Active Learning of Modular Plant Models

Available online at www.sciencedirect.com

ScienceDirect

IFAC Papers Online CONFERENCE PAPER ARCHIVE

IFAC PapersOnLine 53-4 (2020) 296–302

Active Learning of Modular Plant Models ⋆

Ashfaq Farooqui, Fredrik Hagebring, Martin Fabian

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

Abstract: Model-based techniques are these days being embraced by the industry in their development frameworks. While model-based approaches allow for offline verification and validation of the system, and have other advantages over existing methods, they do have their own challenges. One of the challenges is to obtain a model describing the behavior of the system. In this paper we present the Modular Plant Learner (MPL), an algorithm that explores the state-space and constructs a discrete model of a system. The MPL takes as input a hypothesis structure of the system – called the PSH – and using this information, interacts with a simulation of the system to construct a modular discrete-event model. Using an example we show how the algorithm uses the structural information provided – the PSH – to search the state-space in a smart manner, mitigating the state-space explosion problem.

1. INTRODUCTION

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 certain *specifications*. The obtained supervisor can be used to control the system and this *controlled system* is known to behave according to the 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 (Ramadge and Wonham, 1987b). 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.

In this paper, we propose a state-based active learning approach that is able to learn a modular model of a target system. The main benefit being the ability to decrease the search space of the learning process by exploiting the modular structure of the system and, thus, mitigating the state-space explosion problem. To that end, we

⋆ 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.2021.04.028

assume the possibility to simulate the plant behavior in a simulation environment. We present the *Modular Plant Learner* (MPL) algorithm that explores the various states reachable in the simulator to construct a modular model of the simulated system. Additionally, the specific implementation of MPL 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.

This paper is organized as follows. First, Section 2 introduces the relevant notation. Section 3 then presents the main components that enable learning a discrete model, followed by the Modular Plant Learner. Section 4 takes an example and walks through the algorithm demonstrating the different concepts and showing its feasibility, before concluding in Section 5.

2. PREREQUISITES

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'

Let Σ, called an *alphabet*, be a finite set of *events*. Denote by τ the *silent event*, not part of Σ.

Definition 1. (DFA). A (*deterministic finite*) *automaton* is defined as a 4-tuple $\langle Q, \Sigma, \delta, q_0 \rangle$,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*.

Plant modules Let $G_i = \langle Q_i, \Sigma_i, \delta_i, q_{0i} \rangle$ be a DFA that describes the behaviour of a plant module. The full behaviour of the plant is then described by a set of all its individual modules given by $G = \{G_1, G_2, ...G_n\}$, known as the *modular model*. The synchronous composition (Cassandras and Lafortune, 2009) of all the modules $G_1 \parallel G_2 \parallel ... \parallel G_n$ results in the complete behaviour of the plant referred to as the *monolithic model*.

3. TOWARDS LEARNING A MODULAR PLANT

Farooqui and Fabian (2019) showed how a monolithic discrete event model can be learned from a simulation.

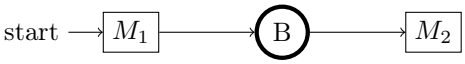
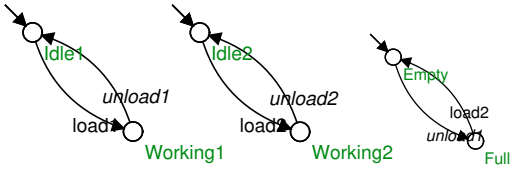


Fig. 1. Two machines and a buffer

However, learning a monolithic model requires the learning algorithm to traverse and visit all reachable states, resulting in the state-space explosion problem. Thus, in this section we show the possibility to instead learn a modular plant, where included modules are learned in parallel using the MPL algorithm. The algorithm interacts with a simulation of the plant that is to be modeled, and actively queries the simulation in a smart way to learn what states are reachable from the initial state for the respective modules. To be able to do this, the algorithm uses an initial *plant structure hypothesis* (PSH) to split the acquired learning into a set of modules.

3.1 Running Example

In this section, we present a well known example of two machines and a buffer (Ramadge and Wonham, 1987a). Here, two identical machines M_1 and M_2 are connected with a buffer B in-between. Each machine can load a part and then unload it. The setup is shown in Figure 1. When machine M_1 unloads a part, the part moves to the buffer; when M_2 loads a part it does so from the buffer. Models representing the behavior of the three subsystems are shown in Figure 2. States are represented by circles, and the arrows represent the transitions between the states. An arrow with no tail state indicates the initial state. The alphabet of the system is $\{ load_1, unload_1, load_2, unload_2 \}$.



(a) Machine 1 (b) Machine 2 (c) Buffer

Fig. 2. Models for the two machines and the buffer.

3.2 The Simulation

Simulations provide several advantages in comparison to using the real system. Unlike the real system, the simulation can be 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

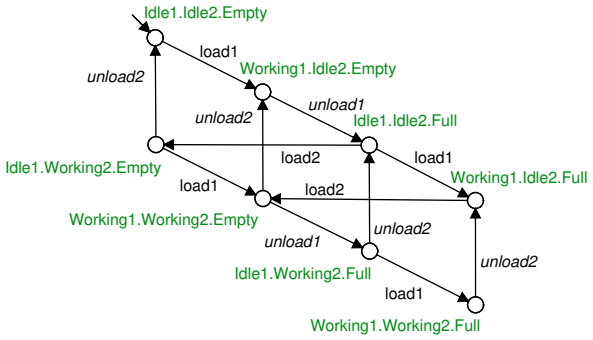


Fig. 3. The complete behaviour of the simulated system when the buffer size is one

to another state. In the case that 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 here consists of a set of variables and their corresponding values in the simulation; these could, for example in a manufacturing context, be the status of sensors, actuators, and product position.

For this purpose, let us 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.

Following up from the example, a simulator is created that mimics the complete system. That is, the simulator does not differentiate the two machines from the buffer specification. Thus, the simulator behaves like the synchronous composition of the three models of Figure 2, shown in Figure 3.

Furthermore, three variables, $varM_1$, $varM_2$ and $varB$, are used in this simulation to keep track of the state of each subsystem. The domains of these variables are given by the respective state names in the automata models of Figure 2. That is, variables $varM_i$ are initially set to `Idle` and can be updated to the value `Working` and then reset back to `Idle` by executing the corresponding load and unload events, respectively. Similarly the variable $varB$ is initially set to `Empty` and can then be updated to `Full`, and reset to `Empty` when events $unload_1$ and $load_2$ are executed, respectively. The initial state of the complete simulator is $q_0 = \langle \texttt{Idle}, \texttt{Idle}, \texttt{Empty} \rangle$. The values of the three variables are continuously monitored.

3.3 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 the plant is composed of; in the example these are M_1, M_2, and B. This can then be exploited by the MPL to divide the learned information into separate plant 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 that is to be learned. 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 Σ 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. Consequently, to minimize the size of the final modular plant, events should only be included in event mappings of those modules that have a vital role in their execution. Finally, a mapping S, called *state mapping*, 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 $q_i, q_j \in V^D$ 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 2. (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*;

To guarantee that the MPL explores the full plant, the union of all event mappings should encompass the whole alphabet Σ and the union of all state mappings should 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, respectively. This is, however, only a lower bound on the PSH. For any given system there may exist multiple PSH of various coarseness; the coarsest one being a PSH defining only a single module m with $E(m) = \Sigma$ and $S(m) = V$. This does satisfy the criteria and will ensure that a full plant model is learned but the learning will be very inefficient, since there is no modular information to exploit. In many cases a PSH can be refined by considering the physical structure of the plant, defining separate modules from subsystems, such as machines, robots or vehicles. In other cases it may be more efficient to combine multiple strongly connected subsystems into a single module, since their shared behavior otherwise needs to be represented redundantly in each module, or to define modules that capture specific operations or actions regardless of the subsystems involved.

Additionally, what modules that can be learned also depends on the available state variables in the simulation.

For example, assume that there is a second simulation of the running example that outputs a single variable that merge the values of the variables $varM_1$, $varM_2$ and $varB$ into a single string of text. That is to say, a state $\langle \texttt{Idle}, \texttt{Idle}, \texttt{Empty} \rangle$ in the original simulation would in the new simulation be, for example, "`Idle Idle Empty`". In this case, the information available in the state variable is the same and, hence, the system is unchanged but the MPL can no longer learn the modular plant model. With no distinction between the state variables, the state and event mapping of all modules have to include the single variable and the full alphabet respectively, since the variable is affected by all events and any change to this variable is of interest to the entire system.

Ideally, the event and state mappings should be as restrictive as possible and only include events and state variables that affect the collaboration between modules. This will maximize the structural information that can be exploited, which minimizes the search space of the learning algorithm. Furthermore, the state mapping of each module typically must include every such variable; but in some cases, as shown in the example below, the learning can be completed even without this constraint fulfilled. This however, depends strongly on the specific structure of the plant and requires extensive understanding of both the plant and the learning process to identify, and hence, cannot be expected in most situations.

To summarize, defining a PSH is a complex task and there is typically no unique solution. There are multiple properties to consider when defining the modules in a PSH, such as the extent of the state-space explosion compared to the amount of redundancy between the modules. In general, an event/state mapping *should* be as restrictive as possible but *must* be an overestimation of the required events/variables while conforming to the simulation. It is, however, not always trivial to identify all events and state variables that can be left out from a specific module. Thus, the creator of a PSH must have sufficient knowledge about the system and the connection between events/variables and the defined modules.

Example Continued

To better understand the notion of a PSH, let us define the PSH for the example of Section 3.1. Here, we are interested in learning the behaviour of the physical subsystems M_1 and M_2 and of their interaction through B. Following the given PSH guidelines, a suitable PSH can then be defined as:

- $M = \{M_1, M_2, B\}$,
- $E(M_1) = \{load_1, unload_1\}$,
- $E(M_2) = \{load_2, unload_2\}$,
- $E(B) = \{unload_1, load_2\}$,
- $S(M_1) = \{varM_1, varB\}$,
- $S(M_2) = \{varM_2, varB\}$,
- $S(B) = \{varM_1, varM_2, varB\}$,

where the set of modules represents the physical subsystems, the event mapping includes all events of these subsystems, and the state mapping for each module includes all variables that contribute to its behaviour. However, this PSH is very inefficient since $S(B)$ includes all state variables, which would result in a monolithic search of the plant to learn this module. Fortunately, typically only in very small systems with strongly connected subsystems would a state mapping cover the complete set of state variables. Moreover, as will be shown in Section 4, the learning of this specific system can done much more efficiently by omitting some variables from the state mappings.

3.4 Learning a Modular Plant

Given an interface to a simulator capable of simulating the complete plant, the MPL will compute modules, as defined by the PSH. This section presents the algorithm and demonstrates its working with the help of the example.

The Modular Plant Learner

The algorithm consists of three procedures, *Main*, *Explorer*, and *ModuleBuilder*, see Algorithm 1, and is constructed so that the latter two procedures execute concurrently. The program is initiated by *Main* that launches the *Explorer* and one instance of *ModuleBuilder* for each module defined in the PSH. 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 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 $T(m) : Q(m) \times E(m) \rightarrow Q(m)$. The *ModuleBuilder*, on receiving the broadcast, finds all the variables V' that have been updated in the reached state. The builder then evaluates if the received transition is of interest to the particular module. There are two conditions for the builder to be interested in a transition.

(i) $\sigma \in E(m)$, the event in the transition is also defined by the event mapping of the particular module.
(ii) $S(m) \cap V' \neq \emptyset$, i.e at least one variable defined in the state mapping has been updated in the transition.

The *ModuleBuilder* processes the transition if one or both of the above conditions hold, leading to three possible scenarios.

When both conditions (i) and (ii) are satisfied, then the transition $(P_m(q), \sigma) \rightarrow P_m(q')$ is appended to the set of known transitions for module m. Additionally, if $P_m(q')$ is a new state, i.e, $P_m(q') \notin Q(m)$, then q' is added to the global exploration queue used by the *Explorer*, and $P_m(q')$ is added to $Q(m)$.

When only condition (ii) is satisfied, then a τ-transition $(P_m(q), \tau) \rightarrow P_m(q')$ is added to the module. This is done to monitor changes made by other modules that may lead to new states where the current module can continue its

execution. Additionally, similar to the above case, if q' is a new state, then it is added to the global exploration queue and added to the local set of states in $Q(m)$.

When only condition (i) holds, then the resulting transition refers to a self loop; since, $q = q'$. The *ModuleBuilder* adds the transition $(P_m(q), \sigma) \rightarrow P_m(q')$ to its set of transitions.

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 $Q(m)$ and $T(m)$.

Note that the criteria $P_m(q') \notin Q(m)$ implies that q' is a new, not seen before, state. However, the converse does not hold; q' can be a new, not seen before state, but $P_m(q')$ may still already be in $Q(m)$. This lets a new state q' be further expanded if and only if it also represents a new state in at least one of the modules. This property constitutes the main search space reduction of the modular approach. However, it relies heavily on the correctness of the PSH. If one of the state mappings is too narrow, this might ignore some important combinations of variables and, thus, fail to explore the full system behavior.

The τ-transitions mentioned above represent actions that, since they do not involve any event in the event-mapping, should not represent any vital behavior within the module. Moreover, this implies that any state variable that is changed by these transitions can be ignored by the module. That is to say, if the value of a state variable is changed by a τ-transition, which by definition can be ignored in the module, then this variable can also be ignored. To show this, consider the contradiction where a τ-transition changes the value of a state variable that is actually important to the module, this implies that the transition represents a vital behavior within the module, which is contradicted by the definition of the event mapping of the module. Thus, assuming a correct event mapping, the state mapping can be refined by removing all state variables that are changed by any τ-transitions. All τ-transitions will then become self-loops, since the state projection of the source and target states now are equal. Finally, these self-loops can safely be removed.

To further illustrate this concept, consider M_2 in the example. Given the PSH defined in Section 3.3 the state mapping is $S(M_2) = \{varM_2, varB\}$. The reason why $varB$ is interesting to this module is that, from an initial state where the machine is `Idle` and $varB$ is `Empty`, there are no transitions available in the module. Instead it requires $varB$ to change to `Full` in order for the event $load_2$ to become possible. It then makes sense to monitor $varB$, such that the module identifies that this state, where the buffer is full, does occur and continues the exploration. Filling the buffer does not, however, directly involve the second machine and, hence, the event $unload_1$ is omitted from $E(M_2)$ and consequently generates a τ-transition. This is illustrated in Figure 4.

Fig. 4. Illustration of τ-transitions as a means to monitor external changes to the state variables of the current module. (left) M_2 including a τ-transition, indicating that $varB$ has to change before the local event $load_2$ can execute. (right) The final M_2 with τ-transition and $varB$ removed, representing exclusively the local behavior.

4. ILLUSTRATIVE EXAMPLE

Taking the example discussed before, the working of the algorithm will be illustrated here.

Assume that there exists a simulation of the example as described in Section 3.2, and that the starting state of the simulator is known; that is, the initial global state q_0 is defined. Assume further that the PSH of the example is defined to be:

- $M = \{M_1, M_2, B\}$,
- $E(M_1) = \{load_1, unload_1\}$,
- $E(M_2) = \{load_2, unload_2\}$,
- $E(B) = \{unload_1, load_2\}$,
- $S(M_1) = \{varM_1\}$,
- $S(M_2) = \{varM_2, varB\}$,
- $S(B) = \{varB\}$,

where, compared to the more generic PSH presented in Section 3.3, variable $varB$ has been omitted from $S(M_1)$ and variables $varM_1$ and $varM_2$ have been omitted from $S(B)$. The variable $varB$ could have been omitted also from $S(M_2)$ but has been kept to illustrate how τ-transitions are generated and how they are removed in the final result. This simplification of the PSH is possible due to each module being very basic and only including a single looping sequence with no alternative behavior. Generally, a PSH cannot be simplified as much as in this case, but there are usually *some* variables that can be omitted in order to improve efficiency.

Sending the above PSH as input to the MPL will result in the modular plant to be learned, including the buffer specification.

First, the *Main* procedure generates the initial search queue Q_G, starts *ModuleBuilder*(M_1), *ModuleBuilder*(M_2) and *ModuleBuilder*(B), representing the two machines and the buffer. The search is then initiated by running the *Explorer*. After this, the work is performed asynchronously in the individual processes, while the *Main* procedure just waits for the search to terminate. For simplicity of illustration the description of the execution is divided into iteration steps, where each step includes one iteration of the main loop in *Explorer* followed by one iteration of the main loop in each *ModuleBuilder*. For each step, the status of modules are illustrated by figures 5-10.

Recall from Section 3.2 that a global state in this example is given by the state vector $\langle varM_1, varM_2, varB\rangle$. Based on the state mapping of the PSH defined above, we know that the modules M_1, M_2 and B have local state vectors $\langle varM_1\rangle$, $\langle varM_2, varB\rangle$ and $\langle varB\rangle$, respectively.

Input: An interface to the simulator as specified in Section 3.2, the initial global state, q_0, of the system, and a PSH $H = \{M, E, S\}$.
Result: A set G of modular plant components, $G_m \in G, \forall m \in M$.
begin
  Procedure *Main*
    $Q_G \longleftarrow \{q_0\}$
    - 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 G_m returned by ModuleBuilders into a set G
    - Remove τ-transitions in all modules contained in G and their corresponding variables
    return G
  Procedure *Explorer*
    while *Run* **do**
      for $q \in Q_G$ **do**
        for $\sigma \in \Sigma$ **do**
          - find q' by executing σ 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)*
    $Q(m) \longleftarrow \{P_m(q_0)\}$, $T(m) \longleftarrow \emptyset$
    while *Run* **do**
      if (q, σ, q') *received* **then**
        $V' \longleftarrow \{v \in V \mid q(v) \neq q'(v)\}$
        $newState \longleftarrow False$;
        if $\sigma \in E(m)$ **or** $S(m) \cap V' \neq \emptyset$ **then**
          $\sigma' \longleftarrow$ **if** $\sigma \in E(m)$ **then** σ **else** τ
          $T(m) \longleftarrow T(m) \cup \{(P_m(q), \sigma') \rightarrow P_m(q')\}$
          if $P_m(q') \notin Q(m)$ **then**
            $Q(m) \longleftarrow Q(m) \cup \{P_m(q')\}$
            $newState \longleftarrow True$
          end
        end
        if *newState* **then**
          $Q_G \longleftarrow Q_G \cup \{q'\}$
        end
      else
        "waiting for broadcasts"
      end
    end
    return G_m
end

Algorithm 1: The Modular Plant Learner to learn a modular plant model from a simulation model and a PSH.

Initialization After initialization by *Main*, $Q_G = \{q_0\}$ with $q_0 = \langle \texttt{Idle}, \texttt{Idle}, \texttt{Empty} \rangle$, $Q(M_1) = \langle \texttt{Idle} \rangle$, $Q(M_2) = \langle \texttt{Idle}, \texttt{Empty} \rangle$, and $Q(B) = \langle \texttt{Empty} \rangle$

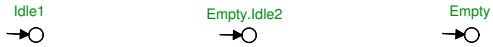


Fig. 5. Initialization. (left) M_1, (mid) M_2, and (right) B.

Iteration 1

- *Explorer*: Expanding Q_G yields a single possible transition: $t = (\langle \texttt{Idle}, \texttt{Idle}, \texttt{Empty} \rangle, load_1) \rightarrow \langle \texttt{Working}, \texttt{Idle}, \texttt{Empty} \rangle$. This is then being broadcast to the three *ModuleBuilder*s.
- *ModuleBuilder*(M_1): Since $varM_1 \in S(M_1)$ is changed by t from `Idle` to `Working` and $load_1 \in E(M_1)$, a local transition $(\langle \texttt{Idle} \rangle, load_1) \rightarrow \langle \texttt{Working} \rangle$ is added to $T(M_1)$, the target state $\langle \texttt{Working} \rangle$ is added to $Q(M_1)$ and the global target state is added to Q_G.
- *ModuleBuilder*(M_2): No local transition is added to M_2.
- *ModuleBuilder*(B): No local transition is added to B.

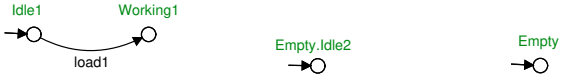


Fig. 6. Iteration 1. (left) M_1, (mid) M_2, and (right) B.

Iteration 2

- *Explorer*: Expanding Q_G yields, again, a single transition: $t = (\langle \texttt{Working}, \texttt{Idle}, \texttt{Empty} \rangle, unload_1) \rightarrow \langle \texttt{Idle}, \texttt{Idle}, \texttt{Full} \rangle$.
- *ModuleBuilder*(M_1): Since $varM_1 \in S(M_1)$ is changed by t from `Working` to `Idle` and $unload_1 \in E(M_1)$, a local transition $(\langle \texttt{Working} \rangle, unload_1) \rightarrow \langle \texttt{Idle} \rangle$ is added to $T(M_1)$ and the global target state is added to Q_G.
- *ModuleBuilder*(M_2): Since $varB \in S(M_2)$ is changed by t from `Empty` to `Full` and $unload_1 \notin E(M_2)$, a local transition $(\langle \texttt{Idle}, \texttt{Empty} \rangle, \tau) \rightarrow \langle \texttt{Idle}, \texttt{Full} \rangle$ is added to $T(M_2)$ and state $\langle \texttt{Idle}, \texttt{Full} \rangle$ is added to $Q(M_2)$.
- *ModuleBuilder*(B): Since $varB \in S(B)$ is changed by t from `Empty` to `Full` and $unload_1 \in E(B)$, a local transition $(\langle \texttt{Empty} \rangle, unload_1) \rightarrow \langle \texttt{Full} \rangle$ is added to $T(B)$ and state $\langle \texttt{Full} \rangle$ is added to $Q(B)$.

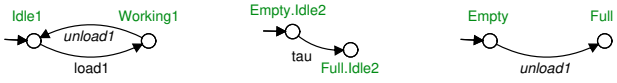


Fig. 7. Iteration 2. (left) M_1, (mid) M_2, and (right) B.

Iteration 3

- *Explorer*: Expanding Q_G yields two transition: $t_1 = (\langle \texttt{Idle}, \texttt{Idle}, \texttt{Full} \rangle, load_1) \rightarrow \langle \texttt{Working}, \texttt{Idle}, \texttt{Full} \rangle$ and $t_2 = (\langle \texttt{Idle}, \texttt{Idle}, \texttt{Full} \rangle, load_2) \rightarrow \langle \texttt{Idle}, \texttt{Working}, \texttt{Empty} \rangle$.
- *ModuleBuilder*(M_1): Even though transition t_1 does change a variable connected to M_1, the corresponding local transition is identical to what was previously seen in iteration 1, since only variable $varM_1$ is considered, and adds no local transition to M_1. Transition t_2 adds no local transition to M_1.
- *ModuleBuilder*(M_2): Transition t_1 adds no local transition to M_2. Since both variables that are changed by t_2 are in $S(M_2)$ and $load_2 \in E(M_2)$, a local transition $(\langle \texttt{Idle}, \texttt{Full} \rangle, load_2) \rightarrow \langle \texttt{Working}, \texttt{Empty} \rangle$ is added to $T(M_2)$, state $\langle \texttt{Working}, \texttt{Empty} \rangle$ is added to $Q(M_2)$ and the global target state is added to Q_G.
- *ModuleBuilder*(B): Transition t_1 adds no local transition to B. Since $varB \in S(B)$ is changed by t_2 from `Full` to `Empty` and $load_2 \in E(B)$, a local transition $(\langle \texttt{Full} \rangle, load_2) \rightarrow \langle \texttt{Empty} \rangle$ is added to $T(B)$.

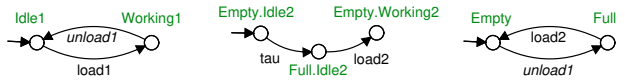


Fig. 8. Iteration 3. (left) M_1, (mid) M_2, and (right) B.

Iteration 4

- *Explorer*: Expanding Q_G yields, again, two transition: $t_1 = (\langle \texttt{Idle}, \texttt{Working}, \texttt{Empty} \rangle, load_1) \rightarrow \langle \texttt{Working}, \texttt{Working}, \texttt{Empty} \rangle$ and $t_2 = (\langle \texttt{Idle}, \texttt{Working}, \texttt{Empty} \rangle, unload_2) \rightarrow \langle \texttt{Idle}, \texttt{Idle}, \texttt{Empty} \rangle$.
- *ModuleBuilder*(M_1): Transition t_1 is the same is identified in iteration 3 and adds no local transition to M_1. Transition t_2 adds no local transition to M_1.
- *ModuleBuilder*(M_2): Transition t_1 adds no local transition to M_2. Since both variables that are changed by t_2 are in $S(M_2)$ and $load_2 \in E(M_2)$, a local transition $(\langle \texttt{Idle}, \texttt{Full} \rangle, load_2) \rightarrow \langle \texttt{Working}, \texttt{Empty} \rangle$ is added to the module, but the target state has already been seen so no state is added to $Q(M_2)$ or Q_G.
- *ModuleBuilder*(B): No local transition is added to B.

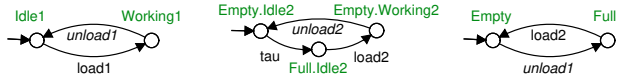


Fig. 9. Iteration 4. (left) M_1, (mid) M_2, and (right) B.

Termination Since Q_G is now empty, the algorithm can terminate. The learning is now complete and, following the discussion in Section 3.4, the final modular plant model, shown in Figure 10, is computed by removing all τ-transitions and all variables that are changed by them.

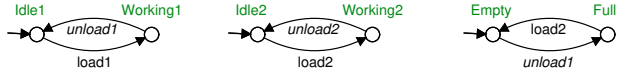


Fig. 10. Final modular system. (left) M_1, (mid) M_2, and (right) B.

5. CONCLUSION

This paper presented an approach to learn a modular discrete-event model of a system using a simulation of it. To this end, the *Modular Plant Learner* is presented along with the required simulation interface and a PSH that makes learning of a modular model possible. Furthermore, an example demonstrates the working of the presented algorithm. The resulting plant model can be used along with user defined specifications to synthesize a supervisor using existing synthesis algorithms (Malik et al., 2017).

The accuracy and performance of this method depends upon the defined 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 the procedure to define the PSH is needed.

Additionally, the natural next step is to learn a modular and compositional supervisor directly instead of the plant model (Mohajerani et al., 2014; Flordal et al., 2007), as well as to learn richer formalisms, in particular Extended Finite State Machines (Malik et al., 2011).

REFERENCES

Aarts, F., de Ruiter, J., and Poll, E. (2013). Formal models of bank cards for free. *IEEE 6th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2013*, 461–468.

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.

Cassandras, C.G. and Lafortune, S. (2009). *Introduction to discrete event systems*. Springer Science & Business Media.

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. and Fabian, M. (2019). Synthesis of supervisors for unknown plant models using active learning. In *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, 502–508.

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 Proceedings.*, volume 5, 13–20.

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

Malik, Åkesson, Flordal, and Fabian (2017). Supremica–an efficient tool for large-scale discrete event systems. In *The 20th World Congress of the International Federation of Automatic Control, 9-14 July 2017*.

Malik, R., Fabian, M., and Åkesson, K. (2011). Modelling large-scale discrete-event systems using modules, aliases, and extended finite-state automata. *IFAC Proceedings Volumes*, 44(1). 18th IFAC World Congress.

Mohajerani, S., Malik, R., and Fabian, M. (2014). A framework for compositional synthesis of modular nonblocking supervisors. *IEEE Transactions on Automatic Control*, 59(1), 150–162. doi:10.1109/TAC.2013.2283109.

Ramadge, P.J. and Wonham, W.M. (1987a). 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. *Proceedings of the IEEE*, 77(1).

Ramadge, P. and Wonham, W. (1987b). Modular feedback logic for discrete event systems. *IFAC Proceedings Volumes*, 20(9), 93 – 98.

Steffen, B., Howar, F., and Merten, M. (2011). Introduction to active automata learning from a practical perspective. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*.

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.