Towards Model-Based Support for Regression Testing

Anna Gujgiczer, Márton Elekes, Oszkár Semeráth, András Vörös
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, Hungary
Email: gujgiczer.anna@gmail.com, marci543@gmail.com, semerath@mit.bme.hu, vori@mit.bme.hu

***Abstract*—Software is a continuously evolving product: modifications appear frequently to follow the changing requirements or to correct errors. However, these modifications might introduce additional errors. Regression testing is a method to verify the modified system, whether the development introduces new problems into the system or not. Regression testing involves the execution of numerous tests, usually written manually by the developers. However, construction and maintenance of the test suite requires huge effort. Many techniques exist to reduce the testing efforts. In this paper we introduce a model-based approach to reduce the number of regression tests by using abstraction techniques and focusing on the changing properties of the unit under test.**

I. Introduction

The development of complex, distributed and safety-critical systems yield a huge challenge to system engineers. Ensuring the correct behavior is especially difficult in evolving environments, where the frequent changes in demands lead to frequent redesign of the systems. This rapid evolution raises many problems: the new version of the system has to be verified in order to detect if it still fulfills the specification, i.e. the developments do not introduce additional problems or undesired modifications in the existing functionality. Regression testing is selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements. [5] It uncovers newly introduced software bugs, or regressions. Regression testing can determine whether a change in one part of the software affects other parts or functionalities.

Supporting regression testing is an important though difficult task. Creating a model for the desired behavior of the system could significantly help regression testing and would enable the application of model-based testing approaches. However developers usually do not have time and effort to create the specification model during the development and it is costly to construct the model afterwards from the source code and configuration files.

In this paper we introduce a model-based approach to support the regression testing of software components. We developed a methodology to automatically synthesize behavior models by using automata learning algorithms. However, traditional automata learning algorithms proved to be insufficient for this task, as they are unable to handle the sheer complexity of existing software components. Therefore some kind of abstraction framework is required to simplify the observable behavior of the unit under learning: these improvements can support the construction of a behavior model of even complex software components, which the state-of-the-art learning algorithm fails to learn. In our approach we use a feature model based abstraction on the interface of the software component. Based on the learned models we automatically generate a set of test sequences. Additionally, comparing the behavior models of the different versions of the software components is able to highlight unwanted changes. We also implemented the proposed approach in a prototype framework to prove its feasibility.

The structure of the paper: first of all, in Section II we introduce the required preliminaries and an example to guide the reader through the paper, later, Section III recommends a general approach for regression testing. Section IV introduces language support for defining the relevant behavior of the analysed software component. Section V shows preliminary measurements. Finally, Section VI concludes the paper.

II. Preliminaries

A. Example

A chess clock software used at the System Modeling Course [2] at BME serves as a motivating example. This chess clock has two main functionalities, let call them *menu* and *game* function. The game function enables the switch of the active player and descending its remaining time. In the menu the players can configure the settings of the game. The input interface of this chess clock contains four buttons (Start, Mode, White and Black) and the output interface consists of three displays (Main, White time, Black time).

B. Feature Model

Feature models [6] are widely used in the literature to provide a compact representation for software product lines. A feature model contains features in a tree structure, which represents the dependencies between the features.

The possible relations between a feature and its child- or subfeatures are categorized as:

- *Mandatory*: in this case the child feature is required.
- *Optional*: in this case the child feature is optional.

- *Or*: at least one of the subfeatures must be active, i.e. contained in the input or output, if the parent is contained in the message.
- *Alternative (Xor)*: exactly one subfeature must be selected.

Beside that, cross-tree constraints can be represented by additional edges.

A feature model configuration is a concrete set of features from a feature model which satisfies the constraints expressed by the relations of the feature model: a valid configuration does not violate any parent-child dependency or any cross-tree constraint.

C. Automata Learning

Automata learning is a method for producing the automaton-based behavior model of a unit by observing the interactions, i.e. inputs and outputs with its environment. There are two main types of automata learning, active and passive learning. An active automata learning algorithm was chosen in our work, as it can produce more accurate behavior models.

In active automata learning [1], [9], models of a Unit Under Learning (UUL) are created through active interaction, i.e. driving the UUL and by observing the output behavior. For this procedure the algorithm needs to be able to interact with the target unit in several ways, such as:

- reset the UUL,
- execute action on the UUL, i.e. drive it with an input,
- observe the outputs of the UUL.

Given the possible input and output alphabets of the software component the algorithm learns by constructing queries composed of input symbols from the alphabet, then these queries are asked from the UUL which responds by processing the inputs and providing the outputs.

III. Regression Testing Approach

In this section our regression testing approach and its basic steps are introduced. The proposed approach is depicted on Fig. 1. The method uses a user defined version of the software component as a reference: from this software component version the algorithm synthesize a behaviour model which can be used for test generation. Various test coverage criteria can be supported and many algorithms and tools are available to perform the test generation. This generated set of tests are then used for the later versions of the software component to check its conformance with the reference version.

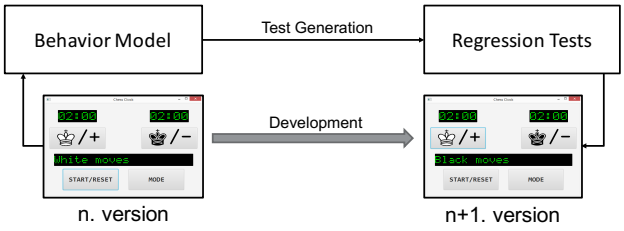

Fig. 1. Regression testing

Challenges. However, the envisioned approach faces some challenges. Automata learning algorithms construct the behavior model of the software component which is a *complex task in itself*. In addition, as a software usually expresses data dependent behavior, learning the automata model often becomes infeasible. Another important issue is that learning the automata model of the former version of the software component and generating test from it yields many test cases which should not hold in the newer version and will lead to many *false positive tests*. Our approach supports the learning algorithm with a specification language and abstraction technique to:

- enable the automata learning of software components with even complex, data dependent behaviour, and to
- focus the regression testing into the relevant parts of the software component, where the test engineer expect no change.

The overview of this process is depicted on Fig. 2. In the first step the framework learns the relevant behavior of the unit, defined by the user using feature models and abstraction. The result of this process is an automaton describing the behavior of the unit. Regression test cases can be easily generated using this automaton: a model-based test generation algorithm or tool can be chosen arbitrarily at this phase, the only question is the expected coverage criterion the tester needs.

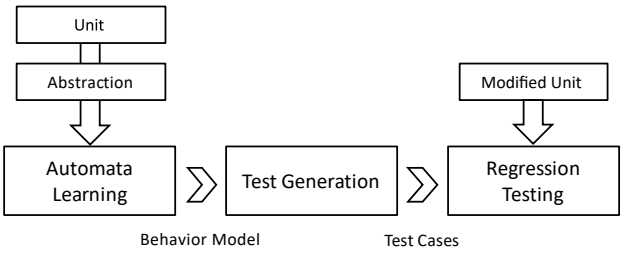

Fig. 2. Overview

In the following we introduce the main steps of the approach – construction of the behavior model and regression testing the modified component – in more details.

A. Constructing the Behavior Model

The first step of the process is the learning of the behavior model. In our work, we have integrated an active automata learning algorithm of LearnLib [8], which produces the behavior model as a Mealy machine [7] – a finite-state machine.

We used abstraction during the learning, as it hides the irrelevant parts of the behaviour and gives the user the means to focus the testing into the functions which are the scope of the regression testing. The user of our approach is able to formulate abstraction rules on the inputs and outputs, i.e. the alphabet of the learning process.

Figure 3. illustrates the role of abstraction during the learning process in the communication between the learning algorithm and the UUL. The queries generated by the learning algorithm are sequences of input symbols of the abstract

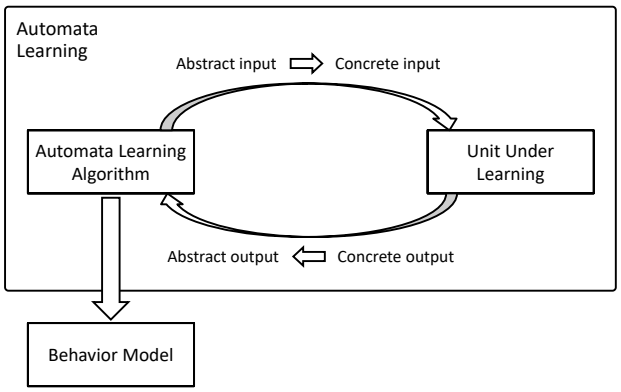

Fig. 3. Automata learning through abstraction

alphabet. These symbols are not directly executable inputs for the UUL, but they represent equivalence classes defined by the abstraction rules. Each abstract input needs to be concretized (choose a concrete executable action). The unit under learning will produce an answer for that particular action. According to the abstraction rules, the concrete response provided by the UUL is mapped to an abstract symbol consumed by the learning algorithm.

B. Regression Testing

Testing consists of the following steps: at first tests are generated from the previously learned behavior model. In the prototype a depth-first-search based test generation method was implemented which provided full state and transition coverage. Various test generation algorithms can be chosen to ensure the desired coverage. These tests can than be executed on the next version of the software component.

The automata learning algorithm constructs an automaton which is an abstract model in the sense that inputs and outputs can not directly drive the software component under test. In order to gain executable test, the abstraction and concretization steps used during the learning are saved so the prototype implementation can use it to produce executable tests.

The verification consists of two main steps:

- At first, the generated tests are executed to examine the new version of the software component.
- If the testing was successful, the framework compares the behavior model of the new version to the former one.

In the first step of verification we run the previously saved test cases on the newer version of the unit. If the result of any tests is an error, manual investigation is needed to decide the reason for the failing test. The reasons for a failing test can be the following:

- A real problem is found. The developers have to fix it.
- False positive occurs because of the inaccurate or not properly defined abstraction.

In order to further increase the efficiency of the analysis, the framework learns the new version of the software component to compare it to the former behavior model by using automaton minimization and equivalence checking.

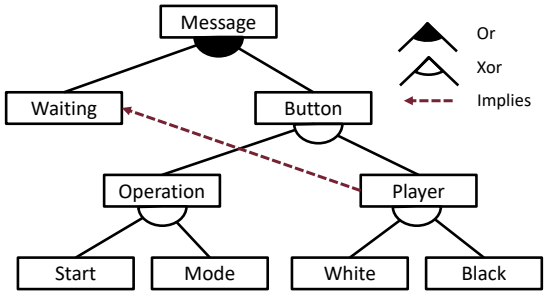

Fig. 4. Feature model

If any of these mentioned verification steps results in failure, we can assume that the modified unit does not conform to the desired behavior model.

IV. Language Support for Regression Testing

The presented approach requires the definition of the interfaces of the UUL, i.e. the possible inputs and outputs. In addition, the user has to define the relevant functionalities for the regression testing, i.e. the abstraction used in the learning process. In the framework, the interfaces and the abstractions are defined with the help of a feature model [6] language.

A. Specifying Communication Interfaces by Feature Models

In our setting, the feature model describes the set of possible input or output messages. Our feature model representation supports two types of features:

- Integer features: have a range of possible values from the integer domain. They can not have child elements.
- Boolean features: have Boolean values.

And it allows one kind of cross-tree constraint:

- Implication: if feature A requires feature B, the selection of A in an input or output implies the selection of B.

An example feature model is depicted by a feature diagram in Fig. 4. It defines the input interface of the chess clock, i.e. the set of all possible input messages. The root is the *Message* feature, which contains a *Waiting* time period, or a *Button* (specified by Or dependency). This represents that the players can wait or push a button to produce a kind of button input. The Waiting feature is an integer feature. The Button feature and its children are of Boolean types. The Button feature can either be an Operation or a Player button press (specified by Xor dependency).

An example input of the chess clock is the following: (1) pushing the white button, then (2) waiting $1s$. This is a possible valid configuration for the previously mentioned feature model.

B. Abstraction and concretization

We have chosen feature model based specification because it supports the formulation of the abstraction and concretization. In our work we implemented the following rules:

- Merge: Representing multiple features as an abstract one.
- Remove: The value of the feature will not be observed.

		Error in Menu 1	Error in Menu 2	Error in Game 1	Error in Game 2	Change
1)	Manual	10/18	2/18	1/18	0/18	8/18
2)	Without abstraction	-	-	-	-	-
3)	Game-focused abstr. 1	0/6	0/6	1/6	0/6	0/6
	Game-focused abstr. 2	0/51	0/51	9/51	0/51	0/51
	Menu-focused abstr. 1	4/6	1/6	0/6	0/6	2/6
	Menu-focused abstr. 2	4/6	1/6	0/6	0/6	2/6
	Menu-focused abstr. 3	6/6	1/6	0/6	0/6	6/6

TABLE I
COMPARISON OF MANUAL AND GENERATED TEST SUITES

Using the feature model of Fig. 4 we illustrate the effects of the abstractions: merging the White and Black buttons will lead to a simple feature model where the merged features will be represented by the Player button. Removing the Start button will result that its value is not observed by the learning algorithms.

V. EVALUATION

In order to evaluate the effectiveness of the proposed framework executed initial measurements on our prototype implementation.

Research questions. The goal of the measurements is to compare the effectiveness of the following sets of tests:

1) Manual test suite
2) Learning and test generation without abstraction
3) Generated test suite using a dedicated abstraction

We are interested in the following questions for each test suite:

RQ1 Is the test suite able to detect randomly injected errors?
RQ2 Is the test suite maintainable? How many modifications are required upon a change of the software?

Measurement method. For the measurements we used the previously presented chess clock statechart developed in YAKINDU [10]. This complex statechart has 12 states and 45 transitions and 9 variables, which results in several billion states when represented as a Mealy machine. In order to evaluate effectiveness of the previously mentioned methods we systematically injected 4 random atomic errors (in different regions of the state machine) to the state machine – motivated by [4] – and an intended change. The manual test suite covered all the transitions of the statechart. For the focused test suite we used 5 dedicated abstractions, e.g. removing the difference between the white and the black player.

Measurement result.: Table I summarizes the test results. Each row represents a testing method (denoted by 1)–3) in the research question). The columns represent various modifications in the software component: the first four are different kind of errors and the last one is an intended change. The cells represent the number of failed test cases in the form of *failed tests / all tests*. '-' represents timeout as we were not able to generate test cases without abstraction, because learning the unit timed out.

Analysis of the results: RQ1 The manually created tests were successfully executed and were able to detect several errors. However, when no abstraction was used, the technique was unable to learn the chess clock unit, thus it can not be directly used to generate test cases. But finally, when using suitable abstractions, the method was able to detect those kind of errors with less test cases. An error can remain hidden from both the manual and the generated test suites. In this case we can assume that we used a too coarse abstraction.

RQ2 Changes in the specification (in the functions of a program) invalidate many of the manual test cases, which have to be (partially) rewritten. However a suitable abstraction will reduce the number of false positive test cases so the test engineering efforts can be decreased.

VI. CONCLUSION AND FUTURE WORK

This paper introduced a model-based regression testing approach utilizing an automata learning algorithm to produce behavior model of a software component. A feature model based language is provided to support the definition of the relevant aspects of the interfaces and serve as the basis of the abstraction. User defined abstractions can drive the learning to focus on those parts of the software component that can be used as a specification model for the testing of the later versions. Our initial experiments showed that the direction is promising and hopefully it can reduce the regression testing efforts needed for testing software components.

In the future we plan to use an automatic abstraction refinement technique – based on the well-known CEGAR [3] – to automatically calculate abstractions.

ACKNOWLEDGMENT

This work was partially supported by the MTA-BME Lendület Research Group on Cyber-Physical Systems and the ÚNKP-16-1-I. New National Excellence Program of the Ministry of Human Capacities. Finally, we thank to Zoltán Micskei for his insightful comments.

REFERENCES

[1] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and computation*, 75(2):87–106, 1987.
[2] Budapest University of Technology and Economics. *System Modeling course (VIMIAA00)*. https://inf.mit.bme.hu/en/edu/courses/remo-en.
[3] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.
[4] SC Pinto Ferraz Fabbri, Márcio Eduardo Delamaro, José Carlos Maldonado, and Paulo Cesar Masiero. Mutation Analysis Testing for Finite State Machines. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 220–229. IEEE, 1994.
[5] ISO/IEC/IEEE. 24765:2017 Systems and Software Engineering-Vocabulary.
[6] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, DTIC Document, 1990.
[7] George H Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.
[8] Harald Raffelt, Bernhard Steffen, and Therese 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.
[9] Bernhard Steffen, Falk Howar, Malte Isberner, et al. Active Automata Learning: From DFAs to Interface Programs and Beyond. In *ICGI*, volume 21, pages 195–209, 2012.
[10] Yakindu Statechart Tools. *Yakindu*. http://statecharts.org/.