as don't-care. It can be deduced that the don't-care states consist of the states at which $\sigma$ is not defined and the states that are not included in the supervisor.

### B. Guards

Recall that a state in $Q^{S_0}$ has the following form:

$$q_k^{S_0} = \left\langle q_{k_1}^{A_1}, q_{k_2}^{A_2}, \ldots, q_{k_n}^{A_n}\right\rangle.$$

For an event $\sigma$, the following propositional function $G^{\sigma} : Q^{A_1} \times Q^{A_2} \times \cdots \times Q^{A_n} \to \mathbb{B}$, referred to as *guard*, is desired

$$G^{\sigma}\left(\left\langle q^{A_1}, q^{A_2}, \ldots, q^{A_n}\right\rangle\right) = \begin{cases} \text{true}, & \left\langle q^{A_1}, q^{A_2}, \ldots, q^{A_n}\right\rangle \in Q_{\mathbf{a}}^{\sigma} \\ \text{false}, & \left\langle q^{A_1}, q^{A_2}, \ldots, q^{A_n}\right\rangle \in Q_{\mathbf{f}}^{\sigma} \\ \text{don't care}, & \text{otherwise} \end{cases}$$

where $\mathbb{B}$ is the set of Boolean values and $q^{A_i}$ represents the current state of automaton $A_i$. In particular, $\sigma$ is allowed to be executed from the state $\left\langle q^{A_1}, q^{A_2}, \ldots, q^{A_n}\right\rangle$ if the guard is true.

The following procedure shows how the corresponding propositional formula of a state set can be created.

Let $Q_{\alpha} \subseteq Q^{S_0}$, where

$$Q_{\alpha} : \left\{\left\langle q_{k_1}^{A_1}, \ldots, q_{k_n}^{A_n}\right\rangle, \ldots, \left\langle q_{\ell_1}^{A_1}, \ldots, q_{\ell_n}^{A_n}\right\rangle\right\}.$$

The corresponding guard is generated by the following procedure.

1) Introduce $n$ new variables $\left\{q^{A_1}, q^{A_2}, \ldots, q^{A_n}\right\}$ where $q^{A_i} \in Q^{A_i}$. We define $q^{A_i}$ to hold the current state of automaton $A_i$ in $S_0$.
2) The corresponding propositional formula of $Q_{\alpha}$, $\mathrm{PF}(Q_{\alpha})$, will be

$$\mathrm{PF}(Q_{\alpha}) : \bigvee_{q \in Q_{\alpha}} \left(\bigwedge_{i=1}^{n} \left(q^{A_i} = \Phi_i(q)\right)\right) \tag{1}$$

where $=$ is the equality operator.

There are two ways to generate the guard for an event $\sigma$, either on the basis of $Q_{\mathbf{a}}^{\sigma}$ as $G_{\mathbf{a}}^{\sigma} = \mathrm{PF}\left(Q_{\mathbf{a}}^{\sigma}\right)$, or on the basis of $Q_{\mathbf{f}}^{\sigma}$ as $G_{\mathbf{f}}^{\sigma} = \neg\mathrm{PF}\left(Q_{\mathbf{f}}^{\sigma}\right)$. For the sake of brevity, we denote $\neg\left(q^{A_i} = q_k^{A_i}\right)$ as $\left(q^{A_i} \neq q_k^{A_i}\right)$.

Inspired by minimization methods of Boolean functions, simplified guards can be obtained by utilizing the don't-care states and applying some heuristic techniques.

*Definition III.4 (Allowed Guard): Allowed guard*, denoted by $\mathcal{G}_{\mathbf{a}}^{\sigma}$, is the result of simplifying $G_{\mathbf{a}}^{\sigma}$ by utilizing the don't-care states and some heuristic techniques.

*Definition III.5 (Forbidden Guard): Forbidden guard*, denoted by $\mathcal{G}_{\mathbf{f}}^{\sigma}$, is the result of simplifying $G_{\mathbf{f}}^{\sigma}$ by utilizing the don't-care states and some heuristic techniques.

The number of equality terms, which has either the form $\left(q^{A_i} = q_k^{A_i}\right)$ or $\left(q^{A_i} \neq q_k^{A_i}\right)$, in the propositional formula is referred to as the *size* of the formula. We denote the size of a propositional formula $p$ by $|p|$. From a user perspective, a *smaller* formula would typically be more readable and comprehensible. Our goal is to find the smallest guard.

Since Boolean minimization rules are applied to guards, it can be concluded that $|\mathcal{G}_{\mathbf{a}}^{\sigma}| \leq |G_{\mathbf{a}}^{\sigma}|$ and $|\mathcal{G}_{\mathbf{f}}^{\sigma}| \leq |G_{\mathbf{af}}^{\sigma}|$. The procedure of computing $\mathcal{G}_{\mathbf{a}}^{\sigma}$ and $\mathcal{G}_{\mathbf{f}}^{\sigma}$ will be described later. Depending on the internal structure of each model, either $\mathcal{G}_{\mathbf{a}}^{\sigma}$ or $\mathcal{G}_{\mathbf{f}}^{\sigma}$ can yield a smaller guard. A proper approach is to calculate both $\mathcal{G}_{\mathbf{a}}^{\sigma}$ and $\mathcal{G}_{\mathbf{f}}^{\sigma}$ and select the smallest one.

*Definition III.6 (Adaptive Guard):* The *adaptive guard*, denoted by $\mathcal{G}_{\star}^{\sigma}$, is the smallest guard comparing $\mathcal{G}_{\mathbf{a}}^{\sigma}$ and $\mathcal{G}_{\mathbf{f}}^{\sigma}$, i.e., $\mathcal{G}_{\star}^{\sigma} = \mathcal{G}_{\mathbf{a}}^{\sigma}$ if $|\mathcal{G}_{\mathbf{a}}^{\sigma}| \leq |\mathcal{G}_{\mathbf{f}}^{\sigma}|$, otherwise $\mathcal{G}_{\star}^{\sigma} = \mathcal{G}_{\mathbf{f}}^{\sigma}$.

There is an exceptional case where the supervisor cannot be represented by guards. Since guards restrict the occurrence of

the events, they cannot represent a null supervisor, i.e., $Q_{sup} = \emptyset$. Such a case can be represented in different ways. For example, an additional imaginary state can be considered that is connected to the initial state in $S_0$ with the silent event $\epsilon$. In this manner, a guard can be associated to event $\epsilon$ and if the supervisor is null, that guard will be `false`, meaning that the initial state can never be reached. Another way is to simply show a message to the user indicating that the supervisor is null.

We summarize this section by applying the overall procedure to Example II.1.

*1) Example III.2:* From Example II.1, we have $Q_{\mathbf{f}}^{use_{R_1}^A} = \left\{\left\langle q_1^A, q_2^B, q_1^C, q_2^D\right\rangle\right\}$. The guard $G_{\mathbf{f}}^{use_{R_1}^A}$ will be $q^A \neq q_1^A \vee q^B \neq q_2^B \vee q^C \neq q_1^C \vee q^D \neq q_2^D$ (the size is 4). After applying the simplification procedure, the guard $\mathcal{G}_{\mathbf{f}}^{use_{R_1}^A}$ becomes $q^B \neq q_2^B$. This shows that $use_{R_1}^A$ is not allowed to occur when the current state of automaton $B$ is $q_2^B$. Indeed, by considering the states that enable $use_{R_1}^A$ in Fig. 1(e), it can be observed that the only state where $use_{R_1}^A$ should necessarily be disabled is $q_2^B$. A more detailed and formal description about this type of simplification will be given in Section V-C2. Also, note that an alternative guard is $q^D \neq q_2^D$. Similarly, the guards for the other events can be computed.

## IV. BDD Representation

The presented basic state sets could be very huge for complex systems and representing them explicitly would be computationally expensive in terms of time and memory. This prompts us to represent them symbolically using BDDs [5]; 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 dramatically [6], [18]–[20].

Given a set of Boolean variables $V$, a BDD is a Boolean function $f: 2^V \rightarrow \{0,1\}$ which can be recursively expressed using Shannon's decomposition [21]

$$f := (\neg v_j \wedge f|_{v_j=0}) \vee (v_j \wedge f|_{v_j=1}) \quad v_j \in V$$

where $f|_{v_j=0}$ and $f|_{v_j=1}$ refer to assigning 0 and 1 to all occurrences of Boolean variable $v_j$, respectively. A BDD is represented as a directed acyclic graph, 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*. The low- and high-child corresponds to the cases in the above equation where $v_j$ is 0 and 1, respectively.

The power of BDDs lies in their simplicity and efficiency to perform binary operations. A binary operator op between two BDDs $f$ and $g$ can be computed as

$$f \,\mathtt{op}\, g := [\neg v_j \wedge (f|_{v_j=0} \,\mathtt{op}\, g|_{v_j=0})] \vee [v_j \wedge (f|_{v_j=1} \,\mathtt{op}\, g|_{v_j=1})].$$

If the operator is implemented based on dynamic programming, the time complexity of the algorithm will be $O(|f| \cdot |g|)$ [22], where $|f|$ and $|g|$ are the *sizes* of the BDDs referring to the number of nodes excluding the terminal nodes. A BDD operation that is used extensively in our implementation is the *existential quantification* over Boolean variables

$$\exists\, v \cdot f := f|_{v_j=0} \vee f|_{v_j=1}.$$

The time complexity for quantification is exponential in the worst case. The implementation of the BDD operators has been discussed in more detail in [23].

To represent models such as automata by BDDs, a characteristic function can be used. Having a finite set $S$, for every $A \subseteq S$, the characteristic function is defined as follows:

$$\chi_A(\alpha) = \begin{cases} 1 & \alpha \in A \\ 0 & \alpha \notin A \end{cases}.$$

The elements of a set can be expressed as a Boolean vector. So, a set with $n$ elements, requires a Boolean vector of length $\lceil \log_2 n \rceil$. Note that, to represent a transition function of an automaton, two Boolean vectors with different sets of Boolean variables are needed to distinguish between source-states and target-states.

A variable $v_1$ has a lower (higher) *order* than variable $v_2$ if $v_1$ is closer (further) to the root and is denoted by $v_1 \prec v_2$ ($v_2 \prec v_1$). The variable ordering will impact the size of the BDD, however, finding an optimal variable ordering of a BDD is an NP-complete problem [24] and will not be considered in this paper.

We denote the BDD representation of an object $obj$, which could be a transition, a set of states or events, etc., by $\mathcal{B}(obj)$; and the Boolean variables that are required to represent $obj$ are denoted by $B^{obj}$. For a more elaborate and verbose exposition of BDDs and the implementation of different operators, refer to [23] and [25].

*1) Example IV.1:* We will compute the corresponding BDD for the transition function of automaton $A$ in Fig. 1(a). Since automaton $A$ has three states, a binary vector of length 2 is required to represent the states and the events can be represented by a binary vector of length 1. The state and event encoding for automaton $A$ is shown in Table I.

TABLE I
STATE AND EVENT ENCODING TABLE FOR AUTOMATON $A$ IN FIG. 1(A)

| State | $v_1^A v_0^A$ | Event | $e_0^A$ |
|---|---|---|---|
| $q_1^A$ | 0 0 | $use_{R_1}^A$ | 0 |
| $q_2^A$ | 0 1 | $use_{R_2}^A$ | 1 |
| $q_3^A$ | 1 0 | | |

Based on this encoding information, a logic model can be constructed for the transition function as follows:

$$\begin{aligned}\chi_\delta\left(v^A, \acute{v}^A, e^A\right) &= \left(\neg v_1^A \wedge \neg v_0^A \wedge \neg \acute{v}_1^A \wedge \acute{v}_0^A \wedge \neg e_0^A\right) \\ &\vee \left(\neg v_1^A \wedge v_0^A \wedge \acute{v}_1^A \wedge \neg \acute{v}_0^A \wedge e_0^A\right)\end{aligned}$$

where $v^A$ and $\acute{v}^A$ are the binary vectors representing the source- and target-states, respectively, and $e^A$ represents the event which consists of a single element. Fig. 2 shows the corresponding BDD for the transition function of automaton $A$. The variable ordering of this BDD is $v_1^A \prec \acute{v}_1^A \prec v_0^A \prec \acute{v}_0^A \prec e_0^A$.

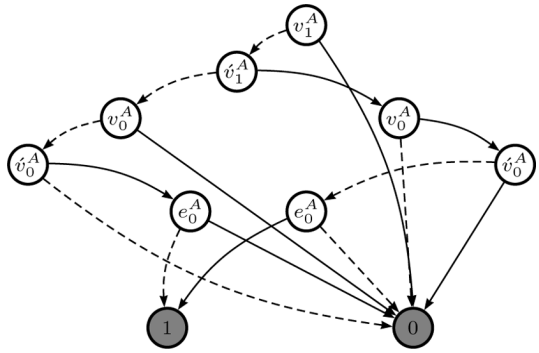


Fig. 2. Corresponding BDD for the transition function of automaton $A$ in Fig. 1(a).

Correspondingly, the BDD representation for the other automata can be computed in an analogous manner.

In this specific example, the BDD representation is larger than the automaton. However, the value of BDDs will be revealed for large and complex automata where the BDDs become much more compact.

Note that in the graphical representation of BDDs, dotted and solid edges refer to low- and high- child, respectively. For a more detailed description on how an automaton is translated to a BDD refer to [6].

In our implementation, a BDD follows a fixed variable ordering based on the method presented in [26]. In this method, the variable ordering is influenced by the ordering of interacting automata based on weighted search in the Process Communication Graph (PCG). A PCG for a set of automata is a weighted undirected graph, where the weight between two automata $A_1$ and $A_2$ is defined as $|\Sigma^{A_1} \cap \Sigma^{A_2}|$. In some cases, the ordering can be improved [6]. This is however beyond the scope of this paper.

## V. From BDDs to Guards

The process of converting BDDs to guards can be divided into three consequent steps.

1) Computing the corresponding BDDs for the basic state sets.
2) Converting the BDDs to integer decision diagrams (IDDs).
3) Converting the IDDs to guards.

In this section, each step will be explained separately. An example describing this procedure is presented in Section VII.

### A. BDD Computation

The first step of generating the guard, is to compute the corresponding BDDs for the basic state sets as described in Section III-A.

For an automaton, the BDD representation for its transition function is used as the basis for generating these state sets. As it was described in Example IV.1, two distinct sets of BDD variables are needed to represent the source- and target-states in an automaton, denoted by $B^Q$ and $B^{\acute{Q}}$, respectively. In the following, we show how $\mathcal{B}(Q^\sigma)$, $\mathcal{B}(Q^\sigma_{\mathbf{f}})$, and $\mathcal{B}(Q^\sigma_{\mathbf{a}})$ are constructed:

$$f = \mathcal{B}(\delta) \wedge \mathcal{B}(\sigma) \tag{2}$$
$$g = \exists\, B^{\acute{Q}}.f \tag{3}$$
$$\mathcal{B}(Q^\sigma) = \exists\, B^{\Sigma}.g \tag{4}$$

First, the BDD representation of all transitions that include event $\sigma$ is extracted (2). Next, the BDD-variables used for representing the target-states are excluded, giving the transitions in $g$ without their target-states (3). Finally, the BDD-variables used for representing the events are excluded (4). Hence, the computed BDD represents all the states that enable $\sigma$. Similarly, the corresponding BDDs for the basic state sets are computed as below

$$f = \mathcal{B}(\delta) \wedge \mathcal{B}(\acute{Q}_x) \tag{5}$$
$$g = f \wedge \mathcal{B}(Q^\sigma) \wedge \mathcal{B}(Q_{sup}) \tag{6}$$
$$\mathcal{B}(Q^\sigma_{\mathbf{f}}) = \exists\, (B^{\Sigma} \cup B^{\acute{Q}}).g \tag{7}$$
$$\mathcal{B}(Q^\sigma_{\mathbf{a}}) = g \wedge \neg \mathcal{B}(Q^\sigma_{\mathbf{f}}) \tag{8}$$
$$\mathcal{B}(Q^\sigma_{\mathbf{dc}}) = \neg(\mathcal{B}(Q^\sigma_{\mathbf{a}}) \vee \mathcal{B}(Q^\sigma_{\mathbf{f}})). \tag{9}$$

Initially, the BDD for all the transitions that lead to a forbidden state is computed (5). Among those transitions, the ones where their source-states belong to the supervisor are extracted (6). Finally, to merely keep the source-states, the target-state and event variables are removed from BDD $g$, yielding $\mathcal{B}(Q^\sigma_{\mathbf{f}})$. Similarly, the other two state sets are computed.

Regarding the computation of $\mathcal{B}(Q_{sup})$ in the supervisory synthesis process, there are various implementations based on BDDs such as [2] and [6]. We compute $\mathcal{B}(Q_{sup})$ based on the algorithms in [6].

As stated, the don't-care states will be utilized in simplifying the guard expressions. In our implementation, the simplification process is carried out directly on the BDD representation of the state set, and the guards will be generated on the basis of the simplified BDD. The SIMPLIFY operator is applied for this purpose, which is based on Coudert and Madre's restrict function [27]. Given two BDDs $f$ and $g$, $f' = \mathrm{SIMPLIFY}(f, g)$ simplifies $f$ under a constraint $g$, so that $f \wedge g = f' \wedge g$. Hence, $f'$ is logically equal to $f$ on the domain defined by $g$ and is *often* smaller than $f$. In this manner, we can simplify the BDD representations of the state sets by constraining them under $\neg\mathcal{B}(Q^\sigma_{\mathbf{dc}})$, i.e., the states that we care about. Although the `SIMPLIFY` operator does not always yield the most simplified BDD, in most cases, the guards are significantly simplified.

### B. IDD Generation

As mentioned in the previous section, BDDs are used to improve the efficiency of various operations. However, our desire is to obtain a model that has the automata elements as its variable domain. For this reason, we use IDDs [7]. 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 $A_1 \parallel \cdots \parallel A_n$, 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 $|Q_i^A|$. 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.

We emphasize that in this work we have only utilized the structure of an IDD; particularly, it has only been used as an interface between the BDD and the guard. Hence, we do not perform any IDD-operations.

Generate-IDD($bdd$)

1. **If** $bdd$ is 0
2. **then return** 0-IDD
3. **If** $bdd$ is 1
4. **then return** 1-IDD
5. $v_0^{A_s} \Leftrightarrow \texttt{root}(bdd)$
6. $idd \leftarrow$ create a new IDD rooted by $A_s$
7. TraverseBDD($\texttt{low}(bdd), \neg\mathcal{B}\left(v_0^{A_s}\right), idd$)
8. TraverseBDD($\texttt{high}(bdd), \mathcal{B}\left(v_0^{A_s}\right), idd$)
9. **return** $idd$

BDD-to-States $\left(B^{A_s}\right)$

1. $Q_{\texttt{edge}}^{A_s} \leftarrow \{\}$
2. **for each** $q$ **in** $Q^{A_s}$
3. **do if** $\left(B^{A_s} \wedge \mathcal{B}(q)\right)$ is not 0
4. **then** add $q$ to $Q_{\texttt{edge}}^{A_s}$
5. **return** $Q_{\texttt{edge}}^{A_s}$

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; and 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.

A BDD is converted to an IDD by traversing it in a top-down depth-first manner and performing the following main steps.

1) For each new BDD-node $v_i^{A_s}$ that is reached, create an IDD rooted by $A_s$, $idd$.
2) Continue traversing until a variable $v_j^{A_t}$ is reached where $A_t \neq A_s$.
3) Create an IDD rooted by $A_t$, $child$.
4) Extract the sub-BDD between $v_i^{A_s}$ and $v_j^{A_t}$ that represents some states of automaton $A_s$, i.e., $B^{A_s}$.
5) Add $child$ to $idd$'s children and label the edge with $Q_{\texttt{edge}}^{A_s}$.
6) Repeat the procedure from step 1.

The result is correct under the assumption that the BDD has a fixed variable ordering.

GENERATE − IDD shows a pseudo algorithm that works as the mentioned procedure (some parts of the algorithms are self-explanatory and are therefore not explained in the sequel). If $bdd$ is not 0 or 1, the algorithm starts to extract the root of the BDD, $v_0^{A_s}$–the BDD variable that is used to represent automaton $A_s$–and creates an IDD rooted by $A_s$ (lines 5 and 6). Then, the BDD is traversed in a depth-first manner. $\texttt{low}(bdd)$ and $\texttt{high}(bdd)$ give the low and high children of $bdd$. $\neg\mathcal{B}\left(v_0^{A_s}\right)$ and $\mathcal{B}\left(v_0^{A_s}\right)$ are two sub-BDDs that keep track of the variables belonging to a single automaton, used to compute the states belonging to an automaton. Since only the paths that lead to the 1-terminal is of interest, in the algorithm we disregard the 0-terminal of the IDD. In the TRAVERSE − BDD algorithm, while traversing the BDD, the sub-BDDs that correspond to the states in a single automaton are created (lines 3, 15 and 16). In line 4, the sub-BDD is converted to its corresponding states, $Q_{\texttt{edge}}^{A_s}$, by BDD-TO-STATES. If the current BDD node has not been visited, a new IDD rooted by $A_t$, $iddChild$, is created and then rest of the BDD is traversed to compute $iddChild$ (lines 7–9). In line 10, $iddChild$ is added as one of the $idd$'s children and the edge between them is labeled by $Q_{\texttt{edge}}^{A_s}$. If the BDD node $v_j^{A_t}$ has been visited before and if the edge connecting $idd$ and $iddChild$ already exists, the label of the edge is extended by $Q_{\texttt{edge}}^{A_s}$, but if there does not exist such an edge a new edge is created (lines 11–14).

The time complexity of the algorithm is $O\left(|bdd| \cdot \left|Q^{A_M}\right| \cdot \left|\mathcal{B}(Q^{A_M})\right| \cdot \lceil \log\left(|Q^{A_M}|\right) \rceil\right)$, where $A_M$ is the automaton with most number of states. Since the algorithm works in a depth-first manner without checking a BDD node more than one time, it performs $(\text{number of nodes} + \text{number of edges})$ checks. Since the number of edges in a BDD is twice as many as its nodes, the algorithm will perform $2 \cdot |bdd| + |bdd| = 3 \cdot |bdd|$ checks. In each check, BDD-TO-STATES is called which has time complexity $O\left(\left|Q^{A_M}\right| \cdot \left|\mathcal{B}\left(Q^{A_M}\right)\right| \cdot \lceil \log\left(|Q^{A_M}|\right) \rceil\right)$. This shows that the time complexity of the algorithm only depends on the number of states of the biggest automaton. Since the automata in the models, typically, do not contain many states, the IDD generation can be performed efficiently.

### C. Heuristic Minimization Techniques

Since the minimization is carried out on the Boolean variables, some information, related to the structure of the automata is lost, which impacts the size of the guard. We introduce two heuristic techniques in an attempt to obtain smaller guards.

*1) Complement States (CS):* The corresponding propositional formula for an IDD edge (containing some states that belong to a single automaton) will be the logical disjunction or conjunction of those elements depending on the type of the guard. In cases where the number of states on an edge $Q_{edge}^{A_i} \subseteq Q^{A_i}$ is greater than the complement (remaining) states of $A_i$, $C\left(Q_{edge}^{A_i}\right)$, the guard generated based on the latter state set will be smaller. If the propositional formula for $Q_{edge}^{A_i}$ is $\bigvee_{q \in Q_{edge}^{A_i}} \left(q^{A_i} = q\right)$, then, the complement formula based on $C\left(Q_{edge}^{A_i}\right)$ will be $\bigwedge_{q \in \left(Q^{A_i} \setminus Q_{edge}^{A_i}\right)} \left(q^{A_i} \neq q\right)$. For

instance, if $q^A \neq q_1^A \wedge q^A \neq q_2^A$ holds for automaton $A$ in Fig. 1(a), we can instead write $q^A = q_3^A$.

Traverse − BDD $\left(bdd, B^{A_s}, idd\right)$

1. $v_j^{A_t} \Leftrightarrow \texttt{root}(bdd)$
2. **if** $bdd$ is not 0
3. **then if** $bdd$ is 1 or $A_t \neq A_s$
4. **then** $Q_{\texttt{edge}}^{A_s} \leftarrow$ BDD-to-States $\left(B^{A_s}\right)$
5. $\texttt{sourceNode} \leftarrow$ get the node in $idd$ representing $A_s$
6. **if** $v_j^{A_t}$ has not been visited
7. **then** $iddChild \leftarrow$ create a new IDD rooted by $A_t$
8. Traverse-BDD(low-child of $bdd, \neg\mathcal{B}\left(v_j^{A_t}\right), iddChild$)
9. Traverse-BDD(high-child of $bdd, \mathcal{B}\left(v_j^{A_t}\right), iddChild$)
10. add $iddChild$ to the children of $idd$ and label the edge with $Q_{\texttt{edge}}^{A_s}$
11. **else if** the edge "$idd \rightarrow iddChild$" already exists
12. **then** $\texttt{label}(idd, iddChild) \leftarrow \texttt{label}(idd, iddChild) \cup Q_{\texttt{edge}}^{A_s}$
13. **else** add $iddChild$ to the children of $idd$
14. $\texttt{label}(idd, iddChild) \leftarrow Q_{\texttt{edge}}^{A_s}$
15. **else** Traverse − BDD $(\texttt{low}(bdd), B^{A_s} \wedge \neg\mathcal{B}\left(v_j^{A_t}\right), idd)$
16. Traverse − BDD$(\texttt{high}(bdd), B^{A_s} \wedge \mathcal{B}\left(v_j^{A_t}\right), idd)$

Since this heuristic is applied to each edge on the IDD, the final guard will consist of a mix of both equality $\left(q^{A_i} = q_j^{A_i}\right)$ and non-equality $\left(q^{A_i} \neq q_r^{A_i}\right)$ terms.

*2) Independent States (IS):* We begin to explain this heuristic by Example III.2. In the example, we have $Q_{\mathbf{a}}^{use_{R_1}^A} = \left\{\left\langle q_1^A, q_1^B, q_1^C, q_1^D\right\rangle, \left\langle q_1^A, q_3^B, q_1^C, q_1^D\right\rangle\right\}$, $Q_{\mathbf{f}}^{use_{R_1}^A} = \left\{\left\langle q_1^A, q_2^B, q_1^C, q_2^D\right\rangle\right\}$, and $G_{\mathbf{f}}^{use_{R_1}^A} = q^A \neq q_1^A \vee q^B \neq q_2^B \vee q^C \neq q_1^C \vee q^D \neq q_2^D$. An interesting feature about this case is that the substate $q_2^B$ is not included in $Q_{\mathbf{a}}^{use_{R_1}^A}$. Hence, it suffices to merely include $q^B \neq q_2^B$ in the guard without concerning about the other terms. In other words, if $q^B = q_2^B$, no matter what the current states of the other automata are, event $use_{R_1}^A$ should be disabled.

*Definition V.1 (Independent State):* An independent state is a state in an automaton $A_i$, where $A_i$ is determined by the following equation:

$$\exists q \in Q_{\mathbf{1}}^{\sigma},\ \exists\ A_i \mid \forall\ q' \in Q_{\mathbf{2}}^{\sigma} : \Phi_i(q) \neq \Phi_i(q'),$$

where $Q_{\mathbf{1}}^{\sigma} = Q_{\mathbf{a}}^{\sigma}$ and $Q_{\mathbf{2}}^{\sigma} = Q_{\mathbf{f}}^{\sigma}$ or *vice versa*, i.e., $Q_{\mathbf{1}}^{\sigma} = Q_{\mathbf{f}}^{\sigma}$ and $Q_{\mathbf{2}}^{\sigma} = Q_{\mathbf{a}}^{\sigma}$. The independent state is $\Phi_i(q)$.

Hence, for the corresponding propositional formula of a state $q$ belonging to $Q_{\mathbf{a}}^{\sigma}$ or $Q_{\mathbf{f}}^{\sigma}$, it suffices to merely include the independent state (if there exists such a state) $\Phi_i(q)$.

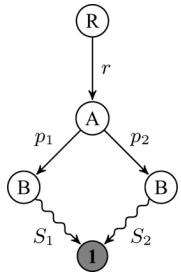


Fig. 3. Recursive representation of an IDD.

### *D. Guard Generation*

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.

The pseudo algorithm GENERATE-GUARD shows how $\mathcal{G}_{\mathbf{a}}^{\sigma}$ is generated based on an IDD. $\mathcal{G}_{\mathbf{f}}^{\sigma}$ can be similarly computed. 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. 3 is

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

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. By following the lines 4, 9, 14, 19, 22 ,and 25–30 of the algorithm, the mentioned procedure can be realized. If the condition for the CS heuristic is satisfied (line 7), described in Section V-C, the expression of the edge is generated based on the remaining states of the automaton (line 8). PF is computed according to(1). When the IS heuristic can be applied (line 12) all the states in the label should be independent states as described in Section V-C. In the algorithm, we represent the independent states by their corresponding expression and set $indp[stateExpr]$ to true indicating that $stateExpr$ represents independent states. If IS is applicable, the IDD will not be traversed anymore and the generated expression can be replaced by all of the expressions computed in the preceding IDDs. For instance, in Fig. 3, if there exists an independent expression in $S_1$, say $d$, the expression can be reduced to

$$d \vee (r \wedge p_2 \wedge S_2).$$

Thus, in the algorithm, when an independent expression is identified, it will backtrack to the first sub-IDD that has less than two children and replace the entire path by the independent expression (lines 12–18). The parentheses handling for the statements are not included in the algorithm. The algorithm uses a lookup table that saves the corresponding expressions of the sub-IDDs. So, if an IDD is already visited, its corresponding expression will be obtained by the lookup table and thus the algorithm

will call each sub-IDD only once. Consequently, since the algorithm generates the guard in a depth first manner, the time complexity of the algorithm will be $O(\text{number of nodes of } idd + \text{number of edges of } idd)$.

Generate-Guard($idd$)

1. **if** $idd$ is 1-terminal
2. **then return** empty string
3. $i \leftarrow 0$
4. **for each** $child$ in `children`($idd$)
5. **do** $A_s \leftarrow$ root of $idd$
6. $Q_{\texttt{edge}} \leftarrow$ `label`($idd, child$)
7. **if** CS heuristic is applicable
8. **then** $stateExpr \leftarrow \neg\texttt{PF}\left(Q^{A_s} \backslash Q_{\texttt{edge}}\right)$
9. **else** $stateExpr \leftarrow \texttt{PF}(Q_{\texttt{edge}})$
10. $expr \leftarrow$ empty string
11. **if** $child$ has not been visited
12. **then if** IS is applicable
13. **then** $indp[stateExpr] \leftarrow$ `true`
14. **else** $temp \leftarrow$ Generate-Guard($child$)
15. **if** $indp[temp]$
16. **then** $stateExpr \leftarrow temp$
17. **if** $idd$ has less than 2 children
18. **then** $indp[stateExpr] \leftarrow$ `true`
19. **else** $expr \leftarrow temp$
20. **else** $expr \leftarrow$ `lookup`($child$)
21. **if** $expr$ is not empty
22. **then** $terms[i] \leftarrow stateExpr + \text{“}\wedge\text{”} + expr$
23. **else** $terms[i] \leftarrow stateExpr$
24. $i \leftarrow i + 1$
25. $guard \leftarrow$ empty string
26. **if** $terms$ is not empty
27. **then** $guard \leftarrow terms[0]$
28. **for** $j \leftarrow 1$ **to** `length`($terms$)
29. **do** $guard \leftarrow guard + \text{“}\vee\text{”} + terms[j]$
30. **return** $guard$

## VI. From Guards to EFA

A major advantage of representing the supervisor as a set of guards is that they can be attached to the original models, leading to a modular supervisor represented by *extended finite automata* (EFAs) [15]. An EFA is a modeling formalism with automata extended with variables, guard expressions and action functions used to update the variables. There are no "restrictions on how variables are shared between extended automata, thus all extended automata are allowed to update all variables as long as the composition is well defined" [15]. EFAs gives "compact representations of huge state-spaces, and hence simplify the modeling of systems of industrially interesting sizes" [15].

From a controller implementation perspective, there are two main advantages of having a modular supervisor represented by EFA.

1) Representing a system by EFAs can make it easier to implement a supervisor on a controller, because the guards can easier be transformed to controller programming languages that are based on logic expressions.
2) Typically, a modular supervisor consumes less memory in a controller. The reason is that the synchronization will be performed online in the controller, see [17], [28], and [29], which will alleviate the problem of exponential growth of the number of states in the synchronization.

Furthermore, if the plants and specifications are modeled by EFAs, then the users can work on a seamless framework where they start by modeling the system by EFAs and end by getting the supervisor in form of EFAs. The users can then repeat this procedure iteratively by adding modifications to the resulting supervisor and perform synthesis again.

Basically, the guards can be attached to the plants and specifications in three main steps.

1) Introduce variables $q^{A_i}$ which hold the current state of automaton $A_i$.
2) For each transition, add an action function that updates $q^{A_i}$ to the new state.
3) Attach $\mathcal{G}^{\sigma}_{\star}$ to all transitions that have $\sigma$ as their event.

From an implementation perspective, the first two steps can be performed implicitly so that they become transparent to the user. The EFA framework has been implemented in the supervisory control tool Supremica [30], [31]. For more details about EFAs refer to [15].

## VII. Case Study—Car Manufacturing Cell

The guard generation procedure discussed in the previous sections will be applied to a relatively complicated example in a *car manufacturing cell*.

We have generated the guards with different implementations:

- $I_1$. All the state sets are represented by BDDs and the operations are performed on the BDDs.
- $I_2$. $I_1+$ minimizing the BDD by utilizing don't-care states using the SIMPLIFY operator.
- $I_3$. $I_2+$ generating IDDs and applying heuristic techniques.

For each case, the adaptive guard $\mathcal{G}^{\sigma}_{\star}$ has been computed and the sizes have been compared.

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

Consider a car manufacturing cell with five manufacturing devices (machines): two robots, a fixture, a turntable, and a conveyor. The task of the cell is to spot weld plates to both sides of the floor of a car. To avoid collisions between the machines, some physical volumes (zones) are defined in the cell so that only one resource at a time may be in a specific zone. In order

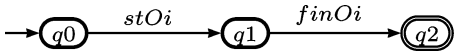


Fig. 4. The automaton model for operation $i$.

TABLE II
THE SIZE OF THE GENERATED GUARDS FOR THE CAR MANUFACTURING CELL. THE NUMBERS WITHIN THE PARENTHESES INDICATE THE SIZES OF THE CORRESPONDING BDDS ($I_1$ AND $I_2$) AND IDDS ($I_3$)

| | | Implementations | | |
|---|---|---|---|---|
| | | $I_1$ | $I_2$ | $I_3$ |
| Events | $stO33$ | 2295 (1087) | 8 (6) | 7 (4) |
| | $stO54$ | 1012 (775) | 1 (1) | 1 (1) |
| | $stO58$ | 2571 (726) | 6 (5) | 4 (4) |
| | $stO72$ | 791 (411) | 1 (1) | 1 (1) |
| | $stO73$ | 587 (299) | 1 (2) | 1 (1) |
| | $stO75$ | 1459 (813) | 1 (4) | 1 (3) |
| | $stO77$ | 1214 (791) | 1 (2) | 1 (1) |
| | $stO79$ | 552 (555) | 1 (1) | 1 (1) |
| | $stO83$ | 1207 (561) | 6 (5) | 4 (4) |
| | $stO85$ | 1207 (561) | 21 (6) | 21 (5) |
| | $stO87$ | 1207 (561) | 15 (6) | 15 (5) |
| | $stO113$ | 11521 (5717) | 57 (12) | 17 (10) |

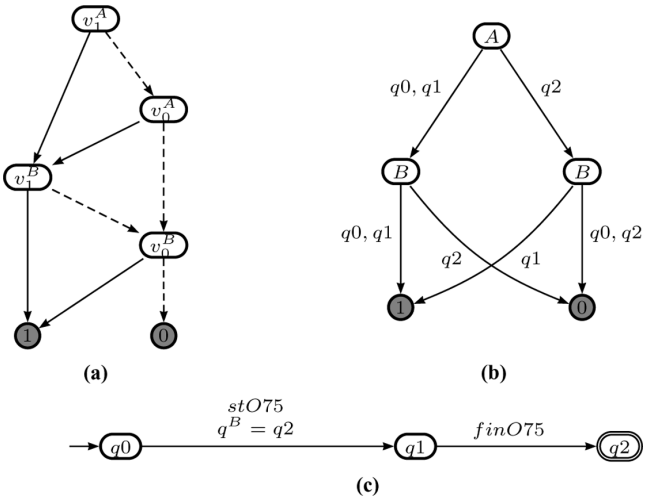


Fig. 5. Corresponding BDD (generated based on $Q_{\mathbf{f}}^{stO75}$), IDD and EFA for event $stO75$. (a) BDD. (b) IDD. (c) EFA of operation 75 after synthesis, i.e., adding guard.

to accomplish the task, the machines should operate in a specific order. The cell has been structured into *sequences of operations* [33], [34]. "An operation represents a piece of work that a machine should perform without interrupts and without communication with other machines" such as welding of a plate or the movement of a fixture [4]. Each machine executes a number of operations according to specific orders and interactions with other machines. An expanded version of the cell is described in detail in [33].

In this model, the operations are considered as the plants and the orders in which they should be executed are considered as the specifications. The automaton model for an operation consists of two events: $stOi$ meaning that operation $i$ has started and $finOi$ meaning that operation $i$ has finished its work. Events $stOi$ and $finOi$ are controllable and uncontrollable, respectively. Fig. 4 shows the automaton model for an operation. In this example, the model contains 24 plant automata, i.e., operations, and 46 specification automata modeling the relations between the individual operations.

The closed-loop automaton consists of 8871 reachable states, where 7646 of those are blocking. After the synthesis, the nonblocking supervisor consists of 1225 states, 48 controllable events, and 3480 transitions. Table II shows the size of the guards for different implementations. The events that are always enabled by the supervisor (when the guard becomes true) are not included in the table. The numbers within the parentheses indicate number of nodes for the corresponding BDDs ($I_1$ and $I_2$) and IDDs ($I_3$). The supervisory synthesis, which is merely computed once for all events, was performed by BDD operations and was completed in 2 s. The computation time for generating the guards for all events was less than a second.

From Table II, it can be observed that by converting the simplified BDDs to IDDs and applying the heuristic techniques the guards become smaller. For instance, by comparing $I_1$ and $I_2$ for event $stO113$, we can see that the guard is significantly reduced when applying the heuristics. Another point is that the sizes of the BDDs have been reduced extremely after applying the `SIMPLIFY` operator (compare $I_1$ and $I_2$), which in turn impacts the sizes of the guards.

As an example, we explain how $\mathcal{G}_{\star}^{stO75}$ is generated, and in this case $Q_{\mathbf{f}}^{stO75}$ will yield the minimal guard. Fig. 5 shows $\mathcal{B}(Q_{\mathbf{f}}^{stO75})$ and its corresponding IDD; together with the generated EFA for event $stO75$.

The BDD is transformed to its corresponding IDD by algorithm `GENERATE−IDD`. In this example, we assume that for a three-state automaton $A$, $\chi_{\{q0\}}(v^A) = v_1^A \wedge \neg v_0^A$, $\chi_{\{q1\}}(v^A) = \neg v_1^A \wedge v_0^A$, and $\chi_{\{q2\}}(v^A) = \neg v_1^A \wedge \neg v_0^A$.

Initially, for the BDD-node $v_1^A$, an IDD rooted by $A$ is created. The next new BDD-node that does not represent $A$ is $v_1^B$, thus a new IDD rooted by $B$ is created. There are two paths from $v_1^A$ to $v_1^B$: $v_1^A \vee \left(\neg v_1^A \wedge v_0^A\right)$, which represents states $q0$ and $q1$ of automaton $A$. Hence, $B$ is added as one of $A$'s children and the edge is labeled by $q0, q1$ (for simplicity, we mix the two edges and represent the states on a single edge). Continuing the traverse from the BDD-node $v_1^B$, the next new BDD node that does not represent $B$ is the 1-terminal. From $v_1^B$, there are two paths to the 1-terminal representing states $q0$ and $q1$ of automaton $B$. Thus, the IDD node 1 is added as one of $B$'s children and the edge is labeled by $q0, q1$. Similarly, the other paths are traversed and the IDD-edges are created.

If we transform the IDD to its corresponding expression, without considering the heuristic techniques, the following expression is obtained:

$$\neg\Big(((q^B = q0 \vee q^B = q1) \wedge (q^A = q0 \vee q^A = q1)) \vee (q^B = q1 \wedge q^A = q2)\Big). \tag{10}$$

In this expression, the states $q0$ and $q1$ of automaton $B$ are independent states. However, this conclusion cannot be derived directly because $Q_{\mathbf{a}}^{stO75}$ is not given (due to lack of space). Thus,

by applying the IS heuristic, the above expression is simplified to

$$(q^B \neq q0 \wedge q^B \neq q1) \wedge (q^B \neq q1) \tag{11}$$

and, finally, the CS heuristic simplifies the expression to

$$q^B = q2 \wedge q^B \neq q1 \tag{12}$$

due to the fact that $Q^B = \{q0, q1, q2\}$. Hence, since both terms in (12) are related to a single automaton, we get

$$\mathcal{G}_\star^{stO75}: \quad q^B = q2. \tag{13}$$

This expression is indeed more understandable and tractable for a user compared to a BDD or an expression with 6 terms (before the heuristics was applied), see Table II. Fig. 5(c) depicts the generated EFA, described in Section VI, where $\mathcal{G}_\star^{stO75}$ has been added to the transition in operation 75 that contains event $stO75$. Consequently, nearly 8000 blocking states are avoided by only a number of small guard expressions.

Worth mentioning that the algorithm has also been applied to larger examples where the guards have been computed efficiently. The above example was, however, more illustrative and industrially relevant, and was therefore selected. In a larger example, where the number of reachable states and the states of the supervisor were around $10^{11}$ and $10^9$, respectively, the generated guards became small and tractable as well (similar to Table II) and were generated in almost 3 s [35].

## VIII. Conclusion and Future Works

In this paper, we introduced a method for generating tractable and comprehensible propositional formulae, i.e., guards, representing the supervisor's behavior. The overall procedure of our method can be divided into five steps. First, the supervisor is generated using BDDs. Second, for each event, a BDD is generated specifying the states where the event must be forbidden or allowed to occur. Third, this BDD is then simplified by using a don't-care BDD. The don't-care BDD is constructed based on a set of don't-care states, such as states that can never be reached from the initial state. This minimization process is efficiently performed by BDD-based operations. Fourth, the simplified BDD is converted to an IDD, which makes it easier to generate and manipulate the propositional formulae. Finally, the guard is generated by transforming the IDD to its corresponding propositional formula. At this step, some heuristic techniques are also applied, which typically reduces the size of the guard significantly.

Furthermore, 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 case study shows that by applying this approach, the guards are computed efficiently and become more tractable for the users due to smaller sizes.

There are some directions in which we could extend and apply our method. For instance, it is possible to apply additional heuristics to further decrease the size of the guards. Furthermore, in this paper, we have assumed that the initial models are modeled by DFAs and can then be extended to EFAs. However, from a user and modeling perspective it would be better to use EFAs from the beginning. The next step is to, given a system modeled by EFAs, return a supervisor represented by the given EFA, extended with additional guards on the transitions [36].

## References


[1] P. J. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proc. IEEE*, vol. 77, no. 1, pp. 81–89, 1989.

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

[3] L. Feng, W. M. 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 Trans. Control Syst. Technol.*, vol. 18, no. 2, pp. 414–429, 2010.

[5] S. B. Akers, "Binary decision diagrams," *IEEE Trans. Comput.*, vol. 27, pp. 509–516, Jun. 1978.

[6] A. Vahidi, M. Fabian, and B. Lennartson, "Efficient supervisory synthesis of large systems," *Control Eng. 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, Dept. Elect. Eng., Linköping Univ., Linköping, Sweden, 1997.

[8] E. J. McCluskey, "Minimization of boolean functions," *Bell Syst. Tech. J.*, 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 *Proc. 8th Int. Workshop on Discrete Event Systems, WODES'06*, pp. 161–167.

[10] Y. Li and W. M. Wonham, "Control of vector discrete-event systems. II. Controller synthesis," *IEEE Trans. Autom. 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 Trans. Autom. Control*, vol. 37, no. 5, pp. 692–697, May 1992.

[12] A. Giua and F. DiCesare, "Blocking and controllability of petri nets in supervisory control," *IEEE Trans. Autom. 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. M. Wonham, "Nonblocking supervisory control of state tree structures," *IEEE Trans. Autom. Control*, vol. 51, no. 5, pp. 782–793, May 2006.

[15] M. Skoldstam, K. Åkesson, and M. Fabian, *Modeling of Discrete Event Systems Using Finite Automata With Variables*. Piscataway, NJ: IEEE Press, 2007.

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

[17] K. Åkesson, "Methods and tools in supervisory control theory: Operator aspects, computation efficiency and applications," Ph.D. dissertation , Signals and Systems, Chalmers Univ. Technol., Göteborg, Sweden, 2002.

[18] 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 *Proc. 5th Annu. IEEE Symp. Logic Comput. Sci., LICS'90*, Jun. 1990, pp. 428–439.

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

[20] C. Ma and W. M. Wonham, "STSLib and its application to two benchmarks," in *Proc. 9th Int. Workshop Discrete Event Syst., WODES'08*, May 2008, pp. 119–124.

[21] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, pp. 379–423, 1948.

[22] R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, 1986.

[23] H. R. Andersen, An introduction to binary decision diagrams Dept. Inform. Technol., Tech. Univ. Denmark, Lyngby, Denmark, Tech. Rep., 1997.

[24] 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.

[25] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Comput. Surv.*, vol. 24, no. 3, pp. 293–318, 1992.
[26] A. Aziz, S. Tasiran, and R. K. Brayton, "BDD variable ordering for interacting finite state machines," in *Proc. 31st Annu. Design Autom. Conf., DAC'94*, New York, 1994, pp. 283–288.
[27] O. Coudert and J. C. Madre, "A unified framework for the formal verification of sequential circuits," in *Proc. IEEE Int. Conf. Comput.-Aided Design, ICCAD-90. Dig. Tech. Papers*, Nov. 1990, pp. 126–129.
[28] A. Hellgren, M. Fabian, and B. Lennartson, "Synchronized execution of discrete event models using sequential function charts," in *Proc.38th Decision and Control*, Phoenix, AZ, 1999, pp. 2237–2242.
[29] A. Hellgren, B. Lennartson, and M. Fabian, "Modelling and PLC-based implementation of modular supervisory control," in *Proc. 6th Int. Workshop Discrete Event Syst.*, 2002, pp. 371–376.
[30] K. Åkesson, M. Fabian, H. Flordal, and R. Malik, "Supremica—an integrated environment for verification, synthesis and simulation of discrete event systems," in *Proc. 8th Int. Workshop on Discrete Event Syst.*, 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 *Proc. 11th Mediterranean Conf. Control Autom.*, Rhodos, Greece, 2003.
[32] Javabdd [Online]. Available: http://javabdd.sourceforge.net
[33] J. Richardsson and M. Fabian, "Modeling the control of a flexible manufacturing cell for automatic verification and control program generation," *Int. J. Flexible Manuf. Syst.*, vol. 18, no. 3, pp. 191–208, 2006.
[34] B. Lennartson, K. Bengtsson, C. Yuan, K. Andersson, M. Fabian, P. Falkman, and K. Åkesson, "Sequence planning for integrated product, process and automation design," *IEEE Trans. Autom. Sci. Eng.*, vol. 7, no. 4, pp. 791–802, Oct. 2010.
[35] M. R. Shoaei, B. Lennartson, and S. Miremadi, "Automatic generation of controllers for collision-free flexible manufacturing systems," in *Proc. IEEE Int. Conf. Autom. Sci. Eng.*, Aug. 2010, pp. 368–373.
[36] S. Miremadi, B. Lennartson, and K. Åkesson, "A BDD-based approach for modeling plant and supervisor by extended finite automata," Chalmers Univ. Technol., Gothenburg, Sweden, 2010.

**Sajed Miremadi** was born in 1983 in Linköping, Sweden. He received the B.Sc. degree in computer engineering from Sharif University of Technology, Tehran, Iran, in 2006 and the M.Sc. degree in computer science from Linköping University, Linköping, Sweden, in 2008. Since 2008, he has been working towards the Ph.D. degree in automation at Chalmers University of Technology, Gothenburg, Sweden.

His current research interests include supervisory control and optimization of (timed) discrete-event systems using formal methods.

**Knut Åkesson** received the M.S. degree in computer science and engineering from the Lund Institute of Technology, Lund, Sweden, in 1997 and the Ph.D. degree in control engineering from the Chalmers University of Technology, Gothenburg, Sweden, in 2002.

Currently, he is an Associate Professor at the Department of Signals and Systems, Chalmers University of Technology, where his main research interest is to develop and applying formal methods for verification and synthesis of control logic.

**Bengt Lennartson** (M'10) was born in 1956 in Gnosjö, Sweden. He received the Ph.D. degree in automatic control from the Chalmers University of Technology, Gothenburg, Sweden, in 1986.

Since 1999, he has been a Professor and the Chair of Automation, Department of Signals and Systems. He was Dean of Education at the Chalmers University of Technology from 2004 to 2007, and since 2005, he is a Guest Professor at University West, Trollhä ttan. He is (co)author of two books and 180 peer reviewed international papers with 2200 citations (GS). His main areas of interest include discrete-event and hybrid systems, especially for manufacturing applications, as well as robust feedback control.

Prof. Lennartson is currently a Member of the Advisory Board for the IEEE TRANSACTIONS ON AUTOMATION SCIENCE AND ENGINEERING. He was the Chairman of the Ninth International Workshop on Discrete-Event Systems, WODES'08 and an Associate Editor for *Automatica*.