Integrating Model Verification and Self-Adaptation

Rafael V. Borges
Department of Computing
City University London
EC1V 0HB, London, UK
rafael.borges.1@soi.city.ac.uk

Artur d’Avila Garcez
Department of Computing
City University London
EC1V 0HB, London, UK
aag@soi.city.ac.uk

Luis C. Lamb
Institute of Informatics
Federal University of Rio Grande do Sul
Porto Alegre, RS, Brazil
LuisLamb@acm.org

ABSTRACT
In software development, formal verification plays an important role in improving the quality and safety of products and processes. Model checking is a successful approach to verification, used both in academic research and industrial applications. One important improvement regarding utilization of model checking is the development of automated processes to evolve models according to information obtained from verification. In this paper, we propose a new framework that make use of artificial intelligence and machine learning to generate and evolve models from partial descriptions and examples created by the model checking process. This was implemented as a tool that is integrated with a model checker. Our work extends model checking to be applicable when initial description of a system is not available, through observation of actual behaviour of this system. The framework is capable of integrated verification and evolution of abstract models, but also of reengineering partial models of a system.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software / Program Verification;
I.2.6 [Artificial Intelligence]: Learning

General Terms
Design, Verification

Keywords
Model Checking, Machine Learning, Neural-Symbolic Systems

1. INTRODUCTION
In the process of software development, formal verification plays an increasingly important role in improving the quality and safety of products and processes [3, 4, 7, 13, 21]. In the case of complex, autonomous and safety-critical systems, where the software may have to manage a large number of different scenarios, the use of model checking [3, 7] has been proven an effective option, increasing system reliability. Recent developments in autonomous systems have demanded new tools and theories capable of offering answers to the growing complexity and needs of such systems [14, 15]. Autonomous systems demand a new systems engineering paradigm, since one has to cater for the so-called self-* objectives or “broad requirements” [15]: self-configuring, self-healing, self-optimizing and self-protecting, which are met by the following attributes: self-awareness, self-situation, self-monitoring and self-adjustment. Formal methods in general, and model checking, in particular, thus have to develop new theories and tools to respond to such needs and challenges [7, 15, 20].

Model checking has been widely used in industry, including applications in hardware, software and artificial intelligence systems [4, 7, 13]. However, when specification problems or inconsistencies are found, model checking tools currently provide limited information about what should be corrected, usually as counter-examples of specific properties that have not been observed. Such information is then normally used by the developers to revise the system’s model in an ad-hoc process which is then followed by a new verification process - in a loop that will continue until a properly validated specification is achieved. Different techniques have been proposed to allow the automatic evolution of specifications [1, 4]. However, the use of verification tools is normally restricted to cases where an abstract description of the system follows a modeling language that is specific to the tool [21]. The use of different verification techniques, capable of learning an abstract description from the observed behaviour of the system, as proposed in this paper, would allow the application of formal verification methods and tools in the validation process.

In this paper, we propose to integrate the concepts of verification and adaptation through the use of hybrid neural-symbolic systems. Neural-symbolic systems are intelligent systems designed to combine certain advantages of two paradigms of Artificial Intelligence (AI): connectionist learning and symbolic reasoning (computation). This is done in AI in order to build robust tools that can offer principled knowledge representation, learning and computation [10, 23]. We propose a verification framework that allows the adaptation and evolution of models, and works in an integrated fashion with a model checking tool. In terms of evolution, the framework is capable of generating an improved model, based on the original description and the counter-examples produced by the model checker. The system can also work in the case where an initial description of the model is incomplete, incorrect or simply not given, through the learning of an abstract description based on the actual observed behaviour of the system under analysis. The main contribution of the work is the unification, in an automated process, of the verification of properties of a model, the evolution of this model according to such properties and the possibility of building the knowledge model from the observation of an actual system.
2. BACKGROUND AND RELATED WORK

The use of temporal models and logics have found a large number of applications in Computer Science [7, 16, 22]. Propositional logics, when extended with modal operators to represent a temporal dimension, provide a powerful framework for the representation and inference of a broad range of dynamic and temporal models [22]. One of the most successful applications of temporal logics in Software Engineering consists of Model Checking, a set of automated tools to perform formal verification of systems. In this approach, the software is described as a temporal model, in which the satisfiability of a property described in a temporal logic can be verified automatically [7]. An example of model checker is the NuSMV (New Symbolic Model Verification) [6], which implements different techniques for the verification of properties that can be expressed in different temporal logics languages.

The other main foundation of our work comes from the area of machine learning: the automated acquisition of knowledge. In the past decade, the discipline has been applied in different areas of Software Engineering. Among these, one may cite Black Box checking [21], which allows formal verification when abstract models of the system are not available by building a high level description of a system through the observation of its behaviour. Also, frameworks like CEGAR (Counter-example Guided Abstraction Refinement) [8] and the works of [11, 1] propose different approaches to integrate verification and learning, where the information generated by the verification process (usually presented as counter-examples) is used to guide the learning process to allow the evolution of software descriptions.

However, a strong criticism of symbolic machine learning approaches points them as too brittle to accomplish good learning performance in some real-world situations, such as the error-prone process of software systems specification [5, 18]. An alternative approach consists of noise-tolerant neural networks: computational models inspired by the human brain, where the adaptation task is performed through consecutive small changes of numerical parameters [17]. The subtlety of the changes in the evolution of knowledge, as well as the distribution of the processing in a parallel architecture, cater for greater robustness in the process of learning, with good performance even when considering noisy or missing information [17]. On the other hand, in trained neural networks, the learned knowledge encoded in the large set of numerical weights in the network structure cannot be understood or explained by an external entity [2]. This is a strong criticism of neural networks. In order to overcome this limitation (and extend the applicability of neural networks to include also symbolic disciplines), neural-symbolic systems translating knowledge from symbolic representations into neural networks and vice-versa have been proposed [5, 9, 19]. These neural-symbolic systems allow for robust learning, reasoning and explanation, and can, therefore, be integrated with symbolic model checking tools. In this paper, we will use the neural-symbolic methodology to overcome some of the limitations of purely symbolic learners in the process of model adaptation.

3. THE VERIFICATION AND ADAPTATION (V&A) FRAMEWORK

In this paper, we use NuSMV as the verification tool and the syntax used to describe the models. More specifically, NuSMV is used to verify if a described model satisfies properties expressed in temporal logic, returning a counter-example in the negative case. This counter-example is given in the form of a sequence of inputs and states that lead to an undesired state. Our proposed framework will consider the information from this counter-example in order to adapt the original model to satisfy the specified properties. Also, the framework will be able to deal with the case where no original model is given, as mentioned above, through the observation of the behaviour of an existing system. The observed behaviour can serve as training examples to create a model from scratch.

The adaptation part of our system is based on a neural-symbolic technique called Sequential Connectionist Temporal Logic (SCTL) [19]. In SCTL, background information represented in the form of a temporal logic program can be translated into the initial architecture of a neural network and then subject to adaptation through the use of standard network learning algorithms.

Figure 1 shows the different modules and information flow in our framework. The process can start with an initial knowledge model about the system, described in NuSMV (1a), or a set of observed examples of system behaviours (2). If a symbolic NuSMV description is given, the description is converted into a temporal logic program (1b), and then into a neural network (5) through the SCTL neural-symbolic tool, which will be used in the adaptation part. If, on the other hand, only the examples are given, the neural network is still capable of learning system behaviours that are more general than the examples themselves. The results can be described in the form of a new symbolic model (1) using a rule extraction neural-symbolic tool. In both cases, the symbolic model can be subject to verification by a model checker (4), where a counter-example (3) will be generated if such a model does not satisfy certain specified properties. Those counter-examples can be fed into the neural network, in a similar way that the examples are, starting a new adaptation process hopefully to improve the existing model. The adaptation process is repeated until the neural network (and its associated model) can deal properly with the counter-example. Finally, a new symbolic model can be obtained, again through rule extraction, and the process repeated. The evolved model will be subject to a new verification process, restarting the cycle that should be repeated until all the properties have been satisfied.

In what follows, we describe each of these individual steps, and the process of translating between the different representations. The framework is modular with the possibility of each module being transparent to a certain user; the neural-symbolic translations are computationally efficient and fully automated to allow this. Nevertheless, as with all machine learning systems, when learning is not successful to a predefined level of accuracy, expert intervention may be needed to fine-tune the system. In other words, the translations in the framework of Figure 1 are fully-automated, although the processes themselves both of model checking and adaptation are expected to be human-driven and elaborated, as usual.

![Figure 1: Structure of the proposed framework](image)

Given our choice of NuSMV as a tool for verification, and SCTL
for reasoning and learning, we have to consider two notations for the description of models: NuSMV descriptions and temporal logic programs. For the representation of models to be subject to the model checking procedure, we focus on a fragment of the NuSMV language, limiting the types of variables to allow only boolean or scalar variables and assuming determinism of the model. This allows the model to be translated into a temporal logic program following the SCTL language. Each logic program is based on temporal propositional logic, as defined by a set of clauses extended with information about the variables. Each atom (propositional variable) used in the description of a model will be either an input or a state variable. Input variables are those whose values are set externally to the model, while state variables have their values defined according to the model’s behaviour. To represent temporal sequences, we use the \( \diamond (\text{next time}) \) operator. Our approach to allow the treatment of scalar variables generates a group of atoms in the logic program to represent each possible value, inserting extra information into the program in a way that only one of the variables in the group is possible at any specific time point.

When the NuSMV model is subject to the verification of properties by the model checker, the system will return counter-examples whenever the properties are violated. The sequences of events given by each counter-example can then be used as a guide by the adaptation engine when deciding which changes are to be made to the model. The sequences used in the adaptation process consist of constraints regarding the initial and final states of the system, as well as the sequence of inputs between them. Our experience shows that this process of converting counter-examples into adaptation sequences should not be fully automated, allowing instead the intervention of an expert who can identify undesirable states in parts of a counter-example or reduce the specificity of the counter-example to represent a broader range of undesired cases. Automating this process would not be difficult, but would probably lead to a one-fits-all approach that could produce poor results in certain applications. A degree of flexibility is needed, notably at this point in the framework, to allow effective adaptation.

In order to create the neural network for the adaptation engine, we use the algorithm proposed in [12], which translates propositional logic programs \( P \) into a corresponding neural network \( N \). Positive assignments to the variables are represented by positive real numbers, and negative assignments by negative real numbers. The network can be seen as a black box with inputs and outputs representing the variables of the logic program, and the network computing a function \( f(N) \) mapping inputs to outputs which depends on the set of weights in the network’s connections. Following [19], the propagation of information through the neural network over time is implemented by recurrent connections from the output to the input layer of the network. In this way, a network output can be used to denote a system state at time \( t+1 \) given the input presented at time \( t \) and the transition function dictated by \( f(N) \). At the next time point, the current state becomes the new input and computation continues through the network, until a stable state is reached or a limit number of time points is reached. In this setting, adaptation is the process of changing the function \( f(N) \) computed by the network by progressively changing its weights according to input/output patterns (training examples) of desired network behaviour. For this task, we use a standard gradient-descent algorithm, called backpropagation, which has been used successfully in a number of applications [17].

At each timepoint \( t \), two steps are performed: in the feedforward step, the network computes the next state according to the values received from the assignments to the current input and state variables; in the backpropagation step, information needs to be given to the system indicating the desired value for the next state (i.e. the output), so that an error can be estimated as the difference between the actual output and the desired output. Backpropagation is known as a supervised learning algorithm because the adaptation process consist of presenting sequences of inputs to the network, informing also a suggestion for the next state, in our case through the values of a subset of the state variables. When we want the system to learn from examples of an observed behaviour, this process consists of using each example of the observed system to assign values to the input variables, as well as desired values to a subset of the state variables. When we want the system to learn from properties, the input and output values for the network are obtained from the counter-examples derived by model checking, as mentioned above and further detailed below. The flexibility of using a subset of the state variables allows the use of learning from examples when some of the state variables are not observable, such as in the case where this subset of states represents the actual outputs of the system.

In order to learn from counter-examples, instead of individual input/output pairs of examples, entire sequences are obtained from the model checker. These sequences will be used to determine the desired values of the output. During the adaptation process, every time the system reaches the initial state of a counter-example, this state is inserted into a list of active sequences together with an index to keep track of the network’s current position in the sequence of inputs. Therefore, for each new input presented, the system can verify which of these active sequences comply with the next applied value, eliminating from the list those that are violated. The definition of the inputs to be presented to the network happens according to the list of active sequences. For each timepoint, a property in the active list is selected randomly, and the conditions associated with such property are used to define the input to be applied to the network. For the definition of the desired value of the next state, the process uses all those active sequences that have reached the last input. In this case, information about the final state of the counter-example is used in the definition of the desired value. Computational tools have been implemented to deal with this process and the process of learning from examples of an observed system. The tool deals also with the case where there is a conflict between information provided by different sources (sequences and examples).

After the execution of the learning process, the new model description is hidden in the network in the form of numerical weights distributed across its structure. This has been pointed out by several authors as a major disadvantage [5, 2]. Several different approaches exist in the literature to overcome this problem. Below, we consider a simple pedagogical approach [2], whereby one considers the network as a black box and samples input vectors from a distribution, producing output values by querying the network and generating logical rules by combining the obtained input/output patterns.

Given a sequence of inputs obtained either from the set of examples or the learned properties, we store a record of all the state transitions happening in the system, generating a state transition diagram. For each input, a transition is stored containing information about the current state, the applied input and the obtained state/output, together with a weight that measures the confidence of the adaptation engine in this state transition. The obtained state will be defined by the sign of the numerical values in the network’s output, while the weight depends on the actual output values. All the transitions with the same source and target states and applied input are then grouped into one, being subject to a filtering according to the weights and the number of occurrences during the process. The remaining transitions after filtering can be displayed as a state transition diagram or written as a new set of logic program clauses and presented as a new NuSMV description.
4. WORKED EXAMPLE

We now illustrate the different steps of the framework. We use the Pump System example from [1]. The pump system monitors and controls the levels of water in a mine to avoid the risk of overflow, through three state variables: CrMeth indicating that the level of methane is critical, HiWater indicating a high level of water, and PumpOn, indicating that the pump is turned on. In order to turn on and off such indicators, six possible values of an input variable are considered. An original description of the system is described as a logic program in Table 1. This program was generated from the translation of an original description given in NuSMV.

\[
\begin{align*}
\text{CrMeth} &\leftrightarrow s : \text{Mon} \\
\text{HiWater} &\leftrightarrow s : \text{HiW} \\
\text{PumpOn} &\leftrightarrow s : \text{POn}
\end{align*}
\]

Table 1: Clauses for Pump System Example

We have run experiments using the pump system as testbed to evaluate the behaviour of the framework in the different adaptation scenarios. We started with the “black box checking” case, where a description is to be learned from a set of examples of a system’s behaviour, in such a way that this description can be used for further verification. The examples were given by a sequence of 1000 inputs and subsequent outputs (in this case, we assumed that the observable outputs of the system correspond to the entire set of states). A neural network was generated containing no prior knowledge about the domain, and was subject to the successive presentation of these examples. Rule extraction was then applied to the network to produce a state transition diagram and corresponding rule set. The network was capable of adapting its internal structure to correspond to the set of rules in Table 1.

We have then run the iterative process of adapting to properties. The process considers the following scenario: the original NuSMV model was presented to the model checker in order to verify the property that the pump should not be on when the level of methane is critical and the water is high. The model checker returned a counter-example, which was then used in the adaptation of the neural model. Information was extracted and a new verification process produced a new counter-example. After a third process of adaptation was carried out using yet another counter-example, a model and rule set was finally produced that does not violate the property.

5. CONCLUSION AND FUTURE WORK

In this paper, we have presented an integrated approach to model verification and adaptation. The role of adaptation in the presented hybrid neural-symbolic system is twofold: to evolve an initial model according to a specified property to be satisfied, and to create an abstract model from the observation of an existing system. The choice of NuSMV as the model checking tool allowed the representation of a general class of models and the potential use of different verification strategies. The adaptation engine can deal with incorrect prior knowledge and with noise in the examples in a supervised learning application. Neural-symbolic systems have been shown capable of learning static relations between inputs and outputs, but also intermediate states that represent observed sequences.

This paper has contributed towards promoting the use of formal methods, artificial intelligence and machine learning models and techniques in software systems development and engineering. A major current challenge in computer science consists of turning existing techniques into robust and practical tools for modeling complex and autonomous systems. This paper has contributed towards this challenge. Our next steps include further validation of the framework in realistic and noisy domains, theoretical and more practical studies of the use of counter-examples in the adaptation process, and formal description and presentation of the translation steps between NuSMV and neural networks, which would promote a more systematic study of this area.

6. REFERENCES