This manuscript version is made available under the CC-BY-NC-ND 4.0 license
http://creativecommons.org/licenses/by-nc-nd/4.0/

Policing Functions for Machine Learning Systems

Thai Son Hoang[1], Naoto Sato[2], Tomoyuki Myojin[2], Michael Butler[1], Yuichiroh Nakagawa[2], and Hideto Ogawa[2]

[1] ECS, University of Southampton, Southampton, U.K.
{t.s.hoang,mjb}@ecs.soton.ac.uk
[2] CTI, Hitachi Ltd., Yokohama, Japan
{naoto.sato.je,tomoyuki.myojin.fs,yuichiroh.nakagawa.hk,hideto.ogawa.cp}@hitachi.com

Abstract. Machine learning (ML) systems typically involve complex decision making mechanisms while lack clear and concise specifications. Demonstrating the quality of ML systems therefore is a challenging task. We propose an approach combining formal methods and metamorphic testing for improving the quality of ML systems. In particular, our framework enables the possibility of developing policing functions for runtime monitoring ML systems based on metamorphic relations.

Keywords: Policing function; Metamorphic Testing; Formal Methods; Machine Learning Systems;

1 Introduction

Machine learning (ML) systems typically involve complex decision making mechanisms and lack clear and concise specifications. ML systems build their knowledge by employing complex learning algorithms, which are often unpredictable. Specifications of these systems are often vague and ambiguous even at an intuitive level. As a result, demonstrating the quality of machine learning systems is a challenging task. Even though ML systems may produce correct output for a large set of input data, it cannot be guaranteed to work with all input data. One solution to this problem is to monitor the outputs of ML systems at runtime to ensure that incorrect behaviours can be detected.

In this paper, we propose an approach combining formal methods and metamorphic testing for improving the quality of ML systems. On the one hand, formal methods are mathematical-based technique for ensuring system dependability. In particular, formal methods allow system specifications to be captured and reasoned about precisely. On the other hand, metamorphic testing [3] is one possible approach to alleviate the *oracle problem*, i.e. testing of programs without a test oracle, which is often the case for ML systems. Metamorphic testing works by applying known transformations to inputs and checking that the outputs are consistent with the output produced for the untransformed input.

Our generic approach is as follows. To address the vagueness and ambiguous of the system specifications, we propose to use Event-B [1], a formal notation, to precisely capture system requirements and metamorphic relations. The metamorphic relations are then used as a basis for developing policing functions. Our framework is generic which can be applied to different types of machine learning systems with different metamorphic relations.

The rest of the paper is as follows. Section 2 gives some background information about the Event-B modelling method and metamorphic testing. Section 3 presents our approach for formally developing policing functions based on metamorphic relations. Section 4 discusses related work and Section 5 presents our conclusion.

2 Background

In this section, we give an overview of the Event-B modelling method (Section 2.1) and metamorphic testing (Section 2.2).

2.1 The Event-B Modelling Method

Event-B [1] is a formal method for system development. The main features of Event-B include the use of *refinement* to introduce system details gradually into the formal model. An Event-B model contains two parts: *contexts* and *machines*. Contexts contain *carrier sets*, *constants*, and *axioms* constraining the carrier sets and constants. Machines contain *variables* v, *invariants* $I(v)$ constraining the variables, and *events*. An event comprises a guard denoting its enabled-condition and an action describing how the variables are modified when the event is executed. In general, an event e has the following form, where t are the event parameters, $G(t, v)$ is the guard of the event, and $v := E(t, v)$ is the action of the event.[3]

---

1 $e =$ **any** t **where** $G(t,v)$ **then** $v := E(t, v)$ **end**

A machine in Event-B corresponds to a transition system where *variables* represent the states and *events* specify the transitions.

Contexts can be *extended* by adding new carrier sets, constants, axioms, and theorems. Machine M can be *refined* by machine N (we call M the abstract machine and N the concrete machine). The state of M and N are related by a gluing invariant $J(v, w)$ where v, w are variables of M and N, respectively. Intuitively, any "behaviour" exhibited by N can be simulated by M, with respect to the gluing invariant $J(v,w)$. Refinement in Event-B is reasoned event-wise. Consider an abstract event e and the corresponding concrete event f. Somewhat simplifying, we say that e is refined by f if f's guard is stronger than that of e and f's action is simulated by e's action, taking into account the gluing invariant J.

[3] Actions in general can be non-deterministic.

More information about Event-B can be found in [5]. Event-B is supported by the Rodin platform [2], an extensible toolkit which includes facilities for modelling, verifying the consistency of models using theorem proving and model checking techniques, and validating models with simulation-based approaches.

2.2 Metamorphic Testing

The idea of metamorphic testing was first proposed by Chen et al [3] as one possible way to alleviate the *oracle problem*, i.e. testing of a program without a test oracle. Often the test oracle knows the expected output for a given input, i.e., the oracle is aware of the input/output relationship for the program. In general, this might not be the case, e.g., for programs with complex inputs, that perform complicated computation, or programs produced by various machine learning techniques. For these kind of programs, predicting the correct output for a given input and compare it with the actual output of the program is non-trivial and error-prone [11].

Metamorphic testing based on the idea that it is easier to reason about the expected relations between different inputs and corresponding outputs of a program than to fully understand the input/out relationship of the program. An example of such a program is a machine learning *classifier* program for traffic sign images. It would be impossible to define an oracle that can map any traffic sign image to the corresponding traffic sign. However, if we consider image manipulation functions, such as, sharpen the image, we expect the the program will give the same as classifying the original image. Such a relationship between an input manipulation and the expected output relation is called a metamorphic relation. Metamorphic testing has been shown to find erroneous behaviours in image classification programs. For example, in [9], the Nvidia DAVE-2 self-driving car platofrm decides to turn left for an image, but incorrectly decides to turn right for a slightly darker version of the same image.

Since its introduction, metamorphic testing has been used for a range of applications [11]: embedded systems, web services and applications, computer graphics, simulation and modelling, machine learning, bioinformatics, variability and decision support, components, compilers, numerical programs, etc. For example, it has been considered for machine learning classifiers by Xie et al [12].

3 A Formal Approach for Policing Functions Based on Metamorphic Testing

The summary of our approach can be seen in Figure 1. The main idea of each step is as follow.

1. In the first step, we specify the expected inputs and outputs precisely but the precise relationship between inputs and outputs is specified loosely.
2. Subsequently, from the system specification, we construct the metamorphic relations, which are “properties” of the systems.

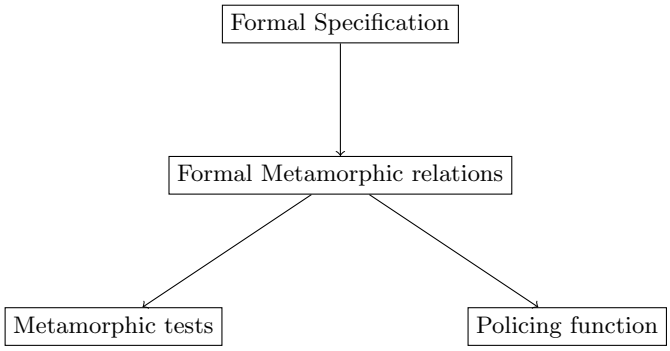

Fig. 1. Approach Overview

3. In the last step, the metamorphic relations are used (a) to construct the metamorphic tests and (b) as the specification for a policing function, which is developed formally using refinement technique.

In this paper, we focus on the development of the policing function using refinement technique.

In the subsequent sections, we discuss the main ideas: formal specifications of machine learning systems (Section 3.1), the construction of metamorphic relations (Section 3.2), and an architecture for formal policing functions (Section 3.3).

3.1 Formal Specifications of Machine Learning Systems

To address the vagueness and ambiguous of machine learning system specification, we use formal notations to specify the expected inputs and outputs precisely, but the precise relationship between inputs and outputs is specified loosely. In this paper, we use Event-B, a formal modelling language for this purpose, which support first-order logic with set theory.

At the abstract level, a deterministic program p can be seen as implementation of some function linking between the input i and output o. Note that, i and o can be a vector of inputs and outputs, respectively. For a non-deterministic program (e.g., a machine learning program continuously improve its algorithm at runtime), p is a *binary relation* linking the input i and the possible output o, i.e., $\mathsf{i} \mapsto \mathsf{o} \in \mathsf{p}$. (Here we use the the notation $\mathsf{i} \mapsto \mathsf{o}$ to denote an ordered pair). Without loss of generality, we assume that we consider that our programs are non-deterministic.

Similarly, the "ideal" specification of the program of can be defined *axiomatically*, as a binary relation s between the input i and the output o. A input/output pair of the program is incorrect if it does not "satisfied" the specification, i.e., $\mathsf{i} \mapsto \mathsf{o} \notin \mathsf{s}$. Note that the specification s can be defined constructively (e.g., finding shortest paths in a graph) or axiomatically (traffic sign image recognition).

3.2 Construction of Metamorphic Relations

Metamorphic relations are properties the systems, in particular, it specifies the expected relationship between the output of the system when the input is manipulated. The metamorphic relations are properties of the systems that can be formally and precisely specified.

In general, given a manipulation relation m on the input, we expect the relationship n holds for the output, i.e.,

$$\mathsf{i1} \mapsto \mathsf{o1} \in \mathsf{s} \land \mathsf{i2} \mapsto \mathsf{o2} \in \mathsf{s} \land \mathsf{m}(\mathsf{i1}, \mathsf{i2}) \Rightarrow \mathsf{n}(\mathsf{o1}, \mathsf{o2})$$

The pairs of relationships of the form (m, n) are the metamorphic relations. We call m the (metamorphic) input relation and n the (metamorphic) output relation. Note that (m, n) is defined for the ideal specification s.

The main idea of metamorphic testing is captured by the following

$$\begin{aligned}\mathsf{i1} \mapsto \mathsf{o1} \in \mathsf{p} \land \mathsf{i2} \mapsto \mathsf{o2} \in \mathsf{p} \land \mathsf{m}(\mathsf{i1}, \mathsf{i2}) \land \neg\mathsf{n}(\mathsf{o1}, \mathsf{o2}) \\ \Rightarrow \mathsf{i1} \mapsto \mathsf{o1} \notin \mathsf{s} \lor \mathsf{i2} \mapsto \mathsf{o2} \notin \mathsf{s}.\end{aligned}$$

The implication states that if the metamorphic input relation m holds between two observed input/output pairs $(\mathsf{i1}, \mathsf{o1})$ and $(\mathsf{i2}, \mathsf{o2})$ of program p but the metamorphic output relation n does not hold for the outputs $(\mathsf{o1}, \mathsf{o2})$, then one of the input/output pair does not satisfy the specification s.

3.3 Developing Formal Policing Functions from Metamorphic Relations

The metamorphic relations developed in the previous steps can be used for specifying a runtime policing function. The policing function is running alongside the ML system. The architecture of the policing function for the machine learning systems can be seen in Figure 2. The policing function reads the input and out-

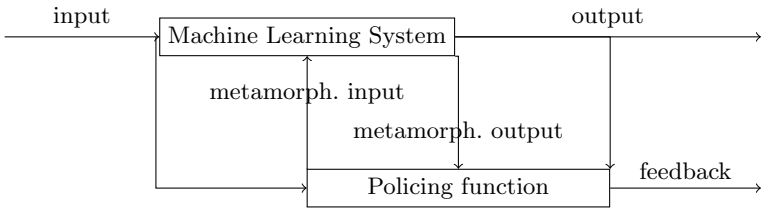

Fig. 2. Architecture of Policing Functions for Machine Learning Systems

put of the machine learning system, creates metamorphic input (according to the metamorphic input relation), and asks the machine learning system to produce the metamorphic output. The metamorphic output is then verified against the original output with respect to the metamorphic output relation.

The generic structure of the formal development of policing functions is as follows. At the abstract level, we model the program with the appropriate input, output. We first define a context c0 which declares the input, output types, i.e., I and O, and the type of the program p and the specification s. In the following, we present the abstract example on the left and a concrete example (traffic sign recognition) on the right.

```
context c0
sets
 I // Input type
 O // Output type
constants
 p
 s
axioms
 @typeof_p:
  "p ∈ ℙ(I × O)"
 @typeof_s:
  "s ∈ ℙ(I × O)"
end
```

```
context c0_trafficsign
sets
 SIGN_IMAGE // Input type
 SIGN // Output type
constants
 SignRecognitionProg
 SignRecognition
axioms
 @typeof_p:
  "SignRecognitionProg ∈ ℙ(SIGN_IMAGE × SIGN)"
 @typeof_SignRecognition:
  "SignRecognition ∈ ℙ(SIGN_IMAGE × SIGN)"
end
```

The behaviour of the policed system is modelled abstractly by machine m0 which contains two events, normal and error. Event normal corresponds to the normal execution of the program and event error specifies the situation where our policing function detects some errors. We model the input and output of the program using parameters i and o of the events. Event error has an additional parameter error_set to denote a set of input/output pairs which are produced by the program p (grd2), contains the original input/output pair (grd3), and contains a input/output pair $x \mapsto y$ which does not satisfy the specification s. Note that due to the nature of metamorphic testing, it is not possible to know exactly which input/output pair does not satisfy the specification s. We will show in a refinement of mo how violation of the specification is detected by failure of metamorphic testing.

```
machine m0
sees c0
events
 normal
 any i o where
  @grd1: "i ↦ o ∈ p"
 end

 error
 any i o error_set where
  @grd1: "i ↦ o ∈ p"
  @grd2: "error_set ⊆ p"
  @grd3: "i ↦ o ∈ error_set"
  @grd4: "∃x, y · x ↦ y ∈ error_set
      ∧ x ↦ y ∉ s"
 end
end

```

```
machine m0_trafficsign
sees c0_trafficsign
events
 normal
 any si s where
  @grd1: "si ↦ s ∈ SignRecognitionProg"
 end

 error
 any si s error_set where
  @grd1: "si ↦ s ∈ SignRecognitionProg"
  @grd2: "error_set ⊆ SignRecognitionProg"
  @grd3: "si ↦ s ∈ error_set"
  @grd4: "∃x, y · x ↦ y ∈ error_set ∧ x ↦ y ∉
      SignRecognition"
 end
end

```

In the next refinement, we introduce the metamorphic relations (m, n) an extended context as follows. The example of traffic sign recognition is on the right-hand side.

```
context c1
extends c0
constants
 m
 n
axioms
 @typeof_m: "m ∈ I ↔ I"
 @typeof_n: "n ∈ O ↔ O"
 @metamorphic_relation: "∀i1, o1
     , i2, o2 · i1 ↦ o1 ∈ s ∧ i2 ↦
     o2 ∈ s ∧ i1 ↦ i2 ∈ m ⇒ o1
     ↦ o2 ∈ n"
end
```

```
context c1_trafficsign
extends c0_trafficsign
constants
 Sharpen
    // We use = for output relation.
axioms
 @typeof_m: "Sharpen ∈ SIGN_IMAGE ↔
     SIGN_IMAGE"
 @metamorphic_relation: "∀si1, s1, si2, s2 · si1
     ↦ s1 ∈ SignRecognition ∧ si2 ↦ s2 ∈
     SignRecognition ∧ si1 ↦ si2 ∈ Sharpen ⇒
     o1 = o2"
end
```

We refine the event error as follows (abstract on the left-hand side, concrete example on the right-hand side).

```
error
any i o i2 o2 where
 @grd1: "i ↦ o ∈ p"
 @grd2: "i2 ↦ o2 ∈ p"
 @grd3: "i ↦ i2 ∈ m"
 @grd4: "o ↦ o2 ∉ n"
with
 @error_set: "error_set = {i ↦ o, i2 ↦ o2}"
end
```

```
sharpen
any i o i2 o2 where
 @grd1: "i ↦ o ∈ SignRecognitionProg"
 @grd2: "i2 ↦ o2 ∈ SignRecognitionProg"
 @grd3: "i ↦ i2 ∈ Sharpen"
 @grd4: "o ≠ o2"
with
 @error_set: "error_set = {i ↦ o, i2 ↦ o2}"
end
```

The witness for error_set (using the **with** keyword) indicates that the error set containing two input/output pairs (the original pair (i, o) and the metamorphic input/output pair (i2, o2)), does not satisfy the metamorphic relation (m, n). The correctness refinement proof (omitted here) relies on the definition of the metamorphic relation (m, n).

Note that a program can have several metamorphic relations. Each metamorphic relation corresponds to an event refinement with similar structure as above. We have applied the above architecture to define a policing function for a traffic sign image recognition program, and a program finding shortest paths. Due to the page limit, we omit the details here, the developments can be found in http://users.ecs.soton.ac.uk/tsh2n14/developments/VaVas2018/.

4 Related Work

A survey on metamorphic testing is presented in [11]. It shows that machine learning corresponds to 7% of the application domains. *Metamorphic validation* is an extension of metamorphic testing in [8]. The proposed framework is applicable to the validation of simulation models, which helps to building confidence in the validity of complex simulation models. The authors created a systematic process of discovery and application of "metamorphic relations". Here, the metamorphic relations in metamorphic validation will often be a definition of changing behaviour given a change in parameters or model design, instead of a relationship between two sets of test cases. Ding et al [4] applied the idea of metamorphic validation for deep learning frameworks and three different levels: system level, data set level, and data item level. At the system level, a metamorphic relation is based on the performance (e.g., classification accuracy) of different classifiers. For example, a deep leanring classifier is expected to have better accuracy than a corresponding Support Vector Machine (SVM) classifier. Metamorphic relations at the data set level include manipulation of various data sets, i.e., training data set, validation data set or test data set. As an example, adding 10% of new images into each category of the training (or validation or test) data set should not effect the classification accuracy. Metamorphic relations at the data item level consider the performance of the classifiers when some manipulations are applied to all items of the data set. For example, the

accuracy should be almost the same if the training images are cropped from the original images using different stride distances. Overall, metamorphic validation can be used to increase the confidence in the validity of ML systems, but its focus is on the validaity of learning processes rather than the resulting program itself. Manipulation of the data item level is also considered by Xie et al [12] for machine learning classifiers.

In [10], the authors propose a framework called VeriVis for computer vision systems. While they do not use the term metamorphic testing, their definition of *locally safe* basically corresponds to an (abstract) metamorphic relation. In particular, for computer vision system, they studied common transformations (which we call manipulation function) which simulate a wide range of common real-world image distortions and deformations. This includes various smoothing techniques, erosion, dilation, etc. This will be useful for us to build policing functions around these transformations.

5 Conclusion

At the moment, our work is carried out at the *theoretical level*, in particular, focusing on how formal specification can be used to help with the design of metamorphic relations and formal policing functions. We need to evaluate the approach on publicly available examples (benchmarks), e.g., hand-written images of digits (MNIST), imageNet large-scale visual recognition challenge (ILSVRC) [6], small colour image recognition (CIFAR-10) [7]. While most existing benchmarks are for classifications, object detection [6], i.e., having additional output of identifying objects within the image, will enable more metamorphic testing relations to be defined for the new additional output. As for any runtime verification, it is crucial that any runtime monitoring function has acceptable real-time performance. As a result, we need to implement and evaluate the performance of the policing functions.

In general, machine learning programs are only a part of a larger software system. Studying the consequences of the possible defects of the programs within the whole system will allow us to focus our efforts in testing and verifying the important aspect of the program. For example, recognising a green signal as a red signal might not be harmful, but recognising a red signal as a green signal can lead to significant safety issues for the whole system.

All data supporting this study are openly available from the University of Southampton repository at https://doi.org/10.5258/SOTON/D0528

References

1. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
2. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, November 2010.

3. Tsong Yueh Chen, T. H. Tse, and Zhiquan Zhou. Fault-based testing in the absence of an oracle. In *25th International Computer Software and Applications Conference (COMPSAC 2001), Invigorating Software Development, 8-12 October 2001, Chicago, IL, USA*, page 172. IEEE Computer Society, 2001.
4. Junhua Ding, Xiaojun Kang, and Xin-Hua Hu. Validating a deep learning framework by metamorphic testing. In *2nd IEEE/ACM International Workshop on Metamorphic Testing, MET@ICSE 2017, Buenos Aires, Argentina, May 22, 2017*, pages 28–34. IEEE Computer Society, 2017.
5. Thai Son Hoang. An introduction to the Event-B modelling method. In *Industrial Deployment of System Engineering Methods*, pages 211–236. Springer-Verlag, 2013.
6. ImageNet. Large scale visual recognition challenge. http://www.image-net.org/challenges/LSVRC/.
7. Alex Krizhevsky. Learning multiple layers of features from tiny images. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf, April 2009.
8. Megan M. Olsen and Mohammad S. Raunak. Metamorphic validation for agent-based simulation models. In Floriano De Rango and José Luis Risco-Martín, editors, *Proceedings of the Summer Computer Simulation Conference, SummerSim 2016, Montreal, QC, Canada, July 24-27, 2016*, page 33. Society for Computer Simulation International / ACM DL, 2016.
9. Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 1–18. ACM, 2017.
10. Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Towards practical verification of machine learning: The case of computer vision systems. *CoRR*, abs/1712.01785, 2017.
11. Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. A survey on metamorphic testing. *IEEE Trans. Software Eng.*, 42(9):805–824, 2016.
12. Xiaoyuan Xie, Joshua W. K. Ho, Christian Murphy, Gail E. Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4):544–558, 2011.