Using Invariant Detection Mechanism in Black Box Inference

Muzammil Shahbaz* and Roland Groz**

*France Telecom R&D
Meylan, France
Muhammad.MuzammilShahbaz@orange-ftgroup.com
**LIG, Computer Science Lab
Grenoble Universités, France
Roland.Groz@imag.fr

Abstract. The testing and formal verification of black box software components is a challenging domain. The problem is even harder when specifications of these components are not available. An approach to cope with this problem is to combine testing with learning techniques, such that the learned models of the components can be used to explore unknown implementation and thus facilitate testing efforts. In recent years, we have contributed to this approach by proposing techniques for learning parameterized state machine models and then use them in the integration testing of black box components. The major problem in this technique left unaddressed was the selection of parameter values during the learning process. In this paper, we propose to use an invariant detection mechanism to select values in the learning process, thus refining model inference and testing approach. Initial experiments with small examples yielded positive results.

1 Introduction

Although formal methods are a key element to automate a number of phases of software development, very few software processes are actually fully based on formal methods and associated tools. Typical problems arise when specifications are not formal or only a part of the specifications is formalized. It is also often the case that formal models are not updated consistently with software evolution.

One approach to alleviate this problem is to revert the usual rigid process that starts from formal specifications. This flexible approach consists in introducing or re-introducing formal models during the software process, typically by retrieving them from various sources, including the code itself. The advantage is that formal methods and tools can be applied on development steps where they can be justified as more efficient and therefore more acceptable for integration in the development process of the company. This approach is commended especially in a testing phase which is a cost-intensive and a time consuming activity and thus automation is highly desirable. Formal methods can support testing through test generation, test interpretation, classification and diagnosis.

Black box testing is an important area where the application of formal methods is still a novice. One view of the application on black box testing is to apply formal reverse engineering methods that combine testing and learning algorithms. The key advantage of this approach is to obtain an in depth knowledge of the internal design of a black box system which is hidden from the user and also to rigorously test the system. The inference is done based upon the observations of the system which can be obtained by stimulation. There are few works, e.g., Hungar et al. (2003); Elkind et al. (2006); Berg et al. (2006) that studied the practicality of the algorithms in the domain of machine learning, esp. automata inference. In recent years, we have also contributed in this domain by introducing methods for learning parameterized finite state machine models and their use in the integration testing of black box systems. The motivation behind introducing parameters is to encapsulate very large (or infinite) input set into few key inputs and run the algorithm on the reduced set. Some efforts have been done in this respect and a parameterized model which can be learnt in polynomial time is presented Berg et al. (2006). However, our model is more expressive in the sense that parameter values can be associated with inputs and outputs of the state transitions. We have developed a learning algorithm Shahbaz et al. (2007) to learn such a model from a black box component. The algorithm tests the component with different parameter values and conjectures a parameterized model through observations of testing results. The procedure is iterative and requires several iterations in which different parameter values are tested. The selection of parameter values is intensively difficult with the motivation of exploring maximum behaviors of the component in minimum iterations. In the algorithm Shahbaz et al. (2007), this selection is intuitive or guided by domain experts.

In this paper, we propose to use an invariant detection mechanism for the selection of parameter values in the learning algorithm. Invariants are basically properties of a program in a white box scenario. They provide a relationship between program variables by observing their values and their use in the program. However, we have noticed that invariant detection methods can be applied to a black box system if there is enough observations collected from the system. In our case, we have observations in terms of input and output parameter values during an initial run of the learning algorithm. The likely invariants over those observations can be inferred thus giving a meaningful relation between input and output parameters. This relation or invariant can be used in selecting new parameter values for the next iterations of the learning algorithm, thus making the learning process iterative and a more realistic model of the black box system can be conjectured.

We organize the paper in the following way. Section 2 describes the learning methodology in general and a brief sketch of the algorithm. Section 4 introduces the dynamic invariant detection mechanism and the tool that we are using in our method. Section 5 illustrates the use of the invariant detection method in our approach with the help of an example. Section 7 concludes the paper.

2 The Learning Methodology

Our approach to learn a black box component and to build its formal model is through active learning techniques. In this technique, a component is tested with different combinations of inputs and a model is learnt incrementally with the help of observations of testing results. We assume that a basic input set through which the component is exercised is known and the interfaces are accessible through which inputs can be sent and their corresponding outputs from the component can be observed. The components in the system communicate with each other by means of message passing and exchange enormous amounts of data in the form of input and output parameters. Testing and learning of such components is hard because regardless of the finite behavioral spectrum of the component, the i/o parameter domain is infinite. It is not known that which values will suddenly change the behavior of the component altogether leading the complete system to an unexpected state. Realistically, we can only learn partial models of a component. Therefore, we exercise some parameter values in in our learning algorithm and conjecture the component behaviors in the form of a state machine model.

The basic idea of the learning process is to systematically explore the component's behaviors by means of testing different input sequences. The input symbols in the sequences are associated with some parameter values from their respective domains. When it is observed that "sufficiently consistent" behaviors have been explored, the algorithm stops testing and conjectures a state machine model[1] consistent with the observed behaviors, i.e., each transition from each state of the model ends in a well-defined state and producing the same input/output relation as observed in the testing.

In this paper, we focus on the solution of selecting parameter values during the learning procedure. We refer to our previous work Shahbaz et al. (2007) for the complete description of the learning algorithm in order to avoid lengthy formal discussion in this paper.

3 A Simple Example

Consider a part of a global avionics system design that is related to controlling of ailerons of a plane. The subsystem is designed to tolerate fault in the working components by means of backup redundancy. There are two components $C1$ and $C2$ that are responsible for controlling the ailerons and another component $Ctrl$ monitors and controls the working of $C1$ and $C2$. In a normal mode, $C1$ is active and sends a periodic signal $s(1)$ to $Ctrl$ where parameter value 1 indicating the activeness of $C1$. $Ctrl$ replies with ack for every signal it receives. Whenever $Ctrl$ receives $s(0)$ meaning $C2$ is down, it sends a request $sc(1)$ to set a counter of failure. The parameter value 1 indicates that failure has occurred. $Ctrl$ periodically checks the counter $c(x)$, where x is the counter value, and increments the value every time. If the counter reaches to a certain threshold then $Ctrl$ sends a command Cmd to $C2$ to take charge of the ailerons.

The complete example of the aileron control system with details of other components is given in Ermont and Boniol (2002). However, we have modified the original example by replacing the timing notions used in the components simply to illustrate our approach. Here we shall just focus on the component $Ctrl$ and assume that its implementation is unknown. A preliminary (partial) parameterized model is learnt, shown in Figure 1, after the first iteration of our learning algorithm (roughly sketched in section 2). The input set used for $Ctrl$ in the algorithm is $\{s, sc, c\}$, where the input parameter domain for inputs s and sc is boolean, and for input c is the set of positive integers. Note that the model is not learnt completely and it lacks the information how $Ctrl$ sends Cmd to $C2$ for activation. The counter c is tested with parameter values between 1 and 4 in this iteration.

[1]A parameterized state machine model can be seen as a Mealy machine where inputs and outputs are associated with parameters. The details and formal definitions can be seen in Shahbaz et al. (2007).

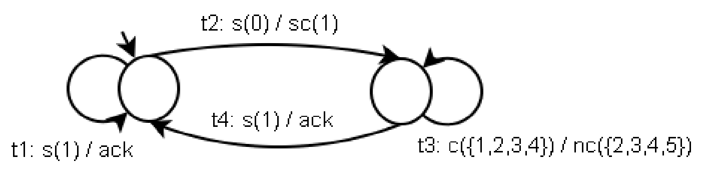

FIG. 1 – *Preliminary model of $Ctrl$ after the first iteration. The interpretation of labels on transition, e.g., $t2 : s(0)/sc(1)$ is as follows: $t2$ is the transition name, $s(0)$ is the input, where 0 is the input parameter value and $sc(1)$ is the output, where 1 is the output parameter value*

4 Dynamic Invariant Detection

A program invariant is a property that holds at certain points in a program. For example, variable x is non-zero ($x \neq 0$), being constant (e.g., $x = 1$), being in a range (e.g., $5 \leq x \leq 10$), linear relationships (e.g., $y = 2x + 1$), ordering (e.g., $x \leq y$), etc. Dynamic detection of likely invariants is a program analysis that infers invariants over variables in the scope by observing their values during the program execution. A dynamic detector of program invariants runs the program on a specific test suite and examines variable values captured during program execution and reports properties and relationships that hold over those values. The basic use of invariants is in program comprehension, in general, and also for the purpose of documentation and maintenance tasks.

The Daikon System Ernst et al. (2006) is an implementation of the dynamic invariant detection that infers invariants over scalar and structured variables from program execution. The essential idea is to run the program over a test suite, collect traces from the program execution and use a generate-and-check algorithm to test a set of potential invariants against the traces. The algorithm initially assumes that all potential invariants over the variables of interest are true and incrementally tests each invariant against the observed values of these variables from the traces. At each step, the invariant is discarded if it is violated by the values to obtain a set of positive invariants. The remaining invariants at the end of the process are reported describing the relations as invariants on the set of values observed in the program behavior.

Daikon is enhanced with a number of optimizations that allows it to scale to both large numbers of invariants and programs of non-trivial size. Currently, it checks over 70 invariants and the list is extendible by the users to accommodate their own domain-specific invariants and derived variables. Some of the invariants it detects that are useful typically for numeric applications are as follows: constant value ($x = a$), small value set ($x \in \{a, b, c\}$), range limits ($x \leq a$), non-zero ($x \neq 0$), modulus ($x \equiv a \pmod b$), linear relationships ($y = ax + b$), functional relationships ($y = f(x)$), comparisons ($x > y, x = y, ...$), polynomial relationships ($z = ax + by + c$) and relationships over all elements of an array.

5 Use of Daikon in Parameter Value Selection

Daikon collects all program executions in a large file called data trace file '.dtrace'. It does not use source code at all when inferring invariants. The inference engine reads data trace files and runs invariant detection algorithm on the variable values collected in the files. This functionality exactly matches our requirements. We do not have access to the source code but have observations from the black box system after its preliminary state machine model is learnt through our learning algorithm. These observations include a (large) set of input and output parameter values generated through testing during the learning phase. We want to pursue for the next iterations of the algorithm in order to refine our models and also to test the system behavior on the other set of parameter values which is never used in the previous iteration. The question is how to select “intelligently” new values and when the iterative process should terminate.

The idea to solve the above stated problems is to use invariants over observed data in the previous iteration to select the new parameter values for the next iterations. The observations in the previous iteration can be written in a data trace file to feed Daikon to detect invariants over those observations. For example, if the values for the input parameter x are given as $\{2, 4, 8\}$ and the values for the corresponding output parameter y are observed as $\{4, 16, 64\}$, where x and y are integers, then Daikon will infer the relationship between x and y as $y = x^2$.

This invariant is meaningful in order to steer the next iteration of the algorithm. The purpose is to make sure that the system will behave according to the inferred invariant for any value of x. To fulfill this purpose, one can select larger or smaller values of x in the next iteration than that are used in the previous iterations. The new observed behavior of the next iteration will be given again to Daikon to infer invariants on the new data. The difference in the invariants of the current and the previous iterations will lead to further iteration of the learning algorithm in which parameter values will be selected according to the new inferred invariants. Otherwise, the procedure will terminate and the transitions of the state model will be accumulated with the respective invariants. Figure 2 is the summary of the whole learning procedure.

6 Experiment

The parameter values labelled on the transitions of the learned model of $Ctrl$, given in Figure 1, will be provided to Daikon in the form of data trace files to infer invariants over those values. A data trace file consists of execution records that list the variable values encountered during the executions. We illustrate a data trace file for the transition $t3$ of the model where the input parameter c is tested with values $\{1, 2, 3, 4\}$ and the corresponding output parameter values are observed as $\{2, 3, 4, 5\}$ respectively. One input parameter value with its corresponding output parameter value is written as one execution record in the file. Figure 3 explains an execution record excerpted from the file.

The lines with '#' are comments. Daikon notes variable values before each procedure entry and after exit for an execution record. We treat this phenomena in our case as input parameter values before testing (vs before procedure entry) and output parameter values observed after testing (vs after procedure exit). We name the procedure as `Controller.aileron()`. Lines 3 to 8 in Figure 3 declare procedure entry and set the value for input parameter c, renamed as $counter$, as 2 (lines 6 and 7). Lines 10 to 18 declare procedure exit and set the value for output parameter nc, renamed as $new\_counter$, as 3 (lines 16 and 17). We run Daikon on the data trace file and obtain the output shown in Figure 4.

The inferred invariants are written under `Controller.aileron():::EXIT` and the most important invariant we learned is: `counter + 1 = new_counter`. This reveals that $Ctrl$ increments by one each time in the $counter$ values which are given from 1 to 4.

```
Let $Inv, NewInv$ be the sets where every $inv_i \in Inv, ninv_i \in NewInv$ is a set of
invariants for a transition $t_i, 1 \leq i \leq n$ of a conjectured model $M$, where $n$ is the
number of transitions in $M$ ;
begin
    Initialize $Inv = NewInv = \emptyset$ ;
    Learn $M$ using the learning algorithm Shahbaz et al. (2007)
    and select the parameter values intuitively ;
    for each transition $t_i$ of $M$ do
        Write dtrace file for i/o parameters observed over $t_i$ ;
        Learn invariants $inv_i$ for $t_i$ using Daikon ;
        Include $inv_i$ in $Inv$ ;
    end
    while $Inv \neq NewInv$ do
        if it is not the first iteration then
            $Inv = NewInv$;
            $NewInv = \emptyset$ ;
        end
        Learn $M$ again using the learning algorithm Shahbaz et al. (2007)
        and select the parameter values using invariants from $Inv$ ;
        for each transition $t_i$ of $M$ do
            Write dtrace file for i/o parameters observed over $t_i$ ;
            Learn new invariants $ninv_i$ for $t_i$ using Daikon ;
            Include $ninv_i$ in $NewInv$ ;
        end
    end
    Output $M$ ;
end
```

FIG. 2 – *Summary of the learning procedure*

```
# Execution record no. 110
# Before execution: counter = 2
Controller.aileron():::ENTER
this_invocation_nonce
110
counter
2
1
# After execution: new_counter = 3
Controller.aileron():::EXIT100
this_invocation_nonce
110
counter
2
0
new_counter
3
0
```

FIG. 3 – *An execution record of the data trace file*

```
Daikon version 4.3.1, released August 2, 2007;
http://pag.csail.mit.edu/daikon.
Processing trace data; reading 1 dtrace file:
[15:14:39]: Finished reading aileron-counter.dtrace
===================================================
Controller.aileron():::ENTER
===================================================
Controller.aileron():::EXIT
counter one of { 1, 2, 3, 4 }
new_counter one of { 2, 3, 4, 5 }
counter == orig(counter)
counter - new_counter + 1 == 0
Exiting Daikon.
```

FIG. 4 – *Daikon output for the data trace file of the transition* $t3$

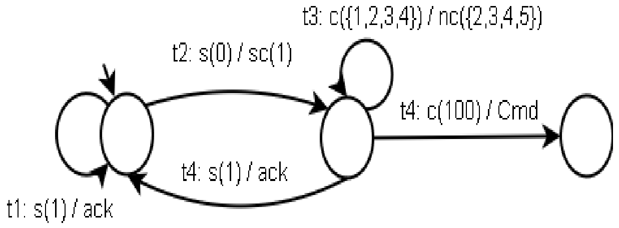

FIG. 5 – *Refined model of* $Ctrl$ *after the second iteration*

In order to start the second iteration of the learning algorithm, we use this invariant to select the values for $counter$ that may change the behavior of $Ctrl$, since our preliminary model is partial. Therefore, we select larger values than the ones given in the counter value set $\{1, 2, 3, 4\}$. We start the next iteration and test $counter$ with value 100, i.e., $c(100)$. This value is larger than the threshold limit of $Ctrl$ upon which it sends Cmd to $C2$. Hence, we find a different behavior of the component from the previous iteration. The refined model is shown in Figure 5 conjectured from the second iteration of the learning algorithm. The new observations will again be fed to Daikon for the update of invariants. The recalculated invariants on the transition $t3$ are not different from the invariants of the previous iteration. Hence, the learning process terminates.

7 Conclusion

We have extended our previous work of black box inference in which we have presented a learning algorithm that tests and learns the component incrementally and conjectures a parameterized state machine model. The selection of parameter values in the algorithm is a difficult task in order to observe maximum behaviors of the component in minimum iterations.

In this paper, we have presented an approach to use a dynamic invariant detection mechanism to select the parameter values in the algorithm. A preliminary model of a component is learnt after the first iteration of the algorithm in which parameter values are selected intuitively. The observations in the first iteration are then used to infer invariants over the parameter values. These invariants are helpful in selecting new parameter values for the subsequent iterations. We use Daikon as an invariant detector and performed a simple experiment with it.

We have performed some additional experiments and found Daikon an efficient tool for numeric applications. However, it has limitations for use in large sized applications where complex formulas are used for computations. Currently, Daikon cannot infer invariants on linear relationship on more than three variables. Moreover, it generates very few invariants that are of any interest unless it is supplied with very large execution data. On the contrary, we believe that even naive invariants of an unknown implementation are meaningful in our case. A thorough evaluation of Daikon and its comparison with other approaches is done in Ernst et al. (2001).

In future work, we intend to apply more formal approaches using invariants to derive efficient test cases. Daikon is being used actively in testing and verification work. But very few works Lorenzoli et al. (2006); Mariani and Pezzè (2005) used this tool in a black box testing framework. Therefore, we continue exploring methods to use Daikon in this domain.

References

Berg, T., B. Jonsson, and H. Raffelt (2006). Regular inference for state machines with parameters. In *FASE*, Volume 3922 of *Lecture Notes in Computer Science*, pp. 107–121. Springer.

Elkind, E., B. Genest, D. Peled, and H. Qu (2006). Grey-box checking. In *FORTE*, pp. 420–435.

Ermont, J. and F. Boniol (2002). Tpap: an algebra of preemptive processes for verifying real-time systems with shared resources. *Electr. Notes Theor. Comput. Sci. 65*(6).

Ernst, M. D., J. Cockrell, W. G. Griswold, and D. Notkin (2001). Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering 27*(2), 99–123.

Ernst, M. D., J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao (2006). The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*.

Hungar, H., O. Niese, and B. Steffen (2003). Domain-specific optimization in automata learning. In *CAV*, Volume 2725 of *Lecture Notes in Computer Science*, pp. 315–327. Springer.

Lorenzoli, D., L. Mariani, and M. Pezzè (2006). Inferring state-based behavior models. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pp. 25–32. ACM Press.

Mariani, L. and M. Pezzè (2005). Behavior capture and test: Automated analysis of component integration. *ICECCS 0*, 292–301.

Shahbaz, M., K. Li, and R. Groz (2007). Learning and integration of parameterized components through testing. In *TestCom/FATES*, pp. 319–334.