Modular Supervisory Synthesis for Unknown Plant Models Using Active Learning

Citation for the original published paper (version of record):
Hagebring, F., Farooqui, A., Fabian, M. (2020)
Modular Supervisory Synthesis for Unknown Plant Models Using Active Learning
IFAC-PapersOnLine, 53(4): 324-330
http://dx.doi.org/10.1016/j.ifacol.2021.04.032

(article starts on next page)

Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnLine 53-4 (2020) 324–330

Modular Supervisory Synthesis for Unknown Plant Models Using Active Learning [⋆]

Fredrik Hagebring, Ashfaq Farooqui, Martin Fabian

Department of Electrical Engineering, Chalmers University of Technology, Göteborg, Sweden 412 96
(e-mail: {ashfaqf, fredrik.hagebring, fabian}@chalmers.se)

Abstract: This paper proposes an approach to synthesize a modular discrete-event supervisor to control a plant, the behavior model of which is unknown, so as to satisfy given specifications. To this end, the Modular Supervisor Learner (MSL) is presented that based on the known specifications and the structure of the system defines the configuration of the supervisors to learn. Then, by actively querying the simulation and interacting with the specification it explores the state-space of the system to learn a set of maximally permissive controllable supervisors.

The *Supervisory Control Theory* (SCT) (Ramadge and Wonham, 1989) provides a general framework to *synthesize supervisors* for Discrete-Event Systems (DES). These DES are models of systems that at each time instant occupy a discrete state, and perform state transition on the occurrence of events. Examples of such systems are manufacturing systems, communication networks, and embedded systems. Given a DES modeling all possible behavior known as a *plant*, a *supervisor* can be synthesized that can control the plant in order to satisfy given *specifications*. However, modeling large complex systems is a challenging task that requires skill, in-depth knowledge of the system, and creativity. Manually defining the behaviour of the plant model is an error prone task; incorrect or incomplete models are misleading, and can unnecessarily complicate the development process. Hence, assuming access to a correct plant model can be limiting.

Though discrete-event models have many advantages, they suffer from one big and fundamental problem – state-space explosion. Here, the discrete models very quickly grow in size making it difficult to store and compute using them. One technique is to have *modular* models that when composed describe the complete behavior. These, modular models, can then be used for computation and synthesis.

Simulation technologies have gained attention in many areas of automation, and hence simulation-based development has become well accepted. In this mode of development, the intended system is first created in a 3D simulation environment where it can be tested and improved upon before constructing the physical system. These simulations implicitly contain within them a behavior of the plant, though this behavior is not accessible in a usable format for supervisory synthesis algorithms. However, there exist active learning algorithms that can be used to infer a discrete model of the plant from a simulation.

Active learning algorithms are a class of machine learning algorithms that aim to deduce a discrete-event model describing the behavior of a system. Active automata learning has been successfully applied to learn and verify communication protocols using Mealy machines (Steffen et al., 2011; Jonsson, 2011); to obtain the formal models of biometric passports (Aarts et al., 2010) and bank cards (Aarts et al., 2013).

Within the SCT community, there has been work on applying active learning algorithms mainly by language based algorithms. That is, they focus on the sequences of events that can be performed. Zhang et al. (2018) look at synthesizing a controller when a plant model is known, and the specification model is not an automaton; Dai and Lin (2014) look at learning decentralized supervisors; Hiraishi (1999) presents a synthesis approach for concurrent systems; and Yang et al. (1995) propose an algorithm to learn optimal controllers. However, to the best of the authors' knowledge, despite most cyber-physical systems being able to employ a state-based formulation, no state-based active learning approaches exist within the automata learning community. Specifically, in the current setting, using a simulator makes it possible to access the state of the system. A *state*, in this paper, is defined by the values assigned to a set of variables, each of which has its own domain. Unique combinations of the values assigned to these variables make up the different states.

Previously in Farooqui et al. (2020), it was shown how a modular plant model could be learned given a definition of its modular structure and interacting with a simulation of the system. It would, however, be useful to directly synthesize a supervisor from the simulation, instead of first learning a plant model and then, by using additional tools, synthesizing a supervisor. In this paper, we extend the idea and show that it is possible to learn a modular maximally permissive controllable supervisor (Åkesson et al., 2002) given the specifications of the system along with a simulation and a modular structure definition, without having first to learn a plant model. To this end, we first present how the modular structure and specifications are used to generate a configuration of supervisors to learn.

[⋆] Work supported by the Swedish Research Council (VR) project SyTeC, the Chalmers Production Area of Advance, and by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation.

10.1016/j.ifacol.2021.04.032

Next, by tracking the state in the supervisor, we present an approach to identify and forbid states that are not controllable. This is followed by the Modular Supervisor Learner algorithm that uses the configuration of supervisors and the controllability algorithm to learn a set of supervisors for the given simulation and specification. Additionally, the specific implementation of MSL proposed in this paper is formulated such that it can employ distributed computation to improve scalability. That is, the search is divided into small independent operations that can be distributed over multiple processors or computers.

In the following, Section 1 introduces the relevant notation. Section 2 presents the modeling framework that enables learning a modular discrete supervisor. Section 3 then presents the Modular Supervisor Learner followed by Section 4 that discusses an example to show the different components before concluding in Section 5.

1. PREREQUISITES

Let Σ, called an *alphabet*, be a finite set of *events*. Partition Σ into the two sets Σ_c, the set of *controllable* events, and Σ_u, the set of *uncontrollable* events. A *language* $\mathcal{L} \subseteq \Sigma^*$ is a set of strings over Σ.

1.1 Deterministic Finite State Automaton (DFA)

Let I be a totally ordered index set. Let $V = \{ v_i \mid i \in I \}$ be a set of variables such that each variable is indexed by one element of the indexing set, that is $|V| = |I|$. Let $V' = \{ v_{i'} \in V \mid i' \in I' \subseteq I \}$ be a subset of variables of V respecting the indexing order, with $|V'| = |I'|$. Each variable v_i has a (finite discrete) domain D_i, and let the domain of V be $D_V = D_{i_1} \times D_{i_2} \times \cdots \times D_{i_{|I|}}$, where the indices $i_j \in I$ (for $j \in 1..|I|$) respect the indexing order. In the same way, the domain of V' is given as the Cartesian product over the domains of the variables of V' in the order defined by the indexing subset $I' \subseteq I$.

Let a *state* q be defined as an element of the domain of V, that is, $q \in D_V$. Thus, a state q is a valuation of the variables of V. Likewise, let a *sub-state* q' be a valuation over V'. Define the *projection* of a state $q \in D_V$ onto a sub-state $q' \in D_{V'}$ as $P_{V'}(q) = q'$, such that all $v_i \in q'$ have the same valuation as in q. For a set of states Q, let $P_{V'}(Q)$ denote the projection of each element in Q on V'

Definition 1. (DFA). A (*deterministic finite*) *automaton* is defined as a 6-tuple $\langle Q, \Sigma, \delta, q_0, Q_m, Q_f \rangle$, where:

- Q is the set of *states*
- Σ is the *alphabet* containing the events
- $\delta : Q \times \Sigma \to Q$ is the partial *transition function*
- $q_0 \in Q$ is the *initial state* of the system
- $Q_m \subseteq Q$ is the set of *marked states*
- $Q_f \subseteq Q$ is the set of *forbidden states*

Given an automaton G, its *language* $\mathcal{L}(G) = \{s \in \Sigma^* \mid \delta(q_0, s) \text{ is defined}\}$ is the set of all sequences of events, *strings*, defined from G's initial state. A state $q \in Q$ such that $\delta(q_0, s) = q$ for some $s \in \mathcal{L}(G)$ is said to be *reachable*. The *marked* language $\mathcal{L}_m(G) = \{s \in \mathcal{L}(G) \mid \delta(q_0, s) \in Q_m\}$ is the set of strings that reach marked states. A state $q \in Q$ such that $\delta(q, t) \in Q_m$ for some $st \in \mathcal{L}_m(G)$ is said to be *coreachable.*

1.2 Supervisory Control Theory

The supervisory control theory (Ramadge and Wonham, 1989) provides a method to synthesize a supervisor S, given a plant G and a specification K such that the behavior of G dynamically restricted by S always fulfills K. The closed-loop system where S controls G can be modeled by *synchronous composition* (Cassandras and Lafortune, 2009) $G \,||\, S$, where events are generated by G only if enabled by S.

However, S does not have full control over the event generation of G, only the controllable events Σ_c can be restricted by S, while the uncontrollable events Σ_u are not susceptible to influence by S. This is captured by the notion of *controllability.*

Definition 2. (Controllability). For a plant G and uncontrollable events Σ_u, a supervisor S is said to be *controllable* if $\forall s \in \mathcal{L}(G \,||\, S), \sigma \in \Sigma_u : s\sigma \in \mathcal{L}(G) \implies s\sigma \in \mathcal{L}(G \,||\, S)$.

A controllable S is able to restrict G to a subset of the state-space that is considered safe. However, while restricting bad states from being reached, S must allow marked states to be reached. This is captured by the notion of *non-blocking*

Definition 3. (Non-blocking). For a plant G, a supervisor S is said to be *non-blocking* if in the closed-loop system $G \,||\, S$ from any reachable state some marked state can be reached, that is, $\mathcal{L}(G \,||\, S) = \overline{\mathcal{L}_m(G \,||\, S)}$.

It is known (Cassandras and Lafortune, 2009) that for a given G and K, there exists a *maximally permissive* controllable and non-blocking S, such that $\mathcal{L}(G \,||\, S)$ is the unique supremal controllable and non-blocking sublanguage $\sup \mathcal{C}(G \,||\, S) \subseteq \mathcal{L}(K)$.

Given a plant $G = \{G_1, G_2, \ldots, G_j\}$, let $K = \{K_1, K_2, \ldots, K_i\}$ be a set of automata describing the *specifications* of the system. A set of supervisors $S = \{S_1, S_2, \ldots, S_i\}$ can then be calculated for each specification K_i in such a way (Åkesson et al., 2002) that the synchronous composition of the supervisors results in a maximally permissive controllable supervisor.

2. THE MODELING FRAMEWORK

Farooqui et al. (2020) showed how a modular discrete event plant model can be learned from a simulation. In some cases, however, it is beneficial to learn a supervisor that satisfies given specifications when controlling the plant. To this end, we modify the previous algorithm to interact with a simulation of the plant for which a supervisor is to be learned, and actively query the simulation in a smart way to learn what states are reachable from the initial state, and also to check if they are controllable with respect to the given specification. To be able to do this modularly, the algorithm uses a *plant structure hypothesis* (PSH) to split the learning into a set of modules. This section highlights the inputs required by the algorithm.

2.1 The Simulation

Simulations provide several advantages in comparison to using a real physical system. Unlike the real system, the simulation can run faster than real-time, even multiple instances in parallel, thereby speeding up the learning process. Dangerous collisions and unforeseen events are avoided and confined to the simulation, providing a safe learning environment. Additionally, the financial investment needed, once a simulation is obtained, relates to obtaining powerful computers – which in today's world is relatively cheap.

It is important to highlight the requirements of the plant simulation. Firstly, it should be possible to, using an interface, execute an event, or a string of events. The plant simulation has at each time a set of enabled events that can be executed to perform specific actions. When these actions are performed the state of the plant is updated resulting in another set of enabled events. Hence, a string of events can be executed taking the plant from one state to another state. In case an event is requested to be executed that is not enabled by the plant in a particular state, the simulation should reply with an error message.

Secondly, it should be possible to observe and set the state of the system. The state is here given by the values of a set of variables in the simulation; these could, for example in a manufacturing context, be the status of sensors, actuators, and product position.

For this purpose, define a function *getNextState* that takes as input an assignment to the variables and an event to be executed. The output of this function is the resulting variable assignment when the given event is executed in the simulator from the given state, or the aforementioned error message.

It is important to note here that we have presented the simulator to be a discrete system. In most cases, these are not discrete, neither in time nor variable values. However, we assume that the simulation can be discretized in order to learn a discrete model.

2.2 Plant Structure Hypothesis

The PSH can be considered the core of the modular learning technique proposed in this paper. It can be viewed as a high-level meta-model that defines the modular structure of the system. The modular structure refers to a division of the complete plant behavior as separate modules, usually, but not necessarily, representing the separate subsystems that together define the behavior of the plant. This can then be exploited by the MSL to divide the learned information into separate modules and to reduce the search space, ultimately mitigating the state-space explosion problem.

The PSH is defined using three pieces of information. Firstly, a set M provides a unique name for each module in the system. The cardinality of M defines the number of modules in the system. Secondly, a mapping E, called *event mapping*, defines which events of the global alphabet $\Sigma = \Sigma_c \cup \Sigma_u$ belong to which module. Thus, $E(m) \subseteq \Sigma$ is the local alphabet of the module $m \in M$. That an event is part of an event mapping implies that the corresponding module is involved in executing the event and, furthermore, that it requires this event to be represented as transitions in the automaton of the module.

A *state mapping* S defines the relation between the modules and the set of variables in the simulator. That is, for all $m \in M$, $S(m) \subseteq V$ contains those variables that either affect or are affected by events in the module. Variables that are not part of a specific state mapping can be ignored by that module. Thus, for a given module $m \in M$, two global states $q_i, q_j \in D_V$ are equal within the module if their projections onto $S(m)$ are equal, that is, if $P_{S(m)}(q_i) = P_{S(m)}(q_j)$. Hereinafter the projection of a state q onto a state mapping $S(m)$ is denoted $P_m(q)$.

Definition 4. (PSH). Formally, the PSH is a 3-tuple $H = \langle M, E, S \rangle$, where:

- M is a set of identifiers for the modules;
- $E : M \to 2^{\Sigma}$ is the *event mapping*;
- $S : M \to 2^{V}$ is the *state mapping*;

For any given system there may exist multiple approaches to define the PSH. To guarantee that the MSL explores the full system, however, requires a *valid PSH*. A valid PSH is one where the union of all event mappings encompass the whole alphabet Σ and the union of all state mappings encompass the whole of V. That is, each event $\sigma \in \Sigma$ and variable $v \in V$ must be included in the event and state mapping of at least one module. In addition, the PSH must conform to the simulation to agree on which module has what events and what variables.

Further details about creating a PSH for a system along with an example can be found in Farooqui et al. (2020).

2.3 Specifications

Along with the possibility to interact with the simulation and a PSH, the MSL requires specifications to calculate supervisors. A specification is an automaton whose language defines the intended behavior of the system. Additionally, specifying the accepted states and forbidden states is done by adding these states in the specification automaton to the appropriate sets.

The input to the algorithm is a set of specifications $K = \{ K_1, K_2, \dots K_n \}$, each with its own alphabet, Σ_{K_i}.

3. THE MODULAR SUPERVISORY LEARNER

This section describes the MSL and its working. Before presenting the algorithm we will define how the supervisors to be learned are calculated from the PSH. Followed by explaining how controllability is tracked in our approach.

3.1 Calculating the modules

It is known (Åkesson et al., 2002), that a controllable modular supervisor can be calculated by selecting for each specification $K_i \in K$, all plant components $G_j \in G$ such that $\Sigma_{K_i} \cap \Sigma_{G_j} \cap \Sigma_u \neq \emptyset$ and performing monolithic synthesis on this sub-system. To guarantee a maximally permissive modular supervisor, also all plant components $G_k \in G$ such that $\Sigma_{G_k} \cap \Sigma_{G_j} \cap \Sigma_u \neq \emptyset$ for each G_j previously selected have to be included. This selection of new plant components sharing uncontrollable events with the already selected ones has to be iterated until a fixpoint. However, these latter G_k plant components can be included incrementally, as needed (Åkesson et al., 2002), to lessen the risk of including the monolithic plant.

In the current case, where plant components are not available, the information about the plant structure available in the PSH is used to select the subsystems.

Define an *event dependence* function $\text{Dep}(M, \Sigma')$ that, given a set of modules M and a set of events Σ', selects the modules that have any events in Σ' in their event mapping.

Definition 5. (Event Dependence). Given an alphabet Σ' and a set $M = \{m_1, m_2, \ldots, m_i\}$ of modules, let

$$\text{Dep}(M, \Sigma') = \{\, m_i \in M \mid E(m_i) \cap \Sigma' \neq \emptyset \,\}$$

The modules are then selected using the following rules. Let M contain the set of plant modules defined in the PSH. Then for each specification component $K_i \in K$:

First initialize

$$\Sigma^{(1)} = \Sigma_u \cap \Sigma_{K_i}$$
$$M^{(1)}_{K_i} = \text{Dep}(M, \Sigma^{(1)})$$

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

$$\Sigma^{(n+1)} = \Sigma^{(n)} \cup (\Sigma_{M^{(n)}_{K_i}} \cap \Sigma_u)$$
$$M^{(n+1)}_{K_i} = \text{Dep}(M^{(n)}_{K_i}, \Sigma^{(n)})$$

The resulting set M_{K_i} is the set of plant modules that are related, directly and indirectly, to specification K_i through uncontrollable events. The state mapping and event mapping for these modules need to be aggregated as well:

$$E_{K_i} = \Sigma_{K_i} \cup \bigcup_{m \in M_{K_i}} E(m), \text{ and } S_{K_i} = \bigcup_{m \in M_{K_i}} S(m).$$

The above steps are repeated for each specification $K_i \in K$ creating the set $M_K = \{\, M_{K_1}, M_{K_2}, \ldots M_{K_j} \,\}$ which is the configuration of supervisors to learn. Similarly, $E_K = \{\, E_{K_1}, E_{K_2}, \ldots E_{K_j} \,\}$ and $S_K = \{\, S_{K_1}, S_{K_2}, \ldots S_{K_j} \,\}$. Additionally, it is possible that some modules of M are not included in any of the module groupings; define $M_G = M \setminus \bigcup_{K_i \in K} M_{K_i}$. These modules need to be learned as plant modules alongside the supervisors.

A new structure, similar to the PSH, is constructed consisting of $\langle M', E', S', K' \rangle$. Where, $M' = M_K \cup M_G$ is a set of identifiers of the modules to learn. Hence the total number of modules to learn is given by the cardinality of $|M'| = |M_K| + |M_G|$. E' and S' are functions that map the modules to be learned in M' to their respective alphabets and variable sets with signatures: $E' : M' \longrightarrow 2^{\Sigma}$ and $S' : M' \longrightarrow 2^{V}$, and are defined as:

$$E'(m) = \begin{cases} E_{K_i}, & \text{if } m \in M_K \text{ and } m = M_{K_i} \\ E(m), & \text{if } m \in M_G \end{cases}$$

$$S'(m) = \begin{cases} S_{K_i}, & \text{if } m \in M_K \text{ and } m = M_{K_i} \\ S(m), & \text{if } m \in M_G \end{cases}$$

Also, K', is a partial function that tracks the specification associated with each module in M', if it exists, and has the signature $K' : M' \longrightarrow K$, where K is the set of specifications:

$$K'(m) = \begin{cases} K_i, & \text{where } m = M_{K_i} \\ \textit{undefined}, & \text{otherwise.} \end{cases}$$

3.2 Checking Controllability

Controllability for any given string can be checked using Definition 2. However, in the present case, as will be shown later, the algorithm works by exploring the simulated system one state at a time. In doing so, there explicitly does not exist a path to any state; only a set of states and transitions Hence, to check controllability the algorithm requires to track states reached in the specification.

To do this, each state in the specification is given a unique name. Then, we add one state variable per specification to the state vector, the valuation of which defines the current state of the corresponding specification:

$$V = V \cup \bigcup_{\forall K_i \in K} var_{K_i}.$$

This variable is updated when a transition is fired in the specification. Similarly, the state mapping S' is updated to contain also the variable corresponding to its specification, $S_{K_i} = S_{K_i} \cup \{var_{K_i}\}$. Furthermore, those states in the specification that are marked as forbidden or accepting can be determined using the valuation of the added variable.

Now, using the above defined setting, it is possible to check for controllability. A state valuation is controllable if it allows an uncontrollable event $\sigma_u \in \Sigma_{K_i} \cap \Sigma_u$ to occur in the simulation and σ_u is defined from the state corresponding to the value of var_{K_i}.

When a state is not controllable, it and all states from which it can be reached by uncontrollable events need to be added to the set of forbidden states. Pseudo-code for this is given in Algorithm 1.

```
begin
    Procedure IsControllable($\langle q, \sigma, q' \rangle$, $K_i$)
        if $\sigma \in \Sigma_u$ then
            if the state in specification $K_i$ given by the
              projection $P_{\{\, var_{K_i} \,\}}(q)$ allows $\sigma$ then
                Return true
            else
                $Q_f = Q_f \cup \{q\}$
                -run processUncontrollableState($q$)
                -Return false
            end
        else
            Return true
        end
    Procedure processUncontrollableState(q)
        - $T = \{\langle q', \sigma_u, q \rangle \in \delta \mid \sigma_u \in \Sigma_u\}$
        - $\forall t \in T$, $Q_f = Q_f \cup \{t.q'\}$
        - processUncontrollableState($t.q'$)
end
```

Algorithm 1: Algorithm to check controllability and process uncontrollable states

3.3 The MSL algorithm

The MSL algorithm consists of three procedures, *Main*, *Explorer*, and *ModuleBuilder*, see Algorithm 2, and is constructed so that the latter two procedures execute concurrently. The functioning of the algorithm is similar to what is presented in Farooqui et al. (2020) with the main addition being the check for controllability and selection of plant components to include in order to learn the supervisor. The algorithm is initiated by *Main* that launches the *Explorer* and one instance of *ModuleBuilder* for each supervisor and plant module that is to be learned, calculated in accordance to Section 3.1. The *Explorer* is responsible for exploring the new states, and the *ModuleBuilder* keeps track of the module as it is learned.

The *Explorer* maintains a queue of states that need to be explored, terminating the algorithm when the queue is empty. The learning is initiated by adding an initial state to the queue, which becomes the starting state of the search. For each element in the queue, the *Explorer* checks if an event defined in the alphabet can be executed. This is achieved using the simulator interface. If a transition is possible, the *Explorer* broadcasts the current state (q), the event (σ) and the state reached (q') to all the *ModuleBuilder*s.

The *ModuleBuilder* tracks the learning of each module as an automaton. This is done by maintaining a set $Q(m)$ containing the states of the module, and a transition function $T(m) : Q(m) \times E(m) \to Q(m)$. The *ModuleBuilder*, on receiving a broadcast of a transition, checks if this transition is controllable as defined in Section 3.2. If a transition is found to be uncontrollable, then the source state needs to be added to the list of forbidden states $Q_f(m)$ followed by processing of all uncontrollable states according to Algorithm 1. Next, the *ModuleBuilder* finds all the variables V' that have been updated in the reached state. The *ModuleBuilder* then evaluates if the received transition is of interest to the particular module and continues following the same procedure as Farooqui et al. (2020). Furthermore, specifications define the markings on the states. If a reached state is marked as accepting or forbidden in the specification, then this state is also added to $Q_m(m)$ or $Q_f(m)$, respectively.

Once the transition is processed the *ModuleBuilder* waits for further broadcasts. The algorithm terminates when all modules are waiting for broadcasts, and the global exploration queue is empty. Each *ModuleBuilder* can now construct and return an automaton based on $Q(m), T(m)$, $Q_m(m)$, and $Q_f(m)$.

The result of this algorithm is a set of supervisors and plant models, with one supervisor for each specification that shares uncontrollable events with one or more modules defined in the PSH. Modules in the PSH that are not in any supervisor module are learned as plant models.

```
Input: An interface to a simulator defined in Section 2.1, the
       initial state, $q_0$, a PSH $H = \langle M, E, S \rangle$, and a set of
       specifications $K$.
begin
    Procedure Main
        - $\langle M', E', S', K' \rangle \longleftarrow$ calculate modules according to
          Section 3.1
        - $Q_G \longleftarrow \{q_0\}$ - Run $\longleftarrow$ true
        foreach $m \in M'$ do
            - run ModuleBuilder($m$)
        end
        - run Explorer()
        - Wait until $Q_G$ is empty AND all ModuleBuilders are
          waiting
        - Run $\longleftarrow$ false
        - Collect $Sup_m$ returned by ModuleBuilders into a set
          Sup
        return Sup
    Procedure Explorer
        while Run do
            for $q \in Q_G$ do
                for $\sigma \in \Sigma$ do
                    - find $q'$ by executing $\sigma$ from state $q$ in the
                      simulator
                    - Broadcast the transition $\langle q, \sigma, q' \rangle$
                end
                - Remove $q$ from $Q_G$
            end
        end
    Procedure ModuleBuilder(m)
        $Q(m) \longleftarrow \{P_m(q_0)\}$, $T(m) \longleftarrow \emptyset$, $Q_m(m) \longleftarrow \emptyset$,
          $Q_f(m) \longleftarrow \emptyset$, $K_i \longleftarrow K'(m)$
        while Run do
            if $\langle q, \sigma, q' \rangle$ received then
                controllable $\longleftarrow$ if $K_i$ is defined then
                  IsControllable($\langle q, \sigma, q' \rangle$, $K_i$) else true
                $V' \longleftarrow \{v \in V \mid q(v) \neq q'(v)\}$
                if $\sigma \in E'(m)$ or $S'(m) \cap V' \neq \emptyset$ then
                    $\sigma' \longleftarrow$ if $\sigma \in E'(m)$ then $\sigma$ else $\tau$
                    $T(m) \longleftarrow T(m) \cup \{(P_m(q), \sigma') \to P_m(q')\}$
                    if $P_m(q') \notin Q(m)$ then
                        $Q(m) \longleftarrow Q(m) \cup \{P_m(q')\}$
                        if $P_{\{var_{K_i}\}}(q')$ is marked in K then
                            $Q_m(m) \longleftarrow Q_m(m) \cup \{P_m(q')\}$
                        end
                        newState $\longleftarrow$ True
                    end
                end
                if newState and controllable then
                    $Q_G \longleftarrow Q_G \cup \{q'\}$
                end
            else
                "waiting for broadcasts"
            end
        end
        return
          $Sup_m \langle Q(m), E'(m), T(m), P_m(q_0), Q_m(m), Q_f(m) \rangle$
end
```

Algorithm 2: The Modular Supervisor Learner algorithm that learns a modular supervisor from a simulation model, a PSH, and a set of specifications without knowing the plant model.

3.4 On Controllability and Non-blocking

Assuming that the modular system is represented by a valid PSH for the given simulation, the resulting supervisor is a maximally permissive controllable, but possibly blocking, modular supervisor. To show that the resulting supervisor is controllable, assume that a supervisor S is learned and is uncontrollable. Then there exists a non-forbidden state q in S, which does not have a transition labelled with an uncontrollable event that exists in the simulation. This would mean that this state q has not been explored by the algorithm. Had the state been explored, the transition labeled with the uncontrollable event would be found and by checking for controllability q would be added to the list of forbidden states. The fact that q was not explored would imply that the PSH and simulation do not define a valid PSH.

The method outlined above results in the set of maximally permissive and controllable modular supervisors. The synchronous composition of these supervisors can still be blocking, though. However, if the obtained result is blocking, the maximally permissive controllable and non-blocking supervisor can be synthesized from the maximally permissive controllable supervisor by existing methods (Ramadge and Wonham, 1989; Cassandras and Lafortune, 2009). Essentially this means extracting from the learned supervisor the largest (in terms of states and transitions) sub-automaton that is both controllable and non-blocking. Note that this can be done while still preserving the modular structure of the supervisor (Flordal et al., 2007; Malik et al., 2017).

3.5 Notes on Efficiency

Running this algorithm with a PSH that defines a monolithic model produces the worst case runtime. The more accurately the PSH defines a modular model, the better is the learning time as it avoids searching the complete monolithic state-space.

However even with an accurate PSH the search could end up being monolithic, and this has to do with the specifications and how supervisor modules are calculated. If specifications share uncontrollable events with some plant modules, and those in turn share uncontrollable events with other plant modules, this could result in a domino effect that leads to all the modules being included in a supervisor. The resulting learning would then be an inefficient monolithic search.

This can be improved by using an incremental approach to learning the supervisor, similar to the one proposed by Åkesson et al. (2002). Doing so requires a slight modification to the calculation of modules. Instead of calculating these supervisor modules iteratively until a fixed-point as in Section 3.1, the learning algorithm is applied only to the modules contained in the first iteration. That is, all plant modules defined in the PSH that directly share an uncontrollable event with the specification are taken together to learn a supervisor. Using the learning algorithm in such a manner will result in one supervisor for each specification and a set of plant models for those modules in the PSH that did not share any uncontrollable event with any specification. These resulting supervisors can then be used offline to check if the supervisor is uncontrollable with respect to other modules learned. If it is found to be uncontrollable, a supervisor is then synthesized using existing synthesis tools like Supremica (Malik et al., 2017).

4. CASE STUDY:THE CAT AND MOUSE PROBLEM

In this section, we will present the working of the algorithm using the Cat and Mouse example introduced by Ramadge and Wonham (1987). This example consists of a cat and mouse that can move in a seven-room maze as seen in Figure 1. The rooms are connected with doors of two sizes, small ones for the mouse (m_i) and larger ones for the cat (c_i). The cat and mouse can only use their respective doors. These are one-way doors, which can be locked. Thus, the cat and mouse can only pass through the door in the direction indicated by the arrows in the figure. Door c_7 though, allows the cat to pass through in both directions and cannot be locked. All files for this example can be found online[1] in Supremica[2] format.

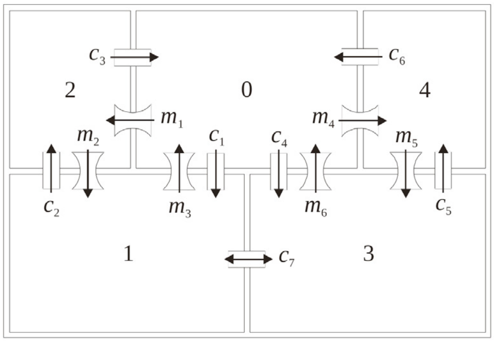

Fig. 1. Five room maze for the cat and mouse. The cat moves between the rooms using events c_i and the mouse using m_i in the direction indicated.

A simulation of this system is built up using two state variables var_c and var_m to track the location of the cat and mouse, respectively. Both these variables have the domain $\{R_0, R_1, R_2, R_3, R_4\}$ to represent the different rooms. Initially, $var_c = R_2$ and $var_m = R_4$.

Hence, the PSH is defined as follows:

- $M = \{Cat, Mouse\}$,
- $E(Cat) = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$,
- $E(Mouse) = \{m_1, m_2, m_3, m_4, m_5, m_6\}$,
- $S(Cat) = \{var_c\}$,
- $S(Mouse) = \{var_m\}$,

There are five specifications (Figure 2), one for each room, defining that both the cat and the mouse cannot be simultaneously in the same room, and each animal should be able to return to its initial room.

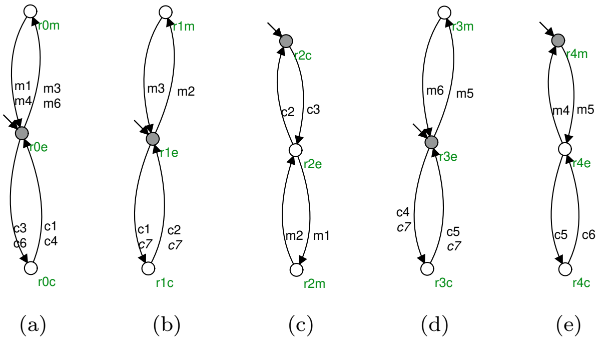

Fig. 2. Specifications for the different rooms ($Kr_0, Kr_1, Kr_2, Kr_3, Kr_4$, in that order) ensuring that only one of either the cat or the mouse can be present at a given time. Each state is identified using a unique name.

The different specifications have the following alphabets:

- $\Sigma_{Kr_0} = \{ c_1, c_3, c_4, c_6, m_1, m_3, m_4, m_6 \}$,
- $\Sigma_{Kr_1} = \{ c_1, c_2, c_7, m_2, m_3 \}$,
- $\Sigma_{Kr_2} = \{ c_2, c_3, m_1, m_2 \}$,
- $\Sigma_{Kr_3} = \{ c_4, c_5, c_7, m_5, m_6 \}$,
- $\Sigma_{Kr_4} = \{ c_5, c_6, m_4, m_5 \}$,

[1] https://github.com/ashfaqfarooqui/ModularSupLearner
[2] www.supremica.org

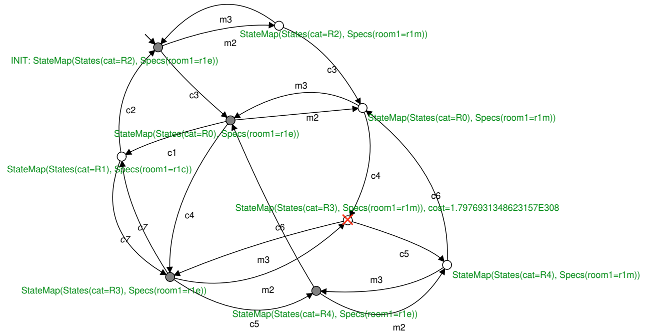
m3
StateMap(States(cat=R2), Specs(room1=r1m))
m2
INIT: StateMap(States(cat=R2), Specs(room1=r1e))
c3
c3
m3
c2
m2
StateMap(States(cat=R0), Specs(room1=r1m))
StateMap(States(cat=R0), Specs(room1=r1e))
c1
StateMap(States(cat=R1), Specs(room1=r1c))
c4
c6
StateMap(States(cat=R3), Specs(room1=r1m)), cost=1.7976931348623157E308
c4
c7
c6
c7
m3
c5
StateMap(States(cat=R4), Specs(room1=r1m))
m3
StateMap(States(cat=R3), Specs(room1=r1e))
m2
StateMap(States(cat=R4), Specs(room1=r1e))
c5
m2

Fig. 3. The resulting supervisor for specification Kr_1 and the plant simulation

As can be seen, only the specifications Kr_1 and Kr_3 contain c_7, the uncontrollable event. Hence, the supervisor configuration to learn consists of two supervisors, one for specification Kr_1 and the other for Kr_3. Calculating the modules results in the following:

$$M_{Kr_1} = \{\, Cat \,\}\,, \text{ and } M_{Kr_3} = \{\, Cat \,\}\,;$$
$$E_{Kr_1} = \Sigma_{Kr_1} \cup E(Cat), \text{ and } E_{Kr_3} = \Sigma_{Kr_3} \cup E(Cat);$$
$$S_{Kr_1} = \{\, var_c, var_{Kr_1} \,\}\,, \text{ and } S_{Kr_3} = \{\, var_c, var_{Kr_3} \,\}\,.$$

Hence, the set of supervisors to learn, M_K, and the plant modules to learn, M_G, are defined as:

$$M_K = \{\, M_{Kr_1}, M_{Kr_3} \,\}\,, \text{ and } M_G = \{\, Mouse \,\}\,;$$
$$V = \{\, var_c, var_m, var_{Kr_1}, var_{Kr_3} \,\}\,.$$

The remaining specifications Kr_0, Kr_2, Kr_4 can be treated as supervisors.

The workings of the learning algorithm are similar to Farooqui et al. (2020). The main difference in this case is the check for controllability. Consider the state defined by the variables $\langle var_c, var_{Kr_1} \rangle$ with valuation $\langle R_3, r1m \rangle$. On exploration, event c_7 is possible from this state in the simulator, leading the cat into room 1. The module builder when receiving the transition $(\langle R_3, r1m \rangle, c_7) \rightarrow \langle R_1, r1m \rangle$ checks if the transition is uncontrollable. To do so, the state $r1m$ in the specification is checked if allows the uncontrollable transition c_7. As can be seen in Figure 2b, transition c_7 is not allowed from the state named $r1m$. Hence, the source state is added to the set of forbidden states. Since no incoming transition is uncontrollable there is no need to further process uncontrollable states.

Figure 3 shows the supervisor for specification Kr_1. As seen, the supervisor has one state marked with a red cross, indicating that this is a forbidden state, since the state is uncontrollable. The forbidden state and its incoming transitions can be removed to obtain the controllable supervisor. Along with these two supervisors, the specifications for $Kr_0, Kr_2, and\ Kr_4$ are treated as supervisors.

5. CONCLUSION

Previously in Farooqui et al. (2020) a method to learn plant models from simulations and a known *plant structure hypothesis* PSH was presented. This paper extends upon the previous approach to directly learn maximally permissive controllable supervisors when supplied with specifications of the unknown plant. To this end, first, an approach to calculate the different grouping of plant modules for each specification is presented. Next, we add additional state variables to track each specification in order to find controllability issues. Finally, the presented Modular Supervisor Learner performs the learning. The result of the MSL is a set of controllable supervisors that can then be made non-blocking using known techniques.

The accuracy and performance of this method depends on the PSH. Defining a correct PSH is crucial and the most difficult aspect of using this method, as it relies on the knowledge, creativity, and experience of the engineer. Further research on how to define the PSH is needed.

REFERENCES

Aarts, F., de Ruiter, J., and Poll, E. (2013). Formal models of bank cards for free. doi:10.1109/ICSTW.2013.60.

Aarts, F., Schmaltz, J., and Vaandrager, F. (2010). Inference and abstraction of the biometric passport. In T. Margaria and B. Steffen (eds.), *Leveraging Applications of Formal Methods, Verification, and Validation*, 673–686. Springer, Berlin, Heidelberg.

Åkesson, K., Flordal, H., and Fabian, M. (2002). Exploiting modularity for synthesis and verification of supervisors. *IFAC Proceedings Volumes*, 35(1), 175 – 180. 15th IFAC World Congress.

Cassandras, C.G. and Lafortune, S. (2009). *Introduction to discrete event systems*. Springer Verlag.

Dai, J. and Lin, H. (2014). A learning-based synthesis approach to decentralized supervisory control of discrete event systems with unknown plants. *Control Theory and Technology*, 12(3), 218–233.

Farooqui, A., Hagebring, F., and Fabian, M. (2020). Active learning of modular plant models. WODES 2020.

Flordal, H., Malik, R., Fabian, M., and Åkesson, K. (2007). Compositional synthesis of maximally permissive supervisors using supervision equivalence. *Discrete Event Dynamic Systems*, 17(4), 475–504.

Hiraishi, K. (1999). Synthesis of supervisors for discrete event systems allowing concurrent behavior. In *IEEE SMC'99 Conference Proc.*, volume 5, 13–20.

Jonsson, B. (2011). *Learning of Automata Models Extended with Data*, 327–349. Springer, Berlin, Heidelberg.

Malik, R., Åkesson, K., Flordal, H., and Fabian, M. (2017). Supremica–an efficient tool for large-scale discrete event systems. *IFAC-PapersOnLine*, 50(1), 5794 – 5799. 20th IFAC World Congress.

Ramadge, P.J. and Wonham, W.M. (1987). Supervisory control of a class of discrete event processes. *SIAM journal on control and optimization*, 25(1), 206–230.

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

Steffen, B., Howar, F., and Merten, M. (2011). Introduction to active automata learning from a practical perspective. In *Lecture Notes in Computer Science, vol 6659. Springer.*

Yang, X., Lemmon, M., and Antsaklis, P. (1995). Inductive inference of optimal controllers for uncertain logical discrete event systems. In *Proceedings of Tenth International Symposium on Intelligent Control.* IEEE.

Zhang, H., Feng, L., and Li, Z. (2018). A learning-based synthesis approach to the supremal nonblocking supervisor of discrete-event systems. *IEEE Trans. on Automatic Control*, 63(10), 3345–3360. doi:10.1109/TAC.2018.2793662.