Pitfalls in Applying Model Learning to Industrial Legacy Software

Omar al Duhaiby[1(✉)], Arjan Mooij[2], Hans van Wezep[3], and Jan Friso Groote[1]

[1] Eindhoven University of Technology, 5612AZ Eindhoven, The Netherlands
{o.alduhaiby,j.f.groote}@tue.nl
[2] ESI (TNO), Eindhoven, The Netherlands
arjan.mooij@tno.nl
[3] Philips Healthcare, Best, The Netherlands
hans.van.wezep@philips.com

Abstract. Maintaining legacy software is one of the most common struggles of the software industry, being costly yet essential. We tackle that problem by providing better understanding of software by extracting behavioural models using the *model learning* technique. The used technique interacts with a running component and extracts abstract models that would help developers make better informed decisions. As promising in theory, as slippery in application it is, however. This report describes our experience in applying model learning to legacy software, and aims to prepare the newcomer for what shady pitfalls lie therein as well as provide the seasoned researcher with concrete cases and open problems. We narrate our experience in analysing certain legacy components at Philips Healthcare describing challenges faced, solutions implemented, and lessons learned.

Keywords: Model learning · Active learning · Legacy software

1 Introduction

As software evolves over years and decades, its very architecture starts to change. And with the original developers unavailable anymore, and the documentation outdated, it becomes increasingly difficult to maintain that software. That is what legacy software is [19]. Not only does maintenance become a more pressing matter, but also a costly and even risky endeavor. As legacy software that has been running a business successfully for decades, refactoring it without complete understanding might lead to unexpected and severe impediments. To achieve that level of understanding, different techniques have been deployed to analyse legacy software, such as process mining [1], static code analysis [9], and our method of choice, active model learning [19].

These techniques aim to model legacy software. With accurate readable abstract models, developers can improve the software in less time, discover hidden behaviour, and generate documentation. Active model learning is a technique that aims to build a finite-state model of a system from observed input and output [19].

© Springer Nature Switzerland AG 2018
T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11247, pp. 121–138, 2018.
https://doi.org/10.1007/978-3-030-03427-6_13

In practice, however, active model learning is not at all an easily realisable endeavour. As many success stories there are [2,17,18], as many pitfalls we faced in our experience applying it—pitfalls such as dealing with obscure proprietary interfaces, unclear code, and lacking documentation; ensuring the accuracy of the learning outcome; interpreting unexpected behaviour; avoiding state space explosion; and a variety of technical problems. Our contribution lies in

- identifying the pitfalls in applying active model learning,
- detailing how they manifested in our industrial setting,
- providing lessons on how to deal with them,
- and suggesting future research directions.

This work is based on our experience with applying model learning to parts of the X-ray imaging software at Philips Healthcare. We use LearnLib [7] as the core learning engine combined with necessary complementary software that we detail in Sect. 2. We first explain the theory in a simple manner, then describe our target system and the learning setup. Section 3 lays out our main contribution describing the practical experience through lessons learned and challenges faced. Then in Sect. 4, we reflect on the practical challenges with suggested future research directions and open questions. We finally conclude with Sect. 5.

2 Background

In this section, we describe the learning method and the component being learned as well as a few relevant technicalities.

2.1 LearnLib, L* and Mealy Machines

As mentioned before, LearnLib [7] is our learning tool. LearnLib houses a few learning algorithms, the most prominent of which is L*, first introduced by Dana Angluin in 1986 [3]. L* learns regular languages by asking whether certain strings belong to that language. This type of querying is not suited for reactive systems such as the ones we mostly face in the industry. To tackle that, Niese in [15] introduced a variant of L* called L^*_{Mealy} which outputs a Mealy machine such as that shown in Fig. 2.

We shall explain some basic concepts, followed by the algorithm and then Mealy machines. Refer to Fig. 1 showing the learning setup. The learner is internally composed of a model builder and a model tester. We assume that the *System under Learning* (*SuL*) responds to every action. The learner can send *actions* as input to the SuL and receive *responses* as output, making a sequence of action/response pairs, called a *trace*. The learner has the ability to *reset* the SuL back to its initial state and thus terminate the current trace. The learning algorithm can be summarised in the following iterative steps:

1. Building: the builder sends/receives action/response traces to/from the SuL, each trace followed by a SuL reset, to build a hypothesis model (as a Mealy machine).

2. Testing: the tester tests the hypothesis model against the SuL similarly through action/response traces.
3. Feedback: if the tester discovers an action/response pair that is not consistent with the hypothesis model, then return to step 1 (building) while using that pair as a counter example to refine further queries; otherwise, the model is verified and the learning is complete.

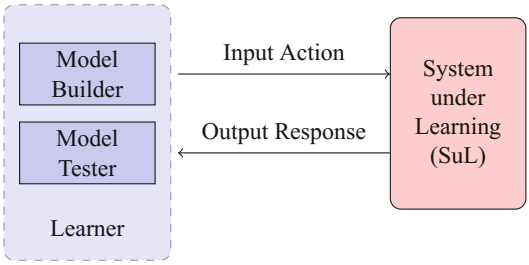

Fig. 1. High-level overview of the learning setup.

This technique requires the input actions to be defined beforehand as well as the SuL's reset routine. We call the set of all admissible input actions the *input alphabet* or the *action set*. The set of all responses is similarly called the *output alphabet*. It is not strictly necessary for the output alphabet to be known beforehand.

Refer to Fig. 2 showing an example Mealy machine produced by this learning technique and describing the following simple behaviour. Suppose that the SuL admits actions: *Init(X)*, which initialises an object X; and *Use(X)*, which uses that object. Both actions give a *Succ* or *Fail* response. The simple behaviour shown in this model is that an object X cannot be used successfully before being initialised.

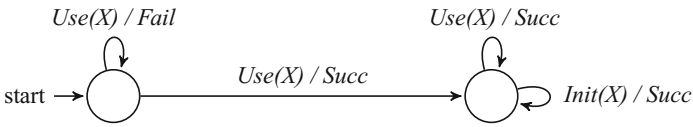

Fig. 2. Example Mealy machine.

2.2 Description of the Legacy Component

Our case study is a software driver for the electrical generator of an X-ray machine responsible for powering the X-ray tube. The driver sets the correct

parameters for electrical requirements depending on the type of X-ray exposure desired. The driver also monitors the hardware's sensors for any necessary intervention. It is called GS for Generator Service. The reasons it was chosen for study were: (1) its reasonable size of 15,000 effective lines of code as measured by TICS[1], (2) the expectation that it can be adequately described by a state machine, (3) that it had undergone recent refactoring which meant that more knowledge was available, (4) an interest in revealing any missed refactoring opportunities or missed behaviour, and (5) that it was relatively easy to isolate and communicate with.

The first step of studying this component as a black box was gaining as much knowledge as possible about its interfaces and a bit about its inside architecture. The ease of isolating it is resembled by the fact that it had only two interfaces as seen in Fig. 3 one interface to the application layer (shown on the top side), and another to the hardware layer (shown on the bottom side). We may refer to them simply as *top* and *bottom* layers, respectively. The bottom interface communicating with the hardware requires a certain HW interface adapter.

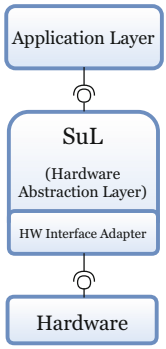

Fig. 3. Interface schematic of the GS component.

The behaviour we aimed to learn was essentially the one shown in Fig. 2, where the component needs to be properly initialised in order to be used. The reason behind learning such seemingly simple behaviour was (1) to confirm that the learned behaviour is equivalent to our expectation (Sect. 3.4 uncovers this result), and (2) to explore the result of learning with a lower-level action set, discussed in Sect. 3.2. Our Experiments showed that the initialisation procedure of our legacy component, as simple as it seems, is not straightforward and would not be learned smoothly. In fact, all the experiments of this paper are merely exploring the GS's initialisation procedure.

Before we take the reader through our model learning experience addressing individual learning experiments, we dive into the practical setup of the learning environment.

[1] TICS (TIOBE Software Quality Framework; www.tiobe.com/tics/tics-framework).

2.3 Practical Learning Setup

The first step of setting up the learning experiment is to determine the interfaces on the SuL, and then, for each interface, to determine the following:

- The input alphabet, which in practice would be the list of functions calls provided by the interface.
- Means for sending actions to the SuL.
- Means for receiving or retrieving responses from the SuL.

We left out the output alphabet as we mentioned earlier that pre-defining it is not absolutely necessary. The only requirement is reading the output regardless of its type; the means of reading should be generic enough.

Note from Fig. 3 that we have two interfaces. This means that we need to identify inputs and outputs through each of these two interfaces. On the top side lies the application layer. In the real environment, commands are sent from the application layer into the GS, which can result in output through any of the two interfaces. We needed to replicate exactly that in order to send our actions. So, we wrote our own component that acts as the application layer. We call that component the *action executer* and it is part of the learning driver.

Note also from Fig. 3 that the bottom component is a hardware device. In our setup, communicating with that device involved many low-level details that were not possible to replicate through the action executer. Therefore we chose to use an abstraction that is the hardware interface adapter.

In such a case, the natural question is how to run the SuL without the actual hardware. Luckily, we had a test environment that provided the answer, namely that the aforementioned hardware interface adapter supported a testing mode where it would also act as a replacement for the real hardware through a test stub. This posed a peculiar case discussed in Sect. 3.5. The action executer communicates with that stub.

Refer to Fig. 4 showing the implemented learning setup. It essentially shows the learning driver being hooked up to the SuL from Fig. 3. The action executer connects to the SuL on the aforementioned interfaces and is responsible for the following:

- Communicating with the Learner. It receives input actions from the learner as strings and sends back responses as strings over a TCP socket connection.
- Translating input actions from strings into executable code. It can call functions that the SuL provides on its interfaces.
- Receiving the relevant response, for each action, from the SuL and translating it into a string.
- Resetting the SuL.

On the left side of Fig. 4 lies the LearnLib client which uses the LearnLib library and is responsible for setting parameters of LearnLib, e.g. selecting a learning algorithm and a testing algorithm, setting testing parameters, as well as fetching the input alphabet, and producing graphs of learned automata. The

source code of the LearnLib client is available online[2]. The LearnLib client and the action executer combined make up what we call the learning driver, shown in Fig. 4, whose main role as a whole is to provide a wrapper around the SuL that acts as Mealy view for LearnLib.

For more information on the requirements and implementation of a learning driver, we refer the reader to the work of Merten et al. [14].

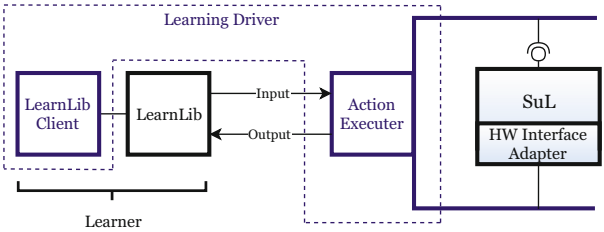

Fig. 4. Implemented learning setup.

3 Lessons Learned from the Case Study

This section details our experience with the case study through individual lessons learned.

3.1 Utilising the Test Environment

The availability of a test environment for the SuL is very valuable for the purpose of model learning. In our case, unit tests are conducted using the CppUnit testing framework [12] and they initially provided insight on how to run the GS without its hardware as mentioned in Sect. 2.3. In a similar use case, Hungar et al. [10] utilised an integrated test environment in their learning setup that is described in detail by Niese et al. in [16]. A major benefit of building such a test environment is having tests of different levels of abstraction from as high as the whole SuL to as low as the lowest subcomponents. In our case, this was achievable with manual work of dissecting the tests into smaller units as discussed in Sect. 3.3. An additional benefit we gained was the ability to extract information about the SuL's interface from individual tests. This point is significant in the case of legacy software, and particularly in our case, where APIs were not well documented. Luckily, the tests had high coverage and their functions acted as an abstraction layer on the SuL's lower-level interface. Moreover, the tests were written in a rather consistent fashion and were quite readable.

Unit tests have a linear structure consisting of three stages: (1) setting preconditions, (2) testing the outcome (usually with an assert statement), and (3)

[2] https://gitlab.science.ru.nl/ramonjanssen/basic-learning.

resetting the environment. This can be mapped to the structure of learning experiments, explained in Sect. 2.3, in the following way. From stage 1, we can extract two things: the initialisation actions and the input actions. It is worth mentioning that classifying a test's pre-conditions into initialisation actions and input actions is a matter of experiment design that we shall revisit in Sect. 3.2. Next, from stage 2, we extract the relevant output which must be configured as the SuL's response to the input actions from stage 1 (to satisfy the Mealy view). Finally, from stage 3, we learn how to undo the initialisation actions of stage 1. Once again, by looking at multiple tests, we extract the requirements for a global (SuL) reset which we configure the learner to perform at the end of each trace. Table 1 summarises this mapping.

Table 1. Mapping elements from unit tests into the learning experiment.

Test structure	Information extracted
Pre-conditions	Initialisation actions
	Input actions
Outcome test	SuL response/output
Reset	Uninitialisation
	Global (SuL) reset

Utilising the test environment is certainly a convenience, but we need to keep a few issues in mind:

- The tests may not cover every possible action. Functions and special arguments that are never used in the tests must be extracted from the SuL's code.
- The tests abstract from certain details. We may fine-tune our level of abstraction as covered in Sect. 3.2.
- Tests are context-specific. We may want to combine different contexts to conduct experiments with more actions. But is learning with more actions always a better idea? We address this question in Sect. 3.4.
- The nature of the learning process—where actions are executed in different possible orders and with different frequencies—can often lead to traces that are not covered in tests or ones that are not even achievable in normal use. A test environment is probably not built to handle such scenarios and will thus cause errors. We allude to this issue in Sect. 4.2.
- One more issue that is rather specific to our environment is the test stub attached to the SuL and the particular challenge of separating the two in regards of actions as well as learned models. This issue is discussed in Sect. 3.5.

We utilise the test environment in experiments of the following subsections, and reuse excerpts of code directly taken from unit tests as we address the challenges mentioned above.

3.2 Fine-Tuning the Level of Abstraction of the Alphabet

When composing an alphabet from code, a level of abstraction must be determined. Consider the actions of the model in Fig. 2. Each of these actions encloses multiple lower level actions that contain lower-level details which we simply hide by choosing the higher level action set. This yields a more abstract and readable model yet linear and without much variety to explore. On the other hand, however, let us explore the result of choosing the lowest level of abstraction. Consider the code from Listing 1.

Listing 1: Top activates GS

```
1 GS = CreateInstance();
2 F = GS.GetFrontalFactory();
3 L = GS.GetLateralFactory();
4 FOp = F.GetOperationalInterface();
5 LOp = L.GetOperationalInterface();
6 FOp.Activate();
7 LOp.Activate();
```

This excerpt is a precondition for most tests in our environment. For the lowest level of abstraction, we chose to set each line of code as a single input action. So this is our fine-grained action set. We set the response for each action to be a simple success/failure check on the call. This setup yields the model in Fig. 5 where transition labels correspond to line numbers in the code, all shown transitions have the output *success* which is omitted, and failed calls are self-loop transitions which are also omitted. We can clearly see the interleaving pattern created by independent actions.

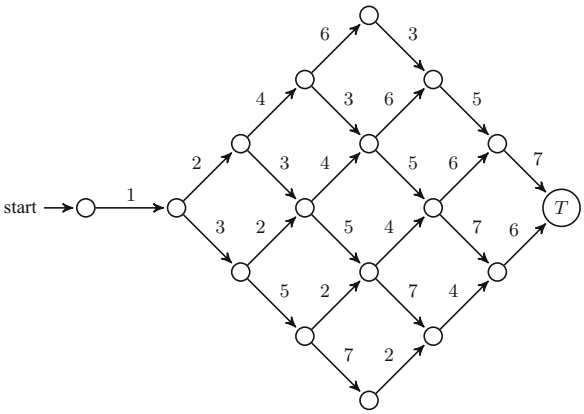

Fig. 5. Learned model of activation with fine-grained actions. Transition labels correspond to the line numbers in Listing 1.

The reason they are independent is because they activate independent components, and that can be done in arbitrary order. So, we can combine actions to yield a more readable model. Action 1 will remain the same and be labelled a_1, while the sequence of actions 2 to 7 will be combined into action a_{2-7}. Then we reach the model in Fig. 6. We simply eliminate interleaving through abstraction.

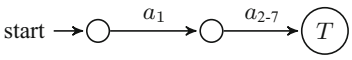

Fig. 6. The learned model of activation with coarse-grained actions.

Practical Results. We conducted two learning experiments, one with six input actions for learning one of the two independent components, and the other with 12 input actions for learning both components together. The first experiment took less than an hour and produced an eight-state model, and the second one lasted 6.5 h and produced a 64-state model. These numbers depend on many factors and are only given for the reader to get a sense of the cost of such an experiment.

Step-Wise Action Refinement. It is not possible to determine which actions are independent without either prior knowledge or experimentation. In the case where domain knowledge is not available, we resort to a technique called stepwise refinement [20]. It simply means we run the experiment with a minimal alphabet, and incrementally increase the size of the alphabet; as soon as interleaving is observed on two actions, such as 2 and 3 from Fig. 5, we abstract those into one single action, and so on. The downside of this technique is that earlier parts of the model will be learned again and again, which is clearly inefficient. Suppose we are only interested in behaviour that occurs at some state R onwards. Then to overcome the aforementioned inefficiency, we configured the learning driver such that every time the SuL is started, a certain action sequence is executed that transitions the state of the SuL to state R, which effectively makes the learning begin from state R. Currently, LearnLib does not have the feature of resuming learning from a certain state it has learned before. This is part of our future work as it will make stepwise refinement much more efficient.

The lesson we learn here is that the objective of a learning experiment decides which details to abstract. More fine-grained actions naturally yield more information in the learned model. On top of that, in this particular experiment, it provided a clue for one more property about the components labelled F and L (Listing 1), namely that they are symmetric, which we explore in the next subsection.

3.3 Exploiting Symmetry

Different patterns of symmetry are observed in software systems [6]. The pattern we refer to is when a certain component can be replaced by another one without

resulting in an observable change in behaviour. Refer to the experiment of the previous section and to Listing 1. We learn from the domain expert that the components labelled F and L are symmetric. So we conduct the same experiment but excluding actions of the L component and we find that the new action set suffices to reach a state we call T_F (Fig. 7) which marks the F component active and ready for executing further actions successfully.

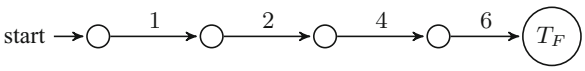

Fig. 7. Learned model of activation with actions of only one of the two symmetric components.

Thus we do not need to repeat the experiment for a certain component once we have already done so on a behaviourally equivalent one. This assumption of symmetry needs to be tested, however. So far, it is verified up to state T, but as we expand our alphabet to learn further parts of the software, we need to repeatedly verify that assumption. Doing so through matching traces from one component against the other is much more efficient than repeating the learning experiment altogether for the other assumed-symmetric component. In other words, the learned model of one component can be used as a hypothesis to be tested against the other symmetric model. We call this initial hypothesis a conjecture and we discuss it in Sect. 4.

So far, we learned that we can reduce our models by abstracting independent actions and by excluding one of two symmetric components from the action set. We continue on learning the next activation procedure and learning more lessons.

3.4 Faster Learning vs. Thorough Testing

Besides L^*_{Mealy} [15], LearnLib offers a faster variant of the L* algorithm, called TTT, introduced by Isberner [11]. Learning with TTT is more efficient simply because it produces a final model sooner. However, it goes through many more iterations and produces many more intermediate hypotheses, which requires more rigorous testing. Thus, optimising the learning process also requires optimising the testing algorithm.

Schuts et al. [17] provide experimental results on learning a small model with TTT combined with various testing algorithms. They conclude that TTT is faster than L* by a factor of 3 regardless of the testing method used. However, producing a correct model is as significant a concern, if not more significant, as speeding up the process. In our experiment, we contrasted between two learning algorithms, L* and TTT, and two testing methods, the W-method [5] and a simple random-walk test that LearnLib provides. The random walk tests random paths in the hypothesis against the SuL. It requires a maximum number of input actions and a probability of resetting the SuL after each action. The more

rigorous W-method requires a parameter that effectively sets an upper bound on the length of the tests. Therefore in both tests, an estimate on the size of the target model must be made. This is quite problematic for the reason made clear by the next experiment.

In this experiment, we have only two actions: *Activate*, which abstracts the SuL activation procedure detailed in Sect. 3.2; and *GetLogicalResource*, which returns a logical resource object to access the hardware. We shorten these two actions as A and B respectively. The output of both actions is an S or F response standing for success and failure respectively. The expected behaviour is exactly that shown in Fig. 2 (if *Init* and *Use* are renamed to A and B respectively) where action B's success depends on action A's success. With this expectation in mind, estimating an upper bound on the number of states to eight, four times larger than the expected model, should be rigorous enough. And indeed we get the expected two-state model. However, with the cheaper random walk testing method, we set the maximum number of actions to 1000, and discover that our previous hypothesis was false and that a more accurate model is the ten-state model of Fig. 8. The model shows that the action *GetLogicalResource* succeeds after *Activate* but calling it eight times fills a certain hidden buffer and causes any subsequent calls to fail, even though *Activate* still succeeds.

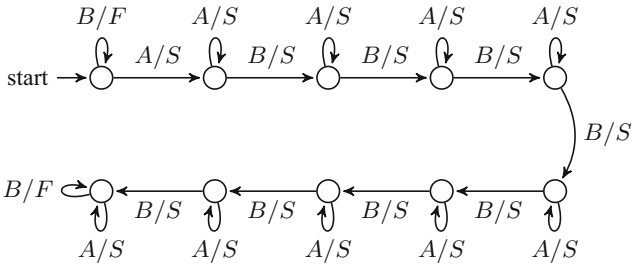

Fig. 8. A learned model showing the hidden buffer of size eight. Input actions are A: *Activate*, and B: *GetLogicalResource*; outputs are S: Success, and F: Fail.

Clearly, we would not want to discover such behaviour in a larger experiment. Such behaviour expands a two-state model to a ten-state model; and in a setup with one more input action, it expanded four states to 28 states. However, we would like to be aware of such behaviour and find a way to abstract from it. For this particular case, such a sequence will never be executed in practice and is therefore deemed uninteresting to learn. Not only that, but it is also expensive to learn and test, and therefore we would like to avoid observing it altogether. We call this an unduly complex model and we discuss it in Sect. 4.2.

Moreover, the bigger the action set, the lower the chance of discovering such behaviour. Thus it is wiser to experiment with a smaller action set before expanding.

We learned that faster learning comes with drawbacks. It comes at the expense of the accuracy of the learned model. We showed a case where not

only would faster learning yield a less accurate model, but even slower learning with wrong assumptions can do the same.

3.5 Utilising the Test Stub

Recall from Sect. 2.3 that the HW interface adapter shown in Fig. 4 supports a testing mode where a built-in test stub would act as a replacement for the real hardware. This subsection contains multiple experiences and multiple lessons in utilising the test stub. We start with a piece of code taken directly from a test case, Listing 2, a procedure called *ToStandby* which runs after the activation procedure of Listing 1.

Listing 2: ToStandby()

```
1 stub.NotifyGS(On);
2 stub.IsRequested(Idle);
3 stub.SetState(Idle);
4 stub.NotifyGS(Idle);
5 stub.IsRequested(Standby);
6 stub.SetState(Standby);
7 stub.NotifyGS(Standby);
```

The goal of the code is to activate the stub and make it reach a standby state. It shows us the proper order of guiding the bottom layer (the stub) to a Standby state. First, the stub notifies the GS that it is powered on and awaits a request to go to an *Idle* state. Once it receives that request, the stub's state is set to Idle, it notifies the GS of that, and awaits the request to go to *Standby*. Again, once it receives the request, the stub is set to Standby and it notifies the GS of that.

Note the calls to `IsRequested` on line 2 and line 5 marked in boldface. These two calls are blocking, i.e. they wait for the output which means we are forced to implement a timeout. In the real setting which uses actual hardware, such calls are asynchronous, but in a testing environment, we are forced to make them synchronous.

When forming the action set out of Listing 2, lack of domain knowledge forced us to take the crude method of making each line of code into a single input action, while configuring the output read to be the success or failure of that particular call; in case of the call `IsRequested`, we get an additional output which is also a boolean value. Additionally, some actions that were added to the setup such as `NotifyGS(Off)` are not part of this particular unit test but were extracted from other tests and from the SuL's source code.

Inseparable Components. The stub is part of HW interface adapter (Fig. 4) and is thus inseparable from the SuL. While learning the SuL, it is probable that part of the learned behaviour is due to the stub. This is not to say that the stub is an undesired component. On the contrary, it provides great benefit in

abstracting away all the obscure low-level communication details necessary in the real hardware connection. Eliminating the stub enforces a greater task which is to reverse engineer this low-level communication and incorporate it into our input and output alphabets, after which we can choose whether to learn it or to abstract away from it; the latter will, in turn, take us back to the situation we are currently in. Moreover, there is no accessible interface between the SuL and the interface adapter, which forces us to learn the combination of these two components rather than the SuL alone. This problem is revisited in Sect. 4.4.

Nondeterminism and Timing. Continuing with the stub experiment and the described alphabet, we discuss a certain problem we faced. As we ran the learner a few times with this setup, it started complaining about nondeterministic behaviour. It would report traces such as the following:

Listing 3: Trace showing nondeterminism

```
1 stub.NotifyGS(Idle);
2 stub.NotifyGS(Idle);
3 stub.NotifyGS(Standby);
4 stub.NotifyGS(On);
5 stub.SetState(Standby);
6 stub.SetState(Idle);
7 stub.SetState(Standby);
8 stub.IsRequested(Standby); -> True/False
```

The arrow at the last line indicates output. The learner here is complaining that the output for this trace is not deterministic, i.e., sometimes true and other times false. In such a setting where blocking is forced on an otherwise-asynchronous communication, we learn from Schuts et al. [17] that simply inserting time pauses after each message is a viable solution. The reason is that the SuL needs time to process and respond to messages. Through trial and error, we were able to determine the shortest pause duration necessary for a deterministic-output run. For this specific environment, the duration was 100 ms. A lesson learned here is that reported nondeterminism may not be so for as simple a reason as needing a time pause.

Expanding the Scope of the Output. A separate attempted solution to solve the nondeterminism problem was applying an abstraction on the actions by grouping them as follows:

```
1 stub.NotifyGS(On);          } — b1
2 stub.IsRequested(Idle);     }
3 stub.SetState(Idle);        }
4 stub.NotifyGS(Idle);        } — b2
5 stub.IsRequested(Standby);  }
6 stub.SetState(Standby);     } — b3
7 stub.NotifyGS(Standby);     }
```

This setup did eliminate the need for inserting time pauses, but it yielded neither new states nor new transitions in the learned model, which pointed our attention to the output. Note that even though we abstracted actions, the scope of the output has not changed. Thus, we decided to revert from the abstraction solution back to the time-pause solution, and in addition do the following: to read the output of both `IsRequested(Idle)` and `IsRequested(Standby)` after each action and remove these two calls from the action set. In other words, we moved them from the input alphabet to the output alphabet. We were able to read both outputs because they were stored as flags in the HW interface adapter. This yields the model in Fig. 9. Output flags are represented by a dash if read *false* and by the first initial of the flag name if read *true*. And to save space, the action names were shortened.

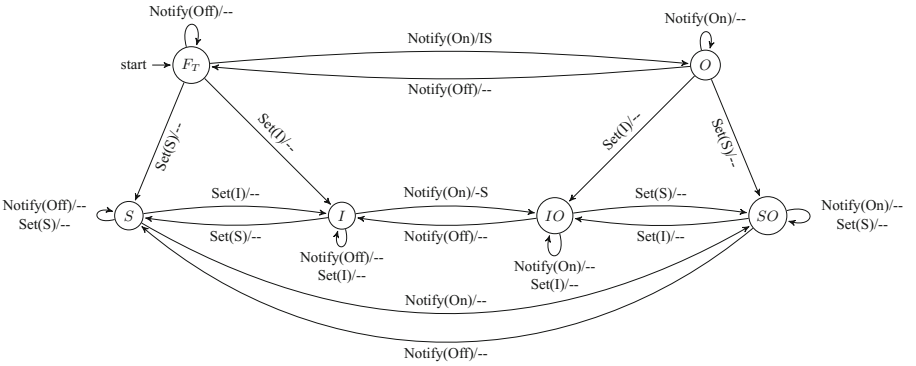

Fig. 9. The learned model of the *ToStandby* procedure with fine-grained actions and aggregated output.

Aggregating output and reading it globally revealed that both flags could be set as response to a single action, a fact that contradicts the implicit assumption of our previous output-reading setup. A more general benefit of aggregating output is that it removes the nondeterminism resulting from the common ambiguity of which output arrived first. The lesson learned here is that the scope of the output should be expanded, i.e. by reading more output, especially if at little or no extra cost.

The model in Fig. 9 also reveals that some actions, namely `NotifyGS(Idle)` and `NotifyGS(Standby)`, have no effect on either the output or the state, which is why they are omitted from the figure. This raises the question: do they cause no change at all; or is the change simply hidden from our view due to this peculiar test stub?

We still do not know the answer but the sure lesson is that in some cases we cannot effectively learn the isolated target system and we are forced to utilise auxiliary components such as the test stub.

4 Future Work

In this section we discuss future solutions to the challenges seen in this case study.

4.1 Learning with Conjectures

We have seen a case in Sect. 3.3 where we use the knowledge that two components are symmetric. To ensure the accuracy of the learned model, we would like to treat this piece of knowledge as an assumption or a conjecture. We were able to verify the symmetry by learning each of the two components separately and finding that they are strongly bisimilar. However, there are two problems with this: first is the obvious unnecessary expense of learning the same behaviour twice, and second is that this only verified symmetry up to state T. Adding further actions to the experiment will require redoing the learning and the verification. So we would like to keep this conjecture of symmetry for all future experiments. When starting a new experiment, the learner would start with the conjecture and test it. Recall the three steps of the learning algorithm explained in Sect. 2.1. Step 2 was testing the already built hypothesis model. Our future direction is that we would like the conjecture to be an initial hypothesis model and that in the first iteration of the loop, we start with step 2, i.e. testing the hypothesis/conjecture.

Symmetry is one property that can be represented as a conjecture. There are certainly other properties that fit into a conjecture, including undesired sequences, explained in Sect. 4.2. This solution was mentioned in [13] as presenting an abstract model to the learner before starting the experiment.

Furthermore, we need a formalism for conjectures such that a property can be uniquely expressed and then translated into a hypothesis model. The output learned model should abstract away unwanted details to produce a more readable model just as Fig. 6 is compared to Fig. 5, while still keeping the information of the expanded model as a formalised property.

In summary, we would like to research the theory necessary to express conjectures, translate them into hypotheses, start learning with a hypothesis as a starting point, and output an abstract readable learned model.

4.2 Avoiding Illegal/Undesired Sequences

Because of the nature of the learning process where actions are executed in different possible orders and with different frequencies, it can often lead to traces that are not covered in tests or ones that are not even allowed in normal use. An example is turning on a device that is already on. A test environment is probably not built to handle such scenarios and will thus cause errors. In such a case, we would like to specify two subsequent *turn on* actions as an illegal sequence. This research direction investigates how to formulate such illegalities and how to keep the learner's traces within certain boundaries.

Another example is the one seen in Sect. 3.4, where eight *Activate* actions in a run is an undesired sequence because it explodes the state space and thus we would like to avoid observing it altogether. This problem can fall under illegalities but we would like to investigate whether such a sequence can be expressed as a conjecture and thus fall under the problem discussed in Sect. 4.1.

4.3 Avoiding Repetition of Traces

In the current learning implementation, every new trace starts from the initial state. In many cases, however, we are interested in a certain state and would like to run multiple traces from that state without repeating the sequences that leads to it after each reset. Bauer et al. [4] introduce the idea of reusing previous traces in what they call the Reuse algorithm. The reuse algorithm acts as an intermediate layer between the learner and the SuL. It would respond to the learner with a previously known response instead of running it explicitly on the SuL, which obviously saves time. They keep the information of previous runs in a reuse tree.

As mentioned in Sect. 4.1, we would like to start learning from a hypothesis. And we see in this context one way to implement this, namely that LearnLib would use the hypothesis as a reuse tree. We believe that this is a more generic approach, but the question remains about which approach is more efficient.

4.4 Composition and Decomposition of Models

Looking at the case of Sect. 3.5 and the learned model of Fig. 9, a pressing question is: what does the model say about the real behaviour of the SuL, and what does it say about the combination of the SuL and its interface adapter? Is there a way to correctly decompose the learned model to deduce one describing the behaviour of the SuL alone? Such a scenario can be seen in practice and the question invites theoretical research. Moreover, the more complex the auxiliary component is, the more difficult it will be to analyse the learned model.

On the other hand, we would like to investigate composition of models. Consider the models of Figs. 7 and 9. The latter model starts at state F_T where the former model ends. Both have different sets of actions and different scopes. We can make the assumption that the action set from one model has no effect on the states of the other model. Based on this assumption, we can take the union of these two models and present the resulting model as a conjecture for further experiments, which would also serve in testing the assumption. Composing models that learn different parts of a system into one that describes the complete behaviour is a problem that relates directly to the scalability of model learning techniques and the efficiency of learning large systems.

4.5 Automating the Learning Setup

Automation is essential for growing the model learning technique into an industrial-scale application. There are certain parts in the learning setup process

that can be automated. For instance, Howar et al. [8] modified their LearnLib driver such that it automatically applies an abstraction on the alphabet when non-determinism is faced, whereas Merten et al. [14] automate the process of setting up the learner with input alphabet and other parameters. The latter is especially viable in our environment for the availability of consistent unit tests.

5 Conclusion

We narrated our experience in applying model learning to industrial legacy software. We faced interleaving, discovered hidden behaviour unintentionally, and dealt with an auxiliary component. We provided lessons about abstracting actions, exploiting symmetry, thoroughly testing the learned models, dealing with asynchronous communication, expanding the scope of read output, and carefully treating auxiliary components. Finally, we discussed future research directions, including learning with conjectures, and learning given an initial hypothesis.

Acknowledgement. We would like to thank Joshua Moerman and Ramon Janssen for their help with LearnLib and several related concepts, and Mathijs Schuts for sharing his knowledge about model learning.

References

1. Van der Aalst, W.M.P., Weijters, A.: Process mining: a research agenda. Comput. Ind. **53**(3), 231–244 (2004). https://doi.org/10.1016/j.compind.2003.10.001
2. Aarts, F., De Ruiter, J., Poll, E.: Formal models of bank cards for free. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, pp. 461–468, March 2013. https://doi.org/10.1109/ICSTW.2013.60
3. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
4. Bauer, O., Neubauer, J., Steffen, B., Howar, F.: Reusing system states by active learning algorithms. In: Moschitti, A., Scandariato, R. (eds.) EternalS 2011. CCIS, vol. 255, pp. 61–78. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28033-7_6
5. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering **SE-4**(3), 178–187 (1978). https://doi.org/10.1109/TSE.1978.231496
6. Coplien, J.O., Zhao, L.: Symmetry breaking in software patterns. In: Butler, G., Jarzabek, S. (eds.) GCSE 2000. LNCS, vol. 2177, pp. 37–54. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44815-2_4
7. Howar, F., Isberner, M., Merten, M., Steffen, B.: LearnLib tutorial: from finite automata to register interface programs. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012. LNCS, vol. 7609, pp. 587–590. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34026-0_43

8. Howar, F., Steffen, B., Merten, M.: Automata learning with automated alphabet abstraction refinement. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 263–277. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_19
9. Huang, Y.W., Yu, F., Hang, C., Tsai, C.H., Lee, D.T., Kuo, S.Y.: Securing web application code by static analysis and runtime protection. In: Proceedings of the 13th International Conference on World Wide Web, WWW 2004, pp. 40–52, ACM, New York (2004). https://doi.org/10.1145/988672.988679
10. Hungar, H., Margaria, T., Steffen, B.: Test-based model generation for legacy systems. In: 2003 Proceedings of the International Test Conference, ITC 2003, vol. 2, pp. 150–159, September 2003. https://doi.org/10.1109/TEST.2003.1271205
11. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 307–322. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_26
12. Madden, B.: Using CPPunit to implement unit testing. In: Game Programming Gems, vol. 6 (2006)
13. Margaria, T., Niese, O., Raffelt, H., Steffen, B.: Efficient test-based model generation for legacy reactive systems. In: Proceedings of the Ninth IEEE International High-Level Design Validation and Test Workshop (IEEE Cat. No. 04EX940), pp. 95–100, November 2004. https://doi.org/10.1109/HLDVT.2004.1431246
14. Merten, M., Isberner, M., Howar, F., Steffen, B., Margaria, T.: Automated learning setups in automata learning. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012. LNCS, vol. 7609, pp. 591–607. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34026-0_44
15. Niese, O.: An integrated approach to testing complex systems. Ph.D. thesis, Technical University of Dortmund, Germany (2003)
16. Niese, O., Steffen, B., Margaria, T., Hagerer, A., Brune, G., Ide, H.-D.: Library-based design and consistency checking of system-level industrial test cases. In: Hussmann, H. (ed.) FASE 2001. LNCS, vol. 2029, pp. 233–248. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45314-8_17
17. Schuts, M., Hooman, J., Vaandrager, F.: Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 311–325. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_20
18. Smeenk, W., Moerman, J., Vaandrager, F., Jansen, D.N.: Applying automata learning to embedded control software. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 67–83. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_5
19. Vaandrager, F.: Model learning. Commun. ACM **60**(2), 86–95 (2017)
20. Wirth, N.: Program development by stepwise refinement. Commun. ACM **14**(4), 221–227 (1971)