On the Use of Machine Learning in Formal Software Verification TR03-294

Lutz Hamel

Department of Computer Science and Statistics
University of Rhode Island, Kingston, R.I. 02881, USA
`hamel@cs.uri.edu`

Abstract. Starting with the notion that a typical program test set encodes the input/output behavior of a program, we use machine learning techniques to induce a first-order equational theory from the test set that abstractly characterizes this behavior. A program can be considered correct if the induced theory implements the program specification and holds for all test cases in the test set. In terms of test set properties developed in this paper: a program is correct if a test set is adequate and coherent. We show formally that, if a test set is adequate and coherent, then the program can be considered a model of the program specification. Our approach based on machine learning addresses the adequate test data selection problem by providing an effective decision procedure and avoids the oracle problem usually associated with formal testing.

1 Introduction

Formal software development in general and algebraic specification and system development in particular is concerned with building systems that are demonstrably correct with respect to a specification. One way to develop a demonstrably correct system from a specification is by successive refinements [20]. Briefly, a specification S' refines a specification S, write $S \sqsupseteq S'$, if and only if $[S'] \subseteq [S]$ where $[S]$ and $[S']$ denote the model classes of specifications S and S', respectively. Refinements can be composed and the correctness of the overall refinement follows from the transitivity property of the refinement relation. Now, a program P implements a specification S correctly if and only if $S \sqsupseteq P$. Unfortunately, this correctness argument is only valid for programs that have a formal semantics so that $[P]$ is defined. Therefore, this approach to demonstrable correctness is unsatisfactory in most production settings, since the majority of industrial programming languages have partially formalized semantics at best.

Another approach to establish the correctness of a system with respect to a specification is through appropriate testing strategies. This was first proposed by Goodenough and Gehart [11] and further developed in the algebraic setting in [8, 7, 15] among other places. Informally, given a set of test cases T, then a program P is correct with respect to a specification S if and only if

$$(\forall t \in T)\ [pass(P,t)\ \wedge\ S \models t]\ \Rightarrow\ P \equiv S$$

where *pass* is a predicate with the value true if P passes a test t and false otherwise, $S \models t$ denotes the fact that S semantically entails test t, and we also let $P \equiv S$ denote the fact that program P can be considered equivalent to the specification S. In other words, a program is correct if and only if a set of successful test cases imply that the program can be considered equivalent to its specification.

Designing the test set T in such a way that the implication above holds is known as the *adequate test data selection problem* and was first identified by [11]. If the test set T is exhaustive then the above implication holds trivially. However, exhaustive test sets are typically infinite and therefore of limited practical usefulness. In order to limit the size of the test set, most contemporary approaches select test cases based on some kind of *a priori* case analysis of the specification [8]. This has some interesting ramifications, since a particular test case derived from a specification might not be directly testable in a given program. This raises the next issue.

Designing an effective procedure that evaluates the expression "$pass(P,t) \wedge S \models t$" is known as the *oracle problem.* Given the fact, that a particular test case might not be directly testable in a program, computing the predicate *pass* can be particularly tricky. Also, typically a program and its specification are based on very different formalism and therefore it is not always straightforward to show that the above conjunction holds for a particular test case. Solutions to this problem have been proposed by considering observational equivalence and allowing the developer of a specification to supply additional information in the specification [15].

Here we propose an approach of demonstrating the correctness of a system with respect to a specification through testing based on machine learning. Our approach can be summarized as follows: given a test set for a particular program that consists of input and output values for each test case, we can induce a theory that fits these data points using machine learning techniques. The induced theory abstractly characterizes the input/output behavior of the program encoded in the test set. Program correctness follows from demonstrating that the induced theory implements the original specification. This is due to the fact that it can be formally shown that, if the induced theory implements the program specification, then the program can be considered a model of the program specification.

We develop two characteristics a test set has to posses in order to demonstrate program correctness:

– we say that a test set is coherent if and only if we can induce a theory that satisfies each test case in the test set,
– we also say that a test set is adequate if and only if we can show that the induced theory implements the specification.

Given this, a program is correct with respect to a specification, if a test set is coherent and adequate.

It is interesting to note that we do not derive test sets from specifications, rather we construct coherent and adequate test sets around programs. Therefore, we avoid the oracle problem since our test sets are testable by construction and our test results are never directly compared to an evaluation of the test cases in the program specification. We also address the adequate test data selection problem by providing an effective decision procedure based on our notion of adequacy rather than relying on an *a priori* case analysis.

The rest of the paper is structured as follows. Section 2 introduces the algebraic notions necessary to make this paper as self-contained as possible. In Section 3 we outline as much machine learning theory as is necessary for this paper. Our notion of program correctness based on testing in introduced in Section 4. We look at a couple of simple, illustrative examples in Section 5 and we finally conclude with Section 6.

2 Algebraic Specification

Algebraic specification is based on equational logic with algebras as models [16, 21, 4]. Here we define the basic algebraic notions necessary for the development of the current paper.

An equational signature defines a set of sort symbols and a set of operator or function symbols[1].

Definition 1. *An* **equational signature** *is a pair (S, Σ), where S is a set of sorts and Σ is an $(S^* \times S)$-sorted set of operation names. The operator $\sigma \in \Sigma_{w,s}$ is said to have arity $w \in S^*$ and sort $s \in S$. Usually we abbreviate (S, Σ) to Σ.*

We define Σ-algebras as models for these signatures as follows:

Definition 2. *Given a many sorted signature Σ, a* **Σ-algebra** *A consists of the following:*

- *an S-sorted set, usually denoted A, called the* **carrier** *of the algebra,*
- *a* **constant** *$A_\sigma \in A_s$ for each $s \in S$ and $\sigma \in \Sigma_{[],s}$,*
- *an* **operation** *$A_\sigma \colon A_w \to A_s$, for each non-empty list $w = s1 \dots sn \in S^*$, and each $s \in S$ and $\sigma \in \Sigma_{w,s}$, where $A_w = A_{s1} \times \dots \times A_{sn}$.*

Mappings between signatures map sorts to sorts and operator symbols to operator symbols.

Definition 3. *An* **equational signature morphism** *is a pair of mappings $\phi = (f, g) \colon (S, \Sigma) \to (S', \Sigma')$, we write $\phi \colon \Sigma \to \Sigma'$.*

A theory or specification is an equational signature with a collection of equations.

[1] Notation: Let S be a set, then S^* denotes the set of all finite lists of elements from S, including the empty list denoted by $[]$. Given an operation f from S into a set B, $f \colon S \to B$, the operation f^* denotes the extension of f from a single input value to a list of input values, $f^* \colon S^* \to B$, and is defined as follows: $f^*(sw) = f(s)f^*(w)$ and $f^*([]) = []$, where $s \in S$ and $w \in S^*$.

Definition 4. *A* **Σ-theory** *or* **specification** *is a pair (Σ, E) where Σ is an equational signature and E is a set of* **Σ-equations**. *Each equation in E has the form*

$$(\forall X)\ l = r,$$

where X is a set of variables distinct from the equational signature Σ and $l, r \in T_\Sigma(X)$ are terms over the set Σ and X. If $X = \emptyset$, that is, l and r contain no variables, then we say the equation is **ground**. *When there is no confusion Σ-theories are referred to as theories and are denoted by their collection of equations, in this case E.*

The above can easily be extended to conditional equations[2]. However, without loss of generality we continue the discussion here based on unconditional equations only.

The models of a theory are the Σ-algebras that satisfy the equations. Intuitively, an algebra satisfies an equation if and only if the left and right sides of the equation are equal under all assignments of the variables. More formally:

Definition 5. *A Σ-algebra A* **satisfies** *a Σ-equation $(\forall X)\ l = r$ iff $\overline{\theta}(l) = \overline{\theta}(r)$ for all assignments $\overline{\theta}\colon T_\Sigma(X) \to A$. We write $A \models e$ to indicate that A satisfies the equation e.*

We define satisfaction for theories as follows:

Definition 6. *Given a theory $T = (\Sigma, E)$, a Σ-algebra A is a T-model if A satisfies each equation $e \in E$. We write $A \models T$ or $A \models E$.*

In general, there are many algebras that satisfy a particular theory. We also say that the class of algebras that satisfy a particular equational theory represent the denotational semantics of that theory.

Semantic entailment of an equation by a theory is defined as follows.

Definition 7. *An equation e is* **semantically entailed** *by a theory (Σ, E), write $E \models e$, iff $A \models E$ implies $A \models e$ for all Σ-algebras A.*

Mappings between theories are defined as theory morphisms.

Definition 8. *Given two theories $T = (\Sigma, E)$ and $T' = (\Sigma', E')$, then a* **theory morphism** *$\phi\colon T \to T'$ is a signature morphism $\phi\colon \Sigma \to \Sigma'$ such that $E' \models \phi(e)$, for all $e \in E$.*

In other words, the signature morphism ϕ is a theory morphism if the translated equations of the source theory T are semantically entailed by the target theory T'.

Burstall and Goguen have shown within the framework of institutions [4] that the following holds for many sorted algebra[3]:

[2] Consider the conditional equation, $(\forall X)\ l = r$ if c, which is interpreted as meaning the equality holds if the condition c is true.

[3] Actually, Burstall and Goguen have shown the much more powerful result that the implication holds as an equivalence relation. However, for our purposes here we only need the implication.

Theorem 1. *Given the theories $T = (\Sigma, E)$ and $T' = (\Sigma', E')$, the theory morphism $\phi\colon T \to T'$, and the T'-algebra A', then $A' \models_{\Sigma'} \phi(e) \Rightarrow \phi A' \models_{\Sigma} e$, for all $e \in E$.*

In other words, if we can show that a given model of the target theory satisfies the translated equations of the source theory, it follows that the *reduct* of this model, $\phi A'$, also satisfies the source theory.

3 Machine Learning and Theory Induction

Among other things, the discipline of machine learning concerns itself with the induction of first-order theories from positive and negative examples [18]. In the case of first-order logic this has been extensively studied under the name of inductive logic programming [6, 19]. Recently, systems that induce first-order equational theories from ground equations have been introduced [14, 13].

Theory induction can be thought of as a search over all admissible theories for an *optimal theory* that satisfies a given set of examples [17]. The notion of an optimal theory is usually defined to be the simplest theory that satisfies the examples. Theories that satisfy the given examples are usually referred to as hypotheses.

It is possible to give a more abstract characterization to the induction of first-order theories as follows: given a theory P of positive examples, and a theory N of negative examples, then an induced theory H can be considered an *hypothesis* if

$$H \models p,\ \text{for every } p \in P \text{ (Completeness)},$$
$$H \not\models n,\ \text{for every } n \in N \text{ (Consistency)}.$$

In our case H is an equational theory and P and N are equational ground theories, i.e., every equation in P and N is a ground equation[4]. Completeness states that an hypothesis satisfies the positive examples. Consistency states that an hypothesis does not satisfy the negative or counter examples. Typically we refer to the positive and negative examples as training examples. Please note that this abstract characterization does not address optimality issues outlined above.

Central to machine learning is the *inductive learning hypothesis* [18] stated here in terms of first-order theories:

> *If an induced theory satisfies the completeness and consistency criteria for a sufficiently large set of training examples, then the induced theory will satisfy unseen examples appropriately.*

The key notion here is that machine learning allows us to generalize beyond a finite set of training examples. By sufficiently large we mean that the training examples are chosen in such a fashion that they represent the domain to be learned appropriately.

[4] A more algebraic formulation of this characterization can be found in [12].

4 Program Correctness and Testing

Our notion of program correctness views programs as Σ-algebras that satisfy an equational specification or theory. Formally,

Definition 9. *Given a program P and a specification S, we say that P is* **correct** *with respect to the specification S iff P satisfies the specification S, we write $P \models_{\Sigma} S$. If it is clear from the context that we view program P as a Σ-algebra we simply write $P \models S$.*

In the following we show that given a specification and a program it is possible to establish that the program is correct with respect to the specification by appropriate testing. We define a set of criteria a set of tests has to fulfill in order to establish the correctness of a program.

We define a test set to be a set of input/output pairs with respect to a program P. Formally,

Definition 10. *Let program P be a Σ-algebra with signature Σ_P, then we define a* **test set** *to be the set T of pairs*

$$(\sigma(v), t_{\sigma(v)}),$$

where v denotes an input value and $\sigma(v)$ denotes the application of the input value to the program P with $\sigma, v \in \Sigma_P$; $t_{\sigma(v)} \in T_{\Sigma_P}$ denotes the result of evaluating the input in the program P. Each pair in the test set is referred to as a **test case**.

It is straight forward to see that we can interpret each test case as an equation[5]. Therefore, we can make the following assertion: a specification satisfies a test set if and only if each test case is satisfied by the specification. Formally,

Definition 11. *A specification S* **satisfies** *a test set T iff $S \models t$ for each $t \in T$, we write $S \models T$.*

Now, we say that a test set is coherent if and only if there exists at least one specification that satisfies the test set. We can state this property in machine learning terms formally as,

Definition 12. *A test set T is* **coherent** *iff we can induce a theory H such that $H \models T$.*

Coherency expresses the fact that we can characterize the input/output behavior of a program encoded in the test set with an hypothesis, that is, an induced theory that satisfies all the test cases. It is interesting to note that our current notion of test set only embodies positive examples, the set of negative examples is empty.

The second property for test sets is defined as follows: a test set is adequate if and only if an induced theory implements the original program specification. Formally,

[5] This can be made precise by considering assignments $\phi\colon T_{\Sigma_P} \to P$ where $\phi(\sigma(v)) = \phi(t_{\sigma(v)}) = P_\sigma(P_v)$.

Definition 13. *We say that a test set T is* **adequate** *iff there exists a theory morphism $\phi\colon\ S \to H$, where S is the program specification and H is a theory induced from the test set T.*

This allows us to propose program correctness,

Proposition 1. *Given a program specification S, a program P, and a test set T; if the test set T is coherent and adequate, then P is correct with respect to the specification S.*

Proof. Let us assume that H is an induced theory such that $H \models T$, i.e., T is coherent. Then by the inductive learning hypothesis we have $P \models H$. Let us also assume that there exists a theory morphism $\phi\colon\ S \to H$, i.e., T is adequate. It follows from ϕ and $P \models H$ that $\phi P \models S$.

The approach taken and the notions defined here were inspired by Weyuker's work on program inference and data adequacy [22]. However, there are some fundamental differences. Weyuker considers equivalence relations between the original specification, the induced program and the program to be tested based on particular test values approximating an ideal test set. Instead, our approach relies on the inductive learning hypothesis and the notion of implementation to show that a program satisfies a specification. Also, since Weyuker's work was published, machine learning has advanced tremendously. Today's machine learning algorithms tend to be fairly robust and will induce sensible theories even in the presence of errors. Therefore, we introduced the notion of coherency to reinforce that for program correctness an induced theory has to satisfy the complete test set.

5 Simple Examples

For exposition purposes we will consider some simple examples here. It is interesting to note that non-trivial software specifications have been induced from programs [5, 1] albeit with different formalisms than the induction of equational theories considered here. Neither of these approaches, however, go beyond the induction of theories to consider formal notions of program verification.

Now, consider the following specification of a stack module. The specification is given in OBJ syntax [10] where the `op` keyword introduces new operations, the keyword `var` defines variables, and the keyword `eq` allows one to define operations with equations:

```
obj STACK is sorts Stack Element .
  op push  : Stack Element -> Stack .
  op top   : Stack -> Element
  var X : Element .  var S : Stack .
  eq top(push(S,X)) = X .
endo
```

This specification states that the top of a stack is the last element pushed.

Example 1. Consider the following test set

$$T1 = \{\ (\texttt{top(push(v,a))}, \texttt{a}),\\ (\texttt{top(push(push(v,b),a))}, \texttt{a})\ \}$$

and the induced theory

```
obj STACKI is sorts Stack Element .
  ops a b : -> Element .
  op v : -> Stack .
  op top : Stack -> Element .
  var X : Stack .
  eq top(X) = a .
endo
```

It is easy to see that the test set is coherent with respect to the induced theory. It is also straight forward to see that the induced theory does not implement the original stack specification above. Therefore, $T1$ is a coherent but not adequate test set for the stack module.

It could be argued that the test set $T1$ violates the inductive learning hypothesis due to the fact that it can be considered “insufficiently large”. However, it serves to illustrate the point.

Example 2. Consider the following test set

$$T2 = \{\ (\texttt{top(push(v,a))}, \texttt{a}),\\ (\texttt{top(push(push(v,a),b))}, \texttt{b}),\\ (\texttt{top(push(push(v,b),a))}, \texttt{a}),\\ (\texttt{top(push(push(v,d),c))}, \texttt{c})\ \}.$$

Given this test set, the system in [13] is able to induce a theory like the following:

```
obj STACKI is sorts Stack Element .
  ops a b c d : -> Element .
  op v : -> Stack .
  op push  : Stack Element -> Stack .
  op top   : Stack -> Element
  var X : Element .  var S : Stack .
  eq top(push(S,X)) = X .
endo
```

It is easy to see that the test set $T2$ is coherent, since the induced theory satisfies each test case. It is also straight forward to see that the induced theory implements the original specification by considering a theory inclusion from the specification to the induced theory. Therefore, the test set is adequate.

6 Conclusions

Starting with the notion that a typical program test set encodes the input/output behavior of a program, we use machine learning techniques to induce a first-order equational theory that abstractly characterizes this behavior. A program can be considered correct if the induced theory implements the program specification and holds for all test cases in the test set. In terms of test set properties elaborated in this paper: a program is correct if a test set is adequate and coherent.

Our approach based on machine learning addresses two major drawbacks in other formal testing approaches: we avoid the oracle problem altogether and we mitigate the test data selection problem by building test sets around programs using our adequacy and coherency criteria as a guide rather than deriving test cases from specifications through some kind of *a priori* case analysis.

A powerful extension to testing is the notion of negative testing, that is, tests which are designed to fail. This kind of test strategy can be easily handled in our framework by considering negative tests to be negative examples to the machine learning algorithm, i.e., examples that an induced theory should not entail.

Here we have only considered algebraic specifications in the sense of abstract data types. An extension to these notions are behavioral equational specifications [9, 2, 3]. How these specification mechanisms can be used in the context of this framework together with the notion of inducing behavioral theories from examples is an open research question at this point.

References

1. G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. of Principles of Programming Languages (POPL02)*, pages 4–16. ACM, January 2002.
2. G. Bernot, M. Bidoit, and T. Knapik. Observational specifications and the indistinguishability assumption. *Theoretical Computer Science*, 139(1-2):275–314, 1995. Submitted in 1992.
3. M. Bidoit and R. Hennicker. Behavioral theories and the proof of behavioral properties. *Theoretical Computer Science*, 165(1):3–55, 1996.
4. R. Burstall and J. Goguen. Institutions: abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
5. W. W. Cohen. Recovering software specifications with inductive logic programming. In *AAAI'94, Proc. of the 12th Natl. Conf. on AI*, pages 142–148, 1994.
6. P. A. Flach. The logic of learning: a brief introduction to inductive logic programming. In *Proceedings of the CompulogNet Area Meeting on Computational Logic and Machine Learning*, pages 1–17, 1998. http://citeseer.nj.nec.com/flach98logic.html.
7. P. Le Gall and A. Arnould. Formal specifications and test: Correctness and oracle. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specification*, volume 1130 of *Lecture Notes in Computer Science*, pages 342–358. Springer-Verlag, 1995. http://citeseer.nj.nec.com/39321.html.
8. M. Gaudel. Testing can be formal too. In *TAPSOFT: 6th International Joint Conference on Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, Berlin, Germany, 1995.
9. J. Goguen and G. Malcolm. Hidden coinduction: Behavioral correctness proofs for objects. *Mathematical Structures in Computer Science*, 9(3):287–319, 1999.

10. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: algebraic specification in action*, pages 3–167. Kluwer, 2000.
11. J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, 1975.
12. L. Hamel. An algebraic view of inductive equational logic programming. Technical Report TR00-278, Dept. of Computer Science and Statistics, University of Rhode Island, December 2000.
13. L. Hamel. Breeding algebraic structures – an evolutionary approach to inductive equational logic programming. submitted for publication, 2002.
14. J. Hernández-Orallo and M. J. Ramírez-Quintana. A strong complete schema for inductive functional logic programming. In S. Džeroski and P. Flach, editors, *Proceedings of the 9th International Workshop on Inductive Logic Programming*, volume 1634, pages 116–127. Springer-Verlag, 1999.
15. P. D. L. Machado. *Testing from Structured Algebraic Specifications: The Oracle Problem.* PhD thesis, University of Edinburgh, 2000. http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-423/index.html.
16. J. Meseguer and J. Goguen. Initiality, induction and computability. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985.
17. T. M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, 1982.
18. T. M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.
19. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.
20. D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. In G. Levi, H. Ehrig, R. Kowalski, and U. Mantanari, editors, *Proc. of the International Conference on Theory and Practice of Software Development (TAPSOFT '87)*, volume 249 of *Lecture Notes in Computer Science*, pages 96–110. Springer-Verlag, Berlin, Germany, 1987.
21. W. Wechler. *Universal Algebra for Computer Scientists.* Springer-Verlag, 1992. EATCS Monographs on Theoretical Computer Science, Volume 25.
22. Elaine J. Weyuker. Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems*, 5(4):641–655, 1983.