Symbolic Execution-Based Approach to Extracting a Micro State Transition Table

Takahiro Shimizu
takahiro915@ertl.jp
Nagoya University
Japan

Norihiro Yoshida
yoshida@ertl.jp
Nagoya University
Japan

Ryota Yamamoto
muku@ertl.jp
Nagoya University
Japan

Hiroaki Takada
hiro@ertl.jp
Nagoya University
Japan

ABSTRACT

During embedded system development, developers frequently change and reuse the existing C source code for the development of a new but behaviorally similar product. Such frequent changes generally decrease the understandability of C source code although the developers have to understand how it behaves and how to reuse it. So far, much research has been done on symbolic execution techniques that statically analyze the behavioral aspect of given source code. In this paper, we propose a symbolic execution-based approach to extracting a Micro State Transition Table (MSTT) that helps developers understanding the behavioral aspect of embedded C source code based on a fine-grained state transition model. As a case study, we applied the proposed approach to a collection of source files and then confirmed the correctness of the extracted MSTTs.

CCS CONCEPTS

• **Software and its engineering** → *Software evolution.*

KEYWORDS

Embedded system, Symbolic execution, State transition table

ACM Reference Format:
Takahiro Shimizu, Norihiro Yoshida, Ryota Yamamoto, and Hiroaki Takada. 2019. Symbolic Execution-Based Approach to Extracting a Micro State Transition Table. In *Proceedings of the 2019 ACM SIGSOFT International Workshop on Testing, Analysis, and Verification of Cyber-Physical Systems and Internet of Things (TAV-CPS/IoT '19), July 16, 2019, Beijing, China.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3341108.3342244

1 INTRODUCTION

In embedded systems software development, changes to the target software specifications are often accompanied by changes to the source code. Even when products are shipped as identical systems, disparities may occur in the hardware in each factory, and there may be gradual changes to the source code in each factory in which it is developed. When developing similar systems to existing products, they are often reused by partially changing the source code of the existing products [2, 3, 12]. From this background, there is a trend in embedded systems development of repeatedly changing and reusing source code.

When developing embedded systems, as high responsiveness is required from low performance computers, source code is often written in the C language [1, 9, 10]. In comparison to object-oriented languages, the C language has inexpresiveness in terms of modularity. For this reason, when changing the source code at the same time the hardware specifications are changed, the level of module complexity tends to increase as a result of the addition of new conditional branch statements. In particular, in projects with rapidly approaching deadlines, easily-available low condition branch statements tend to be added, which decrease maintainability and reusability.

The State Transition Design Research Working Group (hereafter, STDR-WG) in the Japan Embedded System Technology Association has studied reverse engineering targeting source code that includes complex condition branch statements, and Micro State Transition Table (hereafter, MSTT) has been proposed [18]. The MSTT has been proposed to support understanding of state transitions within modules (i.e., compilation units) written in the C language, and is a table that, with one variable declared within the module as a variable expressing a state (transition variable), expresses the process and state transitions corresponding to states and events. If an MSTT exists, it is considered that, even if the module contains complex condition branches, by comparing with the MSTT at the time of maintenance or reuse, it is possible to promote understanding of that module. It is fine-grained, compared with general state transition models, in that it expresses state transitions at the modular rather than the system level.

The STDR-WG proposed the concept of MSTT and a procedure for extracting an MSTT from a module manually; however, with limited human resources, extracting an MSTT manually from a module that includes complex condition branches is not realistic.

There is existing research that aims to achieve reverse engineering with a state transition design; however, these are mainly methods that extract state transition models at the system level

TAV-CPS/IoT '19, July 16, 2019, Beijing, China

ACM ISBN 978-1-4503-6867-4/19/07…$15.00
https://doi.org/10.1145/3341108.3342244

[6, 15, 16], or are methods that target modules written in object-oriented languages [8, 13, 14], and it is difficult to use them to extract an MSTT from a module written in the C language.

In this study, we use symbolic execution as a method of statically analyzing complex condition branches, and attempt to extract an MSTT from modules written in the C language. Symbolic execution technique is an approach to allocating symbols and abstractly executing programs, rather than providing concrete values for certain variables [4, 5, 7]. The proposed method automatically extracts an MSTT using the following procedure, when the user (practitioner) specifies state transitions from variables within the modules.

(1) From the source code, generate a symbolic execution graph (graph that summarizes source code conditions and corresponding process information) [5] based on the symbolic execution tool TRACER [5].
(2) Extract conditions and process from the generated symbolic execution graph and summarize these in tabular form.
(3) Extract the MSTT, based on the table summarizing conditions and process, and state variables selected by the user.

The main contributions of this study are as follows:

- Using symbolic execution, we devised methods for extracting an MSTT from the modules written in the C language using only static analysis.
- We implemented a prototype tool of the proposed method and applied to several source files and confirmed the correctness of the extracted MSTTs.

```
1  int state, out;
2  void task(int s,int t){
3    if(t==1){
4      s++;
5      if(s<10){
6        switch(state){
7        case 1:
8          out=s; break;
9        case 2:
10         out=0; state=1; break;
21       default:
22         s++; state=2;
23 }}}}
24 int main(){
25   int a,b;
26   scanf("%d", &a);
27   scanf("%d", &b);
28   task(a,b);
29   return 0;
30 }
```

Figure 1: Source Code A

2 RELATED TECHNOLOGIES

2.1 Symbolic Execution

Symbolic execution allocates symbols and artificially executes programs, rather than providing concrete values for certain variables [4, 5, 7]. Where program execution on a certain execution path ends, it summarizes conditions appearing within the execution path, and derives the path conditions when executing this execution path. By solving the derived path conditions, it is possible to learn whether the execution path can be executed and, if it can be executed, what kinds of values the variables have at the time of execution.

We shall now explain the TRACER [5] symbolic execution tool used in this study. TRACER generates a graph (hereafter, called a symbolic execution graph) that summarizes the conditions and process for all execution paths acquired using symbolic execution. The symbolic execution graph is given as a DOT file. We introduce an example of the symbolic execution graph using the "task" function in Figure 1. A representation of the task function section from among the symbolic execution graph generated from the source code in Figure 1 is shown in Figure 2.

Here, we shall explain each section of the symbolic execution graph. The symbolic execution graph expresses execution paths by the nodes and edges. From among these nodes, the start and end nodes in the function are colored. Nodes include rectangular and diamond-shaped nodes, and the diamond-shaped nodes represent branches within the source code. Additionally, node labels express positions within the source code, and there is no duplication within the node labels. In the edge label after the branch, the conditions for promoting this edge are written, and the process is written in the edge label in which the process exists. In addition, a dashed arrow with label s indicates that the starting state is subsumed by the ending state[1].

2.2 Micro State Transition Table (MSTT)

The Micro State Transition Table (MSTT) proposed by the STDR-WG takes the group of values taken from the variables within the source code as a state, and the group of combinations of values taken from the other variables as events, and expresses, in tabular form, how process and transitions can occur when there is a combination of a certain state and event [18]. The characteristics of the MSTT are such that the state transitions are expressed based on the conditions branches within the functions, with the assumption that the state transition within the functions will occur. Here, we explain the way of reading the MSTT using the example shown in Figure 3. In Figure 3, the state is shown at the top of the table and the event is expressed on the left side of the table. The process is written in the cells corresponding to the events, and transitions are shown when the state variable values within the process change. Additionally, transitions have (t) written in front of the formula to express this. As an example, if the event $t = 1 \ \& \ 10 > s$ occurs when the state is $state = 1$, the process $s := s + 1$ and $out := s$ is carried out, and when event $t = 1 \ \& \ 10 > s$ occurs when the state is $state = 2$, the process $s := s + 1$ and $out := 0$, and the transition $state := 3$ are performed. Here, the cell formula order in which the process and transition are written are expressed as a time series within the

[1] With the respect to the rigorous explanation of a dashed arrow (i.e., subsumption) in Figure 2, please refer to page 760 of [5].

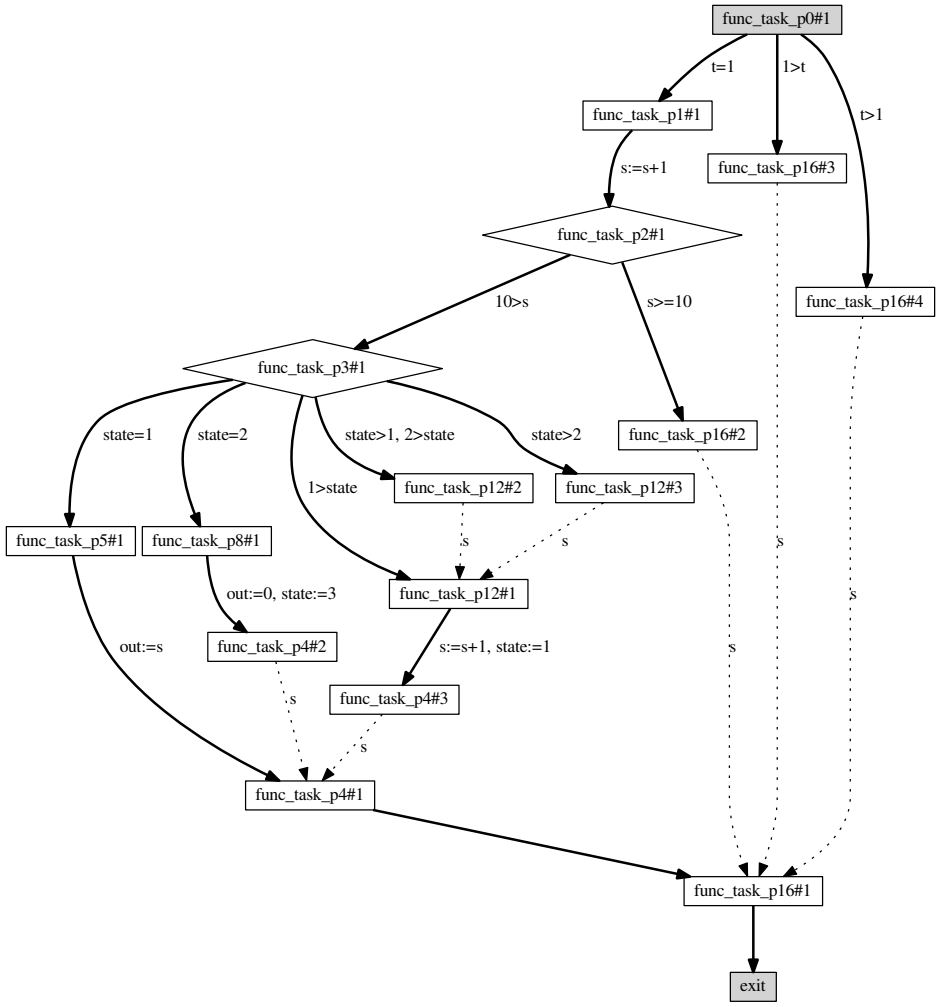

Figure 2: Symbolic Execution Graph for the Function task

state / event	state=1	state=2	1>state	state>2
t=1&10>s	s:=s+1 out:=s	s:=s+1 out:=0 (t)state:=3	s:=s+1 s:=s+1 (t)state:=1	s:=s+1 s:=s+1 (t)state:=1
t=1&s>=10	s:=s+1	s:=s+1	s:=s+1	s:=s+1
1>t	NONE	NONE	NONE	NONE
t>1	NONE	NONE	NONE	NONE

,

Figure 3: An Example of MSTT

source code. Additionally, cells in which the corresponding process and transitions do not exist are expressed as NONE.

3 PROPOSED METHOD

Here, we explain the method of extracting the MSTT proposed in this study. The method in this study consists of the following three steps:

- **Step 1:** A symbol graph is generated from the source code using TRACER.
- **Step 2:** A condition-process table is extracted as an interim state for the purpose of extracting the MSTT from the generated symbolic execution graph.
- **Step 3:** An MSTT is extracted from the condition-process table and state variables selected by the user.

In Section 3.1 and 3.2, we shall explain Step 2 and Step 3 respectively.

3.1 Extracting the Condition-Process Table

Here, we shall explain the method of extracting the condition-process table from the symbolic execution graph generated using TRACER. The condition-process table summarizes the conditions

and process for each execution path within the symbolic execution graph in a table, enabling you to reference what kind of process takes place when certain conditions occur.

Extraction of the condition-process table occurs using the following steps:

Step (a) Finds the start point of the symbolic execution graph and moves to Step (b).

Step (b) Traverses the symbolic execution graph. If it reaches a branch, it moves to Step (c), and if it reaches the end point of the execution path, it moves to Step (d).

Step (c) Progresses to branches it has not reached yet, and then moves to Step (d).

Step (d) Extracts the conditions and process of execution paths it has not reached yet, and writes these to the condition-process table. It returns to the final branch within branches left in unexplored paths and then moves to Step (c). In case there are no more unexplored paths, the symbolic execution graph search ends.

Here, the start point is the node where there are no edges that continue to this node, and the end point is the node where there are no edges that continue from this node.

Here, we explain the method of extracting the condition-process table using the example of the symbolic execution graph (Figure 2) generated from the source code in Figure 1. It passes the symbolic execution graph from the start point func_task_p0#1. Because func_task_p0#1 is a branch node, here it follows the $t = 1$ edge. Branches after that follow, respectively, $10 > s$ with func.task_p2#1 and $state = 1$ edge with func_task_p3#1. When it reaches the end point exit of the symbolic execution graph, it extracts the conditions and process from the edge label for the followed execution path, and writes to the condition-process table. In the case of this execution path, the conditions are $t = 1 \ \ \& \ \ 10 > state = 1$, and the process is $s := s + 1$, $out := s$. After finishing extraction of the conditions and process, it returns to the final branch node func_task_p3#1. The above is repeated until all of the execution paths are followed. Additionally, if there is no process, the process section of the condition-process table is written as NONE. All execution paths of the symbolic execution graph are followed and the completed condition-process table is shown in Table 1. Here, if we consider the variable type in the symbolic execution graph, there are execution paths that have conditions that are not actually executed. In this example, the edge labels from func_task_p3#1 to node func_task_p12#2 are $state > 1, 2 > state$, but because the state is an int type, this execution path is not actually executed. Therefore, the conditions and process for this execution path are not actually added to the condition-process table.

Table 1: Condition-Process Table for Source Code in Figure 1

Condition	Process
$t=1$ & $10>s$ & $state=1$	$s := s + 1$; $out := s$;
$t=1$ & $10>s$ & $state=2$	$s := s + 1$; $out := 0$; $state := 3$;
$t=1$ & $10>s$ & $1>state$	$s := s + 1$; $s := s + 1$; $state := 1$;
$t=1$ & $10>s$ & $state>2$	$s := s + 1$; $s := s + 1$; $state := 1$;
$t=1$ & $s >= 10$	s:=s+1;
$1>t$	NONE
$t>1$	NONE

3.2 Extracting an MSTT

Extraction of the finely-grained state transition table is performed using the following steps:

Step (e) Events and states are determined from the condition-process table and the state variables selected by the user.

Step (f) The content of the process section of the condition-process table is written to the corresponding cell on the finely-grained state transition table.

Here, we shall explain Step (e). The condition formats in which the state variables appear from among the condition section of the condition-process table are extracted in a way that prevents duplication. One column is created for each state and the extracted states are written as column headings. Additionally, the duplicate items from the condition formats after extracting the condition formats related to the state variables are deleted, and the remaining ones are set as events. One row is created for each event and the events are written as row headings.

Now, we shall explain Step (f). Looking from the top of the condition-process table, if the condition format for the state change is included in the condition section, the cell intersecting the event and state corresponding to the condition formula in the condition section is written to the content of the process section. If the condition format for the state change is not included in the condition section, the process section content is written to all cells for the same event in this process section. Here, when writing the content of the process section, formulas expressing substitution to the state variables have (t) written at the front of the formula to show that they are a transition. After referencing all cells in the condition-process table, if there are cells for which process or transitions are not written within the finely-grained state transition table, **** is written to these cells.

In the source code in Figure 1, state is an example of a state variable. There are the four states of $state = 1$, $state = 2$, $1 > state$, and $state > 2$. Additionally, there are the four events of $t = 1 \ \ \& \ \ 10 > 2$, $t = 1 \ \ \& \ \ s >= 10$, $1 > t$, and $t > 1$. process columns and event rows are created for these states and events. By writing the processes and transitions corresponding to these events and states, while referencing Table 1, the micro state transition is completed as shown in Figure 3.

4 CASE STUDY

We developed a tool to automatically perform Steps 2 and 3 of the proposed method and conducted a case study. The developed tool receives a DOT file expressing the symbolic execution graph generated by TRACER and state variable names, and then generates a Tab-Separated Values (TSV) file expressing the MSTT.

The target source code consisted of nine source files created as part of the activities of the STDR-WG (Table 2). Here, the number of conditional branch statements in Table 2 comprises the number of if statements and switch statements in the source code, and the number of execution paths is the number of execution paths discovered when following the symbolic execution graph in Section 3.1.

The source code expresses both state transitions and the occurrence of events.

4.1 Steps

The steps of the case study are as follows:

Step I: Shaping of the source code and extraction of the symbolic execution graph
Step II-1: Manual extraction of the MSTT.
Step II-2: Extraction of the MSTT using the tool.
Step III: Comparison of the MSTTs extracted manually and by using the tool.

First, we shape the target source code so that it can be analyzed by TRACER in Step I. More specifically, this relates to finishing writing the definitions of constants using #define. Then, the symbolic execution graph is generated by executing TRACER for the shaped source code.

Next, we extract the MSTT from the symbolic execution graph in Step II-1. The DOT file is converted to a PDF file, and the symbolic execution graph is schematized. From the schematized symbolic execution graph, the MSTT is created manually using the method in Section 3.

Moving on, the MSTT is extracted from the symbolic execution graph using the tool developed in Step II-2. The symbolic execution graph generated using TRACER is applied as an input for the tool developed in the study, and the MSTT is extracted. Additionally, the state transitions from the source code and symbolic execution graph considered to be the most suitable are specified. The output TSV file can be opened with spreadsheet software and the content confirmed.

Finally, we compare the MSTT extracted using the tool in Step III and the MSTT extracted manually and confirm that the content of the events, states, processes, and transitions is the same.

4.2 Results

Here, we shall explain the results of the case study. First, we will explain the results of automatically extracting the MSTT using the developed tool. The MSTT could be extracted from the symbolic execution graph using the developed tool in all target source files. Additionally, when comparing the events and states between the automatically extracted ones and those created manually, although there were differences in the order of the events and states, the content for state transition matched for all target source files.

As an example, we shall introduce the symbolic execution graph from one of the target source files (Figure 4) and the MSTT (Figure 5) extracted using the tool. There are no format errors in the MSTT in Figure 5, and it matches the MSTT extracted manually.

Table 2: Statistics of Target Source Files

	LoC	#func.	#cond. statement	#variable	#path
max.	139	3	21	8	26
min.	27	2	3	2	4
avg.	72.33	2.667	9.222	4.000	14.56

From the results of the case study, it is able to see that MSTTs were extracted correctly from the symbolic execution graphs in all source files using the developed tool. Extraction of the MSTT using the developed tool in all cases is completed within several seconds, and the cost for extracting the MSTT is reduced compared to when performed manually. Additionally, it is possible to know the information for conditions and process within the source code from the extracted MSTT, and this will aid understanding of source code behavior when handing more complex source code.

5 RELATED WORK

Several approach for extracting state transition diagrams have been proposed, but most of these are based on the traces of concrete execution [11, 17]. The method proposed in this study uses symbolic execution, and can be applied even in environments without the actual equipment. In embedded systems, it can be costly to construct a test environment for acquiring information at the time of execution, and there are times when applying methods requiring information at time of execution is not realistic. Walkinshaw et al. proposed a method for extracting state transition diagrams from source code using extraction execution [15]. The state transition diagrams extracted by their method are coarsely grained and only deal with transitions generated from method call statements and exception process. With the method of Walkinshaw et al., the functions are state transition points, and a state transition diagram for the system as a whole is extracted [15]. In this study, however, we define an MSTT, and handle branch statements within the function for more fine-grained expression than function call statements.

Furthermore, Walkinshaw et al. propose a tool for automatically generating a state transition table from an execution trace [16]. Because the state transition table is dynamically extracted, paths that are not executed are not taken into consideration. Additionally, because the state transition conditions are derived using machine learning, values that are incompatible with the actual trace may be the threshold. There are also methods that conduct reverse engineering on the source code written in object-oriented languages and extract a state transition model [8, 13, 14]. In [14], reverse engineering of the state transition model is conducted using the following three sorts of information:

- Extracted state value domains
- Mapping of abstract state values from concrete state values
- Abstract meanings of all primitive commands within the provided program

The proposed method in this paper can extract state transition tables even when these restrictions are not defined. [13] and [8] attempt to use symbolic execution from Java byte code or C++ source code to extract the state transition model.

6 SUMMARY

In this study, we proposed a method for extracting MSTTs using symbolic execution from source code including state transition design. Furthermore, we developed a tool to extract an MSTT from TRACER generated items and confirmed that MSTTs can be extracted using the developed tool when conducting the case study.

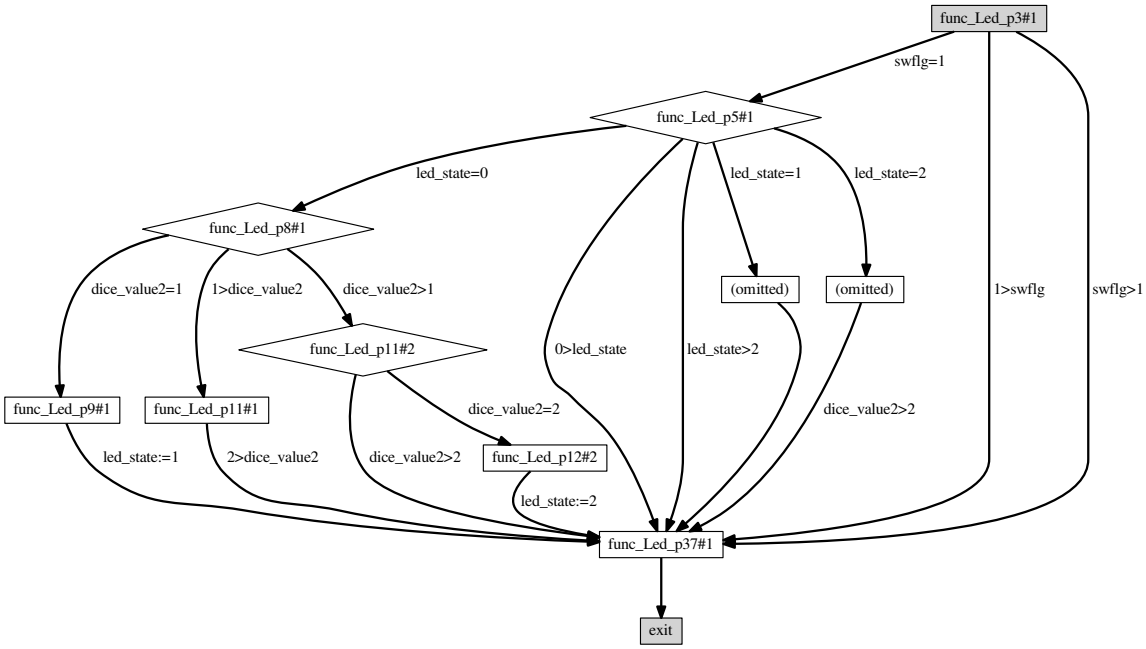

Figure 4: Symbolic Execution Graph from an Experimental Source Code

	led_state=0	led_state=1	led_state=2	0>led_state	led_state>2
swflg=1&dice_value2=1	(t)led_state:=1	(t)led_state:=2	(t)led_state:=0	****	****
swflg=1&1>dice_value2&2>dice_value2	NONE	NONE	NONE	****	****
swflg=1&dice_value2>1&dice_value2=2	(t)led_state:=2	(t)led_state:=0	(t)led_state:=1	****	****
swflg=1&dice_value2>1&dice_value2>2	NONE	NONE	NONE	****	****
swflg=1	****	****	****	NONE	NONE
1>swflg	NONE	NONE	NONE	NONE	NONE
swflg>1	NONE	NONE	NONE	NONE	NONE

Figure 5: An MSTT from Figure 4

REFERENCES

[1] Xavier Allamigeon, Wenceslas Godard, and Charles Hymans. 2006. Static analysis of string manipulations in critical embedded C programs. In *Proc. of SAS 2006*. 35–51.

[2] Gunnar Brataas, Svein Olav Hallsteinsen, Romain Rouvoy, and Frank Eliassen. 2007. Scalability of decision models for dynamic product lines. In *Proc. of SPLC 2007*.

[3] Christopher Brink, Philipp Heisig, and Sabine Sachweh. 2015. Using cross-dependencies during configuration of system families. In *Proc. of PROFES 2015*. Springer, 439–452.

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of OSDI 2008*. 209–224.

[5] Joxan Jaffar, Vijayaraghavan Murali, Jorge A Navas, and Andrew E Santosa. 2012. TRACER: A symbolic execution tool for verification. In *Proc. of CAV 2012*. Springer, 758–766.

[6] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. 2003. Generalized symbolic execution for model checking and testing. In *Proc. TACAS 2003*. Springer, 553–568.

[7] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[8] David Kung, Nimish Suchak, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, and Chris Chen. 1994. On object state testing. In *Proc. COMPSAC 94*. 222–227.

[9] Sangchul Lee and Jae Wook Jeon. 2010. Evaluating performance of Android platform using native C for embedded systems. In *Proc. of ICCAS 2010*. 1160–1163.

[10] Daniel Wesley Lewis. 2002. *Fundamentals of Embedded Software: Where C and Assembly Meet* (1st ed.). Prentice Hall PTR.

[11] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2006. Inferring state-based behavior models. In *Proc. WODA*. 25–32.

[12] Kunming Nie, Tao Yue, Shaukat Ali, Li Zhang, and Zhiqiang Fan. 2013. Constraints: The core of supporting automated product configuration of cyber-physical systems. In *Proc. of MODELS 2013*. 370–387.

[13] Tamal Sen and Rajib Mall. 2016. Extracting finite state representation of Java programs. *Software & Systems Modeling* 15, 2 (2016), 497–511.

[14] Paolo Tonella and Alessandra Potrich. 2005. *Reverse engineering of object oriented code*. Springer. 115–132 pages.

[15] Neil Walkinshaw, Kirill Bogdanov, Shaukat Ali, and Mike Holcombe. 2008. Automated Discovery of State Transitions and Their Functions in Source Code. *Software Testing, Verification & Reliability* 18, 2 (2008), 99–121.

[16] Neil Walkinshaw, Ramsay Taylor, and John Derrick. 2016. Inferring extended finite state machine models from software executions. *Empirical Software Engineering* 21, 3 (2016), 811–853.

[17] Tao Xie and David Notkin. 2004. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proc. ICFEM*. 290–305.

[18] Ryota Yamamoto, Norihiro Yoshida, and Hiroaki Takada. 2018. Towards Static Recovery of Micro State Transitions from Legacy Embedded Code. In *Proc. of WASPI 2018*. 1–4.