# Automatically Generating Learning Setups for GUI-based Programs through Annotation Processing

Hans van der Laan
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
j.h.vanderlaan@student.utwente.nl

## ABSTRACT

The main challenge for widespread adaptation of active automata learning is the effort required to design and implement application specific learning setups. In this paper, we propose an abstraction of GUI elements by dividing them into three categories: widgets, containers and windows. We analyse at how events are handled in Java, JavaFX and how the MVC pattern is structured. With this information in hand we propose a Mealy machine based behaviour model. Learning GUI-based programs has a few challenges: Event handler accessibility, event space transmutations and the order of events. We solve the event handler accessibility by performing checks, event space transmutations by storing the event handlers dynamically and the order of events by analysing the validity of queries through an NFA-ε with as language all possible event sequences at that given moment. With our research, we have paved the way for active automata learning of GUI-based programs.

## Keywords

Learning setup generation, Active automata learning, GUI, GUI learning, GUI abstraction, learning framework, Learnlib, JavaFX

## 1. INTRODUCTION

As Isberner, Howar and Steffen [12] have stated: "The wealth of model-based techniques developed in Software Engineer- such as model checking or model-based testing – is starkly contrasted with a frequent lack of formal models". In an ideal world, a model would be created before a system is deployed. Unfortunately, this often is not the case. Design-time behavioural models are typically hard and time-intensive to construct, especially if they should be complete, and they are almost guaranteed to get out of sync with the actual code. This hampers the application of formal validation techniques such as model-based testing or model checking, and thus most mistakes are most often found only after deployment. Automata learning has been proposed as a technique to solve this by automatically generating automata based behavioural models of systems where models are otherwise unavailable, incomplete or erroneous.

Active automata learning for DFA's with membership queries and equivalence queries was first presented by Angluin in 1987 [4]. It was adapted to Mealy machines by Niese in 2003 [17]. Since then other active automata learning algorithms have been created for learning DFA's and Mealy Machines; Steffen et al. contains a survey [21]. Recently efforts to learn other types of automata have been made, such as nominal automata [16], register automata [11] and I/O automata [3].

As Hower, Isberner and Steffen have shown in practice active automata learning has been applied create behavioural models for a plenitude of systems such as: CTI systems [9], web-applications [19], communication protocol entities [1], the new biometric European passport [2], botnets [6] and a network of integrated controllers in car doors [20]. As far as we know, only once has it been used in combinations with GUI's, namely to automate black-box GUI testing for android apps [8].

The main challenge for widespread adaptation of active automata learning is the effort required to design and implement application specific learning setups. This requires determining a suitable abstraction and finding ways to manage concrete runtime data that influences the behaviour of the target system. In [20], this process of finding a fitting abstraction and creating correct test drivers has been estimated to have taken around 27% of the total effort of analysing the network of integrated controllers in car doors [15].

Active automat learning aims at constructing an abstract model of a target system, better known as an SUL (System Under Learning). Communication with the SUL is often dependent on actions previously taken and data values previously transferred. M. Merten et al., in an effort to reduce test driver setup time proposed a reconfigurable and reusable test driver using interface analysis, mainly targeted at web applications. However, this reconfigurable test driver is not adequate for generating models of GUI-based programs because some of the intricate challenges this domain poses.

In this paper, we will answer the following research question: "*How can a behavioural model of a GUI-based Java program be generated through active automata learning in LearnLib*?" While doing this, we will answer the following sub questions: "*What is a suitable abstraction to understand and capture the behaviour of GUI-based programs?*", "*How could we model this in an automaton?*", "*What challenges does the GUI domain pose*?" and "*how could we generate learning setups for GUI-based programs*?".

*Outline:* In section 2, we will shortly summarise the how automata learning works and how it can learn a target system. Section 3 presents an abstraction of GUI-based programs, analyse how they work in the context of Java and JavaFX and propose an automaton which captures this behaviour. In section 4 we introduce the challenges of GUI-based program learning through a running example. Section 5 presents a learning setup and solutions to the challenges introduced in section 4 while section 6 introduces a framework for automatically generating these setups. In section 7, we will discuss the correctness and efficacy of the framework by applying it to a few use cases. The final sections contain the conclusion, future work, acknowledgements and references.


*27th Twente Student Conference on IT*, July 7st, 2017, Enschede, The Netherlands.

# 2. BACKGROUND

## 2.1 Automata

In this section, we briefly give the definitions of a *deterministic finite automaton* (*DFA*) and a *Mealy machine*.

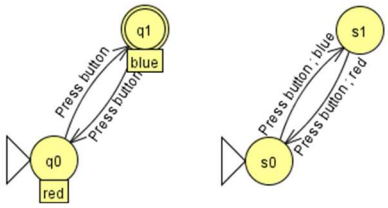


**Figure 1: Example behavioural model of a simple program based on a DFA (left) and a Mealy Machine (right)**

Let Σ be a finite set of *input symbols* $a_1,\ldots,a_k$. Sequences of input symbols are called *words*. The empty word (of length zero) is denoted by ε. We write *uv* when concatenating two words *u* and *v*. Finally, a language $\mathcal{L} \in \Sigma^*$ is a set of words.

**Definition 1 (Deterministic finite state machine).** *A deterministic finite state machine or deterministic finite automaton (DFA) is defined as a tuple* $\langle Q, q_0, \Sigma, \delta, F \rangle$*, where*

- $Q$ *is the finite set of states,*
- $q_0 \in Q$ *is the dedicated initial state,*
- $\Sigma$ *is the finite input alphabet,*
- $\delta : Q \times \Sigma \rightarrow Q$ *is the transition function and,*
- $F \in Q$ *is the set of final states.*

**Definition 2 (Mealy machine).** *A Mealy machine is defined as a tuple* $\langle S, s_0, \Sigma, \Omega, \delta, \lambda \rangle$*, where*

- $S$ *is the finite set of states,*
- $s_0 \in S$ *is the dedicated initial state,*
- $\Sigma$ *is the finite input alphabet,*
- $\Omega$ *is the finite output alphabet,*
- $\delta : S \times \Sigma \rightarrow S$ *is the transition function and,*
- $\lambda : S \times \Sigma \rightarrow \Omega$ *is the output function.*

Let's consider a very simple GUI-based program with one button which changes colour when a user clicks on it. Here above, the reader can find this example simulated as DFA and a Mealy machine. Here $Q = \{q_0, q_1\}$, (which represent that the button is red and blue respectively), $S = \{s_0, s_1\}$, $\Sigma = \{Press\ button\}$, $\Omega = \{blue, red\}$, δ and $\lambda$ are determined by what happens when you "Press the button" and $F = \{q_1\}$. The language $\mathcal{L} \in \Sigma^*$ of the automatons are all valid sequences of interactions which can be performed with the program. In this case it's (*Press button*)*

The difference between a Mealy machine and a DFA is that a Mealy machine has an output function and has defined behaviour for each input in each state while a DFA has a set of accepting states. Additionally, states in the DFA can represent pre-defined states of the program while in a mealy machine states are abstracted representations of conditions which lead to a given output.

## 2.2 Active Automata Learning

In this section, we present *active automata learning* and how it works in practice when used to learn a target system, better known as an *SUL (System Under Learning)*. We only summarise the basic concepts; a more detailed explanation and discussion can be found in [21].

As M.Merten et al. summarised: In active automata learning, "models of an SUL are created by active interaction and by reasoning on the observed output behaviour. This is done by constructing *queries*, which are sequences of input symbols from an alphabet that represents actions executable on the SUL, and answering these queries using actual execution." [19].

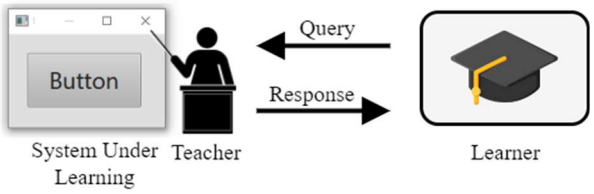


**Figure 2: High-level overview of a Learning Setup**

A *learning setup* generally consists of a *learner* and a *teacher*. The leaner implements the learning algorithm that interrogates the teacher and reasons about the observed output. The teacher answers the learner. The teacher is either the SUL itself or more often an adapter to the SUL. This is because generally the SUL has no interface to apply test cases for realising queries.

In more detail, the learning is done by the learner asking *membership queries* and *equivalence queries* to a teacher as to extract behavioural information and by successively refining its *hypothesis automaton* (the automaton which the learner believes models the SUL's behaviour), such as a DFA or a mealy machine, based on the answers. A membership query tests whether a word is accepted by the target system, that is the word is in the target system's automaton's language. Membership queries have to be independent: two queries with the same word should always lead to the same state. An equivalence query checks the hypothesis automaton for language equivalence with respect to the system (i.e. the constructed automaton correctly represents the behaviour of the SUL) and provides a counter-example if they are not equal. When they are equal, the learning procedure is successfully completed, and the generated model correctly describes the SUL's behaviour.

## 2.3 Challenges in Practical Applications

As Steffen, Howar and Merten have shown, there are five main challenges which one faces when using automata learning to learn real-world systems [10] [21].

1. *Interacting with real systems:* Interaction with real-world systems poses two problems: establishing an interface to apply test cases for realising membership queries and bridging the gap between abstract queries and actions executable on the SUL.

2. *Membership queries*: "Whereas small learning experiments generally require only a few hundred membership queries, learning realistic systems often requires several orders of magnitude more" [10]. Also, the speed of which the experimental simulation test environments can process membership queries is much faster than that of realistic test environments, in which a query could cost as much as a second or a minute.

3. *Resets:* Membership queries have to be independent. This is typically no problem for simulated systems but it may be quite a problem when dealing with realistic systems. This might require resets. which can be a very time expensive task, especially for server-based systems or systems interacting with databases.

4. *Parameters and value domains:* Automata learning is based on abstract communication alphabets. The parameters and interpreted values used in a system have to be expressed in the abstract alphabet. However, applications can have very complex or infinite alphabets.

5. *Equivalence queries*: Realising equivalence queries through testing is typically unrealistic, in particular when learning a

black-box system. In practice, equivalence queries will have to be approximated using membership queries.

Due to equivalence queries having to be approximated through membership queries, black-box active learning in practice is inherently neither correct nor complete. Without assuming extra knowledge, e.g. about the number of states of the system under learning the possibility of not having tested extensively enough will always remain and the model could have incorrect or missing states and transitions.

# 3. GUI-BASED PROGRAM ABSTRACTION

The chosen abstraction has a great influence on the expressiveness and usefulness of the final learned model. In this section, we propose an abstraction of GUI-based programs through abstracting the GUI and the design pattern used to create this program. We will then analyse how this abstraction is connected practically to Java and JavaFX (the new GUI library which is intended to replace Swing as the new standard) and propose an automaton which captures this behaviour.

## 3.1 GUI Abstraction

The abstraction presented in this section is inspired by [13] [14] but modified to better suit the level of abstraction we necessary.

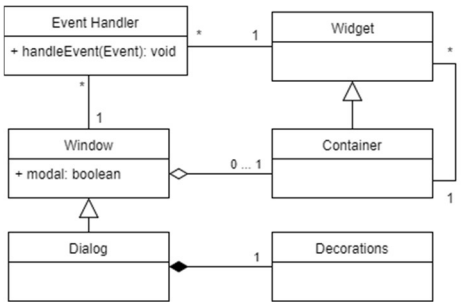


**Figure 3. GUI Class Diagram**

A *Graphical User Interface (GUI)* is a hierarchical, graphical front-end to a program that accepts user-generated and system-generated events as input, from a fixed set of events and produces deterministic output. A GUI contains *windows* that contain *widgets* (e.g. buttons, labels). Widgets have a fixed set of properties. At any time during the execution, these properties have discrete values, the set of which constitutes the state of the GUI. [13] *Containers* are a subclass of widgets. The difference between containers and widgets is that a container can contain other widgets. Examples of containers are grids, tables and menu bars.

At all times during interaction with the GUI, the user interacts with events within a *dialog window* also called a *dialog*. [14] This dialog window contains a container, called the *root container*, which can contain other containers. The difference between a dialog window and a regular window, whether is modal or modeless, is that a dialog window contains a container and has window decorations, typically consisting of a title bar along the top of the window and a minimal border around the other sides.
The most dominant paradigm used in GUI's is event-driven programming. This means that interactions with windows and widgets, such as clicking or dragging them generate events which are captures and processed by the windows' and widgets' event handlers.

## 3.2 MVC Pattern Abstraction

The *Model-View-Controller Pattern*, better known as the *MVC pattern*, is a software architectural pattern for implementing user interfaces. The pattern consists of three components: The *model*, which manages the data and logic of the application, the *view* which manages the output representation of the information in the model and the *controller,* which accepts user inputs and modifies the model and the view presentation of the information in the model.

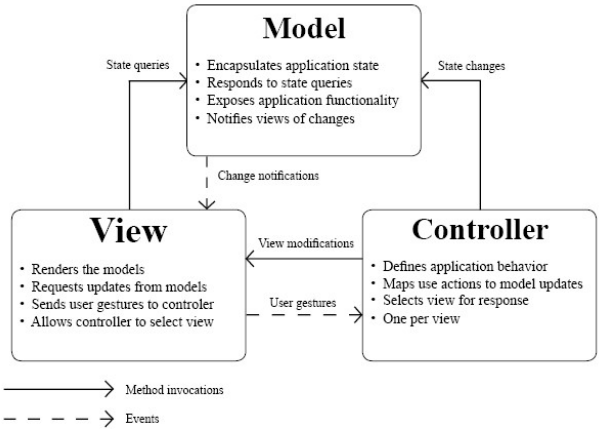


**Figure 4. Structure of the MVC pattern, inspired by [22]**

In GUI-based programs created with the MVC pattern we can observe two different kinds of events: *user gestures* and *change notifications*. These events are processed by *event handlers*. There are thus two different kinds of event handlers: *user gesture handlers* and *change notification handlers*. The set of user gesture handlers which can trigger at a given moment is called the *event space*. We call the union of all possible event spaces the *complete event space.* User gestures handlers can invoke *view modifications* and *state changes*. A view modification in the context of the MVC pattern is a change in the view while a state change is a modification of the model. Change notification handlers can invoke *state queries*. A state query is a query of information in the model.

During our research, we will consider that the GUI is the only means of interaction with the program. This means that change notifications can only be caused by user gestures.

There are four different kinds of view modifications: *widget property modifications, event space expansions, event space contractions and event space transmutations*. Widget property modifications are when the controller directly modifies properties of widgets, for example when a controller disables a button after it has been clicked. Event space expansions expand the set of currently triggerable event handlers. This would for example happen when a pop-up or a menu is opened or when a controller adds a new event handler to an already existing widget. Event space contractions is when event handlers are removed from the event space. This could, for example, occur whenever a window is closed.

## 3.3 Abstractions in Practice

### *3.3.1 GUI's in JavaFX*

The proposed abstraction in section 3.1 and 3.2 map very well with how JavaFX works in practice. In JavaFX, widgets are classes which implement the `EventTarget` interface (more about this in section 3.3.2). Examples of widgets in JavaFX are buttons, labels and menu items.

The containers in JavaFX are the classes which extend the `Parent` class. Examples of containers, which we will also use in our running example in section 4, are `AnchorPanes` (anchoring their children based on an offset from the pane's edges) and

`SplitPanes` (containing two `AnchorPanes` separated by a `Divider`). From this point on, whenever we talk in JavaFX about classes which implement the `EventTarget` interface we will call them widgets. If they also extend the `Parent` class we will call them containers.

In JavaFX, a class designated to act as a controller is both the controller and the view when we look at it through the MVC pattern. The controller is instantiated by the `FXMLLoader` when an FXML file, a file providing the structure of a screen of the user interface, is loaded and has it has specified this particular class to be a controller of this screen.

### *3.3.2 User gesture events & event handlers in JavaFX*

In JavaFX, there are eight classes of events which can be triggered by user gestures: `ContextMenuEvents`, `DragEvents`, `GestureEvents`, `InputMethodEvents`, `KeyEvents`, `MouseEvents`, `TouchEvents` and `Windowevents`.

Every event has a type. Event types are used to further classify the event classes. For example, the `KeyEvent` class contains `KEY_PRESSED`, `KEY_RELEASED` and `KEY_TYPED` event types. These types are hierarchical, and all have a super type. For example, the name of the event for a key being pressed is `KEY_PRESSED`, and the supertype is `KeyEvent.ANY`. This means that whenever a `KeyEvent.ANY` is propagated, event handlers of `KeyEvent.KEY_PRESSED` will also respond. From this point on, we will refer to events combined with their types, such as `KeyEvent.KEY_PRESSED` as `KEY_PRESSED` events.

Summarising the official JavaFX Oracle documentation: "The *event delivery process* consists of four steps: *Target selection, route construction, event capturing and event bubbling*. […] In the target selection phase, the system determines when the action occurs which [widget] is the target, based on internal rules. […] For key events, the target is the [widget] that has focus. For mouse events, the target is the [widget] at the location of the cursor. […] During the route construction phase, the *event dispatch chain* which the event should follow is created. […] The route can be modified when *event handlers* and *event filters* process the event. […] If an event handler or a filter consumes the event at any point, some [widgets] on the route may not receive the event " [18]. We will consider for our research that no route modifications or event consumptions are made by any of the widgets along the dispatch chain. "In the event capturing phase, the event is dispatched by the root [container] of the application and passed down the event dispatch chain to the target widget. […] [Afterwards,] in the event bubbling phase, the event returns along the dispatch chain from the target [widget] to the [root container]. […] Event handling is provided by event filters and event handlers. […] An event filter is executed during the event capturing phase […] [,] an event handler is executed during the event bubbling phase" [18]. During our research, we will not look at event filters, we will only take into account event handling done by event handlers.

Many widgets define *event handler properties*, which provide a way to register event handlers. Setting an event handler property to a user-defined event handler automatically registers the handler to receive the corresponding event type. Examples of such properties are `onKeyPressed`, `onKeyReleased` and `onKeyTyped`. We will focus on event handlers registered by event handler properties because it is the most common way of registering event handlers. It gives us a denumerable set of event handlers and it ensures each widget has no or one unique event handlers for each event. In JavaFX, no new event in response to a user gesture is thrown before the event before is fully processed. However, this means that whenever event handlers are badly designed and too much stress is put on the applications it could happen that events are not captured and skipped.

### *3.3.3 Change notification events & event handlers in Java*

With the MVC pattern the most common way of dealing with change notifications is through the *observer pattern*. In this pattern, one defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

In our simple Java implementation of the MVC pattern, we use a `Singleton` which extends the `Observable` class as a `Model`. The views implement the `Observer` interface. This interface contains the update method which is invoked whenever one of the models' observables changes and will then notify its observers about its change. The update method of an observer will not be executed on a new thread, it will be executed on the thread which notified the observers.

## 3.4 Abstract Model

Learnlib currently supports two types of state machines: DFAs and Mealy Machines. In 2013 plans have been made to also include Register Automata in the open-source version. In 2015 Cassel, Howar and Jonsson have created an extension for Learnlib named RALib that supports register automata [7]. Since the development of this addition seems to have stalled and RALib is not part of the official Learnlib ecosystem we have focused on DFAs and Mealy machines.

In general, it is desirable to use a DFA if the output sought after is a single boolean value that denotes whether the input sequence leads to a desired result and a Mealy machine to display the results itself.

Which kind of automaton is used as behavioural model impacts the way in which it reflects the GUI-based program as follows:

A model based on a DFA would show whether a sequence of user interactions would lead to the desired state of the system, effectively making it a test oracle on itself. Given for example a program with a few buttons which can influence the color of a shape, a DFA could find all button sequences which result in the shape being red or find all button sequences which lead to a bug, for instance the color of the shape not being the same as the color of the shape stored in the model.

A model based on a Mealy machine could show the state of the system by reporting it through its output function. Going back to our last example, a Mealy machine can be used to show which button sequences result in which colour. A Mealy machine could also show if a sequence of user interactions would lead to a desired state of the system by making the output true if this is the case. We could for example find all button sequences that result in the shape being red by making the mealy machine output true whenever the shape has become red.

Since a Mealy machine can do everything a DFA can and more in the context of representing GUI's, a Mealy machine is the best automaton for representing GUI-program behavior.

As defined before, a Mealy machine consists of a finite set of In the context based on the definition of a GUI we made in the section before, this would mean that in our behavioral model, $S$ is the finite set of states the GUI-based program can be in during execution (the combination of the state of the model, view and the controllers). and the alphabet $\Sigma$ the set of pairs of events with their respective event handlers.

The alphabet seems counterintuitive. Since the only way of interacting with the program is through user gestures at first glance the most logical thing would seem that the symbols of our input alphabet would solely consist of the user gestures. However, a user gesture in itself does not cause behaviour. The handling of the user gesture by a user gesture handler causes behaviour. We also cannot take the complete event space of a program as our input alphabet because event handlers behave differently given the same event with different parameters. For example, an event handler of a `MOUSE_CLICKED` event could behave differently given that the event represents a left mouse, right mouse or middle mouse click. Our input symbols are thus pairs of user gestures and the event handlers from the complete event space of the program which are triggered by these events.

Because we are generating the behavioural model through automata learning, they are state minimal. This means that a state $s \in S$ can represent multiple complete states of the program. Two complete states $s1, s2$ are represented by the same set $s \in S$ if the intersection of the properties of the states used in the transition and output functions are equivalent in regards to these functions, i.e. they give the same transition and output for the transition and output functions.

# 4. A RUNNING EXAMPLE

In this paper, we will discuss the generation of learning setups along the example of a simple GUI program in which users can interact with a shape.

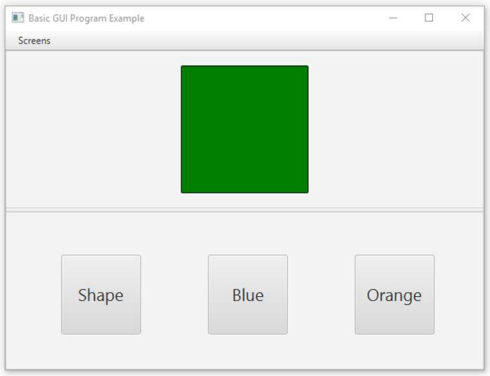


**Figure 5. Screen 1**

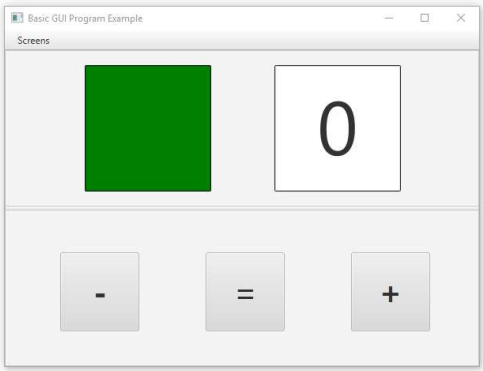


**Figure 6. Screen 2**

On each of the screens, the following interactions are exposed:

On the first screen, shown in figure 5, the user can change the shape of the figure, change the colour to a pre-determined colour and change screens through the screen menu.

On the second screen, shown in figure 6, the user can change the colour of the figure, which is different depending on the current colour stored in the model, with the minus, equal and plus buttons in a traffic light fashion. With the buttons, one can either reverse the traffic light (turning it from red to yellow, yellow to green, or green to red), keep it as it is or advances the traffic light (turning it from green to yellow, yellow to red and red to green). In addition to this, the number next to the can be changed through mouse interactions with the coloured figure and the `Pane` (a container in JavaFX) it is in. To lock this number, the user can click the white square around the number to remove the coloured figures event handlers.

When the program is started, screen 1 is presented to the user. Initially the figure is a green square.

When interacting with this example system, the following challenges have to be addressed, and we will refer to these challenges when elaborating on our learning generator:

*Event handler accessibility:* If a widget is not shown on a screen, is disabled, is not able to get focus or is a window while another modal window is active the event handler should not be triggered. For instance, none of the minus button's event handlers can be triggered while screen 2 is not displayed. In this case, the teacher should not invoke the event handlers referenced in input symbols and return the result of the corresponding output function but do nothing and return an error instead. If we do process the input symbol we create sequences of events handler invocations which would not be possible in practice.

*Widget event space transmutations:* A widget's set of event handlers can change. Event handlers can be added, removed or replaced during runtime. In the example of our program, event handlers can be removed from the shape by clicking the white square.

*Order of events:* Not all user interactions can be performed at any given moment. For example, users cannot click on a button before he moved their mouse to the button. Certain events can only be dispatched after other events, meaning that certain event handlers can only be triggered before or after other event handlers in certain situations. For instance, the figure's event handler on screen 2 of events with the type `MOUSE_CLICKED` or `MOUSE_DRAGGED` will always be triggered after one of the type `MOUSE_PRESSED`, while after an `MOUSE_EXITED` no event handler other than that of the `MOUSE_ENTERED` event can be triggered. For instance, in our program, the number mapped to the `MOUSE_CLICKED` event can never be displayed directly after the number mapped to the `MOUSE_EXITED` event.

With containers and widgets, we have the case that if a widget is contained in a container, a `MOUSE_MOVED` targeting the container will always be dispatched before a `MOUSE_MOVED` targeting the widget is dispatched. This also means that the event handler listening for the `MOUSE_MOVED` of the container is always triggered before the one of the widget. This means that for example the number mapped to the `MOUSE_MOVED` event of the widget can never be displayed after the `MOUSE_EXITED` event of the container.

# 5. LEARNING GUI-BASED PROGRAMS

In this section, we introduce our learning setup architecture and explain how we have solved the challenges presented above.

## 5.1 Learning Setup

`InputSymbol`: We use the `InputSymbol` class to wrap concrete methods and use these as alphabet symbols for the learning algorithm.

`Proxy`: The `Proxy` is the class which contains all controllers created during runtime. When they are created, they have to be added to the `Learner` through the generated setters in this class.

The `Proxy` also contains the information which input symbol corresponds to which widget, which widget listens to which events as well as which query corresponds to which event.

`Mapper`: The `Mapper` is the component which contains all the actions executable on the SUL and which output function it should use given the input symbol. When the query has been executed, it returns the output of the input symbols corresponding output function.

`Teacher`: This is the component which executes the queries. In our implementation, it is as an adapter funnelling learning queries to the SUL. Before an input symbol is mapped and invoked, the `Teacher` checks whether input symbols event handler should be accessible in normal running conditions and whether the next event can be dispatched given the inputs symbols before in the query and the order of events. It also ensures the independence of the queries.

`Leaner`: This is the component which performs the learning experiment. It creates the alphabet of the SUL, consisting of all but the constructor methods in the `Proxy`, the membership oracle based on the `Teacher`, the equivalence oracle based on the membership oracle and performs the learning experiment. When a final hypothesis behavioural model has been reached, it reports it back to the user.

## 5.2 Dealing with the challenges

### *5.2.1 Event handler accessibility*

There are five things which can influence event handler accessibility concerning mouse and keyboard events: whether the widget is disabled, whether a widget/window can get input focus, whether events are consumed when passing through the event dispatch chain, whether it is visible and its location in the rendered scene, e.g. a widget cannot receive certain mouse events if it is hidden behind another widget. Event handler accessibility can be guaranteed by the teacher by checking these conditions individually.

For our research, we have not taken into account the graphical hierarchy of the rendered scene, event consumptions in the event dispatch chain and focus. Furthermore, we have focused on programs with one window.

### *5.2.2 Event space changes*

The event space of a given program can change during runtime. We have solved the problem of event space changes by instead of storing a the event handler or an direct reference to the instantiated event handler, storing an indirect reference which widget variable it belongs and in which view this instance variable is defined. When an event handler is called, the view is obtained through the proxy. This frees us from specific instances of controllers, views, widgets and event handlers and allows them to be added, removed and changed during runtime. Before the event is passed to the event handler, the teacher also verifies that the controller, widget and event handler are instantiated as to prevent null pointers.

### *5.2.3 Order of events*

As introduced in section four, not all user interactions can be performed at any given moment. In term of automata learning, this means that the language of the SUL cannot accept certain words and we want to prevent them from being processed by the SUL. However, the teacher has no knowledge about this, and we have to give it this information. Going back to our example, a user cannot click on a button before they have moved their mouse to the button. This introduces a hierarchy of events and a constraint when each event can happen. In the case of JavaFX, this hierarchy is ensured by the event delivery process, which is deterministic by design.

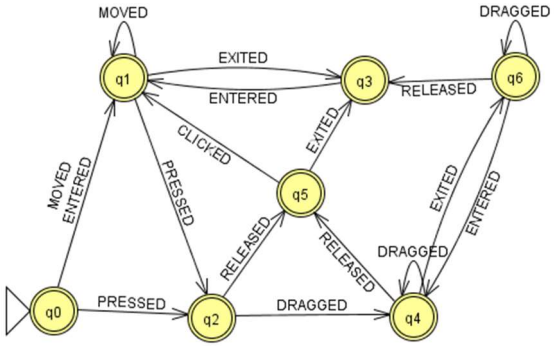


**Figure 7. DFA of valid MouseEvent sequences given a single widget**

Based on Oracle's JavaFX `MouseEvent` documentation, we have constructed a DFA representing the language of all valid `MouseEvent` sequences on a given widget. With this DFA the teacher can know if given an input symbol and the sequence of input symbols before this one in the query the action executable on the SUL represented by the input symbol could happen at that moment. If this is not the case, he should not execute the action but instead return an error.

However, when dealing with input sequences, we often do not deal with the complete `MouseEvent` alphabet but with a subset of the alphabet. For instance, if a widget only reacts when we drag or when we click on it, it means that the widgets have only an `MOUSE_PRESSED` and `MOUSE_CLICKED` event handlers in its complete event space. When we thus try to learn a system with this widget, only the `MOUSE_PRESSED` and with its event handler and the `MOUSE_CLICKED` event with its event handler will be added to the learner's alphabet as input symbols. When we thus test if sequences are valid or not, we only have sequences with these two input symbols. From the DFA we can conclude that `MOUSE_PRESSED` events should always come before a mouse `MOUSE_CLICKED` event, but an `MOUSE_PRESSED` event can occur without being, directly or indirectly, followed by an `MOUSE_CLICKED` event. This means that we need something which accepts words of the form `MOUSE_PRESSED* (MOUSE_PRESSED, MOUSE_PRESSED* MOUSE_CLICKED)* MOUSE_PRESSED*`. Otherwise said, given that we remove event types from the alphabet we will need to be able to reason about the validity of the sequences with words still left in the alphabet. We have resolved this issue by transforming the DFA into an NFA-ε, with ε-transitions replacing transitions of `MouseEvent` types not in the subset.

This leaves us with the problem that widget sequence validity can also depend on the widget hierarchy of a given scene. For instance, in our example in screen two the figure in the top part of the split pane is itself in an anchor pane. Because the figure is contained in a split pane, an `MOUSE_MOVED` event targeting the split pane is always dispatched before an `MOUSE_MOVED` event targeting the figure. Furthermore, when an `MOUSE_EXITED` event has been captured by the container, the widget could not receive any new events before the container captures an `MOUSE_ENTERED` event. This means that, to correctly validate the behaviour of containers and widgets we have to combine their NFA's and adjust them based on the widget's order in the hierarchy of the widgets. For our proof of concept, we have implemented the growing of the NFA in regards to all events except the `MOUSE_DRAGGED` event. This dramatically increased the size and complexity of the NFA without introducing any new theoretical challenges. In the case of the Mouse Event NFA, this is done by taking the current NFA, creating an NFA which describes all valid behaviour when the mouse is within the bounds of the new widget and connecting the new NFA to the

old NFA with the appropriate MOUSE_ENTERED, MOUSE_ENTERED_TARGET, MOUSE_EXITED and MOUSE_EXITED_TARGET events. MOUSE_ENTERED_TARGET and MOUSE_EXITED_TARGET events occurs when mouse enters a widget. It's the bubbling variant, which is delivered also to all parents of the entered widget.

### *5.2.4 Step algorithm*

In this section, we show how the solutions presented above could be implemented into the teacher.

**Algorithm** Processing an input symbol in the Teacher

**Precondition:**
If first step of a query, Pre has been excecuted

**Postcondition:**
If last step of a query, Post will be excecuted

```
1:  function STEP(inputsymbol is)
2:      e ← eventhandler of is
3:      w ← widget of e
4:      if w has e and e accessible then
5:          if hierarchy is empty or w not last in hierarchy
    and w not child of any container ∈ hiearchy then
6:              hierarchy ← [w]
7:              obs ← [w observable events]
8:              nfa ← nfa(obs)
9:              his ← []
10:         else if w not ∈ hierarchy and w child of
    container ∈ hiearchy then
11:             wls ← widgets after container in hierarchy
12:             for all wl ∈ wls do
13:                 pop hierarchy
14:                 pop obs
15:                 his filter wl observable events
16:             end for
17:             nfa ← nfa(obs)
18:         end if
19:         his ← his + [is]
20:         if nfa accepts his then
21:             return sul invoke is
22:         else
23:             return invalid sequence
24:         end if
25:     else
26:         return eventhandler inaccessible
27:     end if
28: end function
```

**Figure 8. Alg. for processing input symbols in the Teacher**

In the algorithm, his stands for history, e.g. the input symbols processed before the current one of this query.

The algorithm checks whether the event handler is accessible and whether the input symbol's mapped event handler can be invoked given the event history.

Intuitively, what this algorithm does is that after checking if the event handler is in the widget's event space and the event handler is accessible it handles 4 different cases: In the case when it is the first input symbol of the query or the widget of the input symbols event handler is not structurally related (i.e.,.is, not a child) to any of the containers of input symbols event handlers before, the NFA is updated to be able to simulate the widget. In the case the widget is not structurally related, we can just update the NFA to only simulate the new widget and clear the history because the new widget's events are not in any way dependent on/for the widgets in the old hierarchy. In the case where the new widget is a child of a widget somewhere in the hierarchy, all the widgets not structurally related to the new widget are removed from the hierarchy, and their events from the observable events because as before the new widgets events are not in any way dependent on the removed widgets' events. The NFA then is updated to be able to simulate this new hierarchy. In the last case, the widget is in the hierarchy, and nothing has to be changed. Afterwards, it adds input symbols mapped event to the event history and checks if the sequence is valid. One case is not shown in this algorithm: the handling of the DRAG_DETECTED event. This can be thrown at any moment after a drag has been detected. This can be dealt with by keeping two flags, one when a drag has started and one whenever a DRAG_DETECTED event has occurred. Whenever either the first flag is false or both are true, additional DRAG_DETECTED the word represents an invalid sequence.

# 6. LEARNING SETUP GENERATOR FRAMEWORK

A learning setup generator framework needs to know six things to generate a working learning setup: how the SUL can be instantiated, how methods can be invoked on the SUL, the alphabet (i.e. the set of methods that are to be invoked) of the SUL, method data values, output functions and how it can guarantee membership query independence. The key to efficient automated creation of learners is a method to easily define this information. We have found that through nine annotations we could mark the necessary information to automatically generate all components of the setup:

`@SystemUnderLearning`: class annotation which marks the JavaFX application class of the SUL.

`@Start`: method annotation to indicate a wrapper for the target system's application initialization and start methods. Methods annotated with `@Start` need to have a Stage parameter and can only be defined in the SUL's application class. In case of JavaFX, these are the `init()` and `start(JavaFX.stage.Stage)` methods.

`@Initialize(int order)`: method annotation which marks methods which have to be called after the application is instantiated but before an learning experiment can be initiated.

`@Model:` class annotation to indicate a singleton acting as a model in the context of the MVC pattern.

`@ViewController`: class annotation to indicate a class is acting as a controller of the JavaFX application. In the context of the MVC pattern, a JavaFX controller is also the view as well as the controller.

`@InputWidget(String[] eventHandlers)`: variable annotation which marks widgets and which event handlers from their complete event space should be added to the learner's alphabet, together with which output function should be used. The corresponding data values are automatically generated. The `eventHandlers` array contains Strings of the format `"[onEvent,outputID"]` in which `onEvent` is the name of the event handler and `outputID` the id of an output function, i.e. a method annotated with `@Output`.

`@InputSymbol:` method annotation to mark an input symbol which can directly be added to the learner's alphabet. Methods annotated with `@InputSymbol` should have no parameters and String as return type.

`@Output(String id):` method annotation to indicate a output function.

`@Pre(int order):` method annotation to mark which methods have to be called before a membership query can be posed so as to ensure query independence. The order variable sets the position of this function when the teacher prepares the SUL.

`@Post(int order):` method annotation to mark which methods have to be called after a membership query has been posed so as to ensure query independence. The order variable sets the position of this function when the teacher resets the SUL.

Through annotation processing all annotated methods and classes are collected, analysed and then the learning setup is generated.

We generate a single event for each event handler with basic predetermined values. The generation does not take any meta-data of the event handler into account except for its accepting event type. The code generation is done with JavaPoet.

# 7. VALIDATION

Our learner generator framework is difficult to verify. Inherently, since we are approximating equivalence queries through membership queries the possibility of not having tested extensively enough will always remain and thus there is always a risk of generating wrong models. Furthermore, large behavioural models are almost impossible to verify by hand. Testing them automatically provides us with the same problem as before: there is always the risk of not having tested extensively enough. Therefore, we have decided to verify our framework through small use-cases of which we know how the behavioural model should look and which can be checked manually.

We consider our framework correct for a gives use-case if it can generate a behavioural model which matches the expected behaviour within one minute.

All tests were executed on an Intel i7-6700HQ based laptop with 8 GB of memory and Learnlib version 0.12.0. The random seed used for the oracles was 42.

For the membership oracle, the learners used an `ExtensibleLStarMealyOracle` with a Classic LStar `ObservationTableCEXHandlers` and a `CloseFirst` closing strategy. For the equivalence oracle, all except the last of the learners used an `RandomWordsEQOracle` with a minimum of 20,a maximum of 30 input symbols per generated membership query, a maximum number of tests of 100 and a batch size of 10. In the last learner, the maximum number of tests was 1000. The batch size is the number of queries the oracle will process at the same time. Since our `Teacher` is not forkable, this has no influence on the results. We use a `RandomWordsEQOracle` instead of a `RandomWalkEQOracle` because the NFA-ε introduces many paths and branches which are easier to find when the oracles searches in breath instead of going in depth.

We have not included the test results for event space transmutation, as this was a trivial test. Furthermore, please note that in our examples, the `OnAction` event handlers are triggered whenever they are clicked. They are thus treated as click events.

## 7.1 Use case tests and results

**Use case 1:** Screen 1: orange, blue and change button. $\Sigma =$ *{<BlueButton.OnMouseClicked, MouseEvent event>, <OrangeButton.OnMouseClicked, MouseEvent event>, < OrangeButton.OnMouseClicked., MouseEvent event>}*

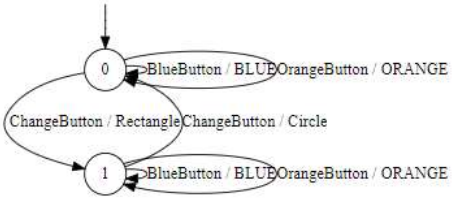


**Figure 9. Behavioural model of use case 1**

This behavioural model correctly captures the expected behaviour. When the blue button is pushed, the figure becomes blue, when the orange button is pushed; the button becomes orange and when the change button is pressed the figure's shape changes.

**Use case 2:** Screen 2: minus, equals and plus button. $\Sigma =$ *{"pairs of onclicked event handler references and mouse events for the minus, equals and plus buttons"}*

This behavioural model correctly captures the expected behaviour. In this test, we have made the program show screen 2 when started. When the plus button is pressed, the colour changes from red to green to yellow to red, the minus button does the inverse, and the equals button does not change the colour. It was created within 2 seconds.

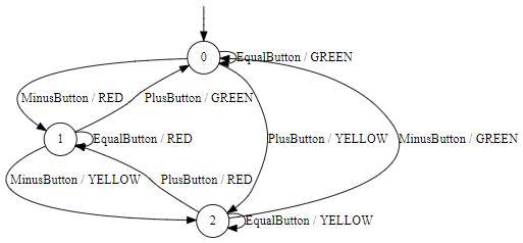


**Figure 10. Behavioural model of use case 2**

**Use case 3:** Screen 1 and 2: equal and blue button and the two menu items. $\Sigma =$ *{"pairs of onClicked event handler references and mouse events for the blue and equal buttons, pairs of onAction event handler references and action events for the menu items"}*

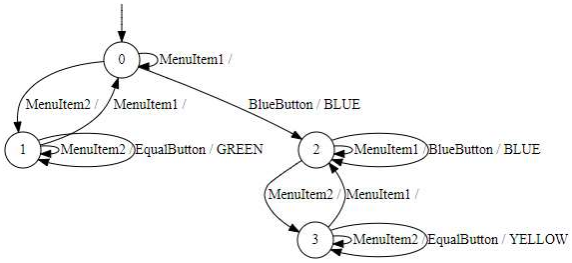


**Figure 11. Behavioural model of use case 3**

This behavioural model correctly captures the expected behaviour. When the program starts, Screen 1 is shown and the color of the shape is green. Going to screen 2 and pressing the equal button while it is green has no effect, the button stays green. Equals does not change the color of the shape except when it is not the shape is not green, yellow or red, in this case it makes the shape to yellow. It was created within 2 seconds.

**Use case 4:** Screen 2: White square. $\Sigma =$ *{"pairs of all mouse event handler references and corresponding mouse events"}*

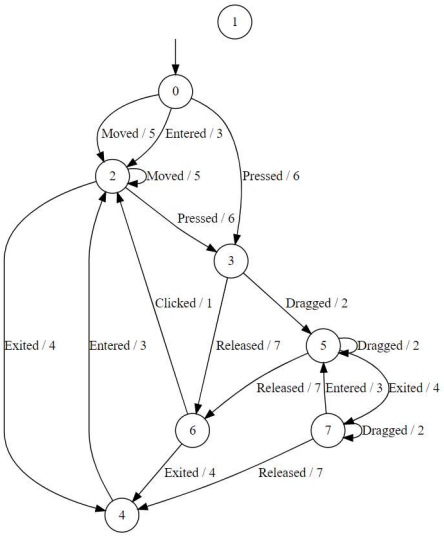


**Figure 12. Behavioural model of use case 4**

This behavioural model correctly captures the expected behaviour. The DFA is isomorph with the `MouseEvent` DFA presented in figure E. It was created within 2 seconds. We have tested the correctness of the `Mouse Event` DFA itself by connecting it to an example program and manually trying out if we can find a sequence which the DFA does not accept. We did not find one.

**Use case 5:** use case 4 but the exited, moved and pressed events are ignored by the event handler. $\Sigma = \{$*"pairs of all mouse event handler references and corresponding mouse events except for the exited, moved and mouse pressed event handlers"*$\}$

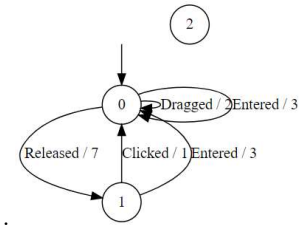


**Figure 13. Behavioral model of use case 5**

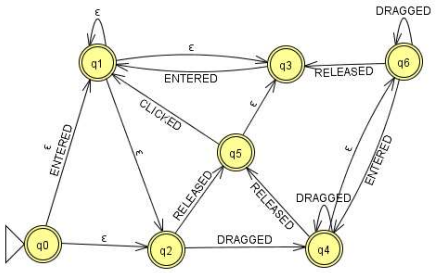


**Figure 14. `MouseEvent` NFA of use case 5**

This behavioral model correctly captures the expected behavior. It was created within 2 seconds. It accepts the same language as the `MouseEvent` NFA with the exited, moved and pressed event transitions being removed and replaced by ε-transitions. It was created within 2 seconds. We have tested the correctness of the `MouseEvent` NFA in a similar fashion as the DFA in use case 4.

**Use case 6:** Screen 1: change button and its container.
$\Sigma = \{$*"pairs of all mouse entered and exited event handlers with their corresponding events for the change button and its container as well as pair containing a reference to the change buttons onAction event handler and an action event"*$\}$

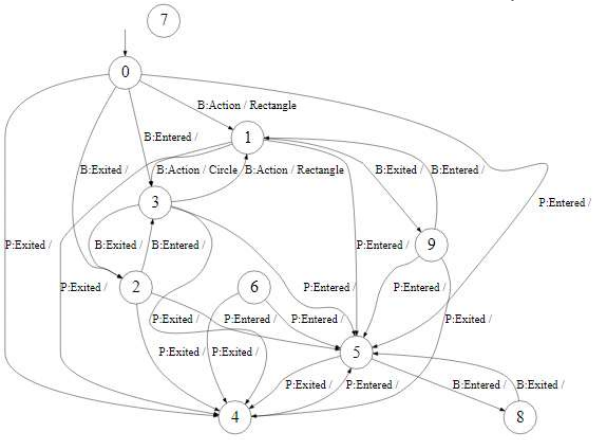


**Figure 15. Behavioural model of use case 6**

This behavioural model correctly captures the expected behaviour. It was created within 4 seconds. All the states where the `onAction` of the button can be triggered lead to states in which the `onAction` of the button can also be triggered. Furthermore, When the underlying pane is excited it has to be re-entered first before other events can be handled and whenever the button is not entered, expect from the beginning in the case was on the button from the start, the buttons `onAction` event handler cannot be triggered. The generated underlying NFA was tested in a similar fashion as use case 4 and use case 5.

## 7.2 Discussion

Although these tests do not prove that our framework and our developed techniques will work for all programs, we can conclude that the framework is able to generate correct learning setups for these basic use cases and the teacher correctly takes into account event handler accessibility, event space transmutation and the order of events for these use cases.

# 8. CONCLUSION

The main challenge for widespread adaptation of active automata learning is the effort required to design and implement application specific learning setups.

To achieve this, we have created an abstraction for GUI-based programs. All GUI elements fall within one of three categories: windows, widgets and containers. Widgets and containers have event handlers which handle user gesture events. We defined the set of user gesture handlers which can triggered at a given moment to be the event space. This information together with how the MVC architectural pattern is organized and how GUI's and the MVC pattern works in practice in Java and JavaFX has allowed us to define a Mealy machine based behavioural modal able to capture the behaviour of a GUI-program.

A learning setup which could learn GUI programs would have to consist of 5 parts: an `InputSymbol` class which wraps concrete methods and use these as alphabet symbols for the learning algorithm, a `Proxy` which contains all created controllers, a `Mapper` which contains all the actions executable on the SUL and which output function it should use given the input symbol, an `Teacher` which executes the queries, deals with the aforementioned challenges and ensures their independence and a `Learner` which performs the experiment. Nine annotations could provide an automatic learner generator with all information necessary to create this learning setup.

Learning GUI-based programs provide three challenges: Event handler accessibility (not all event handlers are accessible at any given time), event space transmutations (the event space can change during runtime) and the order of events (not all events can occur at any given moment). We solved these challenges in the teacher component by checking for event handler accessibility, event space transmutations and the order of events whenever we process an input symbol. We check if the order of events is valid by running the input symbol's mapped event together with those of the input symbols before through as a word through an NFA-ε with as language all valid sequences of mouse events given the set of containers and widgets involved and which could grow and shrink to accommodate a varying number and type of widgets and containers.

With our research, we have paved the way for a new area of automata learning. Now not only can we infer behavioural models of GUI-based programs but also creating learning setups is now a much less time-intensive and complicated endeavour, making practical adaptations much more feasible.

# 9. FUTURE WORK

There are a few more concepts which could be explored for better automated test driver generation. Three of which could be particularly interesting are advanced event generation, focus

handling and dealing with multiple windows. At the moment, we only generate basic events and do not take into account their parameters. However, these parameters can influence runtime behaviour. A future research could be in how to deal with this added complexity. We think this could be done through static code analysis to determine a finite approximation of the alphabet and deriving a symbolic model, as done by Berg, Jonsson and Raffelt to deal with infinite alphabets [5]. Additionally, at the moment we do not deal with focus and multiple windows. These concepts are closely related to the order of events. For example, to move focus from one screen to another screen we can move the mouse from one window to the other and when we click on it we switch focus between the screens. Future research could be to integrate this with/into the NFA-ε which checks the order of events.

## 10. ACKNOWLEDGEMENTS

Firstly, I would like to express my sincere gratitude to my advisor Prof. Arend Rensink for introducing me to this topic, his support and his critical view. I would like to thank Pim van Leeuwen for his support and the numerous mornings reviewing and discussing ideas. Furthermore, I would like to thanks Max Kerkers for taking his time to explain and help me with Learnlib. Last, I would thank my grandmother, Lenie Budde for her moral support and the good coffee.

## 11. REFERENCES

[1] Aarts, Fides, Jonsson, Bengt, and Uijen, Johan. Generating models of infinite-state communication protocols using regular inference with abstraction. In *IFIP International Conference on Testing Software and Systems* ( 2010), 188-204.

[2] Aarts, Fides, Schmaltz, Julien, and Vaandrager, Frits. Inference and abstraction of the biometric passport. *Leveraging Applications of Formal Methods, Verification, and Validation* (2010), 673-686.

[3] Aarts, Fides and Vaandrager, Frits. Learning I/O Automata. In Gastin, Paul and Laroussinie, François, eds., *CONCUR 2010 - Concurrency Theory: 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*. Springer Berlin Heidelberg, Berlin, 2010.

[4] Angluin, Dana. Learning regular sets from queries and counterexamples. *Information and Computation*, 75 (1987), 87-106.

[5] Berg, Therese, Jonsson, Bengt, and Raffelt, Harald. Regular inference for state machines using domains with equality tests. *Fundamental Approaches to Software Engineering* (2008), 317-331.

[6] Bossert, Georges, Hiet, Guillaume, and Henin, Thibaut. Modelling to simulate botnet command and control protocols for the evaluation of network intrusion detection systems. In *Network and Information Systems Security (SAR-SSI), 2011 Conference on* ( 2011), 1-8.

[7] Cassel, Sofia, Howar, Falk, and Jonsson, Bengt. RALib: A LearnLib extension for inferring EFSMs. *DIFTS. hp://www. faculty. ece. vt. edu/chaowang/dis2015/papers/paper*, 5 (2015).

[8] Choi, Wontae, Necula, George, and Sen, Koushik. Guided gui testing of android apps with minimal restart and approximate learning. In *ACM SIGPLAN Notices* ( 2013), 623-640.

[9] Hagerer, Andreas, Margaria, Tiziana, Niese, Oliver, Steffen, Bernhard, Brune, Georg, and Ide, Hans-Dieter. Efficient regression testing of CTI-systems: Testing a complex call-center solution. *Annual review of communication*, 55 (2001), 1033-1040.

[10] Howar, Falk, Isberner, Malte, and Steffen, Bernhard. Tutorial: automata learning in practice. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation* ( 2014), 499-513.

[11] Isberner, Malte, Howar, Falk, and Steffen, Bernhard. Learning register automata: from languages to program structures. *Machine Learning*, 96 (2014), 65-98.

[12] Isberner, Malte, Howar, Falk, and Steffen, Bernhard. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In *RV* ( 2014), 307-322.

[13] Memon, Atif M. An event-flow model of GUI-based applications for testing. *Software testing, verification and reliability*, 17 (2007), 137-157.

[14] Memon, Atif M., Banerjee, Ishan, and Nagarajan, Adithya. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *WCRE* ( 2003), 260.

[15] Merten, Maik, Isberner, Malte, Howar, Falk, Steffen, Bernhard, and Margaria, Tiziana. Automated learning setups in automata learning. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation* ( 2012), 591-607.

[16] Moerman, Joshua, Sammartino, Matteo, Silva, Alexandra, Klin, Bartek, and Szynwelski, Michał. Learning Nominal Automata. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA 2017), ACM, 613-625.

[17] Niese, Oliver. *An integrated approach to testing complex systems*. 2003.

[18] *Oracle Java Documentation JavaFX: Handling Events*. Accessed: 2017-06-27. http://docs.oracle.com/javafx/2/events/processing.htm#CEGJAAFD

[19] Raffelt, Harald, Merten, Maik, Steffen, Bernhard, and Margaria, Tiziana. Dynamic testing via automata learning. *International journal on software tools for technology transfer*, 11 (2009), 307.

[20] Shahbaz, Muzammil, Shashidhar, K. C., and Eschbach, Robert. Iterative refinement of specification for component based embedded systems. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* ( 2011), 276-286.

[21] Steffen, Bernhard, Howar, Falk, and Merten, Maik. Introduction to Active Automata Learning from a Practical Perspective. In Bernardo, Marco and Issarny, Valérie, eds., *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*. Springer Berlin Heidelberg, Berlin, 2011.

[22] *The NetBeans E-commerce Tutorial - Designing the Application*. Accessed: 2017-06-27. https://netbeans.org/kb/docs/javaee/ecommerce/design.html