Towards Static Recovery of Micro State Transitions from Legacy Embedded Code

Ryota Yamamoto
Nagoya University
Japan
muku@ertl.jp

Norihiro Yoshida
Nagoya University
Japan
yoshida@ertl.jp

Hiroaki Takada
Nagoya University
Japan
hiro@ertl.jp

ABSTRACT

During the development of an embedded system, state transition models are frequently used for modeling at several abstraction levels. Unfortunately, specification documents including such model are often lost or not up to date during maintenance/reuse. Based on our experience in industrial collaboration, we present Micro State Transition Table (MSTT) to help developers understanding embedded code based on a fine-grained state transition model. We also discuss the challenges of static recovery of an MSTT.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**;

KEYWORDS

Reverse-Engineering, Code Analysis, State Transition Table

ACM Reference Format:
Ryota Yamamoto, Norihiro Yoshida, and Hiroaki Takada. 2018. Towards Static Recovery of Micro State Transitions from Legacy Embedded Code. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Automated Specification Inference (WASPI '18), November 9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3278177.3278178

1 INTRODUCTION

Recently, it is strongly required to improve not only the efficiency of an embedded system development process but also the reusability of source code for the system since the scale and complexity of the source code have been increased so far [1, 4, 5]. However, when a developer leaves a project or source code documents are unavailable, the left-alone source code can be regarded as legacy code. Main developers in charge of a product often left from their project and only out-dated design documents often exist for the project. Thus, legacy code makes reuse/maintenance more costly. The lack of design documents causes legacy code [6]. It also causes the regression of software maintainability/reusability [6]. Particularly in a development of embedded system development, legacy code is often regarded as a low-maintainability code.

WASPI '18, November 9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-6057-9/18/11…$15.00
https://doi.org/10.1145/3278177.3278178

Typically, embedded systems have a state transition element: an event, a state, a process, and a transition. State transition table is a common form consisting of events, states, processes, and transitions. The table represents state transitions and processes occurring when a target system becomes a state and triggers an event.

According to our experience in industry/university collaboration with Japan Embedded Systems Technology Association[1], many practitioners suggest the following industrial needs:

- Static recovery techniques for extracting state transition models at not only system level but also module level from embedded code
- Common form for representing models that are recovered by the above techniques

As a part of the industry/university collaboration, we defined Micro State Transition Table (MSTT) to help developers understanding embedded code based on a fine-grained state transition model. An MSTT does not treat a function as events such as proposed in [7], instead it treats conditional statements as state transitions or events. We choose a state transition table instead of the state transition model because table style is possible to grasp a combination of all events and states. Therefore, it is easy to find missing elements comparing with other forms of a state transition model. Currently, we are trying to develop a static recovery tool for extracting state transition models a module level from embedded code. In the trial, we found the research challenges of the static recovery.

In this position paper, we present MSTT and then perform a case study using it. After reviewing the related works briefly, we finally discuss the challenges of static recovery of an MSTT.

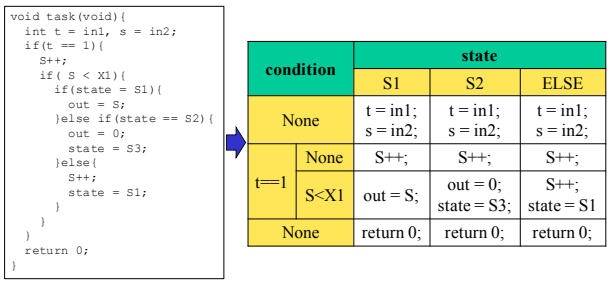
2 MICRO STATE TRANSITION TABLE

In this section, we show the correspondence between Micro State Transition Table (MSTT) and source code, and how to read an MSTT. Moreover, MSTT's advantages are also described in this section. An MSTT treats a conditional statement as a unit instead of function, as defined by Walkinshaw et al.[7, 8], as a state or an events. Therefore an MSTT is more detailed state transition table because it has grained viewpoints. Additionally, a developer can confirm the execution order of events using an MSTT compared with general state transition tables.

2.1 Correspondence with source code

An example of MSTT is shown in Fig.1. First, we describe a correspondence between source code and its MSTT. In an MSTT, each column and row represent a state and an event respectively. The columns represent values that are defined as a state variable chosen

[1] http://www.jasa.or.jp/TOP/data/english/



```
void task(void){
  int t = in1, s = in2;
  if(t == 1){
    S++;
    if( S < X1){
      if(state = S1){
        out = S;
      }else if(state == S2){
        out = 0;
        state = S3;
      }else{
        S++;
        state = S1;
      }
    }
  }
  return 0;
}
```

condition		state		
		S1	S2	ELSE
None		t = in1; s = in2;	t = in1; s = in2;	t = in1; s = in2;
t==1	None	S++;	S++;	S++;
	S<X1	out = S;	out = 0; state = S3;	S++; state = S1
None		return 0;	return 0;	return 0;

Figure 1: Micro State Transition Table Example.

by a developer. Since a state variable represents a system state, it should (1) be used in a conditional statement and be updated in a scope of condition structure including itself, and (2) has a finite value.

If the number of states is enormous, the readability of an MSTT decreases. In Fig.1, the only variable that satisfies the definitions is only *state*. It is an adequate variable to express a state of a system.

Second, each row represents a conditional statement expected in a state variable. In Fig.1, there are events labeled as *None*. The label *None* means the execution has no dependencies on any conditions at any time.

2.2 Representations of MSTT

We describe how to read an MSTT. Generally, a state transition table represents transitions for each unique combination event and state. The table represents one combination of execution process and/or transition in a cell. A next iteration is executed when an event occurs based on a state after transition. A next iteration is executed when an event occurs based on a state after transition.

An MSTT, on the other hand, represents combinations of a state and an event that are not necessarily unique, and the first column to the last column of the table are executed in an iteration. Next, we describe how to read an MSTT in Fig.1. At first, a developer considers the value of condition variable *state* for the first event labeled *None* in Fig.1. The label *None* means if(True) (independent on conditions). In this figure, variable t and s are initialized with no dependencies on *state*. Next, variable s is incremented when event $t == 1$ and *None* occurs. Here, we assume $state = S2$. In this situation, next event $t == 1$ and $S < X1$ are occurred, then state variable *state* are update to $S3$. The final event *None* does not depend on *state*, and execute *return* 0; on state *row* as *state* is $S3$. To sum up, developers read an MSTT up to down, and change reading columns when state variables update.

2.3 Advantages of MSTT

In this section, we describe the advantages of our MSTT compared to related work [7, 8]. Then, we compare the general state transition table and an MSTT.

At first, the MSTT representation has granular viewpoints compared to related work [7, 8] and the granularity has advantages. Unlike related work [7, 8] that define a function as an event unit, an MSTT treats a conditional statement as event unit. Therefore, developers can verify an abstract operation how implemented in the source code, and find potential bugs more easily to use an MSTT. Additionally, our approach can prevent a prejudice caused by the inaccurate function name. When a function name is defined as an event, he/she not be able to comprehend or wrongly comprehend for prejudice if the function name does not show its actual behavior. For example, if there is a function named 'communicate', it is not clear whether it is implemented as an internal communication or an external communication. On the other hand, a developer may infer the kind of communication from system characters, or he/she cannot decide the functionary: whether it is implemented as sender, receiver or both sender and receiver. As a result, to extract a correct model, developers understand details of source code. Therefore, it is confusing for developers to understand source code with extraction from only function name. The main objective of using an MSTT is to understand legacy code, but because of the foresaid reasons, the abstract expression proposed by related work is not suitable for this objective. Alternatively, developers understand the whole of source code behavior hardly because an MSTT causes decreasing of abstractness. In order to improve its readability, developers can read an MSTT with abstract model.

Second, a state transition table can represent all combination of an event and a state. Related work [7, 8] use a state transition diagram instead of a state transition table. Diagrams can represent information as tables, but appearance in the diagrams is more difficult. A table represents all combinations of existing events and states; nevertheless, the diagram represents a behavior not considering whether it is normal or abnormal. Thus, developers may perceive combinations of events and states that cannot be found in the diagram. Additionally, tables have an advantage compared to the diagram. Developers can find a particular combination of events and states clearly using the table. With the table, he/she can find a row and column representing target state and event. Although with diagrams, nodes representing target states, and edges that link nodes up represent target events. Thus, if the diagram is large, it costs a lot only to find the target node.

Next, comparison of existing state transition table and an MSTT is described below. Developers can understand an evaluation sequence of events (an event representing a conditional statement not including a state variable) easily using an MSTT compared to an existing state transition table. For example in Figure 1, developers can understand *None* at the first row is evaluated, then $t == 1$ is evaluated. There are more than one unique combination of an event and a state in an MSTT. On the other hand, although there is a unique combination of an event and a state in an existing state transition table, it cannot represent an event evaluation sequence. It is an advantage of an MSTT because developers may need the sequence to understand the target system. Additionally, there is advantage owing to have more than one unique combination of events and states: to inspect redundant conditional statements. There may be multiple statements representing the same event on a source code. However, the statements can be reduced if the events are redundant. Using events in the MSTT, thus, developers can find redundant events (conditional statements) easily except for pre/post process.

Table 1: Questionnaires

	Source code 1	*	**	F
q_1	Answer combinations of a state and an event when variable spd is increased.	3	1	0
q_2	Answer combinations of a state and an event when variable spd id decreased.	4	0	0
q_3	Assume the state variable is zero and answer all events of including process.	3	0	1
q_4	When event accl == ON occurs, answer all states of including process.	2	1	1
q_5	Describe behavior about this program briefly.	2	1	1
q_6	This program represents a part of a home appliance. What is it?	4	0	0

	Source code 2	**	*	F
q_1	When is cyclic handler 1 handled?	2	1	1
q_2	When is cyclic handler 1 terminated?	4	0	0
q_3	Assume the state variable is two. Answer all events of including process.	3	1	0
q_4	When event stp_b == 1 occurs, answer all states of including process.	4	0	0
q_5	Describe behavior about this program briefly.	3	0	1
q_6	This program represents a part of a home appliance. What is it?	3	0	1

3 CASE STUDY

We performed a case study to confirm the understandability of developers for source code. In it, four subjects (students at Nagoya Univ.) extract two MSTTs. Then the subjects see each MSTT and answer questionnaires about each target source code. At first, we describe about the source code. We prepare two source files as input. Source code 1 is bare metal code presenting a part of automotive controller (110 LoC), and Source code 2 is a microwave application code for real-time operating system (190 LoC). Second, we describe about the experimental procedure. Before the experiment, we do a tutorial of an MSTT modeling procedure to subjects. After that, the subjects performed: (a) extract state variable candidates and (b) choose one state variable and draw an MSTT, and then (c) answer questionnaires (see Table 1). After (b), the experiment conductor check MSTTs and if the MSTT is incorrect (disagreement with the MSTT extracted by the conductor), the conductor advice and make them re-model an MSTT. After that, subjects answer the questionnaires about the MSTT.

Figure 2 is an MSTT for Source code 2 (state variable is *md*). Developers can see "cyclic handler is handled when pushed any buttons except *stp_b* when the system state is 0" from this MSTT. And Table 1 shows the questionnaires and results. In the tables, notation ** means correct answers, notation * means partly correct answers with some missing or incorrectness and F means incorrect answers.

If subjects understand the state, event or combination of a state and an event, he/she can answer questions from q_1 to q_4. In the Table 1, almost all subjects answer correctly. This shows that developers can find state transition elements in source code with an MSTT. Next, if subjects understand the behavior/abstraction of the target source code he/she can answer from q_5 to q_6. A correct answer of q_5 is to describe the source code's behavior briefly and the incorrect answer is to describe it line by line or to describe inexistent behaviors. A correct answer of q_6 is to describe automotive for Source code 1 and microwave for Source code 2. According to these results, the understandability of source code for developers is increased.

4 RELATED WORK

Wasim et al. [9] try to extract state machine models with their concolic approach. Their motivation and ours are close: to help developers program comprehension. Compared to our representation, selecting state variable by a developer is the same point. Each approach introduces a state variable because the readability of the model is quite low owing to the scale and complexity. However, the modeling procedures have a difference, [9] uses a concolic approach and ours is modeled from static information.

Walkinshaw et al. [7] use symbolic execution with a target source code. As a result, their approach generates a finite state machine. In their study, states are defined as a combination of one data and control events. Additionally, a state transition occurs with a function call. Also, Walkinshaw et al. [8] try to extract state transition models applying dynamic execution which generates trace log. The viewpoints of their approach are macroscopic because events and states are defined as a function call. Their macroscopic state transition model is very effective to grasp overview of legacy code. Additionally, there is a possibility that processing with the low-frequency process may be overlooked because of applying dynamic analysis in the study.

5 RESEARCH CHALLENGES

5.1 Conditional statement format

There are conditional statements that are hard to be implemented. The conditional statement for C can be written in various descriptions. Accordingly, it is hard to implement a syntax analysis tool to extract all of the conditional statement formats for C. For example, function calls can be included in conditional statements for C. Hence, there are even cases where comparison operators are not used. In this case, the statement should not be treated as a transition because the statement does not include a variable. However, it should be treated as an event. Moreover, the condition evaluation depends on the execution result of the function, it is necessary to clarify the criteria.

To solve this problem, we have been discussing about an extraction approach using symbolic execution tool KLEE [2] as simple and comprehensive as possible. It may improve the readability of the tool described later.

5.2 Loop Statements

We consider a loop statement as a conditional statement. Essentially, a loop structure should affect state transition table representation. It is desired that target source code has loop structures as state

		md	
		2	![2]
void cyc_1sec()	if(tmp >= 40)	ss = 0; md = 0;	N/A
	else if(tmp >= 80)	N/A	ss = 0; md = 0;

		md	
		0	![0]

void opr()	if(attm_b == 1 && ss	md = 1;
		sta_cyc(C1);
		turn_tlb();
			N/A

	else if(attm_b == 1 &	md = 1;
		sta_cyc(C1);
		turn_tlb();
			N/A

	else if(kt_b == 1)	md = 2;
		temp = gettmp();
		sta_cyc(C1);
		turn_tlb();
			N/A

	else if(stp_b == 1)	N/A	md = 0;
			stp_cyc(C1);
			stp_tlb();

Figure 2: State Transition Table Extracted from Source Code.

transition usually occurs in them. However, functions that are executed asynchronously (e.g. interrupt handler, multi-threading) may represent state transitions even without them. Thus, because loop structures depend on the analysis target, further study is necessary.

Furthermore, there is a case that loop counter variable roles a state variable. For example, a system initialization is executed when the loop counter variable is zero and a system starts calculation when the loop counter variable is one.

5.3 Interruption

In embedded systems, interrupt handler usually exists. The interrupt handler can be called any time except when it is disabled. Hence, unlike a general function, an interrupt handler can be treated as an event. When a developer extracts an MSTT, an interrupt handler should be represented as events without depending on the target function.

5.4 Parallelization and Multitasking

In embedded systems, especially in automotive systems, a large number of electronic control units are used and a manycore environment is frequently used. Additionally, such system with real-time operating system are increasing. Consider a parallel system in an environment where multiple process are executed at the same time. In such case, it is necessary to extract dependencies between tasks and to represent the dependencies in the state transition table.

On the other hand, in case of a multi-task system, the activating condition of handlers (e.g interrupt handler, cyclic handler, alarm handler, and etc.) is treated as an event. Moreover, a running task may be switched when a service call occurs and the task state transits on a real-time operating system. This should be treated as an event, too.

5.5 Readability

Because an MSTT is a state transition table with a small granularity, there is a disadvantage that the scale of the table becomes large. Therefore, it is hard for developer to find whether there is certain state transition or not (developer's concern) using an MSTT.

To improve the readability of an MSTT, we have been discussing about using NuSMV [3]: symbolic model checker. It makes it possible to detect a developer's concern described in temporal logic. NuSMV checks whether it exists or not.

ACKNOWLEDGMENTS

We thank Mr. Lou Yizhi of Nagoya University for proofreading this paper. This work was supported by JSPS KAKENHI Grant Number JP16K16034.

REFERENCES

[1] Berndt Bellay and Harald Gall. 1998. An evaluation of reverse engineering tool capabilities. *Journal of Software Maintenance* 10, 5 (1998), 305–331.
[2] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *Proc. OSDI*. 209–224.
[3] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. NuSMV 2: An opensource tool for symbolic model checking. In *Proc. CAV*. 359–364.
[4] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. 2015. PIE: parser identification in embedded systems. In *Proc. ACSAC*. 251–260.
[5] Benjamin Hummel and Thomas Kinnen. 2015. Incremental Software Quality Analysis for Embedded Systems. In *embedded world Conference 2015*.
[6] M Srinivas, G Ramakrishna, K Rajasekhara Rao, and E Suresh Babu. 2016. Analysis of Legacy System in Software Application Development: A Comparative Survey. *International Journal of Electrical and Computer Engineering* 6, 1 (2016), 292.
[7] Neil Walkinshaw, Kirill Bogdanov, Shaukat Ali, and Mike Holcombe. 2008. Automated discovery of state transitions and their functions in source code. *Software Testing, Verification and Reliability* 18, 2 (2008), 99–121.
[8] 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.
[9] Said Wasim, Jochen Quante, and Rainer Koschke. 2018. Towards Interactive Mining of Understandable State Machine Models from Embedded Software. In *Proc. of MODELSWARD*. 117–128.