Discrete Event Dynamic Systems
https://doi.org/10.1007/s10626-020-00334-w

Compositional coordinator synthesis of extended finite automata

Martijn A. Goorden[1] · Martin Fabian[2] · Joanna M. van de Mortel-Fronczak[1] · Michel A. Reniers[1] · Wan J. Fokkink[3] · Jacobus E. Rooda[1]

Received: 9 May 2019 / Accepted: 6 December 2020 / Published online: 07 January 2021

Abstract
To avoid the state-space explosion problem, a set of supervisors may be synthesized using divide and conquer strategies, like modular or multilevel synthesis. Unfortunately, these supervisors may be conflicting, meaning that even though they are individually non-blocking, they are together blocking. Abstraction-based compositional nonblocking verification of extended finite automata provides means to verify whether a set of models is nonblocking. In case of a blocking system, a coordinator can be synthesized to resolve the blocking. This paper presents a framework for compositional coordinator synthesis for discrete-event systems modeled as extended finite automata. The framework allows for synthesis of a coordinator on the abstracted system in case compositional verification identifies the system to be blocking. As the abstracted system may use notions not present in the original model, like renamed events, the synthesized coordinator is refined such that it will be nonblocking, controllable, and maximally permissive for the original system. For each abstraction, it is shown how this refinement can be performed. It turns out that for the presented set of abstractions the coordinator refinement is straightforward.

Keywords Extended finite automata · Nonblocking · Compositional synthesis · Supervisory control theory

1 Introduction

The design of supervisory controllers for systems with discrete-event behavior becomes increasingly complex, while failures can result in human fatalities or financial losses. Formal methods, such as Supervisory Control Theory proposed by Ramadge and Wonham (1987), can be used to make this complexity manageable. A supervisor is synthesized based

This work is supported by Rijkswaterstaat, part of the Ministry of Infrastructure and Water Management of the Government of the Netherlands, and by the Swedish Science Foundation, Vetenskapsrådet

✉ Martijn A. Goorden
m.a.goorden@tue.nl

Extended author information available on the last page of the article.

on discrete-event models, like finite automata (FAs), of the uncontrolled system and of the control requirements. To cope with the notorious state-space explosion problem, one might deploy a combination of more advanced synthesis techniques and modeling with extended finite automata (EFAs).

There exist several automata-based synthesis procedures to obtain one or more supervisors: modular (Wonham and Ramadge 1988), hierarchical (Zhong and Wonham 1990), decentralized (Rudie and Wonham 1992), distributed (Cai and Wonham 2010), compositional (Mohajerani et al. 2017), and multilevel supervisory control synthesis (Komenda et al. 2016). A set of supervisors, for example obtained with modular supervisory control synthesis, may be conflicting. This gives rise to global blocking of the system, see Wonham and Ramadge (1988), de Queiroz and Cury (2000), Åkesson et al. (2002), Hill and Tilbury (2006), and Cassandras and Lafortune (2008). A nonconflicting verification can be performed on the set of synthesized supervisors to verify this.

The worst-case computational complexity of the nonconflicting verification is the same as of the monolithic supervisory control synthesis, see Cassandras and Lafortune (2008) and Wonham et al. (2017). There exist several nonconflicting verification procedures in the literature that deploy different abstraction techniques to reduce the computational complexity for most cases, see Flordal and Malik (2006), Pena et al. (2008), Su et al. (2010), and Mohajerani et al. (2016).

When such nonconflicting verification reports a conflict, a supervisor can be synthesized to solve it. This supervisor is called a *coordinator*, as introduced in Wong and Wonham (1998). In Su et al. (2009), it is suggested to synthesize a coordinator by applying the monolithic synthesis procedure on the collection of synthesized supervisors. In such cases, the advantages of using a non-monolithic synthesis procedure to synthesize the supervisors may be lost. For some special cases, abstractions employing natural observers can be used to synthesize a coordinator, see for example Feng and Wonham (2008). A study by Zita et al. (2017) suggests to use counterexamples to resolve conflicts by refining the system.

A modeler might also opt to use EFAs to model discrete-event systems. EFAs are finite automata enhanced with discrete variables, see Cheng and Krishnakumar (1996), Chen and Lin (2000), and Skoldstam et al. (2007). This allows for a more compact model representation as shown in Miremadi et al. (2010) and for the usage of state-based requirements as introduced in Ma and Wonham (2005) and Markovski et al. (2010). Furthermore, EFAs allow for efficient symbolic computations with binary decision diagrams, see Fei et al. (2014). Applications of EFAs can be found in several recent case studies of synthesizing supervisors for systems with discrete-event behavior, see for example Fabian et al. (2014), Korssen et al. (2017), and Reijnen et al. (2020). However, theoretical developments in synthesis with EFAs are yet few in numbers. The works published in this context concern monolithic synthesis of Chen and Lin (2001) and Ouedraogo et al. (2011), modular synthesis of Malik and Teixeira (2016) and Malik and Teixeira (2020), where the latter one only solves controllability but gives no guarantees about nonblockingness, and compositional nonconflicting verification of Mohajerani et al. (2016). As non-monolithic synthesis methods for EFAs are sparse, FA-based approaches mixed with monolithic synthesis for EFAs are used in case studies involving EFAs, see for example Reniers (2018). Adapting existing methods of synthesizing multiple supervisors, performing a nonconflicting check, and synthesizing a coordinator for finite automata to the EFA framework allows having theoretically sound results in these case studies. The last step of synthesizing a coordinator for EFAs is currently missing.

This paper builds upon the framework for compositional nonconflicting verification for EFAs as proposed in Mohajerani et al. (2016). The main idea of compositional nonconflicting verification is to apply multiple small and simple conflict equivalence preserving abstractions on a collection of automata until one automaton is obtained, see the top row depicted in Fig. 1. Examples of these abstractions are partial composition, variable unfolding, and event merging. For each of these abstractions, it is proven that the obtained system is nonblocking if and only if the system before abstraction is nonblocking. Therefore, the nonblocking property is preserved resulting in a, so-called, conflict equivalent system. Finally, when a single EFA without variables is obtained (which can be interpreted as an FA), a standard monolithic nonblocking verification procedure is applied. The result of this verification is returned as the result of this compositional nonblocking verification procedure. Numerical results in Mohajerani et al. (2016) show that compositional nonblocking verification can be efficiently performed.

In this paper, we propose a procedure using the result of the nonconflicting verification of Mohajerani et al. (2016) to synthesize a coordinator in case of conflicting supervisors within the framework of EFAs. Such a synthesized coordinator needs to ensure that the final closed-loop system of the plant, supervisors, and coordinator is nonblocking, controllable, and maximally permissive. Using the resulting single automaton of the compositional nonconflicting verification, we propose to synthesize a coordinator, for example by the monolithic synthesis of Ouedraogo et al. (2011), to resolve blocking, see Fig. 1. In general, the synthesized coordinator for the single, abstracted automaton may use notions not present in the original system, such as renamed events, and hence may not directly be suitable as a coordinator. We show in this paper how the synthesized coordinator can be refined in order to be a coordinator for the original system ensuring that the final closed-loop system is nonblocking, controllable, and maximally permissive.

For each abstraction defined in Mohajerani et al. (2016), we investigate if and how the coordinator for the abstracted system can be refined to maintain nonblockingness, controllability, and maximal permissiveness, as illustrated in Fig. 2. It has been shown in, for example, Mohajerani et al. (2014) that for FAs, thus without variables and guards, some conflict equivalence preserving abstraction may not be suitable for synthesis. In this paper, we define the notion of coordinator equivalence to determine which abstractions defined in Mohajerani et al. (2016) are also suitable for synthesis refinement. It turns out that these refinements are simple, but may involve more than just renaming as is the case for FA-based compositional synthesis, see Mohajerani et al. (2014).

The method depicted in Fig. 1 shares similarities with modular compositional synthesis as proposed in Malik and Flordal (2008), Mohajerani et al. (2014), and Mohajerani et al. (2017), but there are three important differences. First, these previous works use finite automata, while this paper uses extended finite automata. Second, the method proposed in Malik and Flordal (2008) deploys a two-pass procedure. In the first pass, a sequence of

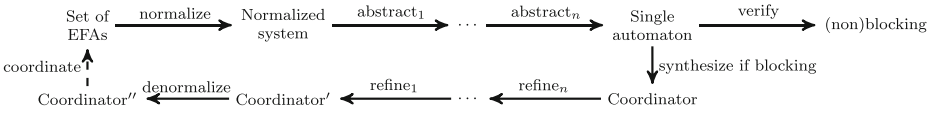


Fig. 1 The top row represents the compositional nonblocking verification of Mohajerani et al. (2016). In this paper, we propose the addition of the bottom row: synthesize a coordinator based on the single, simplified automaton and then transform this coordinator back to the original system

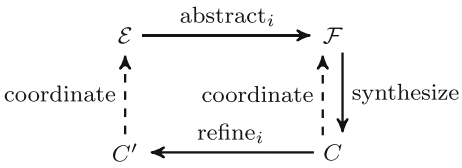


Fig. 2 The structure of the theorems in this paper. The abstraction abstract$_i$ from EFA system $\mathcal{E}$ to $\mathcal{F}$ is one of those mentioned in Mohajerani et al. (2016). The related refinement refine$_i$ from coordinator C to C' is novel work

synthesis equivalence preserving abstractions are performed to simplify the set of models. In the second pass, the abstractions are refined in reversed order by synthesizing a supervisor based on the abstracted system and the synthesized supervisors from the previous refinements. As concluded in Malik and Flordal (2008), the method presented there may result in conflicting supervisors. In our framework, we apply coordinator equivalence preserving abstractions and we synthesize a coordinator only once at the final abstracted system. Third, the methods proposed by Mohajerani et al. (2014) and Mohajerani et al. (2017) synthesize supervisors while abstracting the system. Therefore, their method can guarantee that the resulting set of supervisors is nonconflicting. In our framework, we only synthesize a supervisor after performing abstractions and verifying whether a given set of models is nonconflicting.

This paper is structured as follows. Section 2 provides the preliminaries. The paper continues by introducing the novel notion of coordinator tuples and coordinator equivalence in Section 3, which are the fundament of the framework. Section 4 illustrates the proposed method with an example of a small manufacturing unit. Next, Section 5 shows that normalization is coordinator equivalence preserving, and Section 6 shows several coordinator equivalence preserving EFA-based abstraction-refinement pairs. Section 7 combines the abstractions in an algorithm for compositional nonblocking verification and coordinator synthesis and discusses the computational complexity. Finally, Section 8 concludes the paper.

2 Preliminaries

This section provides a brief introduction to automata and supervisory control theory. This introduction is based on the work of Cassandras and Lafortune (2008), Mohajerani et al. (2016), Ouedraogo et al. (2011), Skoldstam et al. (2007), and Wonham and Cai (2019).

2.1 Finite automata

First we briefly introduce finite automata, since some concepts of extended finite automata relate on their finite automata counterpart. A finite automaton (FA) is a five-tuple $G = (Q, \Sigma, \rightarrow, q_0, Q_m)$, with Q a finite state set of states, Σ a finite set of events, $\rightarrow \subseteq Q \times \Sigma \times Q$ a state transition relation, $q_0 \in Q$ the initial state, and $Q_m \subseteq Q$ a set of marked states.

We denote by Σ^* the set of all finite strings of events in Σ. In the context of supervisory control, the alphabet Σ of a finite automaton (and an extended finite automaton, see below) is partitioned into two disjoint sets of *controllable* Σ_c and *uncontrollable* Σ_u events.

The transition relation can also be written in infix notation $x \xrightarrow{\sigma} y$, and can be extended for traces in Σ^* as follows: $x \xrightarrow{\varepsilon} x$ for all $x \in Q$, and $x \xrightarrow{s\sigma} z$ for all $x, z \in Q$, $\sigma \in \Sigma$, and $s \in \Sigma^*$ if $x \xrightarrow{s} y$ and $y \xrightarrow{\sigma} z$ for some $y \in Q$. The language generated by automaton G is $\mathcal{L}(G) = \{s \in \Sigma^* \mid \exists x \in Q, q_0 \xrightarrow{s} x\}$ and the marked language is $\mathcal{L}_m(G) = \{s \in \mathcal{L}(G) \mid \exists x \in Q_m, q_0 \xrightarrow{s} x\}$.

An automaton is called deterministic if for each state $q \in Q$ and event $\sigma \in \Sigma$ there exists at most one state $y \in Q$ such that $x \xrightarrow{\sigma} y$; otherwise, it is called nondeterministic.

2.2 Extended finite automata

Extended finite automata (EFAs) are FAs extended with bounded discrete variables, see Cheng and Krishnakumar (1996), Chen and Lin (2000), and Skoldstam et al. (2007). In this paper, we follow the EFA concepts of Skoldstam et al. (2007). Nonetheless, these concepts are included in these preliminaries, since several notations and conventions are used in literature. In an EFA, each transition is augmented with a guard and an update (the latter one sometimes also called an action) using variables, constants, the Boolean literals true (**T**) and false (**F**), and the usual arithmetical operators and logical connectives (Ouedraogo et al. 2011). Let V be a finite set of discrete variables. Each variable $v \in V$ is associated with a finite domain $\mathrm{dom}(v)$ of values. A valuation is a mapping $\hat{v} : V \to \bigcup_{v \in V} \mathrm{dom}(v)$ with $\hat{v}(v) \in \mathrm{dom}(v)$ for each $v \in V$. The finite set of all valuations on V is denoted by $\mathrm{Val}(V)$. The initial valuation is denoted by $\hat{v}_0$.

Guards express under which conditions a transition is enabled. A guard is a Boolean expression, or predicate, using variables from V. The set of all guard expressions is denoted by $\mathcal{G}_V$. An example of a guard expression g is $v_1 = 1 \wedge v_2 = 2$. Any valuation $\hat{v} \in \mathrm{Val}(V)$ such that $\hat{v}(v_1) = 1$ and $\hat{v}(v_2) = 2$ ensures that this guard evaluates to true, i.e., $\hat{v} \vDash g$, which we also denote by $g[\hat{v}] = \mathbf{T}$. Guards can be combined with the usual logical connectives. Guards are always evaluated with current-state valuations.

Updates change current-state valuations into next-state valuations after executing an enabled transition. We consider an update to be a collection of n individual update expressions, each updating a single variable from V, where n is the number of variables in V. Formally, let Π_V be the set of all expressions over variables from V. For $p \in \Pi_V$, $p[v_1 \mapsto a_1, \ldots, v_n \mapsto a_n]$ denotes the substitution where each occurrence of variable v_i is replaced by value a_i. For example, $v_1 + v_2[v_1 \mapsto 1] = 1 + v_2$. The valuation function is extended to expressions in the following way: for any $\hat{v} \in \mathrm{Val}(V)$, $\hat{v} : \Pi_V \mapsto \bigcup_{v \in V} \mathrm{dom}(v)$ such that $\hat{v}(p) = p[v_1 \mapsto \hat{v}(v_1), \ldots]$. Therefore, we assume an update expression always evaluates to a single value when all variables are substituted with a value, which is called a *precise update.* Now, an update function is $u : V \to \Pi_V$ such that for all variables the updated value remains within the domain, i.e., for all $v \in V$, $\hat{v} \in \mathrm{Val}(V)$, $\hat{v}(u(v)) \in \mathrm{dom}(v)$. This is called a *domain bounded update function.* Without loss of generality, we also assume that an update function is *total*, i.e., it is defined for all variables in V. Therefore, an update function changes each current-state valuation into a single next-state valuation. The set of all update functions is $\mathcal{U}_V$. Given a valuation $\hat{v} \in \mathrm{Val}(V)$ and an update function $u \in \mathcal{U}_V$, the new valuation $\hat{w}$ can be calculated by $\hat{w}(v) = \hat{v}(u(v))$. For example, update function $u = \{v_1 \mapsto v_1 + v_2, v_2 \mapsto 2\}$ expresses that the new value of v_1 is the current values of v_1 and v_2 summed together and the new value of v_2 is 2. An update function may also be written as a comma separated list of expressions. For example, the update function u may also be written as $v_1 := v_1 + v_2,\ v_2 := 2$. Combining updates can be done as follows. Let update

u_1 be defined on V_1 and update u_2 on V_2. The expression $u_1 \oplus u_2$ denotes the combination of two updates that is defined as follows:

$$(u_1 \oplus u_2)(v) = \begin{cases} u_1(v) & \text{if } v \in V_1 \cap V_2, \forall \hat{v} \in \text{Val}(V_1 \cup V_2) : \\ & \quad \hat{v}(u_1(v)) = \hat{v}(u_2(v)) \\ u_1(v) & \text{if } v \in V_1 \setminus V_2 \\ u_2(v) & \text{if } v \in V_2 \setminus V_1 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

If u_1 and u_2 are not consistent with each other on a variable update, which is the 'otherwise' case, the update function will become undefined for that variable. An example of two non-consistent updates is $x := x + 1$ and $x := x - 1$, as both updates give a different new value of x for the same current value. In the remainder of this paper, we assume that updates are consistent for shared variables (in $V_1 \cap V_2$).

Sometimes, a modeler would like to use (for example due to its conciseness, see the figures in this paper) variable update functions that might have an unclear interpretation. For example, the variable update functions $x := x + 1$ will update a variable outside the range if the current-state value of x is already the upper boundary value. In this paper, we made the assumption that any update function is total and keeps values always within the domain. This excludes currently the formulation of update functions like $x := x + 1$. In the case that future work on EFAs settles down on a specific interpretation of these cases, or EFA-based tooling chooses for a specific interpretation, the theory described in this paper is applicable on the model with the explicit interpretation. In other words, the model as defined by the modeler can be translated under the hood by the tooling into a model with explicit formulations. Therefore, examples illustrating concepts in this paper will use update functions such as $x := x + 1$.

An EFA is a 7-tuple $(L, \Sigma, V, \rightarrow, l_0, \hat{v}_0, L_m)$ where L is a finite set of locations, Σ a finite set of events, V a finite set of variables, $\rightarrow \subseteq L \times \Sigma \times \mathcal{G}_V \times \mathcal{U}_V \times L$ a transition relation, $l_0 \in L$ an initial location, $\hat{v}_0$ the initial valuation, and $L_m \subseteq L$ a set of marked locations.

A transition $(l_1, \sigma, g, u, l_2) \in \rightarrow$ is enabled if $g[\hat{v}_1]$ evaluates to true for the current-state valuation $\hat{v}_1$. After taking the transition the current location of the EFA is l_2 and the global valuation of $\hat{v}_1$ has been updated to $\hat{v}_2(v) = \hat{v}_1(u(v))$ for all $v \in V$. For EFAs, the infix notation is $l_1 \xrightarrow{\sigma,g,u} l_2$.

A state of an EFA is the combination of a location and a valuation. The state space of an EFA captures all these possible states and transitions between these states, see Mohajerani et al. (2016).

Definition 1 (State space) Let $E = (L, \Sigma, V, \rightarrow, l_0, \hat{v}_0, L_m)$ be an EFA. The state space of E is the FA $U(E) = (L_U, \Sigma, \rightarrow_U, l_{U,0}, L_{U,m})$ where

- $L_U = L \times \text{Val}(V)$,
- $((l_1, \hat{v}_1), \sigma, (l_2, \hat{v}_2)) \in \rightarrow_U$ if $(l_1, \sigma, g, u, l_2) \in \rightarrow$, $g[\hat{v}_1] = \mathbf{T}$, and $\hat{v}_2(v) = \hat{v}_1(u(v))$ for all $v \in V$,
- $l_{U,0} = (l_0, \hat{v}_0)$, and
- $L_{U,m} = L_m \times \text{Val}(V)$.

An EFA E is called deterministic if and only if its state space FA $U(E)$ is deterministic. The language of an EFA is defined using the FA-based language definition.

Definition 2 (Language of EFA) Let E be an EFA. The language of E, denoted with $\mathcal{L}(E)$, is defined as $\mathcal{L}(E) = \mathcal{L}(U(E))$. The marked language is defined similarly, i.e., $\mathcal{L}_m(E) = \mathcal{L}_m(U(E))$.

Two EFAs can be combined by using the synchronous composition, see Ouedraogo et al. (2011). As we assume that updates of shared variables are consistent with each other, we only define synchronous composition with consistent updates.

Definition 3 (Synchronous composition EFAs) Let $E^k = (L^k, \Sigma^k, V^k, \rightarrow^k, l_0^k, \hat{v}_0^k, L_m^k)$, $k = 1, 2$ be EFAs such that for shared variables $v \in V^1 \cap V^2$ the initial valuation is the same, i.e., $\hat{v}_0^1(v) = \hat{v}_0^2(v)$. The synchronous composition of E^1 and E^2 is

$$E^1 \parallel E^2 = (L^1 \times L^2, \Sigma^1 \cup \Sigma^2, V^1 \cup V^2, \rightarrow, (l_0^1, l_0^2), \hat{v}_0^1 \oplus \hat{v}_0^2, L_m^1 \times L_m^2)$$

where the transition relation $\rightarrow$ is defined as

- $((l_1^1, l_1^2), \sigma, g, u, (l_2^1, l_2^2)) \in \rightarrow$ if $\sigma \in \Sigma_1 \cap \Sigma_2$, $(l_1^1, \sigma, g^1, u^1, l_2^1) \in \rightarrow^1$ and $(l_1^2, \sigma, g^2, u^2, l_2^2) \in \rightarrow^2$ such that $g = g^1 \wedge g^2$ and $u = u^1 \oplus u^2$;
- $((l_1^1, l_1^2), \sigma, g^1, u^1, (l_2^1, l_1^2)) \in \rightarrow$ if $\sigma \in \Sigma_1 \setminus \Sigma_2$ and $(l_1^1, \sigma, g^1, u^1, l_2^1) \in \rightarrow^1$;
- $((l_1^1, l_1^2), \sigma, g^2, u^2, (l_1^1, l_2^2)) \in \rightarrow$ if $\sigma \in \Sigma_2 \setminus \Sigma_1$ and $(l_1^2, \sigma, g^2, u^2, l_2^2) \in \rightarrow^2$

and the new initial valuation

$$(\hat{v}_0^1 \oplus \hat{v}_0^2)(v) = \begin{cases} \hat{v}_0^1(v) & \text{if } v \in V^1 \cap V^2 \\ \hat{v}_0^1(v) & \text{if } v \in V^1 \setminus V^2 \\ \hat{v}_0^2(v) & \text{if } v \in V^2 \setminus V^1. \end{cases}$$

The synchronous composition of more than two EFAs can be calculated by recursively applying the above definition. For notational simplicity, since the synchronous composition is associative and commutative up to reordering state labels, we write for the synchronous composition of more than two EFAs simply $E^1 \parallel E^2 \parallel \ldots \parallel E^n$ instead of $((E^1 \parallel E^2) \parallel \ldots) \parallel E^n$. Therefore, the location set of $E^1 \parallel E^2 \parallel \ldots \parallel E^n$ is denoted by $L^1 \times L^2 \times \ldots \times L^n$, where L^i is the location set of EFA E^i.

An EFA system is a collection of EFAs $\mathcal{E} = \{E^1, \ldots, E^n\}$. The EFA obtained from the synchronous composition of an EFA system is denoted by $\parallel \mathcal{E}$, with $\parallel \mathcal{E} = E^1 \parallel \ldots \parallel E^n$. In order to successfully apply this synchronous composition, the EFAs in the EFA system need to agree on initial valuation of shared variables and have consistent updates. From now on we assume that this is the case, as otherwise the behavior of an EFA system is undefined. The EFA system's alphabet is $\Sigma_{\mathcal{E}} = \Sigma^1 \cup \ldots \cup \Sigma^n$. Let us denote by $\rightarrow_{\mathcal{E}}$ the union over all transitions of the EFAs in the system $\mathcal{E}$, that is $\rightarrow_{\mathcal{E}} = \bigcup_{E^i \in \mathcal{E}} \rightarrow^i$. Finally, we use the notation $A \parallel \mathcal{E}$ to express the synchronous composition of EFA A and EFA system $\mathcal{E}$, i.e., $A \parallel \mathcal{E} = A \parallel (\parallel \mathcal{E})$.

A renaming, as introduced in Mohajerani et al. (2016), is a surjective function from one alphabet to another, and can be used to change the event labels on transitions. Renaming might also be called masking maps in distinguishers, see for example Teixeira et al. (2011), and relabeling in observers, see for example Wong and Wonham (2004).

Definition 4 (Renaming) Let Σ_1 and Σ_2 be two sets of events. A renaming $\rho : \Sigma_1 \to \Sigma_2$ is a surjective function that preserves the controllability status of renamed events, i.e., for $\sigma \in \Sigma_1$, $\rho(\sigma)$ is controllable if and only if σ is controllable.

Renaming and inverse renaming applied on EFAs are defined as follows. Applying renaming ρ on an EFA results in the EFA where each transition labeled with event σ is replaced by a transition labeled with the renamed event $\rho(\sigma)$ leaving guards and updates unchanged. Similarly, applying inverse renaming ρ^{-1} on an EFA results in the EFA where each transition labeled with event μ is replaced by a set of transitions having for each event σ such that $\rho(\sigma) = \mu$ one transition labeled with σ, again leaving guards and updates unchanged. Renaming and inverse renaming applied on FAs are defined in the same way. Figure 3 illustrates renaming and inverse renaming. Below we only show the formal definitions for EFAs.

Definition 5 (Renamed EFA) Let $G = (L, V, \Sigma, \to, l_0, v_0, L_m)$ be an EFA and let $\rho : \Sigma \to \Sigma'$. Then $\rho(G) = (L, V, \Sigma', \rho(\to), l_0, v_0, L_m)$ where $\rho(\to) = \{(x, \rho(\sigma), g, u, y) \mid (x, \sigma, g, u, y) \in \to\}$. For EFA system $\mathcal{G} = \{G^1, \ldots, G^n\}$, renaming is defined as $\rho(\mathcal{G}) = \{\rho(G^1), \ldots, \rho(G^n)\}$.

Definition 6 (Inverse renamed EFA) Let $G = (L, V, \Sigma, \to, l_0, v_0, L_m)$ be an EFA and let $\rho : \Sigma' \to \Sigma$. Then $\rho^{-1}(G) = (L, V, \Sigma', \rho^{-1}(\to), l_0, v_0, L_m)$ where $\rho^{-1}(\to) = \{(x, \sigma, g, u, y) \mid (x, \rho(\sigma), g, u, y) \in \to\}$. For EFA system $\mathcal{G} = \{G^1, \ldots, G^n\}$, inverse renaming is defined as $\rho^{-1}(\mathcal{G}) = \{\rho^{-1}(G^1), \ldots, \rho^{-1}(G^n)\}$.

Observe that the equality $\rho^{-1}(\rho(G)) = G$ does not hold in general. First, the goal of an abstraction is to get rid of unnecessary information in the model. Renaming does remove information, e.g., events e_1 and e_2 get renamed to event e. Now, inverse renaming does not know whether event e was originally e_1, e_2, or both. This problem is solved by putting a refined coordinator in parallel with the original system, see for example Theorem 5 in Section 6.3. Second, several abstractions, like local normalization in Section 5 and variable unfolding in Section 6.2, are similar to inverse renaming, i.e., they replace a single event by multiple events, for example, event f is replaced by f_1 and f_2. Now, renaming can be used to get the original event names back, as events f_1 and f_2 are both renamed to f. Renaming after inverse renaming uses the fact that the equality $\rho(\rho^{-1}(G)) = G$ does hold. Finally, note that renaming might introduce nondeterminism.

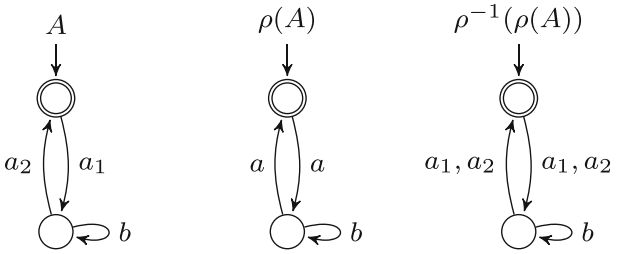


Fig. 3 Example to illustrate renaming and inverse renaming. The renaming ρ maps events a_1 and a_2 to a and event b to b. Multiple events on a single transition are used as a shorthand notation for multiple transitions having the same source and destination locations

2.3 Supervisory control theory

Conceptually, a supervisor is a control function that dynamically disables controllable events in the plant it controls, such that the closed-loop system of the plant and the supervisor obeys some specified behavior. Supervisory control theory of Ramadge and Wonham (1987) and Ramadge and Wonham (1989) gives means to automatically calculate a supervisor given FA models of the plant.

In the EFA context, we define in this paper a supervisor S as a subautomaton of the plant P, which is closely related with the definition used in Ouedraogo et al. (2011). Given two guards g and h, h is said to be a subguard of g, denoted by $h \preceq g$, if h is stronger than g, i.e., $h \wedge g = h$. Now, given two EFAs A and A', we say that A' is a subautomaton of A, denoted with $A' \preceq A$, if A' is obtained from A by replacing guards with subguards.

Definition 7 (Supervisor) Given a plant modeled by a deterministic EFA $P = (L, V, \Sigma, \rightarrow_P, l_0, \hat{v}_0, L_m)$, a supervision map $\mathcal{S}$ for P is a function $\mathcal{S} :\rightarrow_P \rightarrow \mathcal{G}_V$ which maps each transition $e = (l_1, \sigma, g, u, l_2)$ to a guard $\mathcal{S}(e)$ such that $\mathcal{S}(e) \preceq g$. The supervisor S is the subautomaton obtained from G by replacing its guards with those provided by $\mathcal{S}$, i.e., $S = (L, V, \Sigma, \rightarrow_S, l_0, \hat{v}_0, L_m)$ where $\rightarrow_S = \{(l_1, \sigma, \mathcal{S}(e), u, l_2) \mid e = (l_1, \sigma, g, u, l_2) \in \rightarrow_P\}$.

In this definition, the supervisor disables events by strengthening the guards on transitions labeled by these events. Strengthening a guard to **F** effectively removes that transition.

A supervisor S should conceptually adhere to the following control objectives for a given plant model P, as first proposed by Ramadge and Wonham (1987) and Ramadge and Wonham (1989):

- *Controllability*: uncontrollable events may never be disabled by the supervisor, i.e., S is controllable with respect to P.
- *Nonblockingness*: the controlled system should be able to reach a marked state from every reachable state, i.e., $S \parallel P$ is nonblocking.
- *Maximal permissiveness*: the supervisor does not restrict more behavior than strictly necessary to enforce controllability and nonblockingness, i.e., for all other supervisors S' that respect the above two control objectives it holds that $\mathcal{L}(S') \subseteq \mathcal{L}(S)$.

These notions are translated to mathematical definitions for EFAs in Ouedraogo et al. (2011).

Monolithic supervisory control synthesis results in a single supervisor S from a single plant model (Ouedraogo et al. 2011). When the plant model is given as a component system P_s, the monolithic plant model P is obtained by performing the synchronous composition of the component models. Furthermore, S can be obtained by calculating the supremal element of the set of controllable and nonblocking supervisors, i.e., $S = \sup CN(P)$. A nonblocking, controllable, and maximally permissive supervisor can be calculated, for example, by the fixed-point algorithm SSEFA as presented in Ouedraogo et al. (2011). While the desired properties of a supervisor are defined based on the state space of an EFA, the algorithm SSEFA performs the calculations directly on the EFA and not its state space.

Multiple (E)FAs in a component system P_s may be conflicting with each other, i.e., their synchronous composition may be blocking. A supervisor with the aim of resolving a conflict in an (E)FA system is called a coordinator, as introduced in Wong and Wonham (1998).

3 Coordinator equivalence framework

In this paper, we propose a procedure to synthesize, in a compositional manner as schematically illustrated in Fig. 1, a coordinator in case of conflicting supervisors. The main idea is to apply multiple small and simple coordinator equivalence preserving abstractions on a collection of automata. Then, using the abstracted system, nonconflicting verification is performed and in case of a conflict a coordinator is synthesized. Finally, this coordinator is refined in order to be a coordinator for the original system ensuring that the final closed-loop system is nonblocking, controllable, and maximally permissive. In this section, we introduce a new data structure called a coordinator tuple, which keeps track of the abstracted system and the needed refinements, and the notions of coordinator equivalence and coordinator equivalence preserving abstraction-refinement pairs.

3.1 Coordinator tuples

In this paper, we generalize abstractions and refinements, seeing them as functions that generate for a given EFA system a new one. Related abstractions and refinements are called *abstraction-refinement pairs.* Some abstraction functions are straightforward, like for example, taking the synchronous composition of two EFAs, while others are more complex, like for example, the merging of events. Section 4.1 shows four possible abstractions as an illustrative example. As we focus in this paper on refining coordinators, the refinement function is formally defined below.

Definition 8 (Refinement function) A *refinement* function is a function $\xi : 2^{\mathcal{P}} \to 2^{\mathcal{P}}$, with $\mathcal{P}$ the universe of EFAs, that transforms a given EFA system into another EFA system. The composition of two refinement functions ξ_1 and ξ_2, denoted with $\xi_1 \circ \xi_2$, is defined by $\xi_1 \circ \xi_2(\mathcal{E}) = \xi_1(\xi_2(\mathcal{E}))$ for any given EFA system $\mathcal{E}$.

In this paper we limit ourselves to a specific class Ξ of refinement functions, as coordinator equivalence, see next section, can only be proven for specific refinement functions. The set Ξ is defined recursively.

- $\xi \in \Xi$ if ξ is the identity function, i.e., for any abstracted EFA system $\mathcal{F}$: $\xi(\mathcal{F}) = \text{id}(\mathcal{F}) = \mathcal{F}$.
- $\xi \in \Xi$ if ξ is a renaming function, i.e., for any abstracted EFA system $\mathcal{E}$ and abstracted EFA system $\mathcal{F}$: $\exists \rho$ s.t. $\xi(\mathcal{F}) = \rho(\mathcal{F})$ where $\rho : \Sigma_{\mathcal{F}} \to \Sigma_{\mathcal{E}}$.
- $\xi \in \Xi$ if ξ is a renaming function in synchronous composition with the EFA system, i.e., for any EFA system $\mathcal{E}$ and any abstracted EFA system $\mathcal{F}$: $\exists \rho$ s.t. $\xi(\mathcal{F}) = \rho(\mathcal{F}) \parallel \mathcal{E}$ where $\rho : \Sigma_{\mathcal{F}} \to \Sigma_{\mathcal{E}}$.
- $\xi \in \Xi$ if ξ is an inverse renaming in synchronous composition with the EFA system, i.e., for any EFA system $\mathcal{E}$ and any abstracted EFA system $\mathcal{F}$: $\exists \rho$ s.t. $\xi(\mathcal{F}) = \rho^{-1}(\mathcal{F}) \parallel \mathcal{E}$ where $\rho : \Sigma_{\mathcal{E}} \to \Sigma_{\mathcal{F}}$.
- $\xi_1 \circ \xi_2 \in \Xi$ if $\xi_1 \in \Xi$ and $\xi_2 \in \Xi$.

Section 4.3 illustrates four possible refinements, all from the set Ξ. We can now define a coordinator tuple.

Definition 9 (Coordinator tuple) A *coordinator tuple* is a tuple $(\mathcal{E}, \xi)$, where $\mathcal{E}$ is a deterministic EFA system and $\xi \in \Xi$.

Given a coordinator tuple $(\mathcal{E}, \xi)$, the EFA system $\mathcal{E}$ represents the abstracted models and ξ the refinement function used to refine $\mathcal{E}$ or a coordinator synthesized from $\mathcal{E}$. A coordinator tuple represents an intermediate control problem in the compositional coordinator framework and contains all information to either apply a new abstraction or to synthesize a coordinator and refine it back.

Example Consider the example illustrated in Fig. 4. The initial coordinator tuple is given as $(\mathcal{E}, \xi_0)$, with $\mathcal{E} = \{A, B\}$ and $\xi_0 = \text{id}$. From this EFA system, the abstracted EFA system $\widetilde{\mathcal{E}} = \{\widetilde{A}, \widetilde{B}\}$ is obtained. The corresponding refinement function ξ_1 is defined as $\xi_1(\mathcal{F}) = \rho(\mathcal{F})$ for any EFA system $\mathcal{F}$. Renaming ρ maps events a_1 and a_2 onto event a, events b_1 and b_2 onto event b, and event c onto event c. Therefore, $\xi_1 \in \Xi$. The coordinator tuple after performing the first abstraction is then $(\widetilde{\mathcal{E}}, \xi_0 \circ \xi_1)$.

3.2 Coordinator equivalence

In this section, we define coordinator equivalence, which is inspired by synthesis equivalence of triples as defined in Mohajerani et al. (2014). Two coordinator tuples are said to be *coordinator equivalent* if the coordinator synthesized for the abstracted models results in the same closed-loop behavior after applying the refinement functions. This ensures a nonblocking, controllable, and maximally permissive closed-loop system. The notion of coordinator equivalence is captured formally in the following definition.

Definition 10 (Coordinator equivalence) Let $(\mathcal{E}_1, \xi_1)$ and $(\mathcal{E}_2, \xi_2)$ be two coordinator tuples. Then these coordinator tuples are said to be coordinator equivalent, denoted with $(\mathcal{E}_1, \xi_1) \simeq_{\text{co}} (\mathcal{E}_2, \xi_2)$, if

$$\mathcal{L}(\xi_1(\text{sup}CN(\mathcal{E}_1))) = \mathcal{L}(\xi_2(\text{sup}CN(\mathcal{E}_2)))$$

It follows directly from the definition that $\simeq_{\text{co}}$ is indeed an equivalence relation, as it is reflexive, symmetric, and transitive. Therefore, it follows that we can discuss each abstraction-refinement step separately and connect abstraction-refinement pairs and coordinator tuples in a compositional manner.

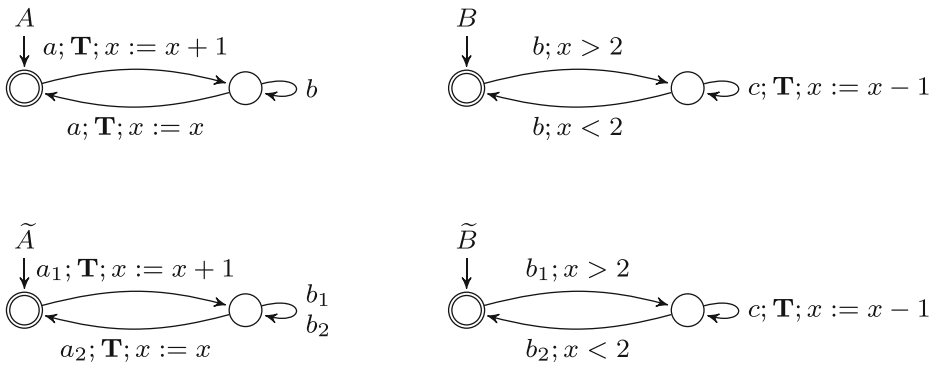


Fig. 4 Example with EFA system $\mathcal{E} = \{A, B\}$ and abstracted EFA system $\widetilde{\mathcal{E}} = \{\widetilde{A}, \widetilde{B}\}$. In this and subsequent figures, we will use the convention that transitions are labeled with 'event;guard;update'. To have concise drawings of the automata, guards equal to 'true' and unspecified updates are sometimes omitted

Figure 5 illustrates the usage of coordinator tuples and coordinator equivalence in the proposed framework of compositional coordinator synthesis. After applying some abstractions on the initial coordinator tuple $(\mathcal{E}_0, \mathrm{id})$, the coordinator tuple $(\mathcal{E}_1, \xi_1)$ is obtained. This coordinator tuple is subsequently abstracted to $(\mathcal{E}_2, \xi_2)$. From the first coordinator tuple $(\mathcal{E}_1, \xi_1)$, a coordinator C_1 can be synthesized, which will become $\xi_1(C_1)$ after refinement. Similarly, a refined coordinator $\xi_2(C_2)$ is obtained from the second coordinator tuple $(\mathcal{E}_2, \xi_2)$. Now the two coordinator tuples are coordinator equivalent if the language of $\xi_1(C_1)$ equals the language of $\xi_2(C_2)$, i.e., they have the same closed-loop behavior. If all used abstractions to obtain the first coordinator tuple $(\mathcal{E}_1, \xi_1)$ are coordinator equivalence preserving abstractions, we know by the transitive property of coordinator equivalence that it is coordinator equivalent to the original coordinator tuple $(\mathcal{E}_0, \mathrm{id})$. Therefore, both $\xi_1(C_1)$ and $\xi_2(C_2)$ result in the same closed-loop behavior as a coordinator C_0 synthesized from the original EFA system directly: all being controllable, nonblocking, and maximally permissive.

We briefly discuss the similarities and the differences between coordinator tuples and synthesis triples. For compositional supervisor synthesis of FAs, the synthesis triple $(\mathcal{E}, \mathcal{S}, \rho)$ is defined in Mohajerani et al. (2014), where $\mathcal{E}$ is a set of abstracted plant models, $\mathcal{S}$ a set of supervisors collected so far, and ρ a renaming function that maps events in $\mathcal{E}$ and $\mathcal{S}$ back to events in the original plant model. Therefore, a synthesis triple represents an intermediate step in the compositional synthesis approach. The equivalence of both coordinator tuples and synthesis triples is based on the closed-loop behavior of the supervised systems. The first difference is that both sets $\mathcal{E}$ and $\mathcal{S}$ contain FAs, while coordinator tuples include EFAs. Second, in our approach, see Fig. 1, no supervisor is collected while performing the abstractions, resulting in the absence of the set $\mathcal{S}$ in a coordinator tuple. Finally, for compositional supervisor synthesis of FAs, it is sufficient to have only renaming as a refinement function, while for compositional coordinator synthesis of EFAs, more complicated refinements are necessary. Therefore, a renaming function is included in the synthesis triple and a refinement function is included in the coordinator tuple.

3.3 Coordinator equivalence preserving abstraction-refinement pairs

Compositional approaches apply several abstractions in sequence. From a collection of abstractions, at each step one abstraction is chosen and applied on a specific coordinator tuple. For the purpose of compositional coordinator synthesis, all applied abstractions should create from any coordinator tuple another one that is coordinator equivalent. Once

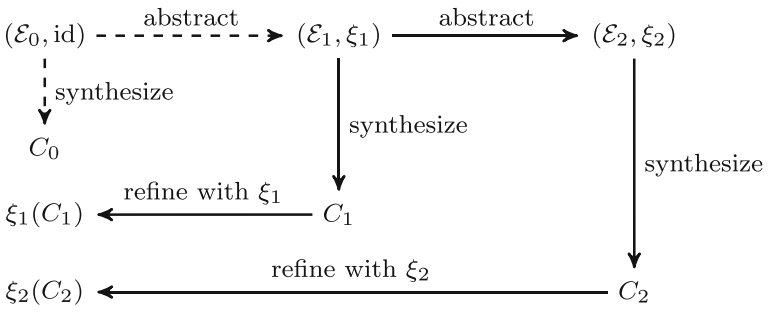


Fig. 5 The usage of coordinator tuples and coordinator equivalence in the framework of compositional coordinator synthesis

an abstraction is proven to always create a coordinator tuple that is coordinator equivalent to the input coordinator tuple, it is called a coordinator equivalence preserving abstraction and it can be added to the collection of abstraction-refinement pairs. In this sense, the framework allows to add new abstraction-refinement pairs easily.

Ten different abstractions, based on the abstractions proposed in Mohajerani et al. (2016), are identified that can be used for compositional coordinator synthesis. These abstractions are enhanced with coordinator refinements to form abstraction-refinement pairs. Table 1 provides an overview of these abstraction-refinement pairs. In this paper, several abstraction-refinement pairs are explained in detail to illustrate the different kinds of refinements. Descriptions of the other abstraction-refinement pairs are included in Online Resource 1.

Several abstractions may introduce nondeterminism: these are the FA-based abstraction (see Online Resource 1, Section 3) and event merging (see Section 6.3). As it is not straightforward how to synthesize supervisors for nondeterministic systems, nondeterminism is circumvented. When an FA-based abstraction results in nondeterminism, we can use the approach of Mohajerani et al. (2011) and Mohajerani et al. (2014) where first renaming is applied and then the FA-based abstraction. This renaming introduces new events to disambiguate between nondeterministic branching behavior. Event merging may also result in nondeterminism. However, then renaming will not help, so the event merging abstraction cannot be used in that case.

4 Illustrative example

The procedure proposed in this paper is in this section illustrated with an example. We use the manufacturing system example as presented in Mohajerani et al. (2016). Since compositional coordinator synthesis is applied on this example, a controllability status (controllable or uncontrollable) is added to each event. For this example we first perform coordinator equivalence preserving abstractions until a single automaton is obtained. While only conflict equivalence preserving abstractions are applied in Mohajerani et al. (2016), applying coordinator equivalence preserving abstractions results in the same final abstracted model. On this automaton we then perform a nonblocking verification. As this example contains a conflict, we synthesize a coordinator with the simplified single automaton. Finally, we refine this coordinator back to the original system.

Figure 6 shows the manufacturing system consisting of a conveyer belt split into two sections CB_1, CB_2 and two machines M_1, M_2. Products are loaded from the environment onto the conveyor belt (event l_1) that has a capacity of 2. Event l_2 represents a product being

Table 1 An overview of all identified abstraction-refinement pairs suitable for inclusion in the proposed compositional coordinator synthesis framework, including a reference to the sections that describe these abstraction-refinement pairs

Abstraction-refinement	Location	Abstraction-refinement	Location
Local normalization	Section 5	Variable unfolding	Section 6.2
Global normalization	Section 5	False removal	Online Resource 1
FA-based abstractions	Online Resource 1	Selfloop removal	Online Resource 1
Partial composition	Section 6.1	Event merging	Section 6.3
Update simplification	Online Resource 1	Update merging	Online Resource 1

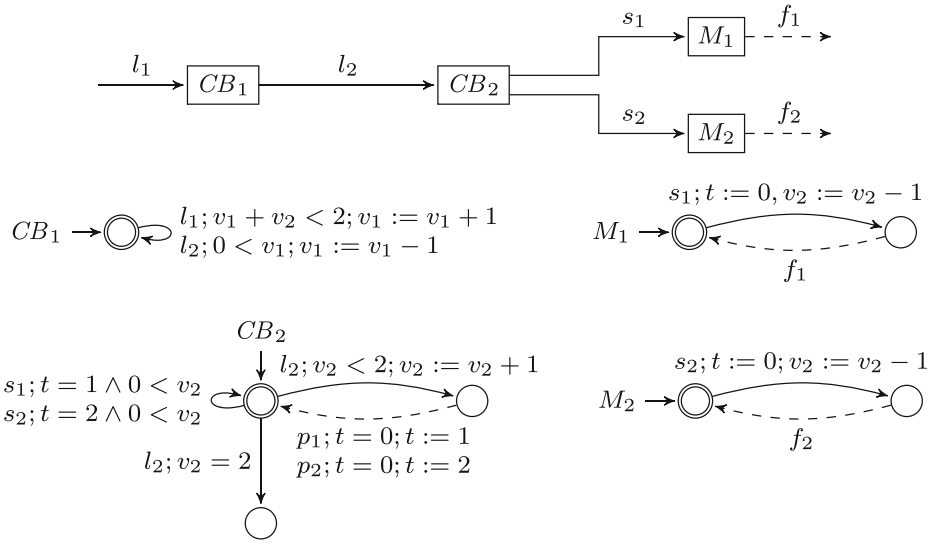


Fig. 6 The manufacturing system example presented in Mohajerani et al. (2016) extended with controllable and uncontrollable events. All variables are initially 0

transported from the first section to the second section. At the beginning of CB_2 a product detection sensor determines the type of products (events p_1 and p_2). Products of type 1 are processed by machine M_1 (event s_1), while products of type 2 are processed by machine M_2 (event s_2). Both machines M_1 and M_2 process products and output them from the manufacturing system (events f_1 and f_2, respectively). In this manufacturing system, events l_1, l_2, s_1 and s_2 are considered to be controllable, while p_1, p_2, f_1 and f_2 are uncontrollable.

Each part of the manufacturing system is modeled by an EFA. Variables v_1 and v_2 represent the number of products on conveyer belt sections CB_1 and CB_2, respectively. Both variables have domain $\{0, 1, 2\}$ and the initial value is 0. Variable t keeps track of the type of product that is last observed by the product detection sensor. The domain of t is $\{0, 1, 2\}$ and the initial value is 0 (representing the fact that the type of a product is not measured yet). The EFA system is given by $\mathcal{E}_0 = \{CB_1, CB_2, M_1, M_2\}$.

The model in Fig. 6 has an incorrect implementation of the type recognition procedure. The sequence $l_1 l_2 p_1 l_1 l_2$ does not reset the variable t back to zero, while now the only possible transitions p_1 or p_2 in CB_2 require $t = 0$. This renders the system to be blocking. Compositional nonblocking verification will show that this system is blocking without exploring its complete state space. After creating an abstracted representation, a nonblocking, controllable, and maximally permissive coordinator is synthesized. As only coordinator equivalence preserving abstractions are deployed, the coordinator can be correctly refined to control the original EFA system.

4.1 Model abstractions

First, a given model is abstracted by applying a sequence of abstractions. The choice of when to apply which abstraction is made heuristically by trying to either create or utilize local events. This is reflected in the algorithm from Mohajerani et al. (2016) as shown in Section 7. The correctness of the approach does not depend on the order of abstractions.

The order only affects the observed computational reduction, because the final EFA system might be different in form. Below, we only describe the first abstraction steps to illustrate the concept of abstraction (and refinement later on); an example of a complete sequence of abstraction steps is described in Mohajerani et al. (2016).

Abstraction step 1 The first steps of compositional nonblocking verification are to normalize the EFA system. In a normalized system, each transition labeled with the same event also has the same guard and update. This facilitates the reasoning about EFAs, as a normalized system shows directly what the effect is on the variables when executing events. Normalization is performed in two parts: first each EFA is normalized locally (where only a single EFA is considered without looking at the other EFAs in the EFA system), and then the EFA system is normalized globally. Each EFA in Fig. 6 is already locally normalized except CB_2, as in this EFA event l_2 is associated with two different pairs of guards and updates. In order to normalize CB_2, event l_2 is renamed to l_{21} and l_{22}, where l_{21} is associated with guard $v_2 < 2$ and update $v_2 := v_2 + 1$, while l_{22} is associated with guard $v_2 = 2$. In all other EFAs, we now need to replace every occurrence of l_2 by l_{21} and l_{22} to maintain synchronization between the EFAs. In CB_1, event l_2 is replaced by l_{21} and l_{22} both having guard $0 < v_2$ and update $v_1 := v_1 - 1$. Thus, local normalization of CB_2 results in two new EFAs C_1 and C_2, shown in Fig. 7, that replace CB_1 and CB_2, respectively, and the renaming ρ that maps events l_{21} and l_{22} to l_2 and keeps all other events the same. The EFA system is now given by $\mathcal{E}_1 = \{C_1, C_2, M_1, M_2\}$.

In case multiple EFAs need to be locally normalized, the procedure illustrated above needs to be applied multiple times. Each time, local normalization of an EFA continues with the EFA system resulting from the previous local normalization. For example, if M_1 also needs to be locally normalized, we would continue with the EFA system $\mathcal{E}_1$ and not the original EFA system $\mathcal{E}_0$.

Abstraction step 2 When all EFAs are locally normalized, the EFA system is globally normalized by taking the conjunction of guards and updates. For example, event l_{21} is associated in EFA C_1 with guard $0 < v_1$ and update $v_1 := v_1 - 1$, while the same event is associated in C_2 with guard $v_2 < 2$ and update $v_2 := v_2 + 1$. The globally normalized guard of l_{21} then becomes $0 < v_1 \wedge v_2 < 2$ and update $v_1 := v_1 - 1, v_2 := v_2 + 1$. Figure 8 shows the globally normalized EFA system. As each transition with the same event has the same guard and update, we can display the guards and updates in a separate table. In this way, the automata can be presented without the guards and updates on the transitions. The EFA system is now transformed into the globally normalized EFA system $\mathcal{E}_2 = \{\mathcal{N}(C_1), \mathcal{N}(C_2), \mathcal{N}(M_1), \mathcal{N}(M_2)\}$.

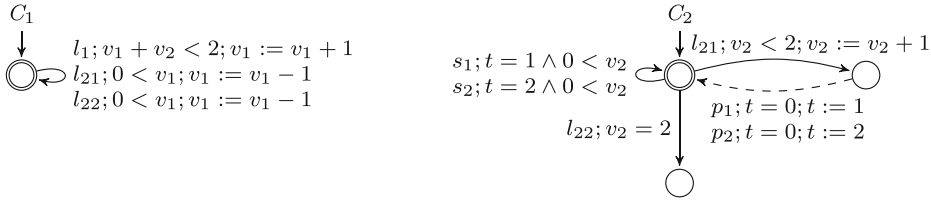
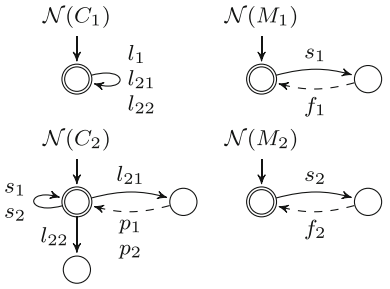


Fig. 7 Locally normalized EFAs C_1 and C_2 obtained from CB_1 and CB_2, respectively



Event	Guard and update
l_1	$v_1 + v_2 < 2; v_1 := v_1 + 1$
l_{21}	$0 < v_1 \wedge v_2 < 2; v_1 := v_1 - 1, v_2 := v_2 + 1$
l_{22}	$0 < v_1 \wedge v_2 = 2; v_1 := v_1 - 1$
p_1	$t = 0; t := 1$
p_2	$t = 0; t := 2$
s_1	$t = 1 \wedge 0 < v_2; t := 0, v_2 := v_2 - 1$
s_2	$t = 2 \wedge 0 < v_2; t := 0, v_2 := v_2 - 1$
f_1	**T**
f_2	**T**

Fig. 8 The globally normalized EFA system. For each event the guard and update is displayed in the table

Abstraction step 3 Event f_1 belongs only to the alphabet of $\mathcal{N}(M_1)$, it has a **T** guard, and does not change the valuation of any variable. Therefore, this event is in the EFA setting a local event. Then, the FA-based abstraction of *weak synthesis observation equivalence* (see Mohajerani et al. (2014)) can be applied to merge the two locations of $\mathcal{N}(M_1)$. This results in the abstracted EFA $\widetilde{M}_1$ as shown in Fig. 9. The EFA system is now given by $\mathcal{E}_3 = \{\mathcal{N}(C_1), \mathcal{N}(C_2), \widetilde{M}_1, \mathcal{N}(M_2)\}$.

Abstraction step 4 Event f_1 appears only on a transition in $\widetilde{M}_1$ and this transition is a selfloop. Furthermore, the guard is true and the event does not change the valuation of variables. Therefore, event f_1 can be safely removed from the EFA system, resulting in $\widetilde{M}_{1\setminus f_1}$ as shown in Fig. 9. The EFA system is now given by $\mathcal{E}_4 = \{\mathcal{N}(C_1), \mathcal{N}(C_2), \widetilde{M}_{1\setminus f_1}, \mathcal{N}(M_2)\}$.

After applying a sequence of coordinator equivalence preserving abstractions, the final abstracted EFA E as shown in Fig. 10 is obtained. Though Mohajerani et al. (2016) did not use only coordinator equivalence preserving abstractions, the same result is obtained. This abstracted EFA shows that the system blocks when a second product enters CB_2 (event l_{21}) before the previous product was sent to one of the machines (event s, a combination of events s_1 and s_2).

4.2 Coordinator synthesis

On the final abstracted result E, we can apply any monolithic synthesis procedure to resolve the blocking issue. Figure 10 shows, on the right-hand side, a nonblocking, controllable, and maximally permissive coordinator S for E. As this coordinator is synthesized for the abstracted system, it cannot be directly deployed on the original system that compositional nonblocking verification started with. Since only coordinator equivalence preserving abstractions were used, the coordinator synthesized for the abstracted system can be refined

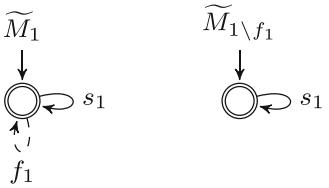


Fig. 9 Simplified EFAs $\widetilde{M}_1$ and $\widetilde{M}_{1\setminus f_1}$ of $\mathcal{N}(M_1)$

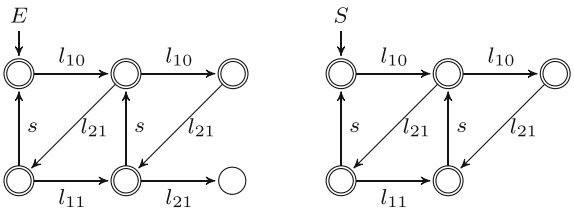


Fig. 10 The final abstracted EFA E and supervisor S obtained from it

back to obtain a nonblocking, controllable, and maximally permissive coordinator for the original EFA system. This refinement is demonstrated in the next section.

4.3 Coordinator refinement

In this section, the coordinator S is refined, which corresponds to C_2 in Fig. 5. Since the system is abstracted step by step, we have obtained a composed refinement function $\xi = \xi_0 \circ \xi_1 \circ \xi_2 \circ \ldots$. Therefore, the refinement function can be applied on the coordinator S by starting with reversing the last abstraction steps (not shown in Section 4.1), all the way back to abstraction step 1. These reversals are called *refinement steps.*

To track the different coordinators in this example properly, a subscript is added to indicate for which system it is a coordinator. After applying the refinement steps related to the not shown abstraction steps, the coordinator $S_4 = S' \parallel \mathcal{E}_4$ is obtained, where S' is shown in Fig. 11. As in this example we are manually refining the coordinator, we do not calculate each coordinator explicitly in the refinement steps. Instead, we indicate how the coordinator can be calculated using known models. This way of working closely resembles supervisor reduction of Su and Wonham (2004).

Refinement step 4 In abstraction step 4, event f_1 was removed, as it only appeared on selfloops with true guards and updates. To ensure that this event is possible in the coordinator, these selfloops need to be placed back. This can be achieved by calculating the synchronous composition of the coordinator with the previous abstracted EFA system, i.e., $S_3 = \xi_4(S_4) = S_4 \parallel \mathcal{E}_3$. This can also be rewritten as $S_4 \parallel \mathcal{E}_3 = S' \parallel \mathcal{E}_4 \parallel \mathcal{E}_3 = S' \parallel \mathcal{E}_3$, where we used the observation for this case that $\mathcal{E}_4 \parallel \mathcal{E}_3 = \mathcal{E}_3$.

Refinement step 3 In abstraction step 3, an FA-based abstraction was applied. In general, a quotient automaton may contain more behavior than the original automaton. In this example, in $\widetilde{M}_1$ it is possible to execute the transitions labeled with s_1 and f_1 in arbitrary order, while in the original model $\mathcal{N}(M_1)$ the events s_1 and f_1 alternate. To refine the coordinator, the synchronous composition is calculated of the coordinator with the previous abstracted EFA system, i.e., $S_2 = \xi_3(S_3) = S_3 \parallel \mathcal{E}_2$. This can also be rewritten as

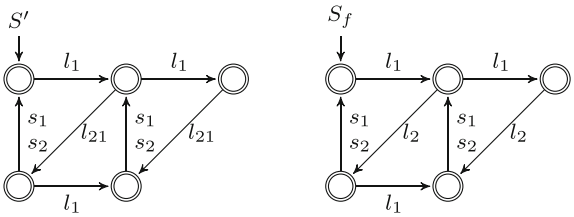


Fig. 11 Intermediate and final refined coordinators

$S_3 \parallel \mathcal{E}_2 = S' \parallel \mathcal{E}_3 \parallel \mathcal{E}_2 = S' \parallel \mathcal{E}_2$, where we used the observation for this case that $\mathcal{E}_3 \parallel \mathcal{E}_2 = \mathcal{E}_2$.

Refinement step 2 In abstraction step 2, global normalization was applied. To refine global normalization, nothing has to be changed, i.e., $S_1 = \xi_2(S_2) = S_2$.

Refinement step 1 In abstraction step 1, an EFA was locally normalized. To refine the coordinator, the newly introduced events l_{21} and l_{22} need to be renamed back to l_2, i.e., $S_0 = \xi_1(S_1) = \rho(S_1)$. This can also be rewritten as $\rho(S_1) = \rho(S_2) = \rho(S' \parallel \mathcal{E}_2) = \rho(S') \parallel \rho(\mathcal{E}_2) = S_f \parallel \mathcal{E}_0$, where S_f is shown in Fig. 11.

We can now verify that the closed-loop behavior of the monolithic coordinator calculated with the original EFA system is the same as the closed-loop behavior of the refined compositional coordinator $S_0 \parallel \mathcal{E}_0 = S_f \parallel \mathcal{E}_0$.

Following the algorithm of Mohajerani et al. (2016), the system was abstracted until a single EFA without variables was obtained. Nevertheless, the proposed method allows the user to stop with abstracting at any given moment, to synthesize a nonblocking, controllable, and maximally permissive coordinator, and to refine this coordinator back to the original EFA system. As shown in Gommans (2016), stopping the refinements before a single EFA without variables is obtained may be beneficial.

5 Normalization

The first step of compositional coordinator synthesis is normalization, see Mohajerani et al. (2016). In a normalized EFA system $\mathcal{E}$, each transition labeled with the same event σ has the same guard and update. To normalize an EFA system $\mathcal{E} = \{E^1, \ldots, E^n\}$, first all individual EFAs of the system are locally normalized, i.e, each EFA is considered separately without looking at the other EFAs in the EFA system, by renaming events with renaming functions ρ_i for $i \in \{1, \ldots, n\}$. When all individual EFAs have been locally normalized, then the EFA system is globally normalized by merging updates to ensure that the complete EFA system is a normalized system. Therefore, the guard and update are related to the event, which we denote by g_σ and u_σ, respectively. These two procedures are treated separately.

Example Figure 12 illustrates the two steps of local and global normalization. Consider the EFA system consisting of E^1 and E^2. EFA E^1 is locally normalized, as for both events a and b it holds that all transitions labeled with the same event have the same guard and update. EFA E^2 is not locally normalized, as not all transitions labeled with event b have the same guard and update. Therefore, events b_1 and b_2 are introduced to replace event b in E^2, resulting in EFA F^2, which is locally normalized. Renaming ρ maps events b_1 and b_2 back to b. Now, EFAs E^1 and F^2 do not synchronize any longer. To restore synchronization, event b in E^1 is replaced by events b_1 and b_2 with the inverse renaming ρ^{-1}. The resulting EFA $\rho^{-1}(E^1)$ is shown in Fig. 12.

Now, each EFA is locally normalized, yet the EFA system of $\rho^{-1}(E^1)$ and F^2 is not globally normalized: transitions labeled by b_1 and transitions labeled by b_2 have different guards and updates. To normalize an EFA system globally, the guards and updates for each event are merged. For example, for event b_1 the guard in EFA $\rho^{-1}(E^1)$ is $x > 0$ and the update is $x := 0$; for the same event the guard in EFA F^2 is $\mathbf{T}$ (not explicitly depicted in the figure) and the update is $y := x$. Merging these will result in a guard $x > 0 \wedge \mathbf{T} = x > 0$

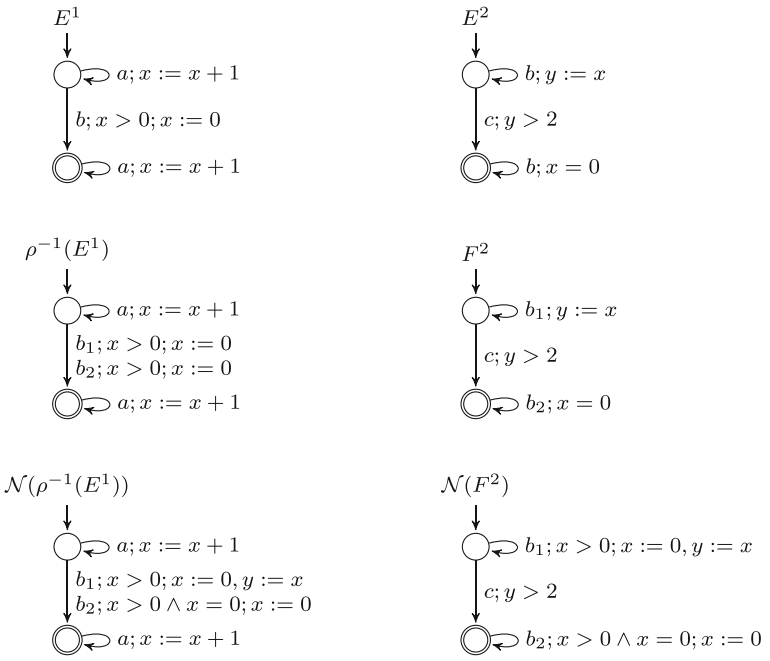


Fig. 12 Example illustrating the process of local and global normalization

and update $x := 0, y := x$. Figure 12 shows the globally normalized EFAs $\mathcal{N}(\rho^{-1}(E^1))$ and $\mathcal{N}(F^2)$.

A normalized system is formally defined as follows, adapted from Mohajerani et al. (2016).

Definition 11 (Normalized system (Mohajerani et al. 2016)) An EFA system $\mathcal{E}$ is normalized if for all pairs of transitions $(l_1, \sigma, g_1, u_1, l_2) \in \rightarrow_{\mathcal{E}}$ and $(l_3, \sigma, g_2, u_2, l_4) \in \rightarrow_{\mathcal{E}}$ it holds that $g_1 = g_2$ and $u_1 = u_2$, i.e., the guards and the updates are the same. An EFA E is normalized if the EFA system $\{E\}$ is normalized.

Definition 12 (Normalized form (Mohajerani et al. 2016)) Let $\mathcal{E} = \{E^1, \ldots, E^n\}$ be an EFA system where each $E^i = (L^i, V^i, \Sigma^i, \rightarrow^i, l_0^i, \hat{v}_0^i, L_m^i)$ is (locally) normalized. The normalized form of $\mathcal{E}$ is denoted by $\mathcal{N}(\mathcal{E}) = \{\mathcal{N}(E^1), \ldots, \mathcal{N}(E^n)\}$ where $\mathcal{N}(E^i) = (L^i, V, \Sigma^i, \rightarrow^i_{\mathcal{N}}, l_0^i, \hat{v}_0, L_m^i)$, $V = \cup_i V^i$, $\rightarrow^i_{\mathcal{N}} = \{(l_1^i, \sigma, g_\sigma, u_\sigma, l_2^i) \mid (l_1^i, \sigma, g_\sigma^i, u_\sigma^i, l_2^i) \in \rightarrow^i\}$, $g_\sigma = \wedge_{i:\sigma\in\Sigma^i} g_\sigma^i$, $u_\sigma = \oplus_{i:\sigma\in\Sigma^i} u_\sigma^i$, and $\hat{v}_0 = \oplus_i \hat{v}_0^i$.

For local and global normalization, we can express the following two theorems, respectively. The proofs of these theorems can be found in Sections 1 and 2 of Online Resource 1. Each theorem in this paper has the same structure, which is depicted in Fig. 2. Given two particular EFA systems $\mathcal{E}$ and $\mathcal{F}$, an already obtained refinement function ξ_1, we show that there exists an refinement function ξ such that $(\mathcal{E}, \xi_1) \simeq_{\text{co}} (\mathcal{F}, \xi_1 \circ \xi)$. The contribution of this paper is reflected in the addition of the refinement to the abstraction and showing that the abstraction-refinement pair is coordinator equivalence preserving. In all theorems, the

order of the individual EFAs in EFA system $\mathcal{E} = \{E^1, \ldots, E^n\}$ is arbitrary. For simplicity, the EFAs that change are placed at the beginning of the set.

Theorem 1 (Local normalization) *Let* $(\mathcal{E}, \xi_1)$ *be a coordinator tuple with* $\mathcal{E} = \{E^1, \ldots, E^n\}$ *a deterministic EFA system, and let* $\rho : \Sigma' \to \Sigma_{\mathcal{E}}$ *be a renaming function such that* $\mathcal{F} = \{F^1, \rho^{-1}(E^2), \ldots, \rho^{-1}(E^n)\}$, $\rho(F^1) = E^1$, *and* F^1 *is a normalized EFA. Then refinement function* $\xi(\mathcal{G}) = \rho(\mathcal{G})$ *for any EFA* $\mathcal{G}$ *ensures that* $(\mathcal{E}, \xi_1) \simeq_{co} (\mathcal{F}, \xi_1 \circ \xi)$.

The intuition behind this theorem is as follows. To refine the coordinator for local normalization, events need to be renamed back to their original names to have the same closed-loop behavior as the coordinator synthesized before local normalization. In Fig. 13, coordinator C is synthesized from the locally normalized EFA system $\{\rho^{-1}(E^1), F^2\}$ of Fig. 12. The refined coordinator C' to control the original EFA system is obtained by applying the renaming ρ, i.e., $C' = \rho(C)$, which is also shown in Fig. 13.

Theorem 2 (Global normalization) *Let* $(\mathcal{E}, \xi_1)$ *be a coordinator tuple with* $\mathcal{E} = \{E^1, \ldots, E^n\}$ *a deterministic EFA system, where each individual EFA* $E^i \in \mathcal{E}$ *is locally normalized. Construct the normalized form of* $\mathcal{E}$ *as* $\mathcal{F} = \mathcal{N}(\mathcal{E}) = \{\mathcal{N}(E^1), \ldots, \mathcal{N}(E^n)\}$. *Then refinement function* $\xi = \mathrm{id}$ *ensures that* $(\mathcal{E}, \xi_1) \simeq_{co} (\mathcal{F}, \xi_1 \circ \xi)$.

Intuitively, the coordinator does not have to be changed to refine global normalization. Global normalization of an EFA system with locally normalized EFAs does not change the behavior of the system as shown in Proposition 3 of Mohajerani et al. (2016). Therefore, a coordinator synthesized for the EFA system just before global normalization would be the same as the coordinator synthesized for the globally normalized EFA system. For example, the coordinator C in Fig. 13 synthesized for the globally normalized EFA system $\{\mathcal{N}(\rho^{-1}(E^1)), \mathcal{N}(F^2)\}$, as shown in Fig. 12, is also a coordinator for the locally normalized EFA system.

6 Coordinator equivalence preserving abstraction-refinement pairs

After normalizing the system, several abstraction-refinement pairs can be applied to simplify the system. The following sections describe several abstraction-refinement pairs suitable for both the nonblocking verification and coordinator refinement. All EFA-based

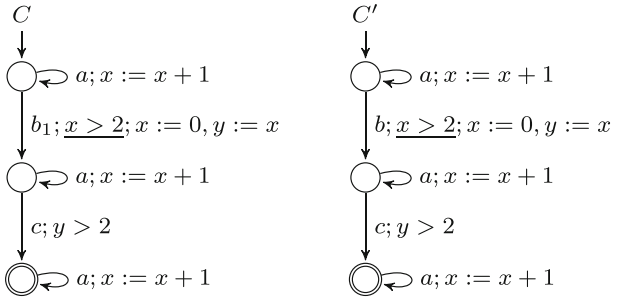


Fig. 13 Example illustrating the process of coordinator refinement for local and global normalization. C is a coordinator synthesized for the locally or globally normalized EFA system and C' is a refined coordinator for the original EFA system. Underlined are the strengthened guards with respect tot the plant

conflict equivalence preserving abstractions presented in Mohajerani et al. (2016) are also suitable for coordinator refinement, while some conflict equivalence preserving FA-based abstractions are no longer suitable for synthesis; for counterexamples, see the work of Mohajerani et al. (2014).

6.1 Partial composition

Partial composition is one of the simplest abstractions, see Mohajerani et al. (2016). Even though the synchronous composition of two EFAs may enlarge the state space, it often increases the applicability of one of the other abstraction-refinement pairs.

It follows from the definitions of synchronous composition and EFA systems (and it is confirmed by Proposition 6 of Mohajerani et al. (2016)) that partial composition does not alter the behavior of the system. Therefore, a monolithic supervisor synthesized for an EFA system before partial composition is the same as one synthesized after partial composition. For example, consider the EFA system consisting of E^1 and E^2 as depicted in Fig. 14. The EFA system consisting of their synchronous composition $E^1 \parallel E^2$ generates the same language. No refinement is needed, as expressed by Theorem 3. The coordinator C in Fig. 14 can be a coordinator for the EFA system consisting of E^1 and E^2 as well as for the EFA system consisting of $E^1 \parallel E^2$. The proof of this theorem can be found in Section 4 of Online Resource 1.

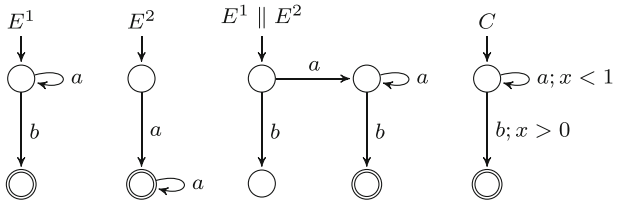
Theorem 3 (Partial composition) *Let $(\mathcal{E}, \xi_1)$ be a coordinator tuple with $\mathcal{E} = \{E^1, \ldots, E^n\}$ a deterministic normalized EFA system. Construct $\mathcal{F} = \{E^1 \parallel E^2, E^3, \ldots, E^n\}$. Then refinement function $\xi = \mathrm{id}$ ensures that $(\mathcal{E}, \xi_1) \simeq_{co} (\mathcal{F}, \xi_1 \circ \xi)$.*

6.2 Variable unfolding

Another useful abstraction is the elimination of a variable from the EFA system by unfolding it into explicit states. In Mohajerani et al. (2016), a particular method of unfolding a variable is presented that keeps a normalized system normalized after the abstraction.

Below, the definitions for variable unfolding as presented in Mohajerani et al. (2016) are briefly discussed and adapted for using guards and updates on the transitions. Details can be found in Mohajerani et al. (2016).

Definition 13 (Variable alphabet (Mohajerani et al. 2016)) Let z be a variable and Σ an alphabet. The variable alphabet of z with respect to Σ is $U_z(\Sigma) = \Sigma \times \mathrm{dom}(z) \times \mathrm{dom}(z)$, where $(\sigma, a, b) \in U_z(\Sigma)$ is controllable if and only if $\sigma \in \Sigma$ is controllable. The variable renaming function ρ_z is defined as $\rho_z((\sigma, a, b)) = \sigma$ for all $(\sigma, a, b) \in U_z(\Sigma)$.



Event	Guard and update
a	$x := x + 1$
b	$x < 2$

Fig. 14 Example of applying synchronous composition, initially $x = 0$. For each event the guard and update is displayed in the table. In the coordinator C, the addition to the guards and updates of the events are depicted

When a variable is unfolded, a new alphabet is created based on the original alphabet and the variable that is unfolded, see the table in Fig. 15. For each event in the original alphabet, new events are created by combining the event with two values from the domain of the variable being unfolded. The first value represents the value of the unfolded variable before, and the second value represents the value of the unfolded variable after taking the transition labeled by the event. For example, in event $(\alpha, 0, 1)$ the value 0 is the value of variable x before, and 1 the value after taking the transition labeled originally by α. The variable alphabet may be larger than strictly necessary: it may contain events that are never enabled in the model. Subsequent abstraction-refinement pairs, like update simplification (see Online Resource 1, Section 5), false removal (see Online Resource 1, Section 7), and event merging (see Section 6.3), can remove such unnecessary events.

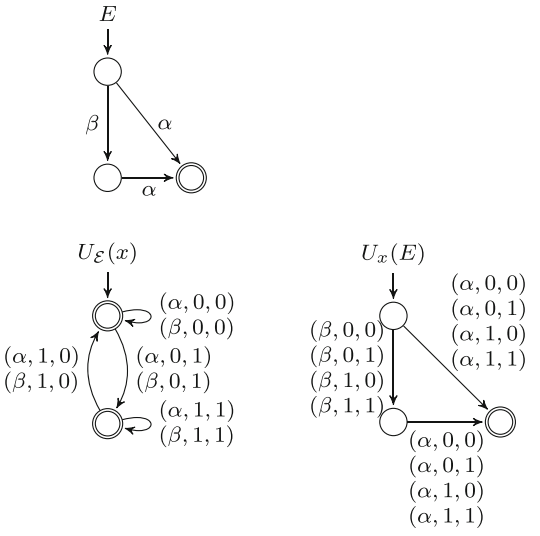
Definition 14 (Normalized variable EFA (Mohajerani et al. 2016)) Let $\mathcal{E}$ be a normalized EFA system with variable set V. The normalized variable EFA of $z \in V$ is

$$U_{\mathcal{E}}(z) = (\mathrm{dom}(z), V \setminus \{z\}, U_z(\Sigma_z), \rightarrow_z, \hat{v}_0(z), \hat{v}_{0\setminus z}, \mathrm{dom}(z))$$

where

$$\begin{aligned}\Sigma_z &= \{\sigma \in \Sigma \mid z \in \mathrm{vars}(g_\sigma) \cup \mathrm{vars}(u_\sigma)\},\\ \rightarrow_z &= \{(a, (\sigma, a, b), g_\sigma[z \mapsto a] \wedge b = u_\sigma(z)[z \mapsto a], u_{\sigma\setminus z}[z \mapsto a], b) \mid\\ &\qquad \sigma \in \Sigma_z, (x, \sigma, g_\sigma, u_\sigma, y) \in \rightarrow_{\mathcal{E}}, a, b \in \mathrm{dom}(z)\},\end{aligned}$$

with $\mathrm{vars}(g_\sigma)$ all variables in guard g_σ, $\mathrm{vars}(u_\sigma)$ all variables in update u_σ, $u_{\sigma\setminus z}$ the update function without the update expression for z, $u_{\sigma\setminus z}[z \mapsto a]$ the update function where in each update expression the variable z is substituted by a, and $\hat{v}_{0\setminus z}$ is the initial valuation without variable z.



Event	Guard and update
α	$\mathbf{T}; x := x, y := x + 1$
β	$\mathbf{T}; x := y + 1, y := y$
$(\alpha, 0, 0)$	$\mathbf{T} \wedge 0 = 0; y := 0 + 1$
$(\alpha, 0, 1)$	$\mathbf{T} \wedge 0 = 1; y := 0 + 1$
$(\alpha, 1, 0)$	$\mathbf{T} \wedge 1 = 0; y := 1 + 1$
$(\alpha, 1, 1)$	$\mathbf{T} \wedge 1 = 1; y := 1 + 1$
$(\beta, 0, 0)$	$\mathbf{T} \wedge 0 = y + 1; y := y$
$(\beta, 0, 1)$	$\mathbf{T} \wedge 1 = y + 1; y := y$
$(\beta, 1, 0)$	$\mathbf{T} \wedge 0 = y + 1; y := y$
$(\beta, 1, 1)$	$\mathbf{T} \wedge 1 = y + 1; y := y$

Fig. 15 Example of unfolding variable x, taken from Mohajerani et al. (2016). In this example, $\mathrm{dom}(x) = \mathrm{dom}(y) = \{0, 1\}$. Initially, $\hat{v}_0(x) = 0$ and $\hat{v}_0(y) = 0$. In the table, the top two events constitute the original alphabet and the bottom ones the alphabet after variable unfolding

The normalized variable EFA captures all theoretically possible assignments of that variable. As this EFA represents the effect of the unfolded variable on the EFA system, only events from the original EFA system that use this variable are included in the normalized variable EFA. Figure 15 shows an example where variable x is unfolded resulting in the normalized variable EFA $U_{\mathcal{E}}(x)$. The guards in a normalized variable EFA after unfolding a variable consists of the original guard where the variable is substituted by its current value in conjunction with the update of that variable. This is done to include the fact that a transition is taken in the normalized variable EFA only for the value that the variable would be updated to in the original system. Observe that several guards of the normalized variable EFA can be simplified with the abstraction-refinement pair called update simplification.

When a variable is unfolded, new events are introduced. To keep the complete EFA system normalized, all other EFAs in the EFA system need to be updated with these new events in a similar manner as local normalization (see Section 5). Furthermore, the guards and updates of events used to create the normalized variable EFA need to be changed in all EFAs in the same manner as in the normalized variable EFA. Both changes are captured below in the definition of variable expansion.

Definition 15 (Variable expansion (Mohajerani et al. 2016)) Let $E = (L, V, \Sigma, \rightarrow, l_0, \hat{v}_0, L_m)$ be an EFA and let $z \in V$. The expansion of E after unfolding variable z is defined by

$$U_z(E) = (L, V \setminus \{z\}, \Sigma^U, \rightarrow^U, l_0, \hat{v}_{0\setminus z}, L_m)$$

where

$$\begin{aligned}
\Sigma^U &= U_z(\Sigma \cap \Sigma_z) \cup (\Sigma \setminus \Sigma_z), \\
\rightarrow^U &= \{(x, (\sigma, a, b), g_\sigma[z \mapsto a] \wedge b = u_\sigma(z)[z \mapsto a], u_{\sigma\setminus z}[z \mapsto a], y) \mid \\
&\qquad \sigma \in \Sigma \cap \Sigma_z, (x, \sigma, g_\sigma, u_\sigma, y) \in \rightarrow\} \cup \\
&\qquad \{(x, \sigma, g_\sigma, u_\sigma, y) \mid \sigma \in \Sigma \setminus \Sigma_z, (x, \sigma, g_\sigma, u_\sigma, y) \in \rightarrow\}
\end{aligned}$$

Figure 15 shows an example of variable expansion of E with respect to variable x.

Given these definitions, unfolding variable z in an EFA system is defined as follows.

Definition 16 (Variable unfolding (Mohajerani et al. 2016)) Let $\mathcal{E} = \{E^1, \ldots, E^n\}$ be a normalized EFA system with variable set V and $z \in V$ a variable. The result of unfolding z in $\mathcal{E}$ is

$$\mathcal{E}_{\setminus z} = \{U_{\mathcal{E}}(z), U_z(E^1), \ldots, U_z(E^n)\}.$$

Two things need to be accomplished to properly refine a coordinator based on the abstracted system back to the original system: events need to be renamed back and variables needs to be reintroduced. The first action is needed as coordinator equivalence is based on having the same closed-loop language, which in essence requires the same alphabet. The second action is needed to ensure that all following refinements of the coordinator can use the unfolded variables. Renaming can be achieved by the renaming function ρ_z. Reintroducing an unfolded variable can be achieved by calculating the synchronous composition of the abstracted coordinator after renaming and the original EFA system. Now Theorem 4 confirms that the coordinator can be refined. The proof of this theorem can be found in Section 6 of Online Resource 1.

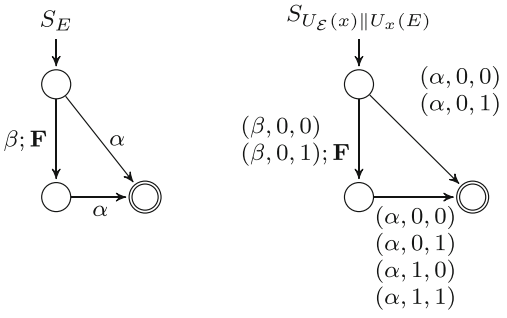
Theorem 4 (Variable unfolding) *Let* $(\mathcal{E}, \xi_1)$ *be a coordinator tuple with* $\mathcal{E}$ *a deterministic normalized EFA system with variable set* V *and* $z \in V$. *Then refinement function* $\xi(\mathcal{G}) = \rho_z(\mathcal{G}) \parallel \mathcal{E}$ *for any EFA system* $\mathcal{G}$ *ensures that* $(\mathcal{E}, \xi_1) \simeq_{co} (\mathcal{E}_{\setminus z}, \xi_1 \circ \xi)$.

Example The coordinators synthesized for the EFA systems before and after variable unfolding as presented in Fig. 15 are shown in Fig. 16. In the initial location, the transition labeled with β needs to be disabled to prevent the system from being blocking after taking this transition. During supervisor synthesis on the abstracted system, the algorithm evaluates the guards and updates on the transitions and finds out that several of them are not possible, as their guards evaluate to false. Therefore, the synthesized coordinator only strengthens the guard on the transition from the initial location labeled with event $(\beta, 0, 1)$; other transitions do not need to be strengthened. If we apply the refinement on coordinator $S_{U_{\mathcal{E}}(x) \parallel U_x(E)}$, we observe that the strengthened transition labeled with $(\beta, 0, 1)$ is now labeled with β. Therefore, it holds that $S_E = \rho_x(S_{U_{\mathcal{E}}(x) \parallel U_x(E)}) \parallel E$.

6.3 Event merging

The abstraction of event merging identifies events that can be merged back into a single event, as proposed in Mohajerani et al. (2016). Events can be merged if they have the same guard and update, for all EFAs except one they appear on transitions with the same source and target location, and they have the same controllability status. Event merging can often be applied directly after variable unfolding, as introduced in Section 6.2. By construction of the EFA system after variable unfolding, all new events introduced for a single event appear everywhere on the same transition except the newly created normalized variable EFA, see the example in Fig. 15.

To refine a coordinator synthesized for the abstracted system, the merged events need to be converted back to original events. As event merging applies a renaming to go from multiple events to a single event, refinement applies an inverse renaming to go from a single event to multiple events. But, in general, for any renaming and EFA A it holds that $\mathcal{L}(\rho^{-1}(\rho(A))) \supseteq \mathcal{L}(A)$, i.e, more behavior may be possible after applying renaming and



Event	Guard and update
α	$\mathbf{T}; x := x, y := x + 1$
β	$\mathbf{T}; x := y + 1, y := y$
$(\alpha, 0, 0)$	$\mathbf{T} \wedge 0 = 0; y := 0 + 1$
$(\alpha, 0, 1)$	$\mathbf{T} \wedge 0 = 1; y := 0 + 1$
$(\alpha, 1, 0)$	$\mathbf{T} \wedge 1 = 0; y := 1 + 1$
$(\alpha, 1, 1)$	$\mathbf{T} \wedge 1 = 1; y := 1 + 1$
$(\beta, 0, 0)$	$\mathbf{T} \wedge 0 = y + 1, y := y$
$(\beta, 0, 1)$	$\mathbf{T} \wedge 1 = y + 1, y := y$
$(\beta, 1, 0)$	$\mathbf{T} \wedge 0 = y + 1, y := y$
$(\beta, 1, 1)$	$\mathbf{T} \wedge 1 = y + 1, y := y$

Fig. 16 The coordinators S_E and $S_{U_{\mathcal{E}}(x) \parallel U_x(E)}$ synthesized for the EFA systems $\{E\}$ and $\{U_{\mathcal{E}}(x), U_x(E)\}$ as shown in Fig. 15, respectively. In the table, the top two events constitute the original alphabet and the bottom ones the alphabet after variable unfolding

inverse renaming than originally possible. This may also be the case for event merging. To solve this, the synchronous composition of the coordinator obtained after inverse renaming and the original system ensures that the languages become equal. Theorem 5 formalizes this. The proof of this theorem can be found in Section 9 of Online Resource 1.

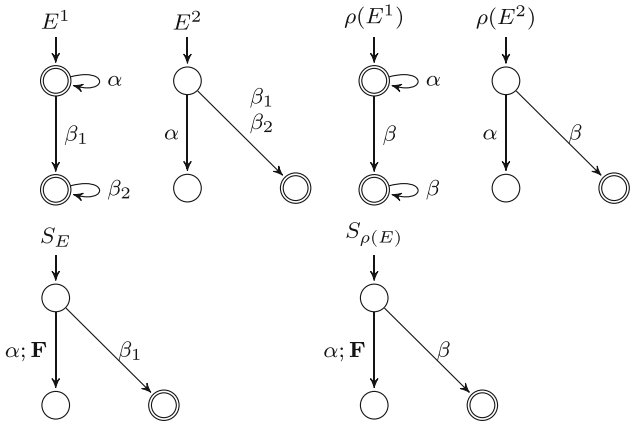
Theorem 5 (Event merging) *Let $(\mathcal{E}, \xi_1)$ be a coordinator tuple with $\mathcal{E} = \{E^1, \ldots, E^n\}$ a deterministic normalized EFA system. Let $E^k \in \mathcal{E}$ and let $\rho : \Sigma_\mathcal{E} \to \Sigma'$ be a renaming such that the following conditions hold for all $\sigma_1, \sigma_2 \in \Sigma_\mathcal{E}$ with $\rho(\sigma_1) = \rho(\sigma_2)$:*

1. *$g_{\sigma_1} = g_{\sigma_2}$ and $u_{\sigma_1} = u_{\sigma_2}$,*
2. *for all $i \neq k$, it holds that $\sigma_1 \in \Sigma^i$ if and only if $\sigma_2 \in \Sigma^i$, and for all $l_1, l_2 \in L^i$ it holds that $l_1 \xrightarrow{\sigma_1, g_{\sigma_1}, u_{\sigma_1}} l_2$ in E^i if and only if $l_1 \xrightarrow{\sigma_2, g_{\sigma_2}, u_{\sigma_2}} l_2$ in E^i,*
3. *$\sigma_1 \in \Sigma_c$ if and only if $\sigma_2 \in \Sigma_c$.*

Then refinement function $\xi(\mathcal{G}) = \rho^{-1}(\mathcal{G}) \parallel \mathcal{E}$ for any EFA system $\mathcal{G}$ with alphabet Σ' ensures that $(\mathcal{E}, \xi_1) \simeq_{co} (\rho(\mathcal{E}), \xi_1 \circ \xi)$.

Keep in mind that we assumed that all abstracted systems remain deterministic. Event merging can be properly refined only under this assumption.

Example Figure 17 shows an example where event merging is applied. In the original EFA system, events β_1 and β_2 have the same guard and update, and they appear on the same transitions in all EFAs except E^1. Therefore, these events can be merged into, for example, β, which results in EFAs $\rho(E^1)$ and $\rho(E^2)$. For the original and the abstracted system a coordinator is synthesized. The strengthened guards are shown directly in the automaton representation of the coordinators. We can now observe that simply applying inverse renaming is insufficient, as a refinement as $\rho^{-1}(S_{\rho(E)})$ allows for β_1 and β_2 from the initial location, while the coordinator based on the original system only allows β_1. This problem is solved by taking the synchronous composition of the renamed abstracted coordinator with the original EFA system, i.e., $\mathcal{L}(\rho^{-1}(S_{\rho(E)}) \parallel (E^1 \parallel E^2)) = \mathcal{L}(S_E)$.



Event	Guard and update
α	$x := 1$
β_1	$x = 0$
β_2	$x = 0$
α	$x := 1$
β	$x = 0$

Fig. 17 Example of event merging and coordinator refinement. In the table, the top three events constitute the original alphabet, while the bottom two the one after event merging. S_E and $S_{\rho(E)}$ are the coordinators for the EFA systems $\{E^1, E^2\}$ and $\{\rho(E^1), \rho(E^2)\}$, respectively

Algorithm 1 EFA-based compositional nonblocking verification and coordinator refinement.

Input: deterministic EFA system $\mathcal{E} = \{E^1, \ldots, E^n\}$
Output: *nonblocking* in case when $\mathcal{E}$ is nonblocking, or *blocking* and refined coordinator C_r in case when $\mathcal{E}$ is blocking

1: $V = V^1 \cup \ldots \cup V^n$
2: $(\mathcal{E}, \underline{\xi}) = \text{normalize}(\mathcal{E})$
3: **while** $|\mathcal{E}| > 1 \wedge |V| > 0$ **do**
4: $\quad (V_c, \mathcal{E}_c) = \text{selectCandidate}(\mathcal{E})$
5: $\quad$ **if** $V_c \neq \emptyset$ **then**
6: $\quad\quad v = \text{selectVariable}(V_c)$
7: $\quad\quad V = V \setminus \{v\}$
8: $\quad\quad (E, \underline{\xi'}) = \text{unfold}(v)$
9: $\quad\quad \underline{\xi = \xi \circ \xi'}$
10: $\quad$ **else**
11: $\quad\quad \mathcal{E} = \mathcal{E} \setminus \mathcal{E}_c$
12: $\quad\quad E = \text{synchronize}(\mathcal{E}_c)$
13: $\quad\quad \underline{\xi = \xi \circ \text{id}}$
14: $\quad$ **end if**
15: $\quad (\mathcal{E} \cup \{E\}, \underline{\xi'}) = \text{removeEvents}(\mathcal{E} \cup \{E\})$
16: $\quad \underline{\xi = \xi \circ \xi'}$
17: $\quad \underline{\Gamma = \text{getLocalEvents}(E, \mathcal{E})}$
18: $\quad (E, \underline{\xi'}) = \text{simplify}(E, \Gamma)$ // Not all simplifications of Mohajerani et al. (2016) can be used.
19: $\quad \underline{\xi = \xi \circ \xi'}$
20: $\quad \underline{(\mathcal{E} \cup \{E\}, \xi') = \text{removeEvents}(\mathcal{E} \cup \{E\})}$
21: $\quad \underline{\xi = \xi \circ \xi'}$
22: $\quad \underline{\mathcal{E} = \mathcal{E} \cup \{E\}}$
23: **end while**
24: **if** monolithicVerification($\mathcal{E}$) is successful **then**
25: $\quad$ **return** *nonblocking*
26: **else**
27: $\quad \underline{C = \text{monolithicSynthesis}(\mathcal{E})}$
28: $\quad \underline{C_r = \xi(C)}$
29: $\quad$ **return** $\underline{\textit{blocking}}$ and $\underline{C_r}$
30: **end if**

7 Algorithm

This section enhances the EFA-based compositional nonblocking verification algorithm of Mohajerani et al. (2016) to refine the coordinator synthesized from the abstracted system. For a detailed discussion on the EFA-based compositional nonblocking verification algorithm, including several heuristics for choosing the order of abstraction-refinement pairs, the reader is referred to Mohajerani et al. (2016).

Algorithm 1 shows the enhanced algorithm. The enhancement, i.e., a contribution of this paper, is visualized by underlining the additions. Algorithm 1 requires a deterministic EFA

system as input, and produces the answer *nonblocking* if the EFA system is nonblocking, or the answer *blocking* together with the refined coordinator C_r if the EFA system is blocking. The algorithm can be split into three parts: the initialization and normalization in Lines 1 and 2, the abstractions in Lines 3-23, and the verification and refinement in Lines 24-29.

The first two parts, initialization and normalization and abstractions, are basically the same as in the EFA-based compositional nonblocking verification algorithm of Mohajerani et al. (2016), where now the refinement function ξ is constructed along the way according to the theorems in Sections 5 and 6, and Online Resource 1. In other words, Algorithm 1 keeps track of which abstractions are performed. Furthermore, the simplify procedure needs to be adjusted, as no longer all FA-based abstractions for verification can be used.

The third part of Algorithm 1 is responsible for synthesizing and refining the coordinator in case a coordinator is needed to solve the blocking issue. Some algorithms available in literature for verification are Abdelwahed and Wonham (2003) and Flordal and Malik (2006), and for synthesis are Ouedraogo et al. (2011) and Fei et al. (2014). If verification of the abstracted EFA $\mathcal{E}_{\text{abstracted}}$ system is successful, then the algorithm returns that the original EFA system provided as input is nonblocking in Line 25. Otherwise, the algorithm continues to synthesize a coordinator C based on the abstracted EFA system. By using only coordinator equivalence preserving abstractions, see Table 1 for an overview, it follows that the initial and abstracted coordinator tuples are coordinator equivalent, i.e. $(\mathcal{E}, \text{id}) \simeq_{\text{co}} (\mathcal{E}_{\text{abstracted}}, \xi)$. Thus we conclude that $\mathcal{L}(\text{id}(\text{sup}CN(\mathcal{E}))) = \mathcal{L}(\xi(\text{sup}CN(\mathcal{E}_{\text{abstracted}})))$. Therefore, in Line 28 the synthesized coordinator C is refined by applying the refinement function ξ. The final result is returned in Line 29.

The theoretical worst-case complexity of this algorithm is the same as the worst-case complexity of monolithic nonblocking verification and monolithic synthesis. This can be seen as follows. Consider an EFA system where each EFA has the same alphabet. As the compositional nonblocking verification algorithm tries to utilize local events, it needs to calculate the synchronous product of all EFAs in this EFA system to have at least one local event. In this case, there is no computational complexity reduction compared to monolithic nonblocking verification and monolithic synthesis. Nevertheless, experimental results as presented in Mohajerani et al. (2016) show that for realistic systems the observed computation time of compositional nonblocking verification is between 1 and 200 seconds. As monolithic verification and monolithic synthesis have the same computational complexity, see Cassandras and Lafortune (2008), and all refinements are straightforward, an implementation of Algorithm 1 is expected to have similar computational results as the ones reported in Mohajerani et al. (2016) for compositional nonblocking verification.

8 Conclusion

In this paper, the general framework of EFA-based compositional nonblocking verification of Mohajerani et al. (2016) is enhanced such that in case of a blocking system a coordinator can be synthesized and refined back to the original system. Such a synthesized coordinator ensures that the closed-loop system of the original EFA system and coordinator is nonblocking, controllable, and maximally permissive. The notion of a general refinement function is introduced, which transforms a given EFA system into another EFA system. All presented abstraction-refinement pairs belong to the particular class of refinement functions Ξ.

To reason about refinement of coordinators, we introduced the notion of coordinator equivalence. All abstraction-refinement pairs presented in this paper are shown to preserve this property. This allows for a compositional approach, where the closed-loop behavior after refining the coordinator synthesized for the abstracted system is the same as the closed-loop behavior of the coordinator synthesized for the original system.

Future work includes the implementation of the framework and the abstraction-refinement pairs in the CIf tooling (van Beek et al. 2014). Such an implementation allows the framework to be used in the development of modular supervisory controllers for large-scale infrastructural applications like, for example, bridges (Reijnen et al. 2020), locks (Reijnen et al. 2017), and tunnels (Moormann et al. 2020).

Acknowledgements The authors thank S. Mohajerani for the several fruitful discussions on compositional nonblocking verification and compositional supervisor synthesis.

References

Abdelwahed S, Wonham WM (2003) Interacting DES: modelling and analysis. In: IEEE International conference on systems, man and cybernetics, vol 5, pp 4222–4229, https://doi.org/10.1109/ICSMC.2003.1245648

Åkesson K, Flordal H, Fabian M (2002) Exploiting modularity for synthesis and verification of supervisors. IFAC Proc 35(1):175–180. https://doi.org/10.3182/20020721-6-ES-1901.00517

Cai K, Wonham WM (2010) Supervisor localization: a top-Down approach to distributed control of discrete-Event systems. IEEE Trans Autom Control 55(3):605–618

Cassandras CG, Lafortune S (2008) Introduction to discrete event systems, 2nd edn. Springer, Boston

Chen YL, Lin F (2000) Modeling of discrete event systems using finite state machines with parameters. In: IEEE International conference on control applications, pp 941–946. https://doi.org/10.1109/CCA.2000.897591

Chen YL, Lin F (2001) Safety control of discrete event systems using finite state machines with parameters. In: American control conference, p conference=975–980,. https://doi.org/10.1109/ACC.2001.945847

Cheng KT, Krishnakumar AS (1996) Automatic generation of functional vectors using the extended finite state machine model. ACM Trans Des Autom Electron Syst 1(1):57–79. https://doi.org/10.1145/225871.225880

de Queiroz MH, Cury JER (2000) Modular supervisory control of large scale discrete event systems. In: Discrete event systems. Springer US, pp 103–110

Fabian M, Fei Z, Miremadi S, Lennartson B, Åkesson K (2014) Supervisory control of manufacturing systems using extended finite automata. In: Campos J, Seatzo C, Xie X (eds) Formal methods in manufacturing, Industrial information technology. Taylor & Francis Inc., pp 295-314

Fei Z, Miremadi S, ÅKesson K, Lennartson B (2014) Efficient symbolic supervisor synthesis for extended finite automata. IEEE Trans Control Syst Technol 22(6):2368–2375. https://doi.org/10.1109/TCST.2014.2303134

Feng L, Wonham WM (2008) Supervisory control architecture for discrete-Event systems. IEEE Trans Autom Control 53(6):1449–1461. https://doi.org/10.1109/TAC.2008.927679

Flordal H, Malik R (2006) Modular nonblocking verification using conflict equivalence. In: 8th International workshop on discrete event systems, pp 100–106. https://doi.org/10.1109/WODES.2006.1678415

Gommans RPA (2016) Modular supervisory control synthesis for automata with data. Master thesis, Eindhoven University of Technology, report number CST 2016.119

Hill R, Tilbury D (2006) Modular supervisory control of discrete-event sytems with abstraction and incremental hierarchical construction. In: 8th International workshop on discrete event systems, pp 399–406

Komenda J, Masopust T, van Schuppen JH (2016) Control of an engineering-structured multilevel discrete-event system. In: 13th International workshop on discrete event systems, pp 103–108. https://doi.org/10.1109/WODES.2016.7497833

Korssen T, Dolk V, Van de mortel-fronczak JM, Reniers MA, Heemels M (2017) Systematic model-based design and implementation of supervisors for advanced driver assistance systems. IEEE Trans Intell Transp Syst 19(2):533–544. https://doi.org/10.1109/TITS.2017.2776354

Ma C, Wonham WM (2005) Nonblocking supervisory control of state tree structures. No. 317 in Lecture Notes in Control and Information Sciences. Springer, Berlin

Malik R, Flordal H (2008) Yet another approach to compositional synthesis of discrete event systems. In: 2008 9th International workshop on discrete event systems, pp 16–21. https://doi.org/10.1109/WODES.2008.4605916

Malik R, Teixeira M (2016) Modular supervisor synthesis for extended finite-state machines subject to controllability. In: 13th International workshop on discrete event systems. IEEE, pp 91–96

Malik R, Teixeira M (2020) Synthesis of least restrictive controllable supervisors for extended finite-state machines with variable abstraction. Discrete Event Dynamic Systems. https://doi.org/10.1007/s10626-019-00302-z

Markovski J, Jacobs KGM, van Beek DA, Somers LJ, Rooda JE (2010) Coordination of resources using generalized state-based requirements. In: 10th International workshop on discrete event systems, pp 300–305

Miremadi S, Åkesson K, Lennartson B, Fabian M (2010) Supervisor computation and representation: a case study. In: 10th International Workshop on discrete event systems, pp 275–280

Mohajerani S, Malik R, Fabian M (2011) Nondeterminism avoidance in compositional synthesis of discrete event systems. In: 2011 IEEE international conference on automation science and engineering, pp 19–24. https://doi.org/10.1109/CASE.2011.6042432

Mohajerani S, Malik R, Fabian M (2014) A framework for compositional synthesis of modular nonblocking supervisors. IEEE Trans Autom Control 59(1):150–162. https://doi.org/10.1109/TAC.2013.2283109

Mohajerani S, Malik R, Fabian M (2016) A framework for compositional nonblocking verification of extended finite-state machines. Discrete Event Dyn Syst 26(1):33–84. https://doi.org/10.1007/s10626-015-0217-y

Mohajerani S, Malik R, Fabian M (2017) Compositional synthesis of supervisors in the form of state machines and state maps. Automatica 76:277–281. https://doi.org/10.1016/j.automatica.2016.10.012

Moormann L, Maessen P, Goorden MA, van de Mortel-Fronczak JM, Rooda E (2020) Design of a tunnel supervisory controller using synthesis-based engineering. In: ITA-AITES World tunnel congress, pp.573–578

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

Pena PN, da Cunha AEC, Cury JER, Lafortune S (2008) New results on the nonconflict test of modular supervisors. In: 9th International workshop on discrete event systems, pp 468–473. https://doi.org/10.1109/WODES.2008.4605991

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

Ramadge PJG, Wonham WM (1989) The control of discrete event systems. Proc IEEE 77(1):81–98

Reijnen FFH, Goorden MA, van de Mortel-Fronczak JM, Rooda JE (2017) Supervisory control synthesis for a waterway lock. In: IEEE Conference on control technology and applications, pp 1562–1568. https://doi.org/10.1109/CCTA.2017.8062679

Reijnen FFH, Goorden MA, Van de mortel-fronczak JM, Rooda JE (2020) Modeling for supervisor synthesis – a lock-bridge combination case study. Discrete Event Dyn Syst 30(3):499–532. https://doi.org/10.1007/s10626-020-00314-0

Reniers MA (2018) An engineering perspective on model-based design of supervisors. FAC-PapersOnLine 51(7):257–264. https://doi.org/10.1016/j.ifacol.2018.06.310

Rudie K, Wonham WM (1992) Think globally, act locally: decentralized supervisory control. IEEE Trans Autom Control 37(11):1692–1708

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

Su R, Wonham WM (2004) Supervisor reduction for discrete-event systems. Discrete Event Dyn Syst 14(1):31–53. https://doi.org/10.1023/B:DISC.0000005009.40749.b6

Su R, van Schuppen JH, Rooda JE (2009) Synthesize nonblocking distributed supervisors with coordinators. In: 17th Mediterranean conference on control and automation, pp 1108–1113. https://doi.org/10.1109/MED.2009.5164694

Su R, van Schuppen JH, Rooda JE, Hofkamp AT (2010) Nonconflict check by using sequential automaton abstractions based on weak observation equivalence. Automatica 46(6):968–978. https://doi.org/10.1016/j.automatica.2010.02.025

Teixeira M, Cury JER, de Queiroz MH (2011) Local modular supervisory control of DES with distinguishers. In: 16th Conference on emerging technologies factory automation, pp 1–8. https://doi.org/10.1109/ETFA.2011.6059038

van Beek DA, Fokkink WJ, Hendriks D, Hofkamp A, Markovski J, van de Mortel-Fronczak JM, Reniers MA (2014) CIf 3: Model-based engineering of supervisory controllers. In: Tools and algorithms for the construction and analysis of systems. Lecture Notes in Computer Science. Springer, Berlin, pp 575–580. https://doi.org/10.1007/978-3-642-54862-848

Wong K, Wonham W (2004) On the computation of observers in discrete-event systems. Discrete Event Dyn Syst 14(1):55–107. https://doi.org/10.1023/B:DISC.0000005010.55515.27

Wong KC, Wonham WM (1998) Modular control and coordination of discrete-event systems. Discrete Event Dyn Syst 8(3):247–297. https://doi.org/10.1023/A:1008210519960

Wonham W, Cai K, Rudie K (2017) Supervisory control of discrete-event systems: a brief history – 1980-2015. In: 20th IFAC World Congress, pp 1791–1797

Wonham WM, Cai K (2019) Supervisory control of discrete-event systems, 1st edn. Springer, Berlin

Wonham WM, Ramadge PJG (1988) Modular supervisory control of discrete-event systems. Math Control Signals Syst 1(1):13–30

Zhong H, Wonham WM (1990) On the consistency of hierarchical supervision in discrete-event systems. IEEE Trans Autom Control 35(10):1125–1134

Zita A, Mohajerani S, Fabian M (2017) Application of formal verification to the lane change module of an autonomous vehicle. In: 13th IEEE Conference on automation science and engineering

Martijn A. Goorden received the M.Sc. degree (cum laude) in systems and control from Eindhoven University of Technology, Eindhoven, The Netherlands, in 2015, and the Ph.D. degree in mechanical engineering from Eindhoven University of Technology, Eindhoven, The Netherlands, in 2019. His current research interests are in the area of model-based systems engineering and supervisory control synthesis.

Martin Fabian received his Ph.D. degree in control engineering in 1995 from Chalmers University of Technology, Gteborg, Sweden. He is currently a Professor with the Department of Signals and Systems, Chalmers University of Technology. His research interests involve modelling and supervisory control of discrete-event systems, modular and compositional methods for complex systems, and generic architectures for flexible production systems.

Joanna M. van de Mortel-Fronczak received the M.Sc. degree in computer science from AGH University of Science and Technology, Cracow, Poland, in 1982 and the Ph.D. degree in computer science from Eindhoven University of Technology, Eindhoven, The Netherlands, in 1993. Since 1997, she has been with the Department of Mechanical Engineering, Eindhoven University of Technology. Her research interests include model-based engineering and synthesis of supervisory control systems.

Michel A. Reniers is currently an Associate Professor in model-based engineering of supervisory control at the Department of Mechanical Engineering, Eindhoven University of Technology. He has authored over 100 journal and conference papers, and is the supervisor of ten Ph.D. students. His research portfolio ranges from model-based systems engineering and model-based validation and testing to novel approaches for supervisory control synthesis. Applications of this work are mostly in the areas of high-tech systems and cyber-physical systems.

Wan J. Fokkink received his Ph.D. degree in Computer Science from the University of Amsterdam, Amsterdam, The Netherlands. Since 2004 he is full professor of Theoretical Computer Science at the Vrije Universiteit Amsterdam. Since 2012 he is moreover professor of Model-Based System Engineering at Eindhoven University of Technology. His research focus is on the design and analysis of distributed computer systems.

Jacobus E. Rooda received the M.Sc. degree from Wageningen University of Agricultural Engineering, Wageningen, The Netherlands, and the Ph.D. degree from Twente University, Enschede, The Netherlands. Since 1985, he has been a Professor of (Manufacturing) Systems Engineering at the Department of Mechanical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands. Since 2010, he is a Professor Emeritus. He is still active in the research fields of engineering design for industrial systems, and of supervisory control thereof.

Affiliations

Martijn A. Goorden[1] · Martin Fabian[2] · Joanna M. van de Mortel-Fronczak[1] · Michel A. Reniers[1] · Wan J. Fokkink[3] · Jacobus E. Rooda[1]

Martin Fabian
fabian@chalmers.se

Joanna M. van de Mortel-Fronczak
j.m.v.d.mortel@tue.nl

Michel A. Reniers
m.a.reniers@tue.nl

Wan J. Fokkink
w.j.fokkink@vu.nl

Jacobus E. Rooda
j.e.rooda@tue.nl

1 Eindhoven University of Technology, Eindhoven, Netherlands

2 Chalmers University of Technology, Gothenburg, Sweden

3 Vrije Universiteit Amsterdam, Amsterdam, Netherlands