Learning Parameterized State Machine Model for Integration Testing

Muzammil Shahbaz
France Telecom R&D
Meylan, France
muhammad.muzammilshahbaz
@orange-ftgroup.com

Keqin Li
LIG, Computer Science Lab
Grenoble Universités, France
Keqin.Li@imag.fr

Roland Groz
LIG, Computer Science Lab
Grenoble Universités, France
Roland.Groz@imag.fr

Abstract

Although many of the software engineering activities can now be model-supported, the model is often missing in software development. We are interested in retrieving state-machine models from black-box software components. We assume that the details of the development process of such components (third-party software or COTS) are not available. To adequately support software engineering activities, we need to learn more complex models than simple automata.

Our model is an extension of finite state machines that incorporates the notions of predicates and parameters on transitions. We argue that such a model can offer a suitable trade-off between expressivity of the model and complexity of model learning. We have been able to extend polynomial learning algorithms to extract such models in an incremental testing approach. In turn, the models can be used to derive tests or for component documentation.

1. Introduction

Many of the software engineering activities such as code analysis, test case generation, component management and documentation etc, can be supported with formal models for their effective implications. Typically in an industry, due to limited exchange of information among component providers and component users [4], engineers find difficulty in providing a required system integration if they have limited knowledge of the behaviors of the components, which they use in the system. Normally, these components that are obtained from external sources, also known as COTS (commercial-off-the-shelf), are not provided with formal models or with precise documentation. In this situation, engineers test each component to get a rough idea of its intended behaviors on typical requests it would have to serve in the assembly, and then test the integrated system of these components based on variations of use cases. Thus absence of formal models is a daunting prospect for providing an effective and quality integrated solution.

Our approach to address this problem is to learn the models directly from the components and then steer the testing effort using these models. This approach provides us with a key towards automata learning to support integration and testing of components that can be regarded as black boxes whose internal structure is unknown. We deal with the domain of complex telecommunication applications. A typical framework is the process associated with designing a service as an assembly of communicating distributed components. Integration entails experimenting with various use cases to elicit component interactions, and in particular potential interoperability problems in data values passed around. Therefore we need to advance from simple state-machine inference to the inference of more expressive models that can maintain the fine granularity of complex systems, i.e., parametric details and also some notion of non-determinism. Thus, our current work proposes a parameterized model as an extension of simple finite state machine that can be learned in polynomial time with modifications in existing learning algorithms in this domain.

In section 2, we present our model and compare it to the state of the art in the learning methodologies. We show in section 3 how we combine testing and model derivation in an iterative approach. The algorithm for model inference is briefly presented and illustrated on a small example.

2. Parameterized Model

In this section we give a formal description of our parameterized model and then a brief comparison with the existing work.

2.1. Definition

A *Parameterized Finite State Machine (PFSM)* M is a tuple $M = (Q, I, O, D_I, D_O, T, q_0)$, where Q is a finite set of states, I is a finite set of inputs, O is a finite set of outputs, D_I is a set of input parameter values, D_O is a set of output parameter values, $q_0 \in Q$ is an initial state, and T is a set of transitions.

A transition $t \in T$ is described as: $t = (q, q', i, o, p, f)$, where $q \in Q$ is a source state, $q' \in Q$ is a target state, $i \in I$ is an input, $o \in O$ is an output, $p \subseteq D_I$ is a predicate on input parameter values and $f : p \longrightarrow D_O$ is an output parameter function. We consider that the model is restricted with the following three properties.

Property 1 (Input Enabled) *The model is input enabled, i.e., $\forall q \in Q$, $\forall i \in I$ and $\forall x \in D_I$, $\exists t \in T$ such that $t = (q, q', i, o, p, f)$, in which $x \in p$.*

The machine can be made input enabled by adding loop back transitions on a state for all those inputs which are not acceptable for that state. Such transitions contain a special symbol $\Omega \in O$ as output. Similarly, there exists transitions which do not take input parameter values into account. Such transitions contain a special symbol $\perp \in D_I$ in the place of parameter value to expresses the absence of parameter value. For the sake of simplicity, we do not write this symbol while modelling with PFSM. For example, in Figure 1, the transition from state 1 to state 2 has an input ON without any parameter value. This is simply represented by ON instead of $ON(\{\perp\})$.

Property 2 (Input Deterministic) *The model is input deterministic, i.e., for $t_1, t_2 \in T$ such that $t_1 = (q_1, q'_1, i_1, o_1, p_1, f_1)$, $t_2 = (q_2, q'_2, i_2, o_2, p_2, f_2)$ and $t_1 \neq t_2$, if $q_1 = q_2 \wedge i_1 = i_2$ then $p_1 \cap p_2 = \phi$.*

Property 3 (Observable) *The model is observable, i.e., for $t_1, t_2 \in T$ such that $t_1 = (q_1, q'_1, i_1, o_1, p_1, f_1)$, $t_2 = (q_2, q'_2, i_2, o_2, p_2, f_2)$ and $t_1 \neq t_2$, if $q_1 = q_2 \wedge i_1 = i_2$ then $o_1 \neq o_2$.*

For a PFSM with properties 1 and 2, we can define the following functions: $\delta : Q \times I \times D_I \longrightarrow Q$ is target state function, and $\lambda : Q \times I \times D_I \longrightarrow O$ is output function.

When M is in state $q \in Q$ and receives an input $i \in I$ along with the parameter value $x \in D_I$, the target state q' and the output o are determined by the functions δ and λ respectively.

For an input sequence $\gamma = i_1 \cdot i_2 \cdot ... \cdot i_k$ and an input parameter value sequence $\alpha = x_1 \cdot x_2 \cdot ... \cdot x_k$, where $i_j \in I, x_j \in D_I (1 \leq j \leq k)$, we define the association of γ and α as $\gamma \otimes \alpha = i_1(x_1) \cdot i_2(x_2) \cdot ... \cdot i_k(x_k)$, where each x_j is associated with i_j. The association of output sequence and output parameter value sequence is defined analogously. Then, for the state $q_1 \in Q$, when applying a complete parameterized input sequence $\gamma \otimes \alpha$, M moves successively from q_1 to the states $q_{j+1} = \delta(q_j, i_j, x_j)(1 \leq j \leq k)$. We extend the functions from input symbols to parameterized

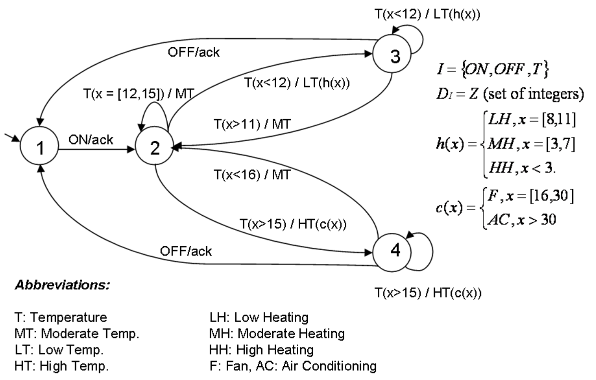

Figure 1. An example of PFSM model

input sequences as $\delta(q_1, \gamma, \alpha) = q_{k+1}$ to denote the final state q_{k+1} and $\lambda(q_1, \gamma, \alpha) = o_1(y_1) \cdot o_2(y_2) \cdot ... \cdot o_k(y_k)$, where each $o_j = \lambda(q_j, i_j, x_j), y_j = \sigma(q_j, i_j)(x_j), \forall 1 \leq j \leq k$, to denote the complete parameterized output sequence, when applying $\gamma \otimes \alpha$ on q_1. An example of PFSM is shown in Figure 1.

2.2. Learning Models in Practice

Our model is an extension of simple finite state machine enriching it with parameters and predicates. At the same time, it is a restricted model that does not capture the in-depth details of a system, e.g., variables, assignment functions and complex guards, etc. This restriction is applied in order to mitigate the complexity of the learning algorithm, since our approach is to learn the formal models from black box components. It is hard to obtain fully accurate models in the form of, e.g., EFSM (Extended Finite State Machines) [16, 15] and ASM (Abstract State Machines) [5], from a black box component. Actually, most works [9] [2] about learning models from the behaviors of components are limited to simple state models. For example, a polynomial time algorithm [1] is well-known that conjectures a DFA through active learning technique. Recently, another approach [11] to learn DFA is proposed using genetic algorithm.

Basic DFA or FSM learning algorithms have been reused in the context of model-checking [7]. Actually, model-checking builds reachability graphs akin to DFA, so the techniques can be integrated at that level. Here, we need to address arbitrarily complex data values in parameters exchanged between components. Modelling at FSM level could result in a combinatorial blow up on transition labels (and the required number of tests to learn them), and in loss of genericity of the models. Therefore, in the study of learning reactive systems [8], an I/O automaton is taken into account with certain optimization in the basic Angluin's DFA algorithm [1]. As an extension, we have presented an algorithm [12] to learn Mealy machines from a black box component. Recently, a parameterized model [3] has been proposed for which the existing algorithm is adapted. This model preserves all the properties of DFA, plus incorporates parameters and guards on transitions. However, it does not associate outputs with actions (or inputs) and assumes only boolean space for the parameter values. There are other similar works, e.g., [14] and [19] that propose techniques to extract more in-depth knowledge and considerably high-level state based models from a component, but they are mostly relying on source code. On the contrary, we have proposed a simple parameterized model [13] and an algorithm to infer it from a black box component. In this paper, we extend our work to enrich this model with the notion of predicates and observable nondeterminism.

Our model (in def. 2.1) exhibits the following extensions: 1) parameterized inputs and outputs, 2) arbitrary (non-finite) domains for parameters, 3) guards on input parameters, i.e., p defines a guard for the corresponding transition, 4) observable nondeterminism when interacting with input parameter values, 5) arbitrary functions f for output parameter calculation, 6) f can be a partial function.

At the same time, it entails the following restrictions w.r.t. to EFSM: 1) single parameter for inputs and outputs, and a single domain for all I/O symbols, 2) no (state) variables: all state information must be encoded in Q.

The first point is not a real restriction to the model, in fact, it is purely a notational simplification. Since we allow arbitrary domains and output parameter functions, it simply means that we need some mapping between the vectors of types of parameters for actual interactions and a suitable domain to represent them. However, the second point is more tricky. The problem stems from the fact that since the internal structure of a component is unknown, the encoding of state information between Q and variables might be arbitrary, so the learning process can hardly infer a meaningful state structure. We can argue that the main limitation is not the absence of variables but the fact that the number of states is finite. Anyhow, since we are set in a testing context, the number of tests sets a limit on observations and we shall learn finite approximations of machines.

Our major contribution has been in the corresponding algorithm needed to learn such a complex model. In the next section, we shall present the key points of our approach, extending previous algorithms.

3 Usage of PFSM Model in Component Integration Testing

In this section, we describe the approach of integration testing guided by machine learning, in which the PFSM model is used.

3.1 Integration Testing Approach

Here we adapt the approaches described in [12] and [13] for PFSM. The components we deal with are viewed as black boxes with known input alphabets. The integrator has a number of test scenarios for the global interaction of the system with its environment.

The outline of the component integration testing approach is as follows:

1. In the first step, for each component C, an input alphabet I_C is defined which corresponds to the invocations on interfaces, and an input parameter domain D_I is defined by mapping from vectors of types of parameters for actual interactions. A simple mapping mechanism is to define D_i as the Cartesian product of the domains of all the related parameters for each input $i \in I_C$, and to define $D_I = \bigcup_{i \in I_C} D_i$.

2. Each component is (unit) tested separately using the learning algorithm until a conjecture can be made. The basic ideas of learning algorithm for PFSM are described in Section 3.2. This provides the first model $C^{(1)}$ for C.

3. The components are integrated. The assembly is tested in two stages.

   In first stage, we systematically test the provided system-wide test scenarios expected from the assembly. In that stage, we select input parameter values according to test scenario to construct parameterized input sequences, observe the parameterized output sequences, and determine whether a test scenario is respected. With the help of domain expertise, faults can be detected, or a discrepancy with the inferred model may be identified, leading to incremental refinement of the model.

   In a second stage, we generate (interoperability) tests from the models of the components. Existing works on integration (interoperability) testing [10] [6] [18] can be adapted to the PFSM model. Since we are dealing with parameterized model, for each test case, we select input parameter values in addition to input symbols. Tests are performed until a discrepancy between predictions from the models is found or some coverage criteria on the models are achieved. Classifying discrepancies as faults may require expert input.

   In our previous work with simplified PFSM [13], from a source state, transition is only determined by the input symbol. Thus, parameter selection in testing is not as important as in this work. In our context, the main focus is to make sure the interactions between the components are correct. Thus, the test generation approach is to cover all the reachable interactions between components. In order to achieve this, we select parameter values used in unit testing which can trigger interactions according to the current models of components. On the other hand, since the current models are only approximations of the component implementations, we want to observe more behaviors of the components. So, we select parameter values which have not been used in unit testing. In test generation, the two parameter selection strategies are combined.

4. In both stages, discrepancies can lead to model refinement. The counterexamples found are injected in the learning algorithm to obtain the new model $C^{(i+1)}$ of component C.

5. At the end of integration testing, for each component, we have a PFSM model, which is consistent with all the tests that have been passed. At the same time, the joint behavior of these components have been systematically tested. Faults could be discovered during integration test execution.

3.2 Unit Testing / Learning Algorithm

Assume an unknown PFSM $M = (Q, I, O, D_I, D_O, T, q_0)$ with known input symbols I and input parameter domain D_I is used to model a component C. Since we can submit any input sequence with parameters to the component and observe the corresponding output sequence with parameters, for any input sequence $r \otimes x (r \in I^*, x \in D_I^*, |r| = |x|)$, $\lambda(q_0, r, x)$ can be known from testing. We also assume that each component can be reset to its initial state before each test. A full formal description and illustration of the learning algorithm are addressed in our recent paper [17]. Here we provide a broad view, covering the basic elements of the algorithm.

3.2.1 Observation Tables

The basic data structure of the learning algorithm is an observation table, denoted by (S, E, R, T). S is a nonempty finite set of *access sequences* as they are used to access the different states of the PFSM we are learning from its initial state. In PFSM, starting from the initial state, the end state is determined not only by input symbol sequence but also input parameter value sequence. Based on this observation, we call the association of an input symbol sequence and a set of input parameter value sequences as *Composite Input Sequence*, and take composite input sequence as access sequence.

	ON	OFF	T
$\epsilon \otimes \epsilon$	$(\perp, ack \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, \Omega \otimes \perp), (12, \Omega \otimes \perp)$
$ON \otimes \perp$	$(\perp, \Omega \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, LT \otimes MH), (12, MT \otimes \perp)$
$ON \cdot T \otimes \perp \cdot 4$	$(\perp, \Omega \otimes \perp)$	$(\perp, ack \otimes \perp)$	$(4, LT \otimes MH), (12, MT \otimes \perp)$
$ON \cdot T \otimes \perp \cdot 12$	$(\perp, \Omega \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, LT \otimes MH), (12, MT \otimes \perp)$
$ON \cdot T \cdot OFF \otimes \perp \cdot 4 \cdot \perp$	$(\perp, ack \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, \Omega \otimes \perp), (12, \Omega \otimes \perp)$
$ON \cdot T \cdot T \otimes \perp \cdot 4 \cdot 4$	$(\perp, \Omega \otimes \perp)$	$(\perp, ack \otimes \perp)$	$(4, LT \otimes MH), (12, MT \otimes \perp)$
$ON \cdot T \cdot T \otimes \perp \cdot 4 \cdot 12$	$(\perp, \Omega \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, LT \otimes MH), (12, MT \otimes \perp)$

Table 1. An Observation Table

E is a nonempty finite suffix-closed set of input strings. They are called *Separating Sequences*, as their goal is to separate between different states of the conjecture.

R is a superset of S. Whenever an access sequence is added into S, we obtain a group of new composite input sequences by extending the input symbol sequence with all $i \in I$ and selecting $x \in D_I$ to extend the set of input parameter value sequences, and add these new composite input sequences to R.

Initially, $R = S = \emptyset$, and $E = I$. The first operation of the learning algorithm will be adding $\epsilon \otimes \epsilon$ [1] into S.

The function T is defined on $R \times E$. In PFSM, starting from a parameterized input sequence, different parameterized output sequences can be observed by inputting the same separating sequence with different input parameter value sequences. So, for $r \in R$ and $e \in E$, the entry $T(r, e)$ is the mapping from input parameter value sequences to parameterized output sequences.

Observation table can be visualized as a two-dimensional array with rows labelled by the elements of R and columns labelled by the elements of E, with the entry for row r and column e equals to $T(r, e)$.

An example of observation table is shown in Table 1 for learning a PFSM model given in Figure 1.

3.2.2 Make Conjecture from Observation Tables

In the learning procedure, the observed behaviors of the component C are recorded in observation table. Finally, we make a conjecture model of the component from observation table. The conjecture has the minimum number of states among the models consistent with observation table. In order to make the conjecture well defined, the observation table should exhibit three properties.

In making a conjecture from observation table, access sequences in S represent states. In order to determine whether two composite input sequences correspond to the same state or not, we need to compare rows in the table. For this purpose, we introduce the following definitions.

For $r_1, r_2 \in R$, if starting from states reached by r_1 and r_2 respectively, we can obtain different parameterized output sequences by inputting the same parameterized input

[1] ϵ is an empty string.

sequence, we say the rows corresponding to r_1 and r_2 are *dissimilar*. In this case we are sure r_1 and r_2 correspond to different states.

For two rows which are not dissimilar, in order to compare them, we need to make sure that the same group of parameterized input sequences has been executed starting from their corresponding states. If for each pair of rows which are not dissimilar, the observation table has this property, we say the observation table is *balanced*.

For balanced observation table, we can compare rows by the normal relation of equality "=". Then, the concepts *closed* and *consistent* can be defined accordingly. In the observation table, the states represented by $r \in R \backslash S$ are successive states of the states represented by $s \in S$. If all these successive states are also represented by some $s' \in S$, then the observation table is *closed*. If for all $s_1, s_2 \in S$ representing the same states, they have the same successive states for all executed input symbols and parameter values, then the observation table is *consistent*.

When the observation table is balanced, closed, and consistent, we can make a conjecture PFSM from the observation table in which states are defined based on S and transitions are derived from T. The information recorded in the cells is used to label transitions with parameterized inputs and their corresponding parameterized outputs and R helps in selecting correct source and target state from S for each transition.

For an interesting reader, we refer to [17] for complete formal definitions of the algorithm and its complexity discussion.

3.3 Illustration

We have given an example of HVAC (Heating-Ventilating-Air-Conditioning) controller in Figure 1. It works on different temperature values (provided externally through temperature sensor etc.) and controls heating and cooling systems with respect to those values. Suppose that this component is integrated in a system of home appliances where various other components are working and interacting with each other. Consider that HVAC controller is a black box component and needs to be tested for this integrated environment. According to our approach, we learn this component with its basic input set $I = \{ON, OFF, T\}$ and using the learning algorithm given in 3.2. We obtain a conjecture of this component, given in Figure 2, from the observation table given in Table 1. Other components are learned individually following the same unit testing procedure.

The conjecture depicts the behavior of the controller for the temperature values exercised in its unit testing, i.e., 4 and 12. In the integration testing, we generate test cases using the learned models to test the interoperability of this

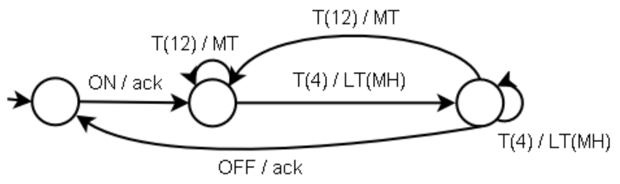

Figure 2. Conjecture of the example in Figure 1 from Table 1

component with others in the system. The model in Figure 2 does not portray the behavior of the controller when weather is extreme. To fulfill the objective of discovering more behaviors, we generate test cases with parameter values other than used in the unit testing, i.e., values higher than 12 and lower than 4. This leads to find more behaviors of the actual component, followed by its refinement using the learning algorithm (by injecting new parameter values in the observation table). The new model will then be derived and put again under integration testing.

We reason that the parametric structure of the model steers a reasonable testing effort that improves comprehension about black box components iteratively (by using learning algorithm) as well as helps in testing their interoperability within an integrated system.

4. Conclusion

We have presented a parameterized model that can be learned from black box components using existing techniques. For that, we have given a flavor of learning algorithm in order to learn this parameterized model. We also illustrated our integration testing approach for components using their learned models with the help of an example.

We advocate the need of expressive models in machine learning that are more adequate for using in model-supported tasks. Actually, our model provides a subset that could be easily integrated into the state-based part of models (such as SDL or UML) and associated tools. The core notation that we use is adequate for the learning algorithm and engine. But it uses a standard semantics extended state models (leaving apart our syntactic restrictions). Our next step is to optimize the overall process of learning and testing of integrated system of COTS with a comparative analysis of our algorithm with the existing ones, e.g., [11] [8] [3]. We are also considering moves towards extended FSM with variables, by assuming some extra information on the structure of the components. The other directions is to start investigating statistical methods that can be used to calculate the relationship between learned models and actual black box components. We intend to use case-studies in this domain to analyze our techniques on real world problems, especially in the telecommunication sector.

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2:87–106, 1987.

[2] J. L. Balcazar, J. Diaz, and R. Gavalda. Algorithms for learning finite automata from queries: A unified view. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72, 1997.

[3] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines with parameters. In *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006.

[4] S. Beydeda and V. Gruhn. The self-testing cots components (stecc) strategy – a new form of improving component testability. In M. Hamza, editor, *Proceedings of the Seventh IASTED International Conference on Software Engineering and Applications*, pages 222–227. ACTA Press, 2003.

[5] E. Börger. The ASM method for system design and analysis. a tutorial introduction. In *FroCos*, volume 3717 of *Lecture Notes in Computer Science*, pages 264–283. Springer, 2005.

[6] M. K. C. Besse, A. Cavalli and F. Zaidi. Two methods for interoperability tests generation: An application to the tcp/ip protocol. In *TestCom*, Berlin, 2002.

[7] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 357–370, 2002.

[8] H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2003.

[9] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory*. MIT Press, Cambridge, MA, USA, 1994.

[10] O. Koné and R. Castanet. Test Generation for Interworking Systems. *Computer Communications*, 7(23):642–652, 2000.

[11] Z. Lai, S. C. Cheung, and Y. Jiang. Dynamic model learning using genetic algorithm under adaptive model checking framework. In *QSIC '06: Proceedings of the Sixth International Conference on Quality Software*, pages 410–417, Washington, DC, USA, 2006. IEEE Computer Society.

[12] K. Li, R. Groz, and M. Shahbaz. Integration testing of components guided by incremental state machine learning. In *TAIC PART*, pages 59–70. IEEE Computer Society, 2006.

[13] K. Li, R. Groz, and M. Shahbaz. Integration testing of distributed components based on learning parameterized i/o models. In *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 436–450. Springer, 2006.

[14] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 229–239, New York, NY, USA, 2002. ACM Press.

[15] A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in efsm testing. *IEEE Trans. Softw. Eng.*, 30(1):29–42, 2004.

[16] T. Ramalingom, K. Thulasiraman, and A. Das. Context independent unique state identification sequences for testing communication protocols modelled as extended finite state machines. *Computer Communications*, 26(14):1622–1633, 2003.

[17] M. Shahbaz, K. Li, and R. Groz. Learning and integration of parameterized components through testing. In *To be appeared in TestCom*, 2007.

[18] W.-T. Tsai, X. Bai, R. J. Paul, W. Shao, and V. Agarwal. End-to-end integration testing design. In *COMPSAC '01: Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, pages 166–171, Washington, DC, USA, 2001. IEEE Computer Society.

[19] N. Walkinshaw, K. Bogdanov, and M. Holcombe. Identifying state transitions and their functions in source code. In *TAIC PART*, pages 49–58. IEEE Computer Society, 2006.