# Extracting a Micro State Transition Table Using the KLEE Symbolic Execution Engine

Norihiro Yoshida*, Takahiro Shimizu*, Ryota Yamamoto† and Hiroaki Takada*
*Nagoya University, Japan
yoshida@ertl.jp, s.takahiro915@gmail.com, hiro@ertl.jp
†National Institute of Technology, Tomakomai College, Japan
r-yamamoto@tomakomai-ct.ac.jp

***Abstract*—In this paper, we suggest an approach for extracting fine-grained state transition tables using the KLEE symbolic execution engine to assist developers in understanding the behavior of C source code for embedded systems.**

## I. Background

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 [1], [2]. There is a trend in embedded systems development of repeatedly changing and reusing source code from this background.

When developing embedded systems, as high responsiveness is required from low-performance computers, source code is often written in the C language [3], [4]. In comparison to object-oriented languages, the C language has inexpressiveness 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 due to 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 [5]. 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, even if the module contains complex condition branches, it is possible to promote understanding of that module by comparing it with the MSTT at the time of maintenance or reuse. Compared with general state transition models, it is fine-grained 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 [6], [7], or are methods that target modules written in object-oriented languages [8], [9]. It is not easy to use them to extract an MSTT from a module written in the C language.

## II. Previous approach using TRACER

In our previous work [10], we use symbolic executor TRACER [11] as a tool of statically analyzing complex condition branches and attempt to extract an MSTT from modules written in the C language. This approach automatically extracts an MSTT when the user specifies state transitions from variables within the modules.

During the application of this approach using TRACER, we found that it is often difficult to generate a correct symbolic execution tree in the case of source code with directives (i.e., macros), pointers, and arrays. Therefore, we decided to try to use a state-of-the-art symbolic executor KLEE.

## III. On-going Work Using KLEE

Currently, we are implementing the previously published approach using KLEE [12] instead of TRACER. A primary challenge of replacing TRACER by KLEE in this work is generating a symbolic execution tree [11]. We found the pull request #114[1] entitled *"Add option to dump proc tree to CSV file"* in the KLEE repository[2]. This pull request has not been merged yet[3] but is expected to help generate a symbolic execution tree. Therefore, we chose to use the implementation of KLEE, which corresponds to this pull request. It is available in the repository https://github.com/KennyMacheka/klee which

[1]https://github.com/klee/klee/pull/1141
[2]https://github.com/klee/klee
[3]accessed 2020-05-11

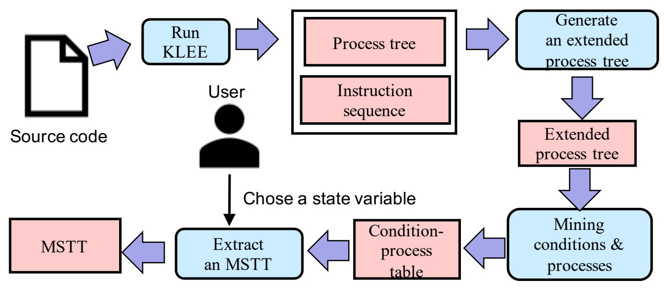


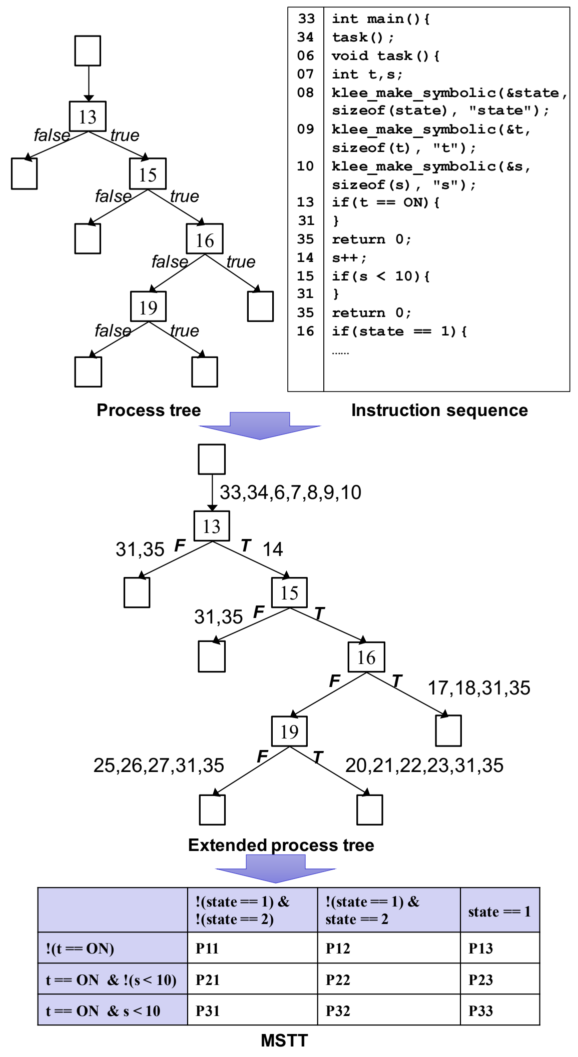
Fig. 1. An overview of the MSTT extraction using KLEE

was forked from the KLEE repository. Hereafter, we call this implementation KLEE.

Figure 1 shows an overview of the extraction of an MSTT from C modules using KLEE. This extraction process is comprised of the following steps:

1) Run KLEE with -search=dfs, -output-exectree and -debug-printf-instructions=src:file options for obtaining a process tree and the corresponding instruction sequence (i.e., depth-first sequence of symbolically executed instructions).
2) Generate an extended process tree by adding the corresponding line number as a label to each edge of the obtained process tree based on the instruction sequence.
3) Generate a condition-process table by extracting pairs of a condition and the corresponding process from the generated extended process tree.
4) Extract an MSTT from the generated condition-process table based on a user-specified state variable.

Figure 2 shows an example of KLEE-based MSTT extraction from a process tree and an instruction sequence. Please note that the detail of each process is abbreviated in the MSTT due to the limited space.



```
33 int main(){
34 task();
06 void task(){
07 int t,s;
08 klee_make_symbolic(&state,
   sizeof(state), "state");
09 klee_make_symbolic(&t,
   sizeof(t), "t");
10 klee_make_symbolic(&s,
   sizeof(s), "s");
13 if(t == ON){
31 }
35 return 0;
14 s++;
15 if(s < 10){
31 }
35 return 0;
16 if(state == 1){
   ......
```

| | !(state == 1) & !(state == 2) | !(state == 1) & state == 2 | state == 1 |
|---|---|---|---|
| !(t == ON) | P11 | P12 | P13 |
| t == ON & !(s < 10) | P21 | P22 | P23 |
| t == ON & s < 10 | P31 | P32 | P33 |

Fig. 2. MSTT extraction from a process tree and an instruction sequence

## Acknowledgment

This work was supported by JST, PRESTO Grant Number JPMJPR21PA, Japan. Also, this work was supported by JSPS KAKENHI Grant Number JP18H04094.

## References

[1] G. Brataas, S. O. Hallsteinsen, R. Rouvoy, and F. Eliassen, "Scalability of decision models for dynamic product lines," in *Proc. of SPLC*, 2007.
[2] K. Nie, T. Yue, S. Ali, L. Zhang, and Z. Fan, "Constraints: The core of supporting automated product configuration of cyber-physical systems," in *Proc. of MODELS*, 2013, pp. 370–387.
[3] D. W. Lewis, *Fundamentals of Embedded Software: Where C and Assembly Meet*, 1st ed. Prentice Hall PTR, 2002.
[4] S. Lee and J. W. Jeon, "Evaluating performance of android platform using native c for embedded systems," in *Proc. of ICCAS*, 2010, pp. 1160–1163.
[5] R. Yamamoto, N. Yoshida, and H. Takada, "Towards static recovery of micro state transitions from legacy embedded code," in *Proc. of WASPI*, 2018, pp. 1–4.
[6] N. Walkinshaw, K. Bogdanov, S. Ali, and M. Holcombe, "Automated discovery of state transitions and their functions in source code," *Software Testing, Verification & Reliability*, vol. 18, no. 2, pp. 99–121, 2008.
[7] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Proc. TACAS*, 2003, pp. 553–568.
[8] P. Tonella and A. Potrich, *Reverse engineering of object oriented code*. Springer, 2005.
[9] T. Sen and R. Mall, "Extracting finite state representation of java programs," *Software & Systems Modeling*, vol. 15, no. 2, pp. 497–511, 2016.
[10] T. Shimizu, N. Yoshida, R. Yamamoto, and H. Takada, "Symbolic execution-based approach to extracting a micro state transition table," in *Proc. TAV-CPS/IoT*, 2019, p. 1–6.
[11] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, "TRACER: A symbolic execution tool for verification," in *Proc. of CAV*, 2012, pp. 758–766.
[12] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of OSDI*, 2008, pp. 209–224.