15th Triennial World Congress, Barcelona, Spain

EXPLOITING MODULARITY FOR SYNTHESIS AND VERIFICATION OF SUPERVISORS

K. Åkesson, H. Flordal, M. Fabian

Corresponding author: ka@s2.chalmers.se
Department of Signals and Systems
Chalmers University of Technology, Gothenburg, Sweden

Abstract: Efficient algorithms for synthesis and verification of supervisors in the Supervisory Control Theory framework are presented. The presented algorithms solve the controllability problem. In many real-world applications both the plant and specification is given as a set of interacting automata or processes. In this work, we exploit this modular structure to reduce the computational effort. First, we present an algorithm that verifies if a given supervisor is controllable with respect to a plant. Second, we show how to synthesize a set of modular supervisors that while interacting with the original supervisors guarantees that the closed system is controllable. Third, we show how the verification algorithm can be used as an efficient language inclusion algorithm. The presented algorithms are benchmarked on a real-world application.

Keywords: Discrete-event systems, Supervisory Control, Verification, Synthesis

1. INTRODUCTION

The Supervisory Control Theory (SCT) as introduced by Ramadge and Wonham presents a framework for synthesizing supervisors satisfying closed-loop specifications. For a detailed overview of the SCT see (Ramadge and Wonham, 1989). The SCT consists of two main components, the *plant* and the *supervisor*. The task of the supervisor is to dynamically disable events generated by the plant so that a given specification is fulfilled. The plant events are divided into two disjoint sub-sets, the *controllable* and the *uncontrollable* events. The supervisor is only allowed to disable controllable events, that is, the supervisor must be *controllable* with respect to the plant. In addition, the supervisor must also be such that from any state reachable in the closed-loop system, some state out of a set of designated states must be reachable. This is known as the *non-blocking* problem, and is a generalization of the deadlock problem. The deadlock problem has been studied extensively in the computer science literature (Corbett, 1996).

To transfer the SCT from academia into industry it is crucial to have efficient algorithms for synthesis and verification of supervisors. A major problem with the SCT is the computational effort, see (Gohari and Wonham, 2000). One approach to handle complexity is to use an efficient representation of the state-space. Binary Decision Diagrams (BDDs) (Bryant, 1992) are widely used for this purpose. BDDs and its variants have been used for supervisory synthesis in (Hoffmann and Wong-Toi, 1992; Zhang and Wonham, 2001; Tronci, 1997). To the authors knowledge, the current BDD-based implementations do not take advantage of the modular structure of the plant and the specification. Thus, BDD based approaches rely on an exhaustive search of the global state-space. Since this state-space might be very large due to the state-explosion problem, even BDDs have their limits. Another approach to handle the complexity is to exploit the inherent *modular structure* of the problem. Algorithms can take advantage of the structure to solve smaller sub-problems that together solve the entire problem. Modular approaches to supervisory control have been presented in (Brandin *et al.*, 2000; Wonham and Ramadge, 1988; Wong and Wonham, 1998). It is our belief that BDD-based approaches should be combined with modular approaches to efficiently handle even larger systems than currently.

In this paper, we attack the controllability verification and synthesis problem by exploiting the modular structure of both the supervisor and plant. Particularly, we show how it is possible to exploit the fact that the alphabets of the different sub-systems might be unequal. This differ from the approach in (Brandin *et al.*, 2000), where is is assumed that all sub-systems have the same alphabet. Efficient algorithms for verification and synthesis, both optimal and sub-optimal, are presented. The presented algorithms are implemented in the verification and synthesis tool `Supremica`. The algorithms are applied to a real-world central-locking system, and we present the computational effort needed for verifying and synthesizing the system.

2. MATHEMATICAL PRELIMINARIES

The notation used in the following sections is introduced in this section. First, the modeling framework is presented then fundamental properties of systems are presented.

2.1 Modeling Framework

Finite automata and regular languages, (Hopcroft and Ullman, 1979), are used as our modeling framework.

Definition 1. (Deterministic Finite Automata). A deterministic finite automaton is a 4-tuple defined as $\mathcal{A} = \langle Q, \Sigma, \delta, q_i \rangle$ where Q is a nonempty finite set of states; Σ is a nonempty finite set of event-symbols, the *alphabet*; $\delta : Q \times \Sigma \to Q$ is the partial transition function and $q_i \in Q$ is the initial state. ■

In this paper the term *automata* will be used to refer to deterministic finite automata. A *string* is a finite sequence of events. A *language* is a set of strings. Let Σ^* be the language that contains all strings that can be constructed with events from the alphabet Σ, including the empty string ε. The *prefix closure* of a string s, denoted $\overline{s}$, is the set of all initial sub-strings of s, i.e.,

$$\overline{s} = \left\{ t \in \Sigma^* \,\middle|\, \exists t' \in \Sigma^* \text{ s.t. } tt' = s \right\}$$

Let $L(A)$ be the language generated by the automaton A. $L(A)$ is prefixed-closed, that is, the prefixe closure of all strings in $L(A)$ are also in $L(A)$. For a state q, $\Sigma(q)$ is the set of events defined from q.

In this paper we use full synchronous composition (FSC) (Hoare, 1985) to model interaction between automata.

Definition 2. (Full Synchronous Composition). The full synchronous composition of two automata A^1, A^2 is defined as $A = A^1||A^2$ where $Q = Q^1 \times Q^2$; $\Sigma = \Sigma^1 \cup \Sigma^2$; $q_i = \langle q_i^1, q_i^2 \rangle$, and the transition function is

$$\delta(\langle q^1, q^2 \rangle, \sigma) = \begin{cases} \langle \delta^1(q^1,\sigma), \delta^2(q^2,\sigma) \rangle & \sigma \in \Sigma(q^1) \cap \Sigma(q^2) \\ \langle \delta^1(q^1,\sigma), \{q^2\} \rangle & \sigma \in \Sigma(q^1) \setminus \Sigma^2 \\ \langle \{q^1\}, \delta^2(q^2,\sigma) \rangle & \sigma \in \Sigma(q^2) \setminus \Sigma^1 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

■

The FSC has a useful property – the composition is associative and commutative with respect to the generated languages. This property implies that composition is easily extended to more than two automata at a time, something that can be very efficient compared to synchronizing automata two and two. The reason is that a potential blow-up of the intermediate state-space is avoided.

The projection operator, *Proj*, is used to restrict a string to an event set Σ, by removing all occurrences of events not in Σ. Formally, the projection operator is defined as follows.

$$\begin{aligned} \text{Proj}_\Sigma(\varepsilon) &= \varepsilon \\ \text{Proj}_\Sigma(\sigma) &= \begin{cases} \sigma \text{ if } \sigma \in \Sigma \\ \varepsilon \text{ if } \sigma \notin \Sigma \end{cases} \\ \text{Proj}_\Sigma(s\sigma) &= \text{Proj}_\Sigma(s)\text{Proj}_\Sigma(\sigma) \end{aligned} \tag{1}$$

$Proj_\Sigma^{-1}(s)$ is the inverse to *Proj*,

$$\begin{aligned} s = \sigma_1\sigma_2\ldots\sigma_n \Rightarrow \\ \text{Proj}_\Sigma^{-1}(s) = \Sigma^*\sigma_1\Sigma^*\sigma_2\Sigma^*\ldots\Sigma^*\sigma_n\Sigma^* \end{aligned} \tag{2}$$

Note that $s \in Proj_\Sigma^{-1}(s)$. The projection and inverse-projection operators are extended to work on a language by applying the operator to all strings in the language.

In an automaton the inverse projection operation can be implemented by adding self-loops with events from Σ to all states. We assume that Σ in the inverse projection is disjoint from the alphabet of the automaton that generated the original language. For two automata P and S, the language of their FSC is $L(P||S) = L^{-1}(P) \cap L^{-1}(S)$, where

$$\begin{aligned} L^{-1}(P) &= Proj^{-1}_{\Sigma^S \setminus \Sigma^P}(L(P)), \\ L^{-1}(S) &= Proj^{-1}_{\Sigma^P \setminus \Sigma^S}(L(S)). \end{aligned} \tag{3}$$

Note that when $\Sigma_P = \Sigma_S$ then $L(P||S) = L(P) \cap L(S)$.

2.2 Controllability

Let P be the automaton modeling the plant, and let S be the supervisor. We will assume that P and S is composed of a set of *sub-plants* and *sub-supervisors*, respectively, i.e., $P = P_1||P_2||\cdots||P_m$ and $S = S_1||S_2||\cdots||S_n$. Both the sub-plants and the sub-supervisors might have different alphabets. A crucial requirement for S to function properly with the respect to the plant, is that it never tries to disable (or prevent) an uncontrollable event that can be generated by the plant; that is, that S is *controllable* with respect to P. The controllability condition can be written as a language inclusion test.

Definition 3. (Controllability).
S is controllable with respect to P if

$$L(S)\Sigma_u \cap L(P) \subseteq L(S). \tag{4}$$

■

We do not assume by definition that the supervisor is controllable with respect to the plant. Instead, we want to restrict the behavior of the supervisor in order to make it controllable. Thus, it is also possible to think of the supervisor as a specification. This is discussed in detail in sections 3 and 4.

Note that Definition 3 presupposes that P and S have the same alphabet, which is a natural assumption. However, in a modular setting, the individual sub-plants and sub-supervisors do not in general have the same alphabets. Since we want to exploit the modularity in verifying controllability and synthesizing controllable supervisors, we have to take the non-equality of the alphabets into account. In this case, Definition 3 does *not* capture the necessary requirements as is illustrated in Example 1. It might seem natural to extend the alphabets by introducing self-loops such that the alphabets become equal. This operation does not change $L(P_i||S_j)$, but unfortunately it not useful for modular controllability verification or synthesis, as is also shown by Example 1.

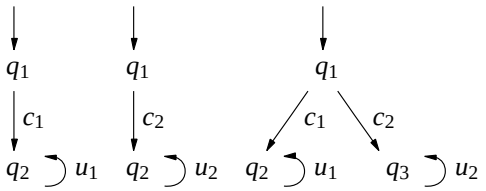

Fig. 1. The leftmost automaton is sub-plant P_1, the middle automaton is sub-plant P_2, and the rightmost automaton is the supervisor S. The plant is $P = P_1||P_2$. The alphabets are: $\Sigma^{P_i} = \{c_i, u_i\}$, $\Sigma^S = \{c_1, u_1, c_2, u_2\}$, $\Sigma_u = \{u_1, u_2\}$.

EXAMPLE 1. In Fig. 1 two sub-plants, P_1 and P_2 (the leftmost automata), together with a supervisor S (the rightmost automaton) for the composed plant $P = P_1||P_2$ are depicted.

Clearly, $\Sigma^P = \Sigma^S$ and S is controllable with respect to P. However, since $\Sigma^{P_1} \neq \Sigma^S$, we cannot directly use the normal definition of controllability, (3), to verify controllability of S with respect to P_1. The usual way to defeat this problem, is to augment Σ^{P_1} with the events of $\Sigma^S \setminus \Sigma^{P_1}$. This is effectively done by introducing self-loops into P_1 on the missing events, thus creating P_1', and then use the standard controllability verification algorithm to verify that S is controllable with respect to P_1'. As this example shows, though, this may surprisingly create the new problem that S is deemed *not to be* controllable with respect to P_1'. The reason for this is that the uncontrollable event u_2 will be self-looped in P_1' at both its states, and thus the strings u_2 and c_1u_2 both belong to $L(P_1')$. On the other hand, those strings do not belong to $L(S)$, while their prefixes ε and c_1 do. In the total plant P, the strings on which S fail controllability will not arise, since P_2 will only allow u_2 after c_2. Not being able to handle non-equal alphabets is clearly a problem when trying to do modular verification since the sub-systems might not look controllable when the total system is, or vice versa. Note also that the the same type of problems manifest themselves when $\Sigma^S \subset \Sigma^{P_i}$, as well as when projecting out events not in both alphabets, see (Flordal, 2001). ■

As Example 1 shows, in a truly modular setting we need to redefine controllability to handle automata with different alphabets. These new definitions will be presented in two forms. First, a definition based on automata is presented. This definition is similar to what the actual implementation looks like. Second, the controllability definition is formulated as a condition on the languages $L(P)$ and $L(S)$. This definition is similar to the standard controllability definition for equal alphabets. In the following definitions P and S are not necessarily the total supervisor and plant, respectively, but rather may be compositions of sub-supervisors and/or sub-plants.

Definition 4. (Controllability - Automata).
Let P be a plant and let S be a supervisor. No restrictions are placed on the alphabets of P and S. S is controllable with respect to P if for each string $s \in L(S||P)$ the following relation holds,

$$\sigma \in \Sigma(p) \cap \Sigma_u \cap \Sigma^S \Rightarrow \sigma \in \Sigma(q) \qquad (5)$$

where

$$p = \delta_P(q_i^P, Proj_{\Sigma^P}(s))$$
$$q = \delta_S(q_i^S, Proj_{\Sigma^S}(s))$$

■

That is, p is the state in P after observing the string s, and q is the corresponding state in S. Sometimes the term *completeness* is used to denote controllability between two automata.

The definition of controllability in terms of languages becomes more complicated with non-equal alphabets, forcing us to use inverse projection in order to be able to intersect languages.

Definition 5. (Controllability - Languages). The same assumptions as in Definition 4. S is controllable with respect to P if their languages fulfill the following relation.

$$L^{-1}(S)(\Sigma_u \cap \Sigma^P) \cap L^{-1}(P) \subseteq L^{-1}(S) \qquad (6)$$

where $L^{-1}(P)$ and $L^{-1}(S)$ are as in (3). ■

Note, that Definitions 4 and 5 are equivalent to Definition 3 when $\Sigma^S = \Sigma^P$.

Definition 6. (Configuration).
A configuration is a finite set of automata. ■

We will regard the sets of sub-plants and sub-supervisors, as well as respective sub-sets thereof, as configurations. Thus, it becomes meaningful to define what we mean by controllability of one configuration with respect to another.

Definition 7. (Controllability - Configurations).
Let $F_1 = \{F_1^1, \ldots, F_1^m\}$ and $F_2 = \{F_2^1, \ldots, F_2^n\}$ be two configurations. F_1 is said to be controllable with respect to F_2 if $F_1^1||\cdots||F_1^m$ is controllable with respect to $F_2^1||\cdots||F_2^n$. ■

If F_1 is empty then it is controllable by definition. To make the notation easier we will introduce the function $Controllable(F_1, F_2)$, which is true if F_1 is controllable with respect to F_2, and false otherwise. We also want to consider controllability for a subset, $\Sigma' \subseteq \Sigma_u$, of the uncontrollable events. This is written as $Controllable(F_1, F_2, \Sigma')$. To include this case (5) is changed to

$$\sigma \in \Sigma(q') \cap \Sigma_u \cap \Sigma^S \cap \Sigma' \Rightarrow \sigma \in \Sigma(q''). \qquad (7)$$

Note, that when implementing this function only one synchronization, if more than two automata are allowed in the synchronization, is needed.

3. CONTROLLABILITY VERIFICATION

Definition 4 together with FSC, Definition 2, can be used to implement a controllability verifier. For each new state found by the FSC algorithm, it is checked if (5) is fulfilled. If it is fulfilled for all, from the initial state, reachable states, then the (sub)system is controllable, otherwise it might not be controllable.

We assumed that the plant and the supervisor were composed of sub-plants and sub-supervisors. This modular structure will be exploited to be able to verify controllability of the global system, i.e., $P||S$ by verifying controllability for a set of sub-systems. How to construct the sub-systems will be presented later, first some more definitions. First, we will introduce a function that given a configuration returns another configuration.

Definition 8. (Event Dependence).
Let $F = \{F_1, \ldots, F_m\}$ be a configuration. Let $Dep(F, \Sigma') = \{F_i | \exists \sigma \in \Sigma' \cap \Sigma^{F_i}\}$. ■

Thus, $Dep(F, \Sigma')$, is the subset of the automata in F, that have an event from Σ' in its alphabet. Specifically, $Dep(P, \{\sigma\})$ is the set of sub-plants, i.e. all P_i's, that have σ in its alphabet.

Definition 9. (Global-, local-, and sub-states).
Let $F = \{F_1, \ldots, F_m\}$ be a configuration, and $F' = \{F_{1'}, \ldots, F_{m'}\}$ a subset of F. With the term *local-state* we will refer to a state in one of the automata in F. With the term *global-state* we will refer to a state in $F = F_1 || \cdots || F_m$. With the term *sub-state* we will refer to a state in $F' = F'_1 || \cdots || F'_{m'}$. Given a global-state $q \in F$ the function $SubState(q, F')$ returns a sub-state that only refer to automata in $F' \subseteq F$.

We are now ready to present a sufficient condition for controllability.

THEOREM 1. Given a supervisor $S = \{S_1, \ldots, S_m\}$, and a plant $P = \{P_1, \ldots, P_m\}$. S is controllable with respect to P if $\forall S_i \in S$

$$\sigma \in \Sigma_u \cap \Sigma^{S_i} \Rightarrow Controllable(\{S_i\}, Dep(P, \{\sigma\})) \quad (8)$$

Proof: Let s be an arbitrary string in $L(S||P)$. Let $q' = \delta_P(q_i^P, Proj_{\Sigma^P}(s))$ and $q'' = \delta_S(q_i^S, Proj_{\Sigma^S}(s))$. To prove the theorem we need to show (from (7))

$$\sigma \in \Sigma(q') \cap \Sigma_u \cap \Sigma^S \Rightarrow \sigma \in \Sigma(q''). \quad (9)$$

Let $\sigma \in \Sigma(q') \cap \Sigma_u \cap \Sigma^S$, i.e., σ is an uncontrollable event enabled by P and in the alphabet of S. To show that σ is enabled in S we have to show that no S_i will disable it from its local-state after observing s. From Definition 2 we can immediately rule out those S_j where $\sigma \notin \Sigma^{S_j}$. For the other S_j, let $q''_j = \delta_{S_j}(q_i^{S_j}, Proj_{\Sigma^{S_j}}(s))$, i.e. q''_j is the local-state of S^j after observing string s. From (8) we know that for all reachable states in S_j, S_j never tries to disable an uncontrollable event that is enabled by P and included in Σ^S, thus $\sigma \in \Sigma(q''_j)$. Note, that all automata in P that included σ were selected in (8). Hence, it is not possible to find a σ such that (9) does not hold. ■

In an implementation it might in some situations be faster to verify all uncontrollable events for a S_i at once, i.e. change (8) to check if

$$Controllable(\{S_i\}, Dep(P, \Sigma_u \cap \Sigma^{S_i})) \quad (10)$$

is true. Theorem 1 can be extended to also handle this case.

If (8) does not hold then there exists an event σ, and sub-states, q' and q'', such that (7) does not hold for σ. Let q be a state in $S||P$ such that $q' = SubState(q, Dep(P, \{\sigma\}))$ and $q'' = SubState(q, \{S_i\})$. If q is reachable from the initial state in $S||P$ then the system is not controllable. Thus the only possibility for the system to still be controllable is when q is not reachable from the initial state. When verifying if such q exists, we may also find a path from the initial state to q. This path is of great importance to the user who wants to use this information to get an idea of what is wrong. In case it is not possible to find a state q reachable from the initial state we have a false-alarm situation. These situations can be resolved by FSC with some of the other sub-automata, making the uncontrollable state unreachable. In our implementation, we use the following simple heuristic rule for selecting these automata. They are selected based on how many common events they have with the problematic configuration. Due to limited space we refer the reader to (Flordal, 2001). The bottom line is that the heuristics work well for those cases we have tried, but there exists pathological cases where these heuristics force FSC of all sub-supervisors and sub-plants, although these situations should be rare.

4. CONTROLLABILITY SYNTHESIS

In this section, we will use insights gained from the previous section to construct modular synthesis algorithms. There are two main advantages with the algorithms presented in this section. (i) They are computationally efficient in many practical applications. This is important since it allows us to solve large synthesis problems faster. (ii) They do not destroy modularity. First, this allows us to use a modular deadlock/non-blocking algorithm after the system has been made controllable. Second, a modular supervisor is easier to understand, than a monolithic supervisor. Third, a modular structure might represent a very large number of states with little memory. If the supervisor is implemented on a device with limited memory, e.g. a PLC, this is important.

Definition 10. (Monolithic Synthesis Algorithm).
Let $S = \{S_1, \ldots, S_m\}$ and $P = \{P_1, \ldots, P_n\}$ be two configurations where S is the supervisor and P is the plant. Let $Synt(S, P)$ be the monolithic controllability synthesis algorithm. More specifically,

$$Synt(S, P) = S_1 || \cdots || S_m || P_1 || \cdots || P_n$$

where all states that violate (4), and those states that can reach these states by a sequence of uncontrollable events, are removed. ■

The synthesized supervisor is more restricting, in the sense of disabling events, than the original supervisor. The previously outlined synthesis algorithm will be our basic synthesis algorithm. We will now introduce the modular synthesis algorithms. First, we present a straightforward modular synthesis algorithm, which unfortunately, may result in a non-maximally permissive supervisor.

THEOREM 2. Extend the set of supervisors according to the following rules. For each S_i such that

$$\neg Controllable(\{S_i\}, Dep(P, \Sigma_u \cap \Sigma^{S_i})), \quad (11)$$

synthesize a supervisor S'_i such that

$$S'_i = Synt(\{S_i\}, Dep(P, \Sigma_u \cap \Sigma^{S_i})). \quad (12)$$

Then extend S by adding all the newly constructed supervisors, resulting in a configuration, of sub-supervisors, S'. Then S' is controllable with respect to P.

Proof: To show this theorem we will rely on (8). First examine S_i together with S_i'. Note that $\Sigma^{S_i} \subseteq \Sigma^{S_i'}$. Now repeat the check for controllability according to (8). Let $q''' = \delta_{S_i'}(q^{S_i'}, Proj_{\Sigma^{S_i'}}(s))$. (4) can now be modified to look like.

$$\sigma \in \Sigma(q') \cap \Sigma_u \cap \Sigma^S \cap \Sigma(q''') \Rightarrow \sigma \in \Sigma(q'') \quad (13)$$

From this we can conclude that the system is controllable. ∎

Note that the synthesis algorithm might remove substates that were not reachable from the initial state. Removing such states are unnecessary but perfectly valid. The problem with Theorem 2 is that it does not give a maximally permissive solution. This is due to that the sub-plants in $Dep(P, \Sigma_u \cap \Sigma^{S_i})$ might have uncontrollable events not included in Σ^{S_i}. Call these events Σ'. The synthesis algorithm has to follow uncontrollable events backward from the initially uncontrollable states. Since not all sub-plants that have Σ' in its alphabet are included in the synthesis, uncontrollable events might be removed when synchronized with these automata. Fortunately, this is fixed by a relatively simple modification.

PROPOSITION 1. Extend the set of supervisors according to the following rules. For each S_i such that

$$Controllable(\{S_i\}, Dep(P, \Sigma_u \cap \Sigma^{S_i})) \quad (14)$$

is false. First initialize $\Sigma^{(1)}$ and $P^{(1)}$ to

$$\begin{aligned} \Sigma^{(1)} &= \Sigma_u \cap \Sigma^{S_i} \\ P^{(1)} &= Dep(P, \Sigma^{(1)}). \end{aligned} \quad (15)$$

Then repeat the following statements until $\Sigma^{(n+1)} = \Sigma^{(n)}$.

$$\begin{aligned} \Sigma^{(n+1)} &= \Sigma^{(n)} \cup (\Sigma^{(P^{(n)})} \cap \Sigma_u) \\ P^{(n+1)} &= Dep(P^{(n)}, \Sigma^{(n)}) \end{aligned} \quad (16)$$

This iteration will always terminate in a finite number of steps, say k. k will always be less than the number of automata in P.

THEOREM 3. Synthesize a supervisor S_i' such that

$$S_i' = Synt(\{S_i\}, P^{(k)}). \quad (17)$$

Then extend S by adding all the newly constructed supervisors. Call the new supervisor set S'. Then S' is controllable with respect to P. Note, k is equal to the number of steps to terminate in the previous proposition.

Proof: This proof is similar to the proof of Theorem 2. The major difference is that each new sub-supervisor is synthesized from a larger set of sub-plants. ∎

PROPOSITION 2. Theorem 3 will result in a maximally permissive supervisor.

Proof: To show this we will make an argument for each sub-state that were removed by the synthesis algorithm. We have two alternatives, to remove or to keep the forbidden state. In Theorem 3 the set of sub-plants were extended until there did not exist a sub-plant outside the set that had an uncontrollable event in common with the sub-plants in the set. This property implies that no uncontrollable event could be prevented from occurring by FSC with another sub-plant. From this we conclude that all sub-states that could uncontrollably reach one of the initially forbidden states must also be forbidden. Since we start the supervisor construction by synchronizing with the plant we know that removing uncontrollable states in the supervisor candidate is equivalent to removing uncontrollable strings from the maximally permissive language as generated by the standard Ramadge-Wonham algorithms. This gurantees the maximally permissiveness of the synthesized supervisor. ∎

5. LANGUAGE INCLUSION CHECK

Language inclusion is a general problem that has been studied extensively in the computer science literature. In verification applications language inclusion can be used to check if an implementation contains a specified behavior. Another application is to check for language equality, where $L_1 \subseteq L_2 \wedge L_2 \subseteq L_1 \Rightarrow L_1 = L_2$. In this section, we will show that we can use the modular controllability verification algorithm to check for language inclusion of prefixed-closed languages. Assume that we want to check if $L_1 \subseteq L_2$.

THEOREM 4. Let L_1 and L_2 be two prefixed-closed regular languages, i.e. A_1 and A_2 can be constructed such that $L_1 = L(A_1)$ and $L_2 = L(A_2)$. We can safely assume that $\Sigma^{A_1} \subseteq \Sigma^{A_2}$. Then it holds that $L^{-1}(A_2)\Sigma^{A_1} \cap L^{-1}(A_1) \subseteq L^{-1}(A_2) \Leftrightarrow L(A_1) \subseteq L(A_2)$

Proof: We are to show that $L^{-1}(A_2)\Sigma^{A_1} \cap L^{-1}(A_1) \subseteq L^{-1}(A_2) \Leftrightarrow L(A_1) \subseteq L(A_2)$ assuming that $\Sigma^{A_1} \subseteq \Sigma^{A_2}$. When this holds then $L^{-1}(A_2) = L(A_2)$ since no events remain to be inserted. Thus we can rewrite the expression as $L(A_2)\Sigma^{A_1} \cap L^{-1}(A_1) \subseteq L(A_2) \Leftrightarrow L(A_1) \subseteq L(A_2)$. Let us also note that the right-hand side is equivalent to $L(A_2)(\Sigma^{A_1})^* \cap L^{-1}(A_1) \subseteq L(A_2)$.

Assume first that the right-hand side holds, that is that $L(A_2)(\Sigma^{A_1})^* \cap L^{-1}(A_1) \subseteq L(A_2)$. Since $L(A_2)$ is prefix-closed $\varepsilon \in L(A_2)$ and in that case the expression becomes $(\Sigma^{A_1})^* \cap L^{-1}(A_1) \subseteq L(A_2)$. Now, when intersecting $(\Sigma^{A_1})^*$ with $L^{-1}(A_1)$, only the Σ^{A_1} events are significant, and we have that $(\Sigma^{A_1})^* \cap L^{-1}(A_1) = L(A_1)$. Obviously, this means that $L(A_2)(\Sigma^{A_1})^* \cap L^{-1}(A_1) \subseteq L(A_2) \Rightarrow L(A_1) \subseteq L(A_2)$.

Assume now that $L(A_1) \subseteq L(A_2)$ and pick a string $s\sigma \in L(A_1) \cap L(A_2)$. Since $s\sigma \in L(A_1)$ it holds that $s \in L(A_1)$ and that $\sigma \in \Sigma^{A_1}$. It also means that $s\sigma \in L^{-1}(A_1)$. That $s\sigma \in L(A_2)$ means that $s \in L(A_2)$ and therefore $s\sigma \in L(A_2)\Sigma^{A_1}$. Thus, $s\sigma \in L(A_2)\Sigma^{A_1} \cap L^{-1}(A_1)$ and therefore it holds that $L(A_1) \subseteq L(A_2) \Rightarrow L(A_2)\Sigma^{A_1} \cap L^{-1}(A_1) \subseteq L(A_2)$. ∎

The intuition behind the theorem is to consider A_1 to be the plant and A_2 to be the supervisor. Since all events in A_1, the plant, are uncontrollable the supervisor, A_2, is never allowed to prevent any of them from occurring. Since $\Sigma^{A_1} \subseteq \Sigma^{A_2}$, the only way this can happen is if A_2 can follow all events generated by A_1.

6. EXAMPLES

The presented results have been implemented in `Supremica`, a tool for supervisory control [1]. To show

[1] A website with more information about `Supremica` can be found at `www.supremica.org`.

Name	States
basic_c	1319
basic_u	2005

Name	States
basic_c	258
basic_u	280

Table 1. Left: Verification examples. Right: Synthesis examples.

that the results are usable on real-world examples we have a variant of the central-locking system used in the Korsys project. The central-locking examples are based on a example distributed with Valid, by Siemens Corporate Research. Due to limited space we cannot present the details of the example here but instead show some numbers that give an indication to the reader about the efficiency of the presented algorithms. The used example consists of 53 automata, 18 sub-plants and 35 sub-supervisors. All sub-plants are two-state automata that only communicate through the supervisor, i.e. their alphabets are disjunctive. The sub-supervisors are of different sizes, from 1 to 27 states. The system has approximately $7.5 \cdot 10^8$ *reachable* states.

We have two versions of the problem, one that is controllable and one that is not, these examples will be called *basic_c* and *basic_u*, respectively. In Table 1, left table, we show how many states that the verification algorithm examines during verification. In Table 1, right table, the total number of states in the new sub-supervisors are presented. Note, in *basic_c* it is not necessary to do any synthesis since the system is controllable before synthesis, but it is always safe to do a synthesis. Instead of having to examine all $7.5 \cdot 10^8$ states we only had to examine a few hundreds to a few thousand states. Somewhat surprisingly, synthesis seems to be cheaper than verification. The reason for this is that verification needs to verify if a found sub-state is reachable from the initial state, while the synthesis can safely proceed with the synthesis. The necessary time for doing synthesis and verification of this example in `Supremica` on a standard desktop computer is well below one second. Even though this is a single application, we are encouraged by the results, but it is necessary to run the algorithms on other large examples before drawing any definitive conclusions. To make it harder for the algorithms we have modified the example by removing modularity, which is easily done by pre-synchronizing sub-models. Our preliminary results show that verification seems to work efficiently in most cases. As expected, optimal synthesis is sensitive to the degree of interaction between different sub-plants.

7. CONCLUSIONS

We have shown how a modular structure of the plant and the supervisor can be exploited to get efficient algorithms for verifying and synthesizing controllable supervisors. Limited usage of both time and space is of great practical importance when dealing with practical applications that usually have a very large state space. In industry, a PLC is very common device for logical controllers. Thus, it is if importance not to synthesize a monolithic supervisor, but instead synthesize a number of supervisors that when interacting with each other accomplish the same result as the monolithic supervisor. Potentially, implementing a set of interacting supervisors instead of one monolithic supervisor requires much less memory. The presented algorithms has been verified on a central-locking example. Both verification and synthesis could be performed with a standard desktop computer within a couple of seconds. We believe that modular algorithms could be combined with BDD-approaches in order to handle problems with little modular structure or when the sub-problems that the modular algorithms give rise too become to large for brute-force approaches. FSC is a special case of prioritized synchronous composition. Extensions of this work to prioritized synchronous composition (PSC), (Heymann, 1990), is presented in (Flordal, 2001). PSC allows the use of broadcast synchronization that is the mechanism used in Statechart and State diagrams in the Unified Model Language (UML). We are currently working on extending the algorithms to handle arbitrary forbidden states and sub-states, we are also working on algorithms for non-blocking verification and synthesis.

REFERENCES

Brandin, B., R. Malik and P. Dietrich (2000). Incremental system verification and synthesis of minimally restrictive behaviors. In: *Proc.of the 2000 American Control Conference*. Chicago, USA. pp. 4056–4061.

Bryant, R. (1992). Symbolic manipulation with ordered binary decision diagrams. *ACM Computing Surveys 24* **24**, 293–318.

Corbett, James C. (1996). Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering* **22**(3), 161–180.

Flordal, H. (2001). Modular controllability verification and synthesis of discrete event systems. Technical report. Department of Signals and Systems, Chalmers University of Technology.

Gohari, P. and W.M. Wonham (2000). On the complexity of supervisory control design in the RW framework. *IEEE Transactions on Systems, Man, and Cybernetics* **30**(5), 643–652.

Heymann, M. (1990). Concurrency and discrete event control. *IEEE Control Systems Magazine* **10**(4), 103–112.

Hoare, C.A.R. (1985). *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science.

Hoffmann, G. and H. Wong-Toi (1992). Symbolic synthesis of supervisory controllers. In: *Proc. of the 1992 American Control Conference*. Chicago, IL. pp. 2789–2793.

Hopcroft, J.E. and J.D. Ullman (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Series in Computer Science, Addison-Wesley.

Ramadge, P. and W.M. Wonham (1989). The control of discrete event systems. *Proc. IEEE* **77**(1), 81–98.

Tronci, E. (1997). On computing optimal controllers for finite state systems. In: *Proc. of 36th IEEE CDC*. USA.

Wong, K. and W.M. Wonham (1998). Modular control and coordination of discrete-event systems. *Journal of Discrete Event Dynamic Systems: Theory and Applications* **8**(3), 247–297.

Wonham, W.M. and P.J. Ramadge (1988). Modular supervisory control of discrete event systems. *Mathematics of Control Signals and Systems* **1**(1), 13–30.

Zhang, Z. and W.M. Wonham (2001). STCT: An efficient algorithm for supervisory control design. In: *Proc. Symposium on the Supervisory Control of Discrete Event Systems. A Satellite Workshop of 13th Conference on Computer Aided Verification*. Paris, France. pp. 82–93.