Applying Automata Learning to Embedded Control Software[⋆]

Wouter Smeenk[1], Joshua Moerman[2], Frits Vaandrager[2], and David N. Jansen[2]

[1] Océ Technologies B.V., Venlo, the Netherlands
[2] Faculty of Science, Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, the Netherlands

Abstract. Using an adaptation of state-of-the-art algorithms for black-box automata learning, as implemented in the LearnLib tool, we succeeded to learn a model of the Engine Status Manager (ESM), a software component that is used in printers and copiers of Océ. The main challenge that we encountered was that LearnLib, although effective in constructing hypothesis models, was unable to find counterexamples for some hypotheses. In fact, none of the existing FSM-based conformance testing methods that we tried worked for this case study. We therefore implemented an extension of the algorithm of Lee & Yannakakis for computing an adaptive distinguishing sequence. Even when an adaptive distinguishing sequence does not exist, Lee & Yannakakis' algorithm produces an adaptive sequence that 'almost' identifies states. In combination with a standard algorithm for computing separating sequences for pairs of states, we managed to verify states with on average 3 test queries. Altogether, we needed around 60 million queries to learn a model of the ESM with 77 inputs and 3.410 states. We also constructed a model directly from the ESM software and established equivalence with the learned model. To the best of our knowledge, this is the first paper in which active automata learning has been applied to industrial control software.

1 Introduction

Once they have high-level models of the behavior of software components, software engineers can construct better software in less time. A key problem in practice, however, is the construction of models for existing software components, for which no or only limited documentation is available.

The construction of models from observations of component behavior can be performed using regular inference (aka automata learning) techniques [4, 19, 37]. The most efficient such techniques use the setup of *active learning,* illustrated in Figure 1, in which a "learner" has the task to learn a model of a system by actively asking questions to a "teacher". The core of the teacher is a *System*

[⋆] This work was supported by STW project 11763 Integrating Testing And Learning of Interface Automata (ITALIA) and by the DFG/NWO Bilateral Research Programme ROCKS. An early version of this paper appeared as [35].

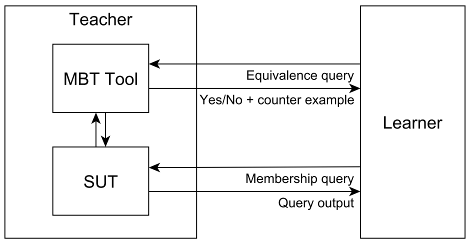

Fig. 1. Active learning of reactive systems

Under Test (SUT), a reactive system to which one can apply inputs and whose outputs one may observe. The learner interacts with the SUT to infer a model by sending inputs and observing the resulting outputs ("membership queries"). In order to find out whether an inferred model is correct, the learner may pose an "equivalence query". The teacher uses a model-based testing (MBT) tool to try and answer such queries: Given a hypothesized model, an MBT tool generates a long test sequence using some conformance testing method. If the SUT passes this test, then the teacher informs the learner that the model is deemed correct. If the outputs of the SUT and the model differ, this constitutes a counterexample, which is returned to the learner. Based on such a counterexample, the learner may then construct an improved hypothesis. It is important to note that it may occur that an SUT passes the test for an hypothesis, even though this hypothesis is not valid.

Triggered by various theoretical and practical results, see e. g. [7, 26, 33, 1, 8, 28, 20], there is a fast-growing interest in automata learning technology. In recent years, automata learning has been applied successfully, e. g., to regression testing of telecommunication systems [22], checking conformance of communication protocols to a reference implementation [3], finding bugs in Windows and Linux implementations of TCP [13], analysis of botnet command and control protocols [9], and integration testing [27, 17].

In this paper, we explore whether LearnLib [33], a state-of-the-art automata learning tool, is able to learn a model of the Engine Status Manager (ESM), a piece of control software that is used in many printers and copiers of Océ. Software components like the ESM can be found in many embedded systems in one form or another. Being able to retrieve models of such components automatically is potentially very useful. For instance, if the software is fixed or enriched with new functionality, one may use a learned model for regression testing. Also, if the source code of software is hard to read and poorly documented, one may use a model of the software for model-based testing of a new implementation, or even for generating an implementation on a new platform automatically. Using a model checker one may also study the interaction of the software with other components for which models are available.

The ESM software is actually well documented, and an extensive test suite exists. The ESM, which has been implemented using Rational Rose Real-Time

(RRRT), is stable and has been in use for 10 years. Due to these characteristics, the ESM is an excellent benchmark for assessing the performance of automata learning tools in this area. The ESM has also been studied in other research projects: Ploeger [31] modeled the ESM and other related managers and verified properties based on the official specifications of the ESM, and Graaf and Van Deursen [16] have checked the consistency of the behavioral specifications defined in the ESM against the RRRT definitions.

Learning a model of the ESM turned out to be more complicated than expected. The top level UML/RRRT statechart from which the software is generated only has 16 states. However, each of these states contains nested states, and in total there are 70 states that do not have further nested states. Moreover, the C++ code contained in the actions of the transitions also creates some complexity, and this explains why the minimal Mealy machine that models the ESM has 3.410 states. LearnLib has been used to learn models with tens of thousands of states [32], and therefore we expected that it would be easy to learn a model for the ESM. However, finding counterexamples for incorrect hypotheses turned out to be challenging due to the large number of 77 inputs. The test algorithms implemented in LearnLib, such as random testing, the W-method [10, 38] and the Wp-method [14], failed to deliver counterexamples within an acceptable time. Automata learning techniques have been successfully applied to case studies in which the total number of input symbols is much larger, but in these cases it was possible to reduce the number of inputs to a small number (< 10) using abstraction techniques [2, 21]. In the case of ESM, use of abstraction techniques only allowed us to reduce the original 156 concrete actions to 77 abstract actions.

We therefore implemented and extension of the algorithm of Lee & Yannakakis [25] for computing an adaptive distinguishing sequence. Even when an adaptive distinguishing sequence does not exist, Lee & Yannakakis' algorithm produces an adaptive sequence that 'almost' identifies states. In combination with a standard algorithm for computing separating sequences for pairs of states, we managed to verify states with on average 3 test queries and to learn a model of the ESM with 77 inputs and 3.410 states. We also constructed a model directly from the ESM software and established equivalence with the learned model. To the best of our knowledge, this is the first paper in which active automata learning has been applied to industrial control software. Preliminary evidence suggests that our adaptation of Lee & Yannakakis' algorithm outperforms existing FSM-based conformance algorithms.

During recent years most researchers working on active automata learning focused their efforts on efficient algorithms and tools for the construction of hypothesis models. Following [7], our work shows that the context of automata learning provides both new challenges and new opportunities for the application of testing algorithms. All the models for the ESM case study together with the learning/test statistics are available at `http://www.mbsd.cs.ru.nl/publications/papers/fvaan/ESM/`, as a benchmark for both the automata learning and testing communities.

2 Engine Status Manager

The focus of this article is the *Engine Status Manager (ESM),* a software component that is used to manage the status of the engine of Océ printers and copiers. In this section, the overall structure and context of the ESM will be explained.

2.1 ESRA

The requirements and behavior of the ESM are defined in a software architecture called Embedded Software Reference Architecture (ESRA). The components defined in this architecture are reused in many of the products developed by Océ and form an important part of these products. This architecture is developed for *cut-sheet* printers or copiers. The term cut-sheet refers to the use of separate sheets of paper as opposed to a *continuous feed* of paper.

An *engine* refers to the printing or scanning part of a printer or copier. Other products can be connected to an engine that pre- or postprocess the paper, for example a cutter, folder, stacker or stapler. Figure 2 gives an overview of the

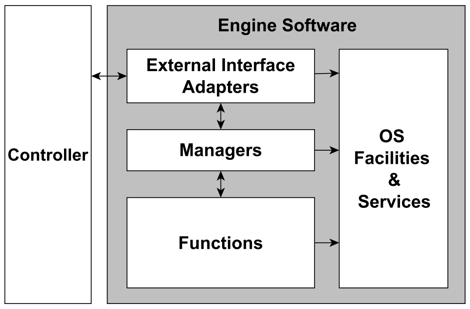

Fig. 2. Global overview of the engine software

software in a printer or copier. The *controller* communicates the required actions to the engine software. This includes transport of digital images, status control, print or scan actions and error handling. The controller is responsible for queuing, processing the actions received from the network and operators and delegating the appropriate actions to the engine software. The *managers* communicate with the controller using the *external interface adapters.* These adapters translate the external protocols to internal protocols. The managers manage the different functions of the engine. They are divided by the different functionalities such as status control, print or scan actions or error handling they implement. In order to do this a manager may communicate with other managers and functions. A *function* is responsible for a specific set of hardware components. It translates commands from the managers to the function hardware and reports the status and other information of the function hardware to the managers. This hardware can for example be the printing hardware or hardware that is not part of the engine hardware such as a stapler. Other functionalities such as logging and debugging are orthogonal to the functions and managers.

2.2 ESM and connected components

The ESM is responsible for the transition from one status of the printer or copier to another. It coordinates the functions to bring them in the correct status. Moreover, it informs all its connected clients (managers or the controller) of status changes. Finally, it handles status transitions when an error occurs.

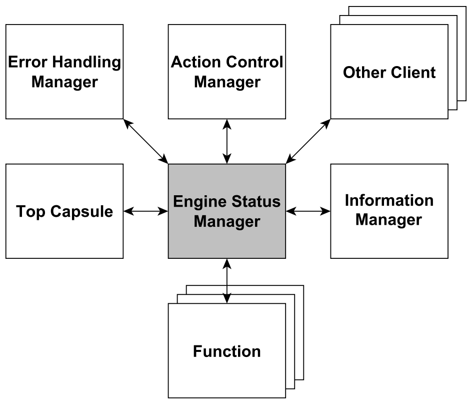

Fig. 3. Overview of the managers and clients connected to the ESM

Figure 3 shows the different components to which the ESM is connected. The *Error Handling Manager (EHM), Action Control Manager (ACM)* and other clients request engine statuses. The ESM decides whether a request can be honored immediately, has to be postponed or ignored. If the requested action is processed the ESM requests the functions to go to the appropriate status. The EHM has the highest priority and its requests are processed first. The EHM can request the engine to go into the defect status. The ACM has the next highest priority. The ACM requests the engine to switch between running and standby status. The other clients request transitions between the other statuses, such as idle, sleep, standby and low power. All the other clients have the same lowest priority. The Top Capsule instantiates the ESM and communicates with it during the initialization of the ESM. The Information Manager provides some parameters during the initialization.

There are more managers connected to the ESM but they are of less importance and are thus not mentioned here.

2.3 Rational Rose RealTime

The ESM has been implemented using *Rational Rose RealTime (RRRT)*. In this tool so-called *capsules* can be created. Each of these capsules defines a hierarchical *statechart diagram*. Capsules can be connected with each other using *structure diagrams*. Each capsule contains a number of ports that can be connected to ports of other capsules by adding connections in the associated structure diagram. Each of these ports specifies which protocol should be used. This protocol defines which messages may be sent to and from the port. Transitions in the statechart diagram of the capsule can be triggered by arriving messages on a port of the capsule. Messages can be sent to these ports using the action code of the transition. The transitions between the states, actions and guards are defined in C++ code. From the state diagram, C++ source files are generated.

The RRRT language and semantics is based on UML [30] and ROOM [34]. One important concept used in RRRT is the run-to-completion execution model [12]. This means that when a received message is processed, the execution cannot be interrupted by other arriving messages. These messages are placed in a queue to be processed later.

2.4 The ESM state diagram

Figure 4 shows the top states of the ESM statechart. The statuses that can be requested by the clients and managers correspond to gray states. The other states are so called *transitory states*. In transitory states the ESM is waiting for the functions to report that they have moved to the corresponding status. Once all functions have reported, the ESM moves to the corresponding status.

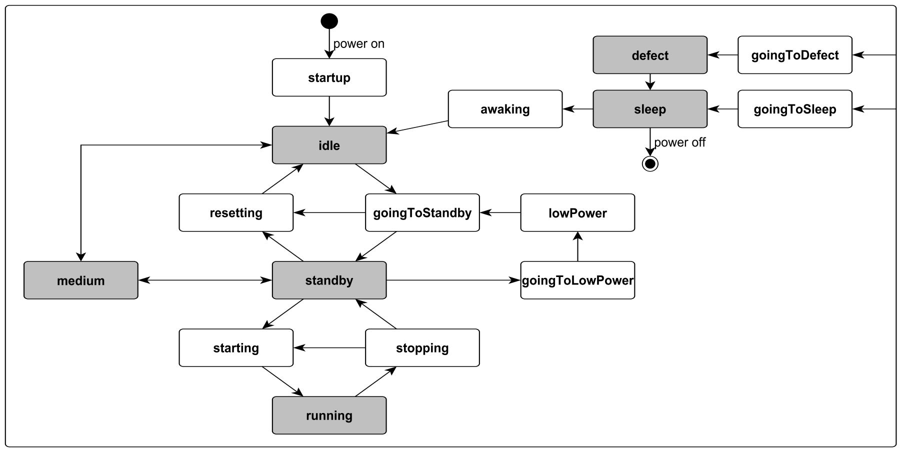

Fig. 4. Top states and transitions of the ESM

The `idle` status indicates that the engine has started up but that it is still *cold* (uncontrolled temperature). The `standby` status indicates that the engine is *warm* and ready for printing or scanning. The `running` status indicates that the engine is printing or scanning. The transitions from the overarching state to the `goingToSleep` and `goingToDefect` states indicate that it is possible to move to the `sleep` or `defect` status from any state. In some cases it is possible to awake from sleep status, in other cases the main power is turned off. The `medium` status is designed for diagnostics. In this status the functions can each be in a different status. For example one function is in standby status while another function is in idle status.

The statechart diagram in Figure 4 may seem simple, but it hides many details. Each of the states has up to 5 nested states. In total there are 70 states that do not have further nested states. The C++ code contained in the actions of the transitions is in some cases non-trivial. The possibility to transition from any state to the sleep or defect state also complicates the learning.

3 Learning the ESM

In order to learn a model of the ESM, we connected it to LearnLib [29], a state-of-the-art tool for learning Mealy machines developed at the University of Dortmund. A *Mealy machine* is a tuple $M = \langle I, O, Q, q_0, \delta, \lambda \rangle$, where I is a finite set of input symbols, O is a finite set of output symbols, Q is a finite set of states, $q_0 \in Q$ is an initial state, $\delta : Q \times I \to Q$ is a transition function, and $\lambda : Q \times I \to O$ is an output function. The behavior of a Mealy machine is *deterministic,* in the sense that the outputs are fully determined by the inputs. Functions δ and λ are extended to accept sequences in the standard way. We say that Mealy machines $M = \langle I, O, Q, q_0, \delta, \lambda \rangle$ and $M' = \langle I', O', Q', q'_0, \delta', \lambda' \rangle$ are *equivalent* if they generate an identical sequence of outputs for every sequence of inputs, that is, if $I = I'$ and, for all $w \in I^*$, $\lambda(q_0, w) = \lambda'(q'_0, w)$. If the behavior of an SUT is described by a Mealy machine M then the task of LearnLib is to learn a Mealy machine M' that is equivalent to M.

3.1 Experimental set-up

A clear interface to the ESM has been defined in RRRT. The ESM defines ports from which it receives a predefined set of inputs and to which it can send a predefined set of outputs. However, this interface can only be used within RRRT. In order to communicate with the LearnLib software a TCP connection was set up. An extra capsule was created in RRRT which connects to the ports defined by the ESM. This capsule opened a TCP connection to LearnLib. Inputs and outputs are translated to and from a string format and sent over the connection. Before each membership query, the learner needs to bring the SUT back to its initial state. This means that LearnLib needs a way to reset the SUT.

Some inputs and outputs sent to and from the ESM carry parameters. These parameters are enumerations of statuses, or integers bounded by the number of functions connected to the ESM. Currently LearnLib cannot handle inputs with parameters; therefore, we introduced a separate input action for every parameter value. Based on domain knowledge and discussions with the Océ engineers, we could group some of these inputs together and reduce the total number of inputs. When learning the ESM using one function, 83 concrete inputs are grouped into four abstract inputs. When using two functions, 126 concrete inputs can be grouped. When an abstract input needs to be sent to the ESM, one concrete input of the represented group is randomly selected, as in the approach of [2]. This is a valid abstraction because all the inputs in the group have exactly the same behavior in any state of the ESM. No other abstractions were found during the research. After the inputs are grouped a total of 77 inputs remain when learning the ESM using 1 function, and 105 inputs remain when using 2 functions.

It was not immediately obvious how to model the ESM by a Mealy machine, since some inputs trigger no output, whereas other inputs trigger several outputs. In order to resolve this, we benefitted from the run-to-completion execution model used in RRRT. Whenever an input is sent all the outputs are collected until quiescence is detected. Next all the outputs are concatenated and are sent to LearnLib as a single aggregated output. In model-based testing, quiescence is usually detected by waiting for a fixed timeout period. However, this causes the system to be mostly idle while waiting for the timeout, which is inefficient. In order to detect quiescence faster, we exploited the run-to-completion execution model used by RRRT: we modified the ESM to respond to a new low-priority test input with a (single) special output. This test input is sent after each normal input. Only after the normal input is processed and all the generated outputs have been sent, the test input is processed and the special output is generated; upon its reception, quiescence can be detected immediately and reliably.

3.2 Test selection strategies

In the ESM case study the most challenging problem was finding counterexamples for the hypotheses constructed during learning.

LearnLib implements several algorithms for conformance testing, one of which is a random walk algorithm. The random walk algorithm works by first selecting the length of the test query according to a geometric distribution, cut off at a fixed upper bound. Each of the input symbols in the test query is then randomly selected from the input alphabet I from a uniform distribution. In order to find counterexamples, a specific sequence of input symbols is needed to arrive at the state in the SUT that differentiates it from the hypothesis. The upper bound for the size of this search space is $|I|^n$ where $|I|$ is the size of the input alphabet used, and n the length of the counterexample that needs to be found. If this sequence is long the chance of finding it is small. Because the ESM has many different input symbols to choose from, finding the correct one is hard. When learning the ESM with 1 function there are 77 possible input symbols. If for example the length of the counterexample needs to be at least 6 inputs to identify a certain state, then the upper bound on the number of test queries would be around 2×10^{11}. An average test query takes around 1 ms, so it would take about 7 years to execute these test queries.

Augmented DS-method. In order to reduce the number of tests, Chow [10] and Vasilevskii [38] pioneered the so called W-method. In their framework a test query consists of a prefix p bringing the SUT to a specific state, a (random) middle part m and a suffix s assuring that the SUT is in the appropriate state. This results in a test suite of the form $PI^{\leq k}W$, where P is a set of (shortest) access sequences, $I^{\leq k}$ the set of all sequences of length at most k, and W is a characterization set. Classically, this characterization set is constructed by taking the set of all (pairwise) separating sequences. For $k = 1$ this test suite is complete in the sense that if the SUT passes all tests, then either the SUT is equivalent to the specification or the SUT has strictly more states than the specification. By increasing k we can check additional states.

We tried using the W-method as implemented by LearnLib to find counterexamples. The generated test suite, however, was still too big in our learning context. Fujiwara et al [14] observed that it is possible to let the set W depend on the state the SUT is supposed to be. This allows us to only take a subset of W which is relevant for a specific state. This slightly reduces the test suite without losing the power of the full test suite. This method is known as the Wp-method. More importantly, this observation allows for generalizations where we can carefully pick the suffixes.

In the presence of an (adaptive) distinguishing sequence one can take W to be a single suffix, greatly reducing the test suite. Lee and Yannakakis [25] describe an algorithm (which we will refer to as the LY algorithm) to efficiently construct this sequence, if it exists. In our case, unfortunately, most hypotheses did not enjoy existence of an adaptive distinguishing sequence. In these cases the incomplete result from the LY algorithm still contained a lot of information which we augmented by pairwise separating sequences.

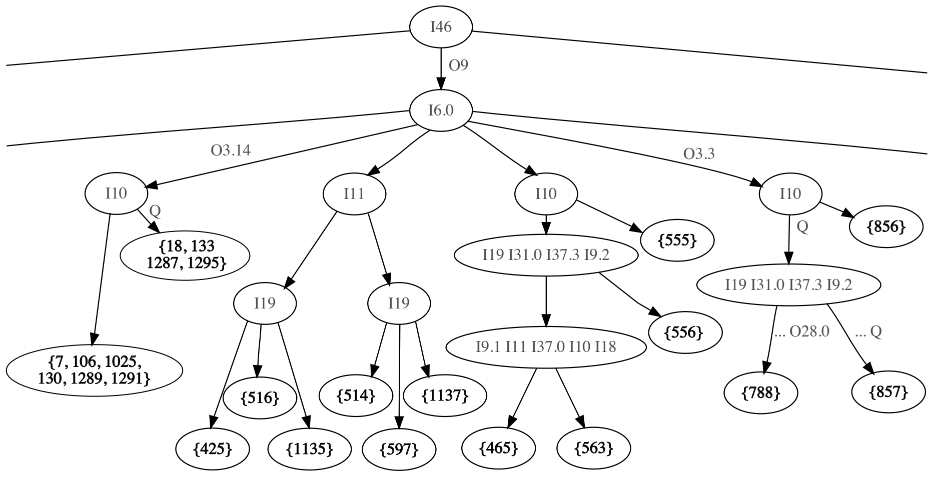

Fig. 5. A small part of an incomplete distinguishing sequence as produced by the LY algorithm. Leaves contain a set of possible initial states, inner nodes have input sequences and edges correspond to different output symbols (of which we only drew some), where Q stands for quiescence.

As an example we show an incomplete adaptive distinguishing sequence for one of the hypothesis in Figure 5. When we apply the input sequence I46 I6.0 I10 I19 I31.0 I37.3 I9.2 and observe outputs O9 O3.3 Q ... O28.0, we know for sure that the SUT was in state 788. Unfortunately not all paths lead to a singleton set. When for instance we apply the sequence I46 I6.0 I10 and observe the outputs O9 O3.14 Q, we know for sure that the SUT was in one of the states 18, 133, 1287 or 1295. In these cases we have to perform more experiments and we resort to pairwise separating sequences.

We note that this augmented DS-method is in the worst case not any better than the classical Wp-method. In our case, however, it greatly reduced the test suites.

Once we have our set of suffixes, which we call Z now, our test algorithm works as follows. The algorithm first exhausts the set $PI^{\leq 1}Z$. If this does not provide a counterexample, we will randomly pick test queries from PI^2I^*Z, where the algorithm samples uniformly from P, I^2 and Z (if Z contains more that 1 sequence for the supposed state) and with a geometric distribution on I^*.

Subalphabet selection. Using the above method the algorithm still failed to learn the ESM. By looking at the RRRT-based model we were able to see why the algorithm failed to learn. In the initialization phase, the controller gives exceptional behavior when providing a certain input eight times consecutively. Of course such a sequence is hard to find in the above testing method. With this knowledge we could construct a single counterexample by hand by which means the algorithm was able to learn the ESM.

In order to automate this process, we defined a subalphabet of actions that are important during the initialization phase of the controller. This subalphabet will be used a bit more often than the full alphabet. We do this as follows. We start testing with the alphabet which provided a counterexample for the previous hypothesis (for the first hypothesis we take the subalphabet). If no counterexample can be found within a specified query bound, then we repeat with the next alphabet. If both alphabets do not produce a counterexample within the bound, the bound is increased by some factor and we repeat all. This method only marginally increases the number of tests. But it did find the right counterexample we first had to construct by hand.

3.3 Results

Using the learning set-up discussed in Section 3.1 and the test selection strategies discussed in Section 3.2, a model of the ESM using 1 function could be learned. After an additional eight hours of testing no counterexample was found and the experiment was stopped. The following list gives the most important statistics gathered during the learning:

- The learned model has 3.410 states.
- Altogether, 114 hypotheses were generated.
- The time needed for learning the final hypothesis was 8 hours, 26 minutes, and 19 seconds.
- 29.933.643 membership queries were required, with on average 35,77 inputs per query.
- 30.629.711 test queries were required, with on average 29,06 inputs per query.

4 Verification

To verify the correctness of the model that was learned using LearnLib, we checked its equivalence with a model that was generated directly from the code.

4.1 Approach

As mentioned already, the ESM has been implemented using Rational Rose RealTime (RRRT). Thus a statechart representation of the ESM is available. However, we have not been able to find a tool that translates RRRT models to Mealy machines, allowing us to compare the RRRT-based model of the ESM with the learned model. We considered several formalisms and tools that were proposed in the literature to flatten statecharts to state machines. The first one was a tool for hierarchical timed automata (HTA) [11]. However, we found it hard to translate the output of this tool, a network of Uppaal timed automata, to a Mealy machine that could be compared to the learned model. The second tool that we considered has been developed by Hansen et al. [18]. This tool misses some essential features, for example the ability to assign new values to state variables on transitions. Finally, we considered a formalism called object-oriented action systems (OOAS) [23], but no tools to use this could be found.

In the end we decided to implement the required model transformations ourselves. Figure 6 displays the different formats for representing models that we used and the transformations between those formats. We used the bisimula-

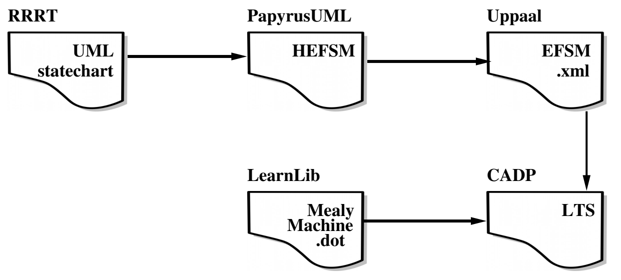

Fig. 6. Formats for representing models and transformations between formats

tion checker of CADP [15] to check the equivalence of labeled transition system models in `.aut` format. The Mealy machine models learned by LearnLib are represented as `.dot` files. A small script converts these Mealy machines to labeled transition systems in `.aut` format. We used the Uppaal [6] tool as an editor for defining extended finite state machines (EFSM), represented as `.xml` files. A script developed in the ITALIA project (`http://www.italia.cs.ru.nl/`) converts these EFSM models to LOTOS, and then CADP takes care of the conversion from LOTOS to the `.aut` format.

The Uppaal syntax is not sufficiently expressive to directly encode the RRRT definition of the ESM, since this definition makes heavy use of UML [30] concepts such as state hierarchy and transitions from composite states, concepts which are

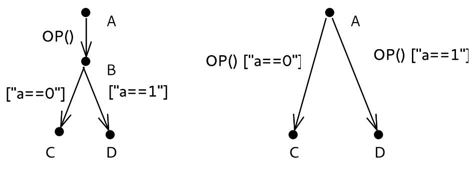

Fig. 7. Example of empty transition transformation. On the left the original version. On the right the transformed version

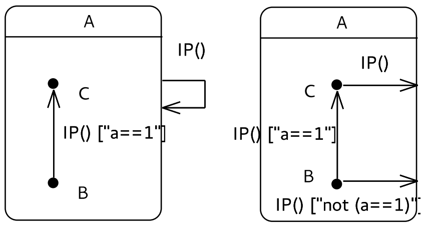

Fig. 8. Example of supertransition transformation. On the left the original version. On the right the transformed version

not present in Uppaal. Using Uppaal would force us to duplicate many transitions and states.

We decided to manually create an intermediate hierarchical EFSM (HEFSM) model using the UML drawing tool PapyrusUML [24]. The HEFSM model closely resembles the RRRT UML model, but many elements used in UML state machines are left out because they are not needed for modeling the ESM and complicate the transformation process.

4.2 Model transformations

We explain the transformation from the HEFSM model to the EFSM model using examples. The transformation is divided into five steps, which are executed in order: (1) combine transitions without input or output signal, (2) transform supertransitions, (3) transform internal transitions, (4) add input signals that do not generate an output, and (5) replace invocations of the next function.

1. Empty transitions. In order to make the model more readable and to make it easy to model `if` and `switch` statements in the C++ code the HEFSM model allows for transitions without a signal. These transitions are called *empty* transitions. An empty transition can still contain a guard and an assignment. However these kinds of transitions are only allowed on states that only contain empty outgoing transitions. This was done to make the transformation easy and the model easy to read.

In order to transform a state with empty transitions all the incoming and outgoing transitions are collected. For each combination of incoming transition a and outgoing transition b a new transition c is created with the source of a as source and the target of b as target. The guard for transition c evaluates to true if and only if the guard of a and b both evaluate to true. The assignment of c is the concatenation of the assignment of a and b. The signal of c will be the signal of a because b cannot have a signal. Once all the new transitions are created all the states with empty transitions are removed together with all their incoming and outgoing transitions.

Figure 7 shows an example model with empty transitions and its transformed version. Each of the incoming transitions from the state `B` is combined with each of the outgoing transitions. This results into two new transitions. The old transitions and state `B` are removed.

2. Supertransitions. The RRRT model of the ESM contains many transitions originating from a composite state. Informally, these *supertransitions* can be taken in in each of the substates of the composite state if the guard evaluates to true. In order to model the ESM as closely as possible, supertransitions are also supported in the HEFSM model.

In RRRT transitions are evaluated from bottom to top. This means that first the transitions from the leaf state are considered, then transitions from its parent state and then from its parent's parent state, etc. Once a transition for which the guard evaluates to true and the correct signal has been found it is taken. When flattening the statechart, we modified the guards of supertransitions to ensure the correct priorities.

Figure 8 shows an example model with supertransitions and its transformed version. The supertransition from state `A` can be taken at each of `A`'s leaf states `B` and `C`. The transformation removes the original supertransition and creates a new transition at states `B` and `C` using the same target state. For leaf state `C` this is easy because it does not contain a transition with the input signal `IP`. In state `B` the transition to state `C` would be taken if a signal `IP` was processed and the state variable `a` equals 1. The supertransition can only be taken if the other transition cannot be taken. This is why the negation of other the guard is added to the new transition. If the original supertransition is an internal transition the model needs further transformation after this transformation. This is described in the next paragraph. If the original supertransition is not an internal transition the new transitions will have the initial state of `A` as target.

3. Internal transitions. The ESM model also makes use of *internal transitions* in RRRT. Using such a transition the current state does not change. If such a transition is defined on a composite state it can be taken from all of the substates and return to the same leaf state it originated from. If defined on a composite state it is thus also a supertransition. This is also possible in the HEFSM model. In order to transform an internal transition it is first seen as a supertransition and the above transformation is applied. Then the target of the transition is simply set to the leaf state it originates from. An example can be seen in Figure 8. If the supertransition from state `A` is also defined to be an internal transition the transformed version on the right would need another transformation. The new transitions that now have the target state `A` would be transformed to have the same target state as their current source state.

4. Quiescent transitions. In order to reduce the number of transitions in the HEFSM model quiescent transitions are added automatically. For every state all the transitions for each signal are collected in a set T. A new self transition a is added for each signal. The guard for transition a evaluates to true if and only if none of the guards of the transactions in T evaluates to true. This makes the HEFSM input enabled without having to specify all the transitions.

5. The next function. In RRRT it is possible to write the guard and assignment in C++ code. It is thus possible that the value of a variable changes while an input signal is processed. In the HEFSM however all the assignments only take effect after the input signal is processed. In order to simulate this behavior the *next* function is used. This function takes a variable name and evaluates to the value of this variable after the transition.

4.3 Results

Figure 9 shows a visualization of the learned model that was generated using Gephi [5]. The large number of states (3.410) and transitions (262.570) makes it hard to visualize this model. Nevertheless, the visualization does provide insight in the behavior of the ESM. The three protrusions at the bottom of Figure 9 correspond to deadlocks in the model. These deadlocks are "error" states that are present in the ESM by design. According to the Océ engineers, the sequences of inputs that are needed to drive the ESM into these deadlock states will always be followed by a system power reset. The protrusion at the top right of the figure corresponds to the initialization phase of the ESM. This phase is performed only once and thus only transitions from the initialization cluster to the main body of states are present.

During the construction of the RRRT-based model, the ESM code was thoroughly inspected. This resulted in the discovery of missing behavior in one transition of the ESM code. An Océ software engineer confirmed that this behavior is a (minor) bug, which will be fixed. We have verified the equivalence of the learned model and the RRRT-based model by using CADP [15].

5 Conclusions and Future Work

Using an extension of the Lee & Yannakakis algorithm for adaptive distinguishing sequences [25], we succeeded to learn a Mealy machine model of a piece of widely used industrial control software. Our extension of Lee & Yannakakis' algorithm is rather obvious, but nevertheless it appears to be new. Preliminary evidence suggests that it outperforms existing conformance testing algorithms. We are currently performing experiments in which we compare the new algorithm with other test algorithms on a number of realistic benchmarks.

There are several possibilities for extending the ESM case study. To begin with, one could try to learn a model of the ESM with more than one function. Another interesting possibility would be to learn models of the EHM, ACM and other managers connected to the ESM. Using these models some of the properties discussed by Ploeger [31] could be verified at a more detailed level. We expect that the combination of LearnLib with the extended Lee & Yannakakis algorithm can be applied to learn models of many other software components.

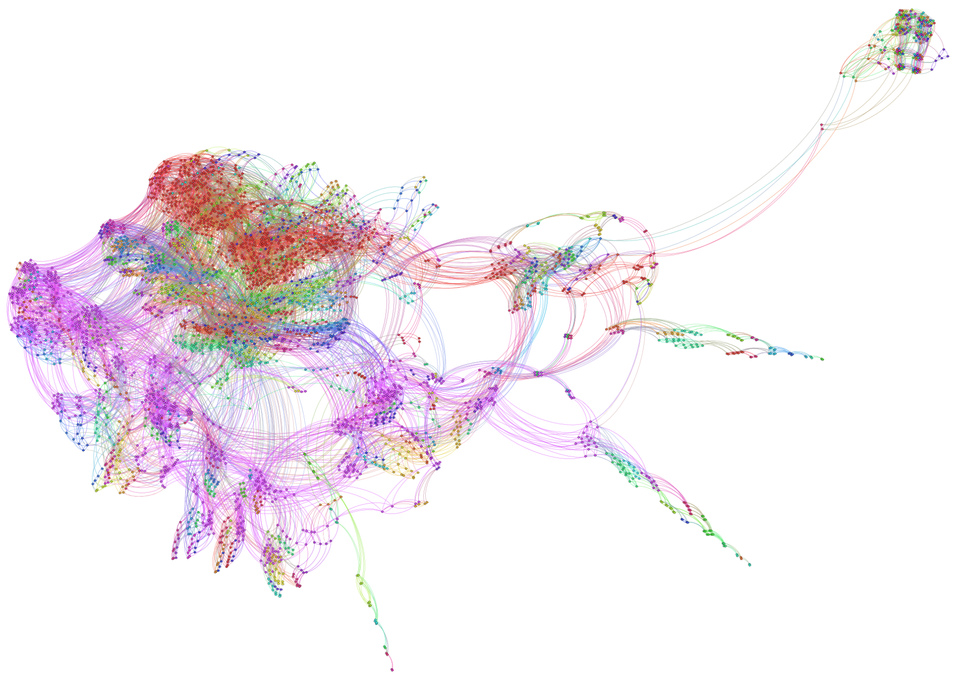

Fig. 9. Final model of the ESM.

In the specific case study described in this article, we know that our learning algorithm has succeeded to learn the correct model, since we established equivalence with a reference model that was constructed independently from the RRRT model of the ESM software. In the absence of a reference model, we can never guarantee that the actual system behavior conforms to a learned model. In order to deal with this problem, it is important to define metrics that quantify the difference (or distance) between a hypothesis and a correct model of the SUT, and to develop test generation algorithms that guarantee an upper bound on this difference. Preliminary work in this area is reported in [36].

Acknowledgments. We thank Lou Somers for suggesting the ESM case study and for his support of our research. Fides Aarts and Harco Kuppens helped us with the use of LearnLib and CADP, and Jan Tretmans gave useful feedback.

References

1. F. Aarts. *Tomte: Bridging the Gap between Active Learning and Real-World Systems.* PhD thesis, Radboud University Nijmegen, Oct. 2014.
2. F. Aarts, B. Jonsson, J. Uijen, and F. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design,* 46(1):1–41, 2015.

3. F. Aarts, H. Kuppens, G. Tretmans, F. Vaandrager, and S. Verwer. Improving active Mealy machine learning for protocol conformance testing. *Machine Learning,* 96(1–2):189–224, 2014.
4. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.,* 75(2):87–106, 1987.
5. M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks. In *ICWSM.* The AAAI Press, 2009.
6. G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *QEST,* pages 125–126. IEEE Computer Society, 2006.
7. T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In *FASE,* LNCS 3442, pages 175–189. Springer, 2005.
8. S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. *J. Log. Algebr. Meth. Program.,* 84(1):54–66, 2015.
9. C. Y. Cho, D. Babic, E. C. R. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. In *ACM Conference on Computer and Communications Security,* pages 426–439. ACM, 2010.
10. T. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.,* 4(3):178–187, 1978.
11. A. David, M. O. Möller, and W. Yi. Formal verification of UML statecharts with real-time extensions. In *FASE,* LNCS 2306, pages 218–232. Springer, 2002.
12. R. Eshuis, D. N. Jansen, and R. Wieringa. Requirements-level semantics and model checking of object-oriented statecharts. *Req. Engineering,* 7(4):243–263, 2002.
13. P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Learning fragments of the TCP network protocol. In *FMICS,* LNCS 8718, pages 78–93. Springer, Sept. 2014.
14. S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Tr. Softw. Eng.,* 17(6):591–603, 1991.
15. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In *TACAS,* LNCS 6605, pages 372–387. Springer, 2011.
16. B. Graaf and A. van Deursen. Model-driven consistency checking of behavioural specifications. In *MOMPES,* pages 115–126. IEEE Computer Society, 2007.
17. R. Groz, K. Li, A. Petrenko, and M. Shahbaz. Modular system verification by inference, testing and reachability analysis. In *FATES,* LNCS 5047, pages 216–233, 2008.
18. H. H. Hansen, J. Ketema, B. Luttik, M. R. Mousavi, J. van de Pol, and O. M. dos Santos. Automated verification of executable UML models. In *FMCO,* LNCS 6957, pages 225–250. Springer, 2010.
19. C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars.* Cambridge University Press, Apr. 2010.
20. F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *VMCAI,* LNCS 7148, pages 251–266. Springer, 2012.
21. F. Howar, B. Steffen, and M. Merten. Automata learning with automated alphabet abstraction refinement. In *VMCAI,* LNCS 6538, pages 263–277. Springer, 2011.
22. H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *CAV,* LNCS 2725, pages 315–327. Springer, 2003.
23. W. Krenn, R. Schlick, and B. K. Aichernig. Mapping UML to labeled transition systems for test-case generation - a translation via object-oriented action systems. In *FMCO,* LNCS 6286, pages 186–207. Springer, 2009.

24. A. Lanusse, Y. Tanguy, H. Espinoza, C. Mraidha, S. Gerard, P. Tessier, R. Schnekenburger, H. Dubois, and F. Terrier. Papyrus UML: an open source toolset for MDA. In *Model-Driven Architecture,* page 1, 2009.
25. D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Trans. Comput.,* 43(3):306–320, 1994.
26. M. Leucker. Learning meets verification. In *FMCO,* LNCS 4709, pages 127–151. Springer, 2006.
27. K. Li, R. Groz, and M. Shahbaz. Integration testing of distributed components based on learning parameterized I/O models. In *FORTE,* LNCS 4229, pages 436–450, 2006.
28. M. Merten, F. Howar, B. Steffen, S. Cassel, and B. Jonsson. Demonstrating learning of register automata. In *TACAS,* LNCS 7214, pages 466–471. Springer, 2012.
29. M. Merten, B. Steffen, F. Howar, and T. Margaria. Next generation LearnLib. In *TACAS,* LNCS 6605, pages 220–223. Springer, 2011.
30. Object Management Group (OMG). Unified modeling language specification: Version 2, revised final adopted specification. `http://www.uml.org/\#UML2.0`, 2004.
31. B. Ploeger. Analysis of concurrent state machines in embedded copier software. Master's thesis, Eindhoven University of Technology, Aug. 2005.
32. H. Raffelt, M. Merten, B. Steffen, and T. Margaria. Dynamic testing via automata learning. *STTT,* 11(4):307–324, 2009.
33. H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT,* 11(5):393–407, 2009.
34. B. Selic, G. Gullekson, and P.T. Ward. *Real-time Object-oriented Modeling.* John Wiley & Sons, Inc., New York, NY, USA, 1994.
35. W. Smeenk. *Applying Automata Learning to Complex Industrial Software.* Master thesis, Radboud University Nijmegen, Sept. 2012.
36. R. Smetsers, M. Volpato, F.W. Vaandrager, and S. Verwer. Bigger is not always better: on the quality of hypotheses in active automata learning. In *ICGI*, *JMLR Proceedings 34*, pages 167–181. 2014.
37. B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In *SFM,* LNCS 6659, pages 256–296. Springer, 2011.
38. M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetics and Systems Analysis,* 9(4):653–665, 1973.