Towards Interactive Learning for Model-based Software Engineering

Áron Barcsa-Szabó, Balázs Várady, Rebeka Farkas, Vince Molnár, András Vörös
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, Hungary
Email: $\{abarcsa, balazs.varady\}@edu.bme.hu, \{farkasr, molnarv, vori\}@mit.bme.hu$

***Abstract*—Model-based technologies improve the efficiency of the design and development of IT systems by making it possible to automate verification, code generation and system analysis based on a formal model. A simple way of describing the behavior of systems is state-based modeling, which - due to the advancements of formal analysis techniques in recent years - can be widely and effectively utilized when analyzing systems. A possible way of synthesizing such models is to apply active automata learning algorithms. Acquiring a correct formal model of a system can be challenging because of the different theoretical and practical obstacles of both manual and automated approaches. We propose a semi-automated solution, that applies automata learning to provide an interactive environment for model development.**

***Index Terms*—Automata Learning, Formal Methods, Model-based Software Engineering, System Design**

I. Introduction

Model-based engineering is the formalized application of modeling during system design and development. It improves the efficiency of designing and developing IT systems by formalizing verification, code generation and system analysis, and in certain cases enables their automation as well. Such models can be designed both manually and in automated ways, both of which have their disadvantages. On one hand, it is difficult for the designing engineer to keep every property of the envisioned system in mind at a given time, partly because of the complexity of the system, and because of possible hidden implications and contradictions. Additionally, to conveniently specify requirements, different scopes, abstractions and formalisms may be applied, which may be difficult to reconcile. On the other hand, there are fully automated solutions – usually stricter in these aspects –, for instance, active automata learning, where the model construction is characterized by a teacher component - which is familiar with the extensive behavior of the system under learning - and a learner component – which synthesizes the model via queries to the teacher component. However, such solutions have practical boundaries when validating the inferred behavior of the system. The objective of our work is to support the design of systems and components from the ground up through a semi-automated solution – *InterActive* automata learning – which utilizes both the frequent input of the designing engineers and automated techniques. As a result of this approach, the users themselves are regarded as the teacher component of the algorithm, resolving the infeasibility of automated equivalence validation. This results in a semi-automated solution driven by declarative behavioral requirement specification, which allows the designing engineers to focus on the expected behavior of the system and on evaluating the model proposed by the algorithm. This paper presents an adaptive state-based modeling framework combining the advantages of manual and automated solutions, into which we designed and integrated the interactive algorithm. We created a proof-of-concept implementation of the approach, allowing system design through different formalisms, and the analysis and development of interactive automata learning algorithms to support model-driven development with an extended scope.

This work was partially supported by the ÚNKP-20-4-II New National Excellence Program of the Ministry for Innovation and Technology and by the National Research, Development and Innovation Fund of Hungary, financed under the 2019-2.1.1-EUREKA-2019-00001 funding scheme.

II. Background

A. Model-Based Engineering

Due to the application of the modeling concept in several completely different domains, first of all, we need to define the meaning of *model*.

Model A model is the simplified image of an element of the real or a hypothetical world (the system), that replaces the the system in certain considerations. Two main types of models exist in the literature: structural and behavioral.

Model-Based Systems Engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases [4]. This concept can also be applied to software engineering. Note, that the models may be the primary artifact of the development process, in which case precisely defined formal models are required. When the models are the primary artifacts, the process is called *Model-Driven Engineering*.

In this paper, our goal is to assist *MBSE* through defining a new metodology for requirement-driven **component** design.

B. Specifying Requirements

During our work, the concept of requirements is used widely, therefore, it is essential to define it precisely.

Requirement [1]

1) A condition or capability needed by the user to solve a problem or achieve an objective.
2) A condition or capability that must be met or possessed by a system component to satisfy a contract, standard, specification or other formally imposed documents.
3) A documented representation of a condition or capability as in (1) or (2).

Requirements are important, as both system design and verification depend on them. They can be specified in many different ways, the most common being textual requirements in traditional feature lists – an informal way of requirement specification.

The rationale behind the precise formalization of requirements is the wide range of automated applications, especially in *formal methods* – such as validation, formal verification or test generation. One possible requirement formalism is Linear-Time Temporal Logic (LTL) expressions over paths of automata models, resembling propositional logic extended with temporal connectives – e.g. X (true in the ne*X*t state), F (at some point in the *F*uture), G (*G*lobally) or U (*U*ntil a certain condition is met). One possible application of LTL is the transformation to Büchi-automata (or in general ω-automata), which then can be utilized in *formal verification* [3].

C. Automata Learning

Due to our solution utilizing automata learning to synthesise formal models, here we define its essential concepts.

Mealy machine A Mealy machine or Mealy automata is a Tuple of $M = (S, s_0, \Sigma, \Omega, \delta, \lambda)$, where:

- S is a finite, non-empty set containing the states of the automata,
- $s_0 \in S$ is the initial state,
- Σ is the input alphabet of the automata,
- Ω is the output alphabet of the automata,
- $\delta : Q \times \Sigma \to Q$ is the transition function and
- $\lambda : Q \times \Sigma \to \Omega$ is the output function.

A commonly used semantic helper can be defined over Mealy machines as $\lambda^* : S \times \Sigma^* \to \Omega$, defined by $\lambda^*(s, \epsilon) = \varnothing$ and $\lambda^*(s, w\alpha) = \lambda(\delta^*(s, w), \alpha)$, essentially providing the output of the automata after running an input sequence from a specified state.

Automata Learning is a way of modeling a system as an automata through observations without having specific knowledge of its internal behavior.

Formally: Automata learning is concerned with the problem of inferring an automata model for an unknown formal language L over some alphabet Σ [5].

Two types of automata learning approaches exist in the literature.

*a) **Passive Automata Learning**:* In case of passive automata learning, the gathering of information is not part of the learning process, but rather a prerequisite. The learning is performed on a pre-gathered positive an/or negative example set of the systems behavior. In passive automata learning, the success of the process is affected greatly by the methodology and time used to gather the data.

*b) **Active Automata Learning**:* In case of active automata learning, the behavioral information is gathered by the learning algorithm via queries. Active automata learning follows the Minimally Adequate Teacher (MAT) model proposed by Dana Angluin [2], which defines the separation of the algorithm to a teacher and a learner component in a way, where the teacher can only answer the minimally adequate queries needed to learn the system. These two queries are described in the following.

Membership query Given a $w \in \Sigma^*$ word, the query returns the $o \in \Omega$ output o corresponding to it, treating the word as a string of inputs. We write $mq(w) = o$ to denote that executing the query w on the system under learning (SUL) leads to the output o: $[\![\text{SUL}]\!](w) = o$ or $\lambda^*(s_0, w) = o$.

Equivalence query Given a hypothesis automata M, the query attempts to determine if the hypothesis is behaviorally equivalent to the SUL, and if not, finding the diverging behavior, and return with an example. We write $eq(H) = c$, where $c \in \Sigma^*$, to denote an equivalence query on hypothesis H, returning a counterexample c. The counter example provided is the sequence of inputs for which the output of system under learning and the output of the hypothesis differ: $[\![H]\!](c) \neq mq(c)$.

The learner component uses membership queries to construct a hypothesis automata, then refines this hypothesis by the counterexamples provided by equivalence queries. Once counterexamples can not be found this way, the learners hypothesis is behaviorally equivalent to the SUL. The learning can terminate and the output of the learning is the current hypothesis.

D. Simple Case Study

In order to illustrate the application of the proposed learning algorithm, we iterate through a running example of an engineer utilizing it to model a composite system of an intersection with three distinct components: a traffic light, a pedestrian light and a controller component. For the sake of conciseness, we only showcase the capabilities of the framework and omit certain steps of the modeling process.

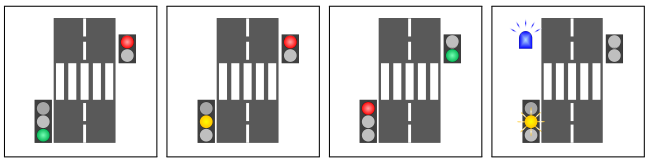

Fig. 2. Possible states of the system: normal operation and the interrupted state.

The functionality of the modeled system is simple: the traffic light loops through the red-green-yellow cycle, while the pedestrian light has a red-green cycle, as visualized on Figure 2. A controller component is responsible for their safe synchronization. Also, the police can interrupt the lights to be

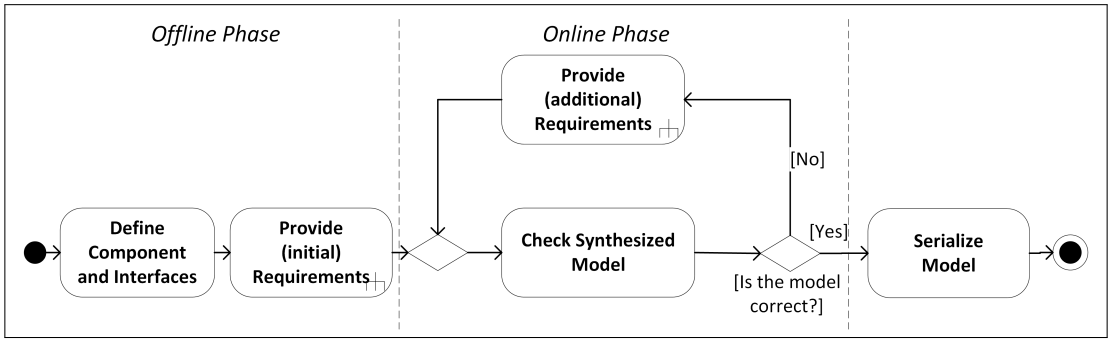

Fig. 1. The Proposed Workflow.

able to pass swiftly, in which case the traffic light switches to blinking yellow, and the pedestrian light switches off. After the police leaves the intersection, the lights return operation through a safe (red) state.

III. Overview of the Approach

Our methodology is heavily based on the interactions of the user with the proposed ***Interactive Learning Entity*** or **ILE** – characterized by the interactive automata learning framework. As Figure 3 illustrates, the two types of queries asked by the ILE are equivalent to that of active automata learning. In contrast to automated equivalence queries – where a non-approximate guarantee of correctness is infeasible – when delegated to a designing engineer, the only obstacles in correctness are the design skills of the engineer. Since no active automata learning algorithm exists in the literature which can learn interactively, while adapting to different types and the changing of reqirements, the ILE encompasses a new interactive algorithm to learn, paired with an oracle to adapt and keep the requirements consistent. The interactions with the ILE take place in a predefined order – the *proposed workflow*, illustrated on Figure 1. The workflow consists of two phases: first, an *offline* one, then an *online* one, and ends with the serialization of the models. During the offline phase, the ILE offers little assistance, the designing engineer must determine the required details by other means. The interactive system design happens during the online phase.

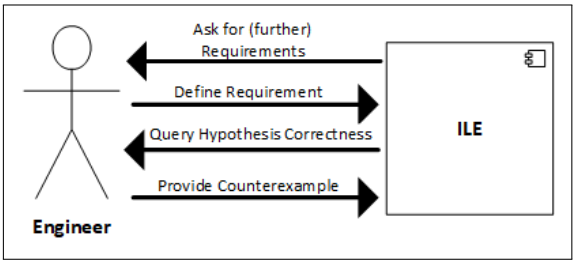

Fig. 3. Queries of the *ILE*.

a) ***Interface and Component Definitions****:* This step happens in the offline phase, as the determination of the system components, their exact boundaries and interfaces are part of the architecture, not the behavior. The engineer must provide the names of the system components, along with their interfaces – in other words their input- and output alphabets – before the workflow can proceed to the next step. During the online phase, the components are handled separately, and are connected later based on their defined interfaces.

In our running example, the engineer first provides the component *TrafficLight* with the input interface(s) containing $\{toggle,\ interrupt\}$ – corresponding to the light-switching sequence and the police interrupt respectively – and output interface(s) containing $\{red,\ green,\ yellow,\ blinkingYellow\}$ – corresponding to the states of a possibly connected basic traffic light display. Then the *PedestrianLight* and the *Controller* components follow, with interfaces defined through the behavior seen in Figure 2 and specified analogously to the previous example.

b) ***Providing Requirements****:* During the workflow, the engineers can provide requirements in both phases. These requirements can vary greatly in their scope – from being specific to individual runs to being generally valid for the whole component – in addition to the differences in the formalism the user defines them through.

In the offline phase, this means that the users add requirements they have formulated in advance. This is useful for more general requirements, with the scope of the whole component, easily formulated as program logic expressions, or long and complex traces.

In the online phase, adding requirements means answering the questions formulated by the automata learning algorithm about a yet unspecified behavior at a specific place in the trace currently being examined. This too can be answered through program logic – e.g. when the engineer realizes a general property during the model construction – but also through traces and through giving the corresponding output directly.

The ILE currently supports the following requirement formalisms:

1) Corresponding Output
2) Valid Trace
3) Invalid Trace
4) Sequence Diagram
5) LTL Expression

Conflicting requirements are expected during the system design, especially when abstraction refinement is applied. However, conflict handling is a difficult and resource intensive task for algorithmic reasons. Consequently, the ILE only guarantees to handle the conflict, when it also interferes with the model synthesis, in which case, the user is asked to remove one of the conflicting models.

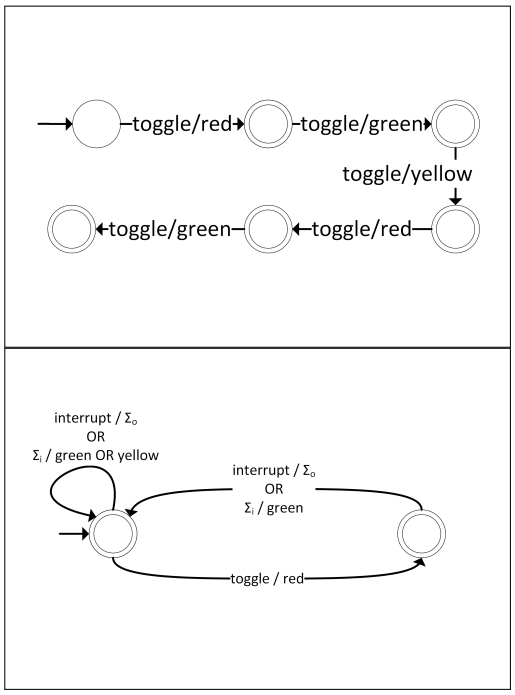

Fig. 4. Examples for attempting to model the requirement 'after toggle/red, for toggle the output is green' through a valid trace automaton (*upper*) and a Büchi automaton (*lower*).

For the traffic light component of our running example, the engineer may provide *toggle/red toggle/green toggle/yellow toggle/red toggle/green* as an initial, Valid Trace-type requirement to illustrate the toggle sequence of the traffic light. A similar, less redundant and more expressive LTL requirement can be formulated for a subset of this behavior, as 'after toggle/red, for the next toggle the output is green': $G(toggle \wedge red \rightarrow X(toggle \rightarrow green))$. The corresponding automata of the above two requirements are illustrated on Figure 4, note the the contrast in expressiveness and information capacity.

c) ***Checking the Correctness of the Synthesized Model:*** During the online phase, whenever the ILE assumes that it has gathered enough information to construct a model for the given component, the engineer is offered a model representing the current state of the model synthesis – the equivalent of an equivalence query in automata learning algorithms. The user can either approve this model – in which case the automata learning and therefore the designing of the behaviour is complete – or provide a counterexample where the model does not meet the – not yet specified – requirements.

The proposed equivalence model is a Mealy machine, which, based on the information provided by the user, can be incomplete in multiple ways. The behavior of the desired model can differ from that of the learned system because of lacking information, in which case the user needs to provide the separating behavior. Another reason for incompleteness can be newly discovered states, whose behavior is unknown based on their input signatures. This case prompts the user to evaluate the validity of state separation and to provide the lacking information. If, for some reason the hypothesized behavior is contradicting that of the desired system (by the users oversight in providing requirements), the actual, conflicting requirement can be provided to guide the learning algorithm through the process described in the previous subsection.

If the model is accepted, the design phase – for the currently learned component – is complete. If a counterexample is provided, the online phase resumes and the system design continues until the next possible model is reached.

In the case of our running example, the engineer might receive a model for equivalence validation, in which after a police interrupt, a consecutive second police interrupt does not end the blinking yellow state. Here a possible counterexample would be the input sequence $\{interrupt\ interrupt\}$, signaling a diverging behavior.

d) ***Creating and Serializing a Composite Model:*** When each of the component models declared during the first step of the workflow are completed, the resulting components are connected into a composite system model, which then can be serialized and handed over to the engineer for further (manual) extensions or usage, e.g. for code generation.

IV. Conclusion

In summary, we designed a new, semi-automated methodology to support component-based system design powered by adaptive automata learning. Additionally, we created a proof-of-concept implementation in order to validate our approach and demonstrated the capabilities and limitations of the implementation and the approach through a case study.

This paper is only the first step of designing the interactive learning methodology, we present our future goals in the following. We plan to thoroughly evaluate the complexity and applicability of our methodology, as they also depend on the skills of designing engineers. Since the by-design correctness of the components does not guarantee the correctness of the system, the inclusion of model-checking capabilities could further improve the methodology. Furthermore, as currently only Mealy machines are supported, we plan to also synthesize MBSE artifacts such as statecharts.

References

[1] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 1987.

[3] Bartha Tamás, Majzik István. *Biztonságra tervezés és biztonságigazolás formális módszerei*. Akadémiai Kiadó, 2019.

[4] Sanford Friedenthal, Regina Griego, and Mark Sampson. Incose model based systems engineering (mbse) initiative. 01 2009.

[5] Falk Howar and Bernhard Steffen. *Active Automata Learning in Practice*, pages 123–148. Springer International Publishing, Cham, 2018.