JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

Integrating model learning with software refactoring in industry

[1]Ammar Osaiweran

Faculty of Computers and Informatics,
Thamar University, Thamar, Yemen

Abstract : Software refactoring is a vital process that ensures the long-term maintainability and evolvability of software systems. However, for large and complex systems, it can be a complex and error-prone task. This work proposes a method of refactoring, guided by model learning and model-based testing, to ensure the correctness of software after the refactoring process. Model learning is used to learn a system under refactoring by constructing models that are used as input for the automatic generation of test cases. These tests are then executed on the system before and after refactoring, to ensure equivalence of behavior and to safeguard that no errors are introduced due to refactoring. The method is applied to a real industrial case and is demonstrated in this paper via application to a simple vending machine system. In this paper, we discuss the strengths and limitations encountered when applying the method to complex industrial applications.

***IndexTerms* -** software engineering, model-based development, software refactoring, software re-engineering, model-based testing, model transformation, software quality, software metrics application, software development, L* algorithm, Mealy state machine

1. Introduction

Code refactoring is one of the essential software engineering practices used by programmers almost on daily basis. Refactoring is frequently employed to address recurrent issues encountered by software engineers such as resolving code smells or technical debts, fixing bugs in a software component, or as an enabler to ease the introduction of a future feature. Moreover, refactoring can be used to enable migration to new software development technologies or to more structured programming languages.

Before establishing the refactoring of a software component, its external interfaces, internal implementation and behavior should be well understood. Understanding the external interface behavior of complex reactive systems via code inspection, or consulting potentially outdated documentation, may lead to errors implemented in the new refactored systems. This is because important information may be overlooked and hence related implementation is missing in the code. Therefore, for complex systems, the use of modern technologies for systematic extraction of external interface behavior may come to the rescue.

Model learning [1] is a promising technology that provides means to extract the behavior of systems automatically, by employing learner components that exercise a system under learning (SUL) and observes its outputs. These learners produce models that represent the external behavior observed for the SUL. This automatic extraction of behavior is attractive to industry because less time and effort are needed to understand the behavior of complex systems and can guide any refactoring activities, especially when documentation and testing are missing in old legacy systems.

In this article, we propose a method for refactoring reactive software systems with the aid of model learning techniques. Model learning is used to learn and automatically extract a behavioral model that resembles how the "to be refactored" system interacts with its environment. This behavioral model is expressed in a Mealy state machine [2] and is used as a basis to verify the correctness of the system before and after refactoring. This is established by employing model-based testing techniques [3], using which the learned model, expressed in a Mealy state machine, is used to automatically generate unique test cases which should first pass on the original system. This set of test cases should also pass on the system after refactoring to ensure it does not deviate in its behavior from the original system.
This work tries to answer the following research questions:

- Can model learning techniques be integrated with software engineering refactoring process?
- Is model learning techniques mature enough to be used for learning complex industrial systems?
- What are the pros and cons of using model learning for refactoring?

This paper is structured as follows. In Section 2, we present the related work established by others that is related to our work. In Section 3, we briefly describe the concepts of refactoring and re-engineering to the limit needed for this paper. Section 4, gives a formal definition of the Mealy state machine which is the formalism produced by the model learner. Section 5, introduces a vending machine system that is used as a use case to discuss the approach of applying model learning. In Section 6, we detail the model learning approach and apply the technology to the use case of the vending machine. We show how the behavioral model of the vending machine is gradually constructed. Section 7, discussed the method we propose to refactor reactive systems with the help of model learning and model-based testing. In Section 8, we discuss the lessons learned from applying model-learning and model-based testing highlighting the pros and cons encountered during the application. Section 9 concludes our work with possible future work.

2. Related Work

Peled et al. [4,5] and Steffen et al. [6,7,8] combined model learning and model checking. A model checker was used to check properties of learned models. Peled [5] proposed an approach called black box checking integrating model learning and model checking. Fitera˘u et al. [11] used model learning and model checking to verify the equivalence of TCP/IP behavior in different operating systems such as Linux, Windows, and FreeBSD. They found that the implementation and behavior differ in these systems. Schuts et al. [9] used model learning to support the refactoring of legacy software. They extracted two models from the old and new systems and compared their behavioral equivalence using model checking.

3. Refactoring, Reengineering and Redesign

There are three main principles used widely in software engineering practices to improve code of a software module: refactoring, reengineering and redesign. In refactoring or reengineering the external interface of a software system remains the same but the underlying implementation may differ. For refactoring, the technology used to implement the original system is used also to implement the new system. Both new and old systems implement the same external interfaces. For example, an original system is implemented using C and the new refactored system is also implemented in C where both use the same header file as an external interface.

Reengineering implies that different (usually modern) technology is used to implement the new system while external interfaces are still the same. For example, an old system may be implemented in C while the new system is implemented in C++ or developed using a model-based technique with automatic code generation. In this work, we use refactoring or reengineering interchangeably.

In addition to having the same interface for the new and old software system after refactoring or reengineering (same interface syntax), the external behavior of the system should be preserved (same interface semantic). Preservation of external behavior implies that, for the same set of call sequence and input values, the resulting set of output values should be equal before and after refactoring. This also means that changes in the code are internal while external interfaces must remain unchanged.

Redesign means that the external interface of a new system and its implementation are completely different than the external interface and implementation of the original system. Redesign is out of the scope of this work and we focus only on the refactoring aspects of reactive systems.

4. Mealy machine models

Models generated by the model learning tool are encoded in a Mealy machine format. Formally, a (deterministic) Mealy machine is described as a 6-tuple $M = (I, O, Q, q0, \delta, \lambda)$, where I and O are finite sets of inputs and outputs, Q is a finite set of states, $q0 \in Q$ is the initial state, $\delta: Q\times I \rightarrow Q$ is a transition function mapping a state and input to another state, and $\lambda: Q\times I \rightarrow O$ is an output function mapping a state and input to an output. Figure 1 gives a graphical representation of a simple Mealy machine with input set $\{0, 1\}$, output set $\{a, b\}$, states $\{q0, q1, q2\}$, and initial state $q0$.

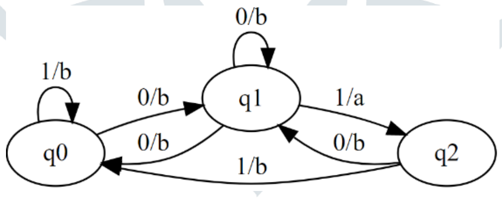

Figure 1: example mealy machine

Function λ is further expanded to sequences of inputs by defining, for all $q\in Q$, $i\in I$, and $\sigma\in I^*$, $\lambda(q, \varepsilon) = \varepsilon$, and $\lambda(q, i\sigma) = \lambda(q, i)\lambda(\delta(q, i), \sigma)$. The behavior of Mealy machine M is defined by function $AM : I^* \rightarrow O^*$ with $AM(\sigma) = \lambda(q0 , \sigma)$, for $\sigma \in I^*$. M and N Machines are equivalent, iff $AM = AN$. Mealy machine M and N are not equivalent if and only if there is a sequence $\sigma \in I^*$ distinguishes M and N, $AM(\sigma) \neq AN(\sigma)$.

5. Case study

Due to the restriction of exposing any confidential information about the industrial case, we replace it with a case study concerning a chocolate vending machine system. We use this system to clarify the concepts around model learning and its application to software engineering refactoring in industry.
The vending machine application accepts input stimuli of *5c* (5 cents) and *10c* coins, and accepts input requests to produce *Mars* (costs 10c), *Twix* (costs 15c) and *Snickers* (costs 25c). The application responds with *OK* if the input is allowed and *NOK* otherwise. So, *I* = {*5c*, *10c*, *Mars*, *Twix*, *Snickers*} and *O* = {*OK*, *NOK*}. The following code gives an example implementation of the vending machine in C++.

```
class ChocolateVendingMachine
```

```
{
public:
  static bool addMoney(int cents);
  static bool subtractMoney(int cents);
  static bool getChoc(std::string chocolate);
  static void reset();

private:
  static int money;
  static map<string, int> chocolates ;

  ChocolateVendingMachine() { money = 0;
  chocolates = {{"mars", 10}, {"twix", 15}, {"snickers", 25}};}
};

bool ChocolateVendingMachine::addMoney(int cents){
  money += cents;
  return true;
}
bool ChocolateVendingMachine::subtractMoney(int cents){
  if(money < cents)
    return false;
  money -= cents;
  return true;
}
bool ChocolateVendingMachine::getChoc(string chocolate){
  if(chocolates.count(chocolate)){
    return subtractMoney(chocolates[chocolate]);
  }
  return false;
}
void ChocolateVendingMachine::reset(){
  money = 0;
}
```

6. Model Learning

Model learning is a technique that constructs a state machine model of a software or a hardware system by providing a sequence of input stimuli to the system and observing its output. The system is seen as a black-box where internal details are irrelevant. This state machine describes the externally visible behavior of the system. There are many algorithms developed for model learning. Among these algorithms is the L* algorithm [10] which is used for this work. Below we give a formal definition and apply the algorithm to the vending machine use case.

6.1. Formal definition

The L* algorithm incrementally constructs an observation table that encodes the Mealy state machine. The table includes entries taken from the set I of input and the set O of outputs. Each row is labeled by a *sequence* in $(S \cup S \cdot I)$, where S is a nonempty finite input sequence called prefix. The columns of the table are labeled by a nonempty finite sequence E called suffix. An observation table is a triple (S, E, row), where row: $S \cup (S \cdot I) \rightarrow (E \rightarrow O)$.
For a prefix s and suffix e, $row(s)(e)$ returns the last output returned by the SUL when the se input sequence is applied. The algorithm starts with the initial state where S contains the empty sequence ε, and E equals set of inputs I. When constructing the state machine, two key properties should apply: closedness and consistency. Observation table (S, E, row) is closed if for all $s \in S \cdot I$ there is a $s' \in S$ with $row(s) = row(s')$. It is consistent if $row(s1) = row(s2)$ for some $s1, s2 \in S$, then $row(s1\ a) = row(s2\ a)$ for all $a \in I$. If a table is closed and consistent, the learner constructs a Mealy machine $H = (I, O, Q, q0, \delta, \lambda)$ with $Q = \{row(s) \mid s \in S\}$, $q0 = row(\varepsilon)$, $\delta(row(s), a) = row(s \cdot a)$, and $\lambda(row(s), a) = row(s)(a)$.

6.2. Applying L* on the vending machine system

L* starts from the initial state of the system represented as the first row in the table where prefix is the ε (empty) input and suffix (E) is the list of possible inputs. In Table 1, the initial state contains OK on inputs ε.5c and ε.10c but NOK on inputs ε.Mars, ε.Twix and ε.Snickers (no sufficient money yet to produce any chocolate).

Prefix	Suffix				
	5c	10c	Mars	Twix	Snickers
ε	OK	OK	NOK	NOK	NOK

Table 1: Observation table containing the initial state

The algorithm continues with adding the input to the prefix (forming S . I as explained formally above), and via another iteration the table is filled in with possible outputs, see Table 2. At this step, the state machine contains only a single state which is the initial state (see the line separating the prefix list in the table).

Prefix	Suffix				
	5c	10c	Mars	Twix	Snickers
ε	OK	OK	NOK	NOK	NOK
5c	OK	OK	NOK	NOK	NOK
10c	OK	OK	OK	NOK	NOK
Mars	OK	OK	NOK	NOK	NOK
Twix	OK	OK	NOK	NOK	NOK
Snickers	OK	OK	NOK	NOK	NOK

Table 2: Prefix of the initial state is extended with each input of I

The L* algorithm will observe a new state when a unique combination of output values appears in the table. For instance, in Table 2, the prefix *10c* (with the suffix *Mars*) makes a unique (state) row (highlighted rows in green denote newly discovered states). This row is then promoted as a new state and moved above the thick line. After that, new prefixes are added to the table appending each input to 10c, see Table 3. New entries are then calculated to fill in the rows and to discover new states.

Prefix	Suffix				
	5c	10c	Mars	Twix	Snickers
ε	OK	OK	NOK	NOK	NOK
10c	OK	OK	OK	NOK	NOK
5c	OK	OK	NOK	NOK	NOK
Mars	OK	OK	NOK	NOK	NOK
Twix	OK	OK	NOK	NOK	NOK
Snickers	OK	OK	NOK	NOK	NOK
10c 5c	OK	OK	OK	OK	NOK
10c 10c	OK	OK	OK	OK	NOK
10c Mars	OK	OK	NOK	NOK	NOK
10c Twix	OK	OK	OK	NOK	NOK
10c Snickers	OK	OK	OK	NOK	NOK

Table 3: prefix 10c is added as a new state and new sequences are formed by appending inputs of I

Two further states are discovered in the table and promoted above the thick line. However, as seen in the table, the current Mealy machine is not consistent because *row(10c 5c) = row(10c 10c)* but their future with appending inputs of set *I* is not the same.

Prefix	Suffix				
	5c	10c	Mars	Twix	Snickers
ε	OK	OK	NOK	NOK	NOK
10c	OK	OK	OK	NOK	NOK
10c 5c	OK	OK	OK	OK	NOK
10c 10c	OK	OK	OK	OK	NOK
5c	OK	OK	NOK	NOK	NOK
Mars	OK	OK	NOK	NOK	NOK
Twix	OK	OK	NOK	NOK	NOK
Snickers	OK	OK	NOK	NOK	NOK
10c Mars	OK	OK	NOK	NOK	NOK
10c Twix	OK	OK	OK	NOK	NOK
10c Snickers	OK	OK	OK	NOK	NOK

Table 4: state are not consistent

Therefore, to make the table consistent, the suffix list *E* needs to be extended. Table 5 shows that the output of *10c.5c.5c.Snickers* is different than *10c.10c.5c.Snickers*. Thus, *5c.Snickers* should be added to the suffix list. Additionally, the output of *10c.5c.Mars.Mars* is different than *10c.10c.Mars.Mars*. Thus, we also add *Mars.Mars* to the suffix list.

Prefix	input	5c	10c	Mars	Twix	Snickers
10c 5c	5c	OK	OK	OK	OK	**NOK**
10c 5c	10c	OK	OK	OK	OK	OK
10c 5c	Mars	OK	OK	**NOK**	NOK	NOK
10c 5c	Twix	OK	OK	NOK	NOK	NOK
10c 5c	Snickers	OK	OK	OK	OK	NOK
10c 10c	5c	OK	OK	OK	OK	**OK**
10c 10c	10c	OK	OK	OK	OK	OK
10c 10c	Mars	OK	OK	**OK**	NOK	NOK
10c 10c	Twix	OK	OK	NOK	NOK	NOK
10c 10c	Snickers	OK	OK	OK	OK	NOK

Table 5: extending suffix list with new sequences to make the observation table consistent

The new table with new prefixes and suffixes looks as depicted in Table 6.

Prefix	**Suffix**						
	5c	**10c**	**Mars**	**Twix**	**Snickers**	**Mars Mars**	**5c Snickers**
ε	OK	OK	NOK	NOK	NOK	NOK	NOK
10c	OK	OK	OK	NOK	NOK	NOK	NOK
10c 5c	OK	OK	OK	OK	NOK	NOK	NOK
10c 10c	OK	OK	OK	OK	NOK	OK	OK
5c	OK	OK	NOK	NOK	NOK	NOK	NOK
Mars	OK	OK	NOK	NOK	NOK	NOK	NOK
Twix	OK	OK	NOK	NOK	NOK	NOK	NOK
Snickers	OK	OK	NOK	NOK	NOK	NOK	NOK
10c Mars	OK	OK	NOK	NOK	NOK	NOK	NOK
10c Twix	OK	OK	OK	NOK	NOK	NOK	NOK
10c Snickers	OK	OK	OK	NOK	NOK	NOK	NOK
10c 5c 5c	OK	OK	OK	OK	NOK	OK	OK
10c 5c 10c	OK	OK	OK	OK	OK	OK	OK
10c 5c Mars	OK	OK	NOK	NOK	NOK	NOK	NOK
10c 5c Twix	OK	OK	OK	NOK	NOK	NOK	NOK
10c 5c Snickers	OK	OK	OK	OK	NOK	NOK	NOK
10c 10c 5c	OK	OK	OK	OK	OK	OK	OK
10c 10c 10c	OK	OK	OK	OK	OK	OK	OK
10c 10c Mars	OK	OK	OK	NOK	NOK	NOK	NOK
10c 10c Twix	OK	OK	NOK	NOK	NOK	NOK	NOK
10c 10c Snickers	OK	OK	OK	OK	NOK	OK	OK

Table 6: extending the suffix with new sequences to make the states consistent

The algorithm continues to discover new states, when it encounters new unique rows, and makes the table consistent when new states have the same outputs in their rows. For example, *10c.5c.10c*, *10c.10c.5c* and *10c.10c.10c* form new states but their rows are identical which means new suffixes should be added to make them unique. Table 7 shows that the suffix list should be extended with *Twix.Twix*.

Pre	input	**5c**	**10c**	**Mars**	**Twix**	**Snickers**
10c 5c 10c	5c	OK	OK	OK	OK	OK
10c 5c 10c	10c	OK	OK	OK	OK	OK
10c 5c 10c	Mars	OK	OK	OK	OK	NOK
10c 5c 10c	Twix	OK	OK	OK	**NOK**	NOK
10c 5c 10c	Snickers	OK	OK	NOK	NOK	NOK
10c 10c 5c	5c	OK	OK	OK	OK	OK
10c 10c 5c	10c	OK	OK	OK	OK	OK
10c 10c 5c	Mars	OK	OK	OK	OK	NOK
10c 10c 5c	Twix	OK	OK	OK	**NOK**	NOK
10c 10c 5c	Snickers	OK	OK	NOK	NOK	NOK
10c 10c 10c	5c	OK	OK	OK	OK	OK
10c 10c 10c	10c	OK	OK	OK	OK	OK
10c 10c 10c	Mars	OK	OK	OK	OK	NOK
10c 10c 10c	Twix	OK	OK	OK	**OK**	NOK
10c 10c 10c	Snickers	OK	OK	NOK	NOK	NOK

Table 7: Twix.Twix will make the states consistent and should be added to the suffix list

Therefore, the new table after adding the new states and the new suffix will look as follows.

Prefix	**Suffix**							
	5c	**10c**	**Mars**	**Twix**	**Snickers**	**Mars Mars**	**5c Snickers**	**Twix Twix**
ε	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
10c	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
10c 5c	OK	OK	OK	OK	NOK	NOK	NOK	NOK
10c 10c	OK	OK	OK	OK	NOK	OK	OK	NOK
10c 5c 10c	OK	OK	OK	OK	OK	OK	OK	NOK
10c 10c 5c	OK	OK	OK	OK	OK	OK	OK	NOK
10c 10c 10c	OK	OK	OK	OK	OK	OK	OK	OK
5c	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
Mars	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
Twix	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
Snickers	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
10c Mars	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
10c Twix	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
10c Snickers	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
10c 5c 5c	OK	OK	OK	OK	NOK	OK	OK	NOK
10c 5c Mars	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
10c 5c Twix	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
10c 5c Snickers	OK	OK	OK	OK	NOK	NOK	NOK	NOK
10c 10c Mars	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
10c 10c Twix	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
10c 10c Snickers	OK	OK	OK	OK	NOK	OK	OK	NOK
10c 5c 10c 5c	OK	OK	OK	OK	OK	OK	OK	OK
10c 5c 10c 10c	OK	OK	OK	OK	OK	OK	OK	OK
10c 5c 10c Mars	OK	OK	OK	OK	NOK	NOK	NOK	NOK
10c 5c 10c Twix	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
10c 5c 10c Snickers	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
10c 10c 5c 5c	OK	OK	OK	OK	OK	OK	OK	OK
10c 10c 5c 10c	OK	OK	OK	OK	OK	OK	OK	OK
10c 10c 5c Mars	OK	OK	OK	OK	NOK	NOK	NOK	NOK
10c 10c 5c Twix	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
10c 10c 5c Snickers	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
10c 10c 10c 5c	OK	OK	OK	OK	OK	OK	OK	OK
10c 10c 10c 10c	OK	OK	OK	OK	OK	OK	OK	OK

10c 10c 10c Mars	OK	OK	OK	OK	NOK	OK	OK	NOK
10c 10c 10c Twix	OK	OK	OK	OK	NOK	NOK	NOK	NOK
10c 10c 10c Snickers	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK

Table 8: Observation table after adding Twix.Twix to the suffix list with further new states

Note that at this step there is no new states can be discovered. However, a valid sequence, which is not part of the observation table calculated so far, which leads to a different state can be ε.*5c.5c*. From the start, this sequence can lead to obtaining *Mars*. Such cases can be added to the learner as a counterexample manually and the learner will start exploring the following states further. This way the learner is guided to explore overlooked states. Note that, the learner can be configured to automatically discover counterexamples via employing several algorithms. Most notable is the W-method algorithm [17] of which detail is outside the scope of this article. Table 9 shows the observation table extended with new states via guiding the algorithm with counterexamples.

Prefix	**Suffix**							
	5c	**10c**	**Mars**	**Twix**	**Snickers**	**Mars Mars**	5c **Snickers**	Twix twix
ε	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
10c	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
10c 5c	OK	OK	OK	OK	NOK	NOK	NOK	NOK
10c 10c	OK	OK	OK	OK	NOK	OK	OK	NOK
10c 5c 10c	OK	OK	OK	OK	OK	OK	OK	NOK
10c 10c 5c	OK	OK	OK	OK	OK	OK	OK	NOK
10c 10c 10c	OK	OK	OK	OK	OK	OK	OK	OK
5c	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
5c 5c	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
Mars	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
Twix	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
Snickers	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
10c Mars	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
10c Twix	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
10c Snickers	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
10c 5c 5c	OK	OK	OK	OK	NOK	OK	OK	NOK
10c 5c Mars	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
10c 5c Twix	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
10c 5c Snickers	OK	OK	OK	OK	NOK	NOK	NOK	NOK
10c 10c Mars	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
10c 10c Twix	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
10c 10c Snickers	OK	OK	OK	OK	NOK	OK	OK	NOK
10c 5c 10c 5c	OK	OK	OK	OK	OK	OK	OK	OK
10c 5c 10c 10c	OK	OK	OK	OK	OK	OK	OK	OK
10c 5c 10c Mars	OK	OK	OK	OK	NOK	NOK	NOK	NOK
10c 5c 10c Twix	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
10c 5c 10c Snickers	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
10c 10c 5c 5c	OK	OK	OK	OK	OK	OK	OK	OK
10c 10c 5c 10c	OK	OK	OK	OK	OK	OK	OK	OK
10c 10c 5c Mars	OK	OK	OK	OK	NOK	NOK	NOK	NOK
10c 10c 5c Twix	OK	OK	OK	NOK	NOK	NOK	NOK	NOK
10c 10c 5c Snickers	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK
10c 10c 10c 5c	OK	OK	OK	OK	OK	OK	OK	OK
10c 10c 10c 10c	OK	OK	OK	OK	OK	OK	OK	OK
10c 10c 10c Mars	OK	OK	OK	OK	NOK	OK	OK	NOK
10c 10c 10c Twix	OK	OK	OK	OK	NOK	NOK	NOK	NOK
10c 10c 10c Snickers	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK

Table 9: Observation table is extended with new states using counterexamples

However, these new states make the table of states not consistent so the suffix list should be extended. Table 10 shows how the new sequence is calculated to make consistent states.

Prefix	input	5c	10c	Mars	Twix	Snickers
ε	5c	OK	OK	NOK	NOK	NOK
ε	10c	OK	OK	OK	NOK	NOK
ε	Mars	OK	OK	NOK	NOK	NOK
ε	Twix	OK	OK	NOK	NOK	NOK
ε	Snickers	OK	OK	NOK	NOK	NOK
5c	5c	OK	OK	OK	NOK	NOK
5c	10c	OK	OK	OK	OK	NOK
5c	Mars	OK	OK	NOK	NOK	NOK
5c	Twix	OK	OK	NOK	NOK	NOK
5c	Snickers	OK	OK	NOK	NOK	NOK

Table 10: calculating new sequences to be added to the suffix list to make the states consistent

Therefore, to make the table consistent, *5c.Mars* and *10c.Twix* sequences should be added to the suffix list.
Table 11 depicts the final states including the final list of prefixes and suffixes of the state machine.

Prefix	Suffix									
	5c	10c	Mars	Twix	Snickers	Mars Mars	5c Snickers	Twix Twix	5c Mars	10c Twix
ε	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK	NOK	NOK
10c	OK	OK	OK	NOK	NOK	NOK	NOK	NOK	OK	OK
10c 5c	OK	OK	OK	OK	NOK	NOK	NOK	NOK	OK	OK
10c 10c	OK	OK	OK	OK	NOK	OK	OK	NOK	OK	OK
10c 5c 10c	OK	OK	OK	OK	OK	OK	OK	NOK	OK	OK
10c 10c 5c	OK	OK	OK	OK	OK	OK	OK	NOK	OK	OK
10c 10c 10c	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
5c	OK	OK	NOK	NOK	NOK	NOK	NOK	NOK	OK	OK
5c 5c	OK	OK	OK	NOK	NOK	NOK	NOK	NOK	OK	OK
		The table can be filled in here following the approach detailed above. Details are omitted for readability purposes.								

Table 11: final observation table with the final resulting state machine

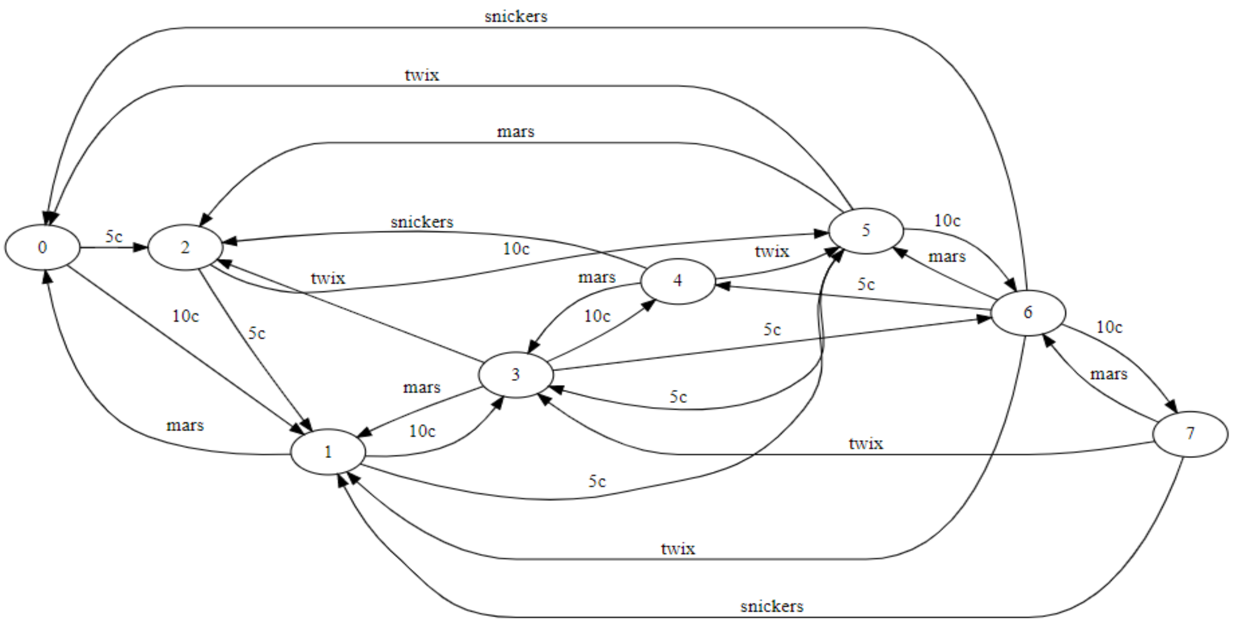

Figure 2: Mealy machine for the learned model

Figure 2 shows graphical representations of the state machine where only transitions that lead to OK are visible. Omitted from the diagram transitions leading to *NOK* (they make self-transitions) for readability purposes. Note that the machine is deterministic and for each missing input in any state, there is a self-transition leading to *NOK*.

6.3. Integrating model learning to learn a software module

To learn the behavior of a software system, the model learner needs to be connected to the system to be learned first. The model learner connects to the system under learning (SUL) via an adapter that translates abstract calls sent from the learner to calls accepted by the SUL via its external interface. Figure 3 shows how the leaner can be connected to a system via an adapter.

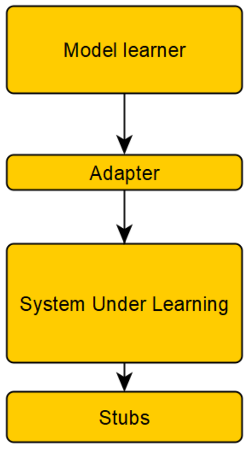

Figure 3: Integrating model learning to learn a system

SUL should be isolated from the rest of the system via stubs. The learning algorithms, which we used throughout this work, are provided via a library called LearnLib [17]. The connection between LearnLib and SUL is done via TCP/IP connection.

7. Proposed refactoring method

In this section, we discuss the proposed methodology of refactoring software components that includes external interfaces with state-full protocols.

Figure 4 depicts the steps applied to the component under refactoring (CUR).

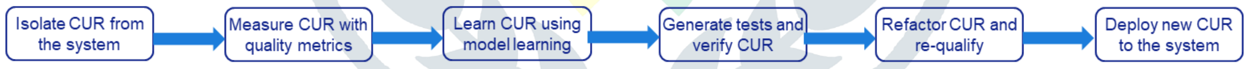
Isolate CUR from the system
Measure CUR with quality metrics
Learn CUR using model learning
Generate tests and verify CUR
Refactor CUR and re-qualify
Deploy new CUR to the system

Figure 4: High-level steps of refactor method with the aid of model learning

1- **Isolate CUR from the rest of the system**. To use the model learning tool, the CUR needs to be isolated from the other surrounding components first. All used components located on the boundary of CUR should be replaced by stubs.
2- **Measure CUR with quality metrics**. Before performing any refactoring activity, the current quality state of CUR needs to be captured using software metrics. These metrics will be compared with the metrics generated from CUR after refactoring to ensure that the quality of CUR is not degraded due to refactoring. To mention a few, metrics may include cyclomatic complexity, nested depth, lines of code per function and code clones. Depending on the programming language used to implement the code, these metrics can be collected systematically using static code analysis tools like TIOBE [12] or CodeSonar [13].
3- **Extract behavioral model of CUR**. This is the starting point of applying model learning techniques. After the CUR is isolated from the rest of the system, the model learner can be connected to the CUR via an adapter. The learner will start learning the CUR and produce a behavioral model in a mealy state machine format. Usually, this is a repeated step where a resulting model is refined iteratively with guided counterexamples as presented for the use case. The final resulting model should be verified by experts who hold sufficient domain knowledge and thus agree on the correctness and completeness of the model.
4- **Test case generation and verification**. After a final model is generated by the model learner, a model-based testing framework can be used to generate test-cases automatically. Most model-based testing frameworks accept state machines as input and generate (offline) test-cases as output. A translation script

is needed to convert the behavioral model of the model learner to the input format accepted by the model-based testing framework. Furthermore, resulting test-cases from the model-based framework need to be converted/adapted to the format of the testing tool used for the actual testing process. All tests should pass on CUR.

5- **Refactor and re-qualify CUR**. At this phase, CUR internal implementation can be refactored. After refactoring is done, quality metrics should be collected and compared to the metrics collected during step 2. New metrics should exhibit improved quality so no traces of quality degradation may be introduced due to refactoring. After ensuring the quality of refactoring via metrics, test cases generated in step 4 should be re-run and passed on the refactored component. This includes any existing test cases used to test CUR before refactoring to ensure regression.

6- **Deploy CUR to the system**. After the refactored component is verified, the old component can be replaced by the new refactored component. Exhaustive system regression tests should be performed to ensure that no errors appear after integrating the new component with the rest of the system.

8. Discussion and Lessons learned

In this section, we present the lesson learned from applying the proposed approach of refactoring with model learning for software refactoring on industrial applications. We highlight the strengths and pitfalls of the approach and the encountered limitations and challenges.

The first challenge was encountered when we applied step 1 of isolating CUR from the rest of the system. The challenging part was that dependency between CUR and other boundary components contains cycles, meaning that other boundary components depend also on CUR. So, to isolate CUR, these cyclic dependencies should be resolved first. To resolve such dependencies, CUR and dependent components were refactored.

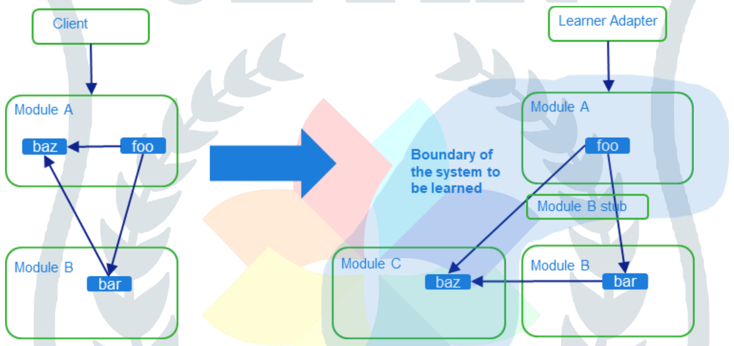

Figure 5: example of cyclic dependency and possible resolution

Figure 5 depicts an example of cyclic dependency where modules A and B depend on each other (a module here can be seen as a *cpp* file that includes functions exposed via a *hpp* header interface). A function *foo* in module A depends on a function *bar* and *baz* in modules B and A respectively. To resolve this dependency, function *baz* is refactored to a new module C so that B cyclic dependency to A is removed. After that, the system under learning is isolated such that it contains module A, module C and a stub replacing module B. The hidden dependency from module B to module C is not part of the CUR but refactoring *baz* to a new module C was done in a way that missing this dependency in CUR does not affect the behavior of module A.

Although this was a small refactoring work, no qualification was performed to ensure that resolving cycles may introduce bugs in CUR. To speed up isolating CUR from the system, we used a tool developed in-house to generate stubs from header interfaces automatically. Manual implementation of stubs would otherwise consume more time and effort. We were planning to automate the separation of CUR but with the existence of cyclic dependencies, we found it hard to achieve this without manual intervention.

In order to integrate the model learner, an adapter should be implemented to bridge the connection with CUR. The adapter should maintain the proper abstraction needed to exchange calls crossing between CUR and the learner. In contrast to the automatic generation of stubs that replace boundary components of CUR, creating the adapter requires human creativity and manual work. For our industrial case, the adapter adds default data parameters sent with the calls towards CUR but abstracts away the real values before forwarding replies from CUR to the learner. Our concern here is that abstracting away real data by replacing it with some defaults may hide potential important external behavior that will appear only in case abstracted data causes internal changes in the logic of CUR. This means that CUR internally may be sensitive to some data values to make a visible change in the external behavior but because such data is absent, due to abstraction, such behavior will not be observed by the learner.

Obtaining the learned model from the model learner was not just a "click of a button" but rather required many iterations. The model learner needed more guidance, using counterexamples, to refine the model and generate a reasonable model that reflects reality. These counterexamples were constructed from generated traces and logs produced by the CUR external interface before refactoring. We had to translate these logs to the format accepted by the model learner which consumed even more time and effort.

The model learner should be restricted otherwise the resulting model can grow very fast in size. Human creativity is needed here to restrict the model. For example, for our case study of the vending machine we had to restrict the total amount of money accepted by the system to *25c*. For example, as can be seen in the resulting state machine of Figure 2, any input of *5c* or *10c* in *state 7* is not accepted.

Another challenge we encountered is the selection of the proper model-based testing framework from the wide range of available tools in the literature. For this, we made some selection criteria that satisfy our needs. The criteria are:

- Publicly available: tool should be available to public and anyone can use it without any restrictions.
- Model-based tool: for its input, the tool should depend on a model to ease translating the learned model to the input model accepted by the tool. This enables the automation of steps.
- Offline testing: Offline testing means that test cases are generated and executed. Online testing implies generating and executing tests on-the-fly. Usually, test cases are generated as a sequence of calls.
- Export functionality: the test tool should be able to export its test sequences to translate them to concrete test cases implementation.
- Full Coverage: all transitions present in the learned model are covered.
- OS independent: our work needs both Linux and Windows systems, so the tool should be easily deployable on both with minimal effort.

We carefully studied 17 tools and selected the HADS (Hybrid Adaptive Distinguishing Sequences) tool [14] because it fits our needs the most. HADS was used to generate test sequences which were translated to the test suite of GTest (google test) [15] and Junit [16]. These test cases were used as the basis for qualifying the component before and after refactoring. All tests were passed on the system before and after refactoring.

The following table summarizes the strong and weak points of applying model learning for our case.

Strengths	**Limitations**
When the learner is executed, potential errors may be discovered in the learned system. This is because the learner will try to cover also unexpected cases by the learned system	Manual effort is required to isolate the system before model learning can be employed. Complexity increases when cyclic dependencies are present.
Behavioral models can be constructed automatically, reducing cost and effort	Manual effort is needed to build an adapter that translates “strings” sent/accepted by the learner to actual calls to the system under learning.
Can help constructing an initial understanding of systems when domain knowledge and documentations are absent	Needs many iterations until a final model is constructed.
The resulting models can be used for future documentation	It is not guaranteed the complete behavior will be captured by the resulting model
Resulting models can be used by model-based technologies to generate interfaces, implementation code and tests via one-to-one transformation.	For complex systems, resulting models can grow very big, making any manual modification or inspection by engineers very complex.
Limited (or even zero) knowledge of the inner details of the system under learning is needed to use the model learning techniques	Tools generate one big model, it is not possible to systematically decompose the model into smaller models.
	The model learner needs guidance using counterexamples to obtain a model.
	Not possible to use function parameters (input/output data). Data parameters should be faked.
	When data parameters cause internal state changes in the component to be learned, they should be translated to dedicated calls, making the adapter very complex and error-prone.
	Control of the SUL is done from the top external interface. If the system changes its state based on callbacks initiated from lower-level components then this is not covered. This is because stubs are used to replace the lower-level components.

As can be seen from the table, from our experience in applying the method to our use case, the limitations of the model learner outweigh its strengths.

9. Conclusion and future work

In this paper, we proposed a method for refactoring complex reactive systems in industrial settings. The method incorporates model learning and model-based testing for software system refactoring. In the following, we will answer the research questions raised earlier.

- Can model learning techniques be integrated with software engineering refactoring process?

As discussed throughout this paper, model learning can be integrated into the software development process. Before refactoring a software module, the learner can learn its external behavior. However, embedding model learning to learn a software model requires time and effort.

- Is model learning techniques mature enough to be used for learning complex industrial systems?

Our work shows that model learning can be suitable for event-driven reactive systems where interfaces are parameter-less or where parameters have no effect on the external behavior of the system. But for other systems where data can change the internal states, extra effort is needed to make a suitable adapter which may grow in complexity and size. The complexity of the adapter may cause errors in its implementation and there is a risk that errors are also reflected in the resulting learned model.

- What are the pros and cons of using model learning for refactoring?

In the previous section, we listed the strengths and the limitation of model learning based on our experience of applying it to the industrial and the use case system.

An apparent limitation of this method is that it requires extensive manual effort at some stages such as separating the system under learning from the rest of the system. As future work, we are investigating how to automatically isolate a software component from its surrounding components.

Another limitation of this method is that the generated model can grow fast. As future work, we are investigating how a model can be decomposed into smaller models. Furthermore, we are investigating how a model size can be reduced by applying abstraction and using compression techniques to reduce the size of the model while still preserving the intended behavior.

References

[1] F. Vaandrager, "*Model Learning*", Communications of the ACM, February 2017, Vol. 60 No. 2, Pages 86-95.

[2] G. Mealy, "*A Method for Synthesizing Sequential Circuits*", Bell System Technical Journal. 1955, pp. 1045–1079.

[3] J.Rushby, "*Automated Test Generation and Verified Software*", Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10–13. pp. 161-172, Springer-Verlag.

[4] A. Groce, D. Peled, M. Yannakakis, "*Adaptive model checking*", Logic J. IGPL 14, 5 (2006), 729–744.

[5] D. Peled, M. Vardi, M. Yannakakis, "*Black box checking*", Autom. Lang. Comb. 7, 2 (2002), 225–246

[6] Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B. "*On the correspondence between conformance testing and regular inference*". In FASE'05, LNCS 3442 (2005). Springer, 175–189 R. Nicole, "Title of paper with only first word capitalized", *J. Name Stand. Abbrev.*, in press.

[7] Hagerer, A., Hungar, H., Niese, O., "*Steffen, B. Model generation by moderated regular extrapolation*". In FASE'02, LNCS 2306 (2002). Springer, 80–95.

[8] de la Higuera, C. Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, 2010.

[9] Schuts, M., Hooman, J., Vaandrager, F. "*Refactoring of legacy software using model learning and equivalence checking: an industrial experience report*". In iFM'16, LNCS 9681 (2016). Springer, 311–325

[10] Angluin, D. "*Learning regular sets from queries and counterexamples*". Inf. Comput. 75, 2 (1987), 87–106. S

[11] Fiteraˇu-Bros¸tean, P., Janssen, R., Vaandrager, F. "*Combining model learning and model checking to analyze TCP implementations*". In CAV'16, LNCS 9780 (2016). Springer, 454–471. S

[12] www.tiobe.com. Last visit March 17, 2023

[13] https://www.verifysoft.com/en_grammatech_codesonar.html, Last visit March 17, 2023

[14] Hybrid adaptive distinguishing sequences for fsm-based complete testing. https://gitlab.science.ru.nl/moerman/hybrid-ads. Accessed: March 17, 2023.

[15] https://github.com/google/googletest, Last visit March 17, 2023

[16] JUnit 5, https://junit.org/junit5/. Last visit March 17, 2023

[17] H. Raffelt, B. Steffen, and T. Berg. "*Learnlib: A library for automata learning and experimentation*". In Proceedings of the 10th international workshop on Formal methods for industrial critical systems, pages 62–71. ACM, 2005.