Regular Paper

Executable Counterexample for Java Model Checker

Chellet Marwan Bernard Hassan, Shinpei Ogata, and Kozo Okano

Shinshu University, Nagano, Japan
19w2074f@shinshu-u.ac.jp,{okano, ogata}@cs.shinshu-u.ac.jp

Abstract - Testing is a mandatory task in the development of software. Effectively, ensuring the reliability of software is a major point of work for a developer. Even after software has been tested, it is not uncommon to encounter bugs that can have consequences on the operation of a system. To improve reliability, developers have started using verification methods in addition to testing. The basic method of model checking verifies whether or not the software or a given fragment of program code satisfies properties that can lead to bugs if they are violated. Model checking has already proved to be an effective tool for verifying software, but there are still some inconveniences in using these techniques. When a software model checker finds a violated property, it will signal it by giving a counterexample as output. This usually involves tracing the path from the initial state to the state that reveals the violated property. The counterexamples are usually given in text format and are not always simple to understand as all of the steps of the process are given with a sequence of machine instructions. This paper aims to improve a Java model checker by translating counterexamples into executable Java code that is more understandable for a developer.

Keywords: Model checking, counterexample, Java, simulation, verification

1 INTRODUCTION

Software testing is one of the best ways to get good software reliability. It is practically impossible to use software without testing it beforehand but with all testing methods, it is not rare to miss some cases which can result in bugs during operation. Even a small bug can have real consequences so software safety becomes a priority. Full coverage of an application is hard to achieve even with advanced testing methods. Such methods are also limited because they use the system itself to find errors which makes it difficult to have an overview of the entire program.

To fill the gaps of testing methods, developers also use a verification method called "model checking." [1][2] This method does not use the system itself, but rather an abstract model of it which is represented by states and transitions. It permits developers to automatically try different scenarios and cases and provides a different field of action than the testing method. Model checking typically works in addition to testing rather than in place of it. Even if software model checking [3] is already recognized as a good way to find errors in software, it is still an underused tool because of its complexity of usability.

One big disadvantage of model checking is the difficulty in treating counterexamples given for a violated property. Unlike the testing method in which such output is usually given in an understandable way by using code, model checking counterexamples are usually presented as sequences of transitions that lead to the error state from the initial state. For software model checking which uses program code as input instead of state machines, the counterexamples are usually represented in a sequence of low-level machine codes due to limitations of the current software model checking tools. For example, Java model checking tools usually take Java Bytecodes instead of Java program code which makes it difficult for people who are not experts in machine language to understand what is being expressed and can be a problem when attempting to localize the bug.

It has been a long time since researchers have tried to study ways to help people to understand the contents of a counterexample [4]. Now we have new technologies and new ways of using counterexamples emerging such as Visual-Studio plugins [5]. One of the most interesting methods reported was a way to make an executable counterexample [6] instead of a complicated list of states in machine language. Our research was especially inspired by the work of Rocha Herbert, Barreto Raimundo, Cordeiro Lucas, and Neto Arilo and their way of creating an executable counterexample for ANSI-C programs [7]. Our paper introduces a method for creating an executable counterexample for Java programs with the Java model checker JBMC [8]. This is done by extracting the necessary data from a counterexample and translating it into lines of Java code to be used in the final version of source code.

This paper is organized as follows: Section 2 introduces the basic concepts and principles of model checking, the model checker we used, and the part of model checking on which we focused. Section 3 explains our proposed method. Section 4 describes our experiments and results. Section 5 presents our observations during the creation of the proposed method. Section 6 describes our future work to improve this proposed method. Section 7 provides all of the existing works related to this proposed method and finally, Section 8 summarizes this paper.

2 MODEL CHECKING

In this section, we introduce the principle of model checking and how it works. After that, we explain our Java model checker choice. The last part focuses on the generation of counterexamples which is the central part of this paper.

2.1 Principle of Model Checking

Model checking has already proven itself as a useful verification method for software [9][10]. The goal of this method is to analyze a system translated into an abstract model to find violated properties. The model is represented as a transition system in the form of an oriented graph: a vertex represents a state of the system and each arc represents a transition, i.e., a possible evolution of the system from a given state to another state. Each state of the oriented graph is labeled by a set of atomic propositions true at this point of execution. The property to check is written by a temporal logic formula. These logical expressions are defined on a set of atomic propositions P or proposition variables. These atomic propositions are combined with a number of logical connectors, including the usual connectors: and, or, not, implication, as well as other operators which are called modalities.

This method works in three phases. First is the modeling phase which takes a real system and translates it into an abstract model. The system becomes a set of states and transitions. States give information about the program such as its variable values. Transitions are used to describe how the system works. The model checker also formalizes the properties to be verified. In the second phase, the model checker evaluates the possible paths of the abstract model to find a violated property. In this work, we are mainly interested in the final phase of analysis. There are three different possible outputs from the second phase. The first possibility is when a property is violated and the model checker traces the path to give a counterexample. The second type of output is when all properties are verified and not violated in which case the model checker reports that no bugs were found in the program. The third type occurs when the software is too complex and the abstract model is too large in which case the model checker runs out of memory in handle it. The model checking principle [1] and its three phases are depicted in Fig. 1.

The model checker we use in this work is a bounded one. The bounded model checking method [11] is a way to use model checking by traversing a finite-state machine with a fixed number of steps k and checking if a property is violated in this bound. The larger the value of k, the better the chances of finding a violation. The principal goal is to be faster and more efficient by giving a limit of the number of steps. This method is also associated with Boolean satisfiability problem (SAT) solvers [12] which, given a propositional logic formula, can determine whether or not there is an assignment of propositional variables that makes a formula true.

2.2 Software Model Checking

This paper is focused on software model checking which is slightly different from general model checking. Effectively the target of software model checking is program code that cannot be transformed into a finite-state model. Instead, code is transformed into transitions written in machine languages. The bounded part is used for program loops where the software model checker unrolls loops with a bound k. Figure 2 is a short explanation of how this works in a while loop with

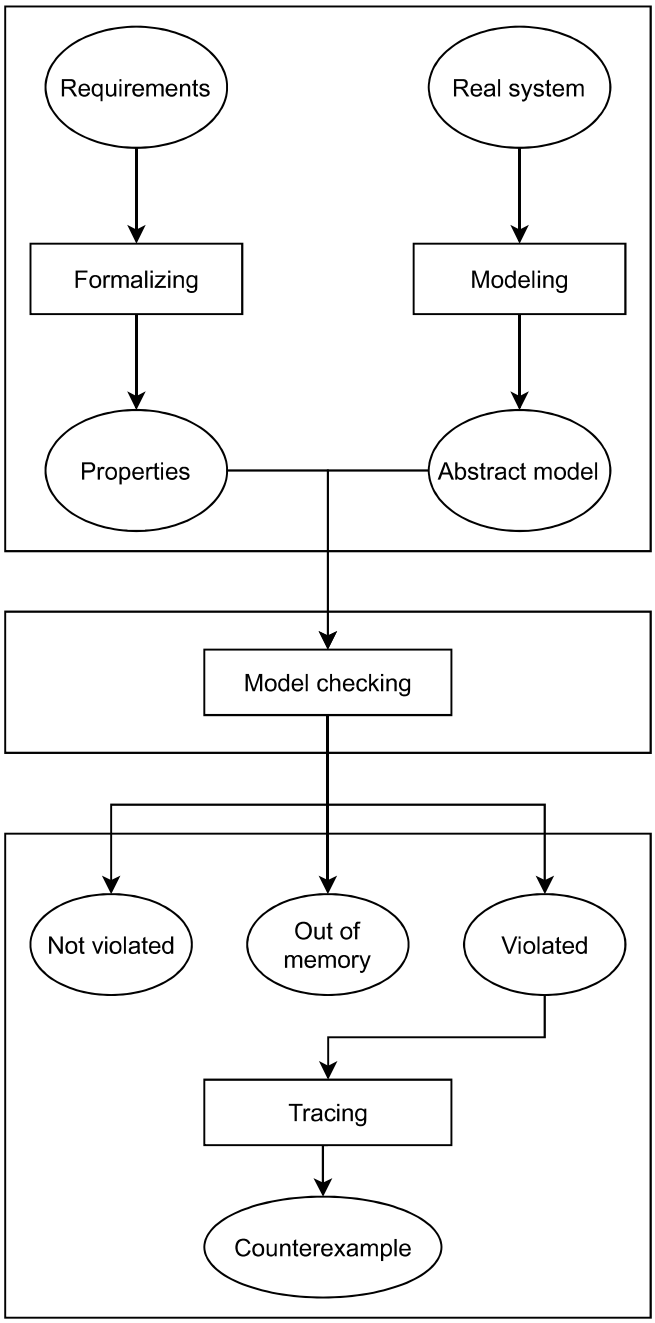

Figure 1: Principle of model checking

a bound of $k = 3$. The specification of a property in software model checking is also different. It is not represented by a temporal logic formula but by an assertion.

2.3 Model Checking with JBMC

JBMC is a bounded Java model checker developed in C++ and is based on the C model checker called CBMC [13]. It is one of the most efficient model checkers for Java programs and takes a Java Bytecode program as input. To be more efficient it also uses an abstract representation of the standard Java libraries called Java Operational Model. This representation was made to simplify the standard Java libraries and speed up the process of model checking. The core of the checker is managed by the CPROVER framework[14]. It is a

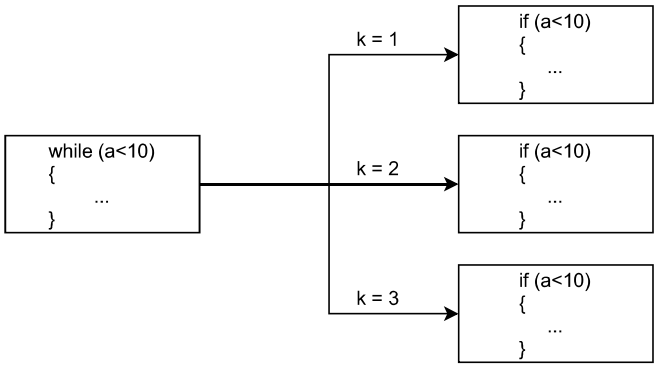

Figure 2: Example of an unrolled loop with a bound of $k = 3$

bounded model checker so the last input will be the number of bounds k that we want to do.

JBMC works in four steps. The first step is to parse the Java Bytecode into a parse tree which corresponds to a translation into a finite-state model. The second step translates the parse tree into a CPROVER control-flow graph representation which is called a GOTO program. The goal of this step is to simplify the Java Bytecode representation and make it easier to analyze with the CPROVER framework. In the third step the checker analyzes the model properties. The last step uses the SAT solver to determine whether or not a property is violated for a given bound k, and returns a counterexample if it is the case.

As mentioned above, JBMC uses the CPROVER framework to produce counterexamples. In addition, to verify assertions made by the user, it also covers such properties as Null Pointer Exception, Division by 0, Index out of bounds, etc. Figure 3 illustrates a simple example of a Null Pointer Exception error which can be found by JBMC.

This example is a simple program for printing the values of an array of strings. The variable "v" takes a random value between 1 and 2. If the value is 1, then only the first value of the array "tab" will be printed, and if the value is 2, then it will also print the second value which is null, and throw a Null Pointer Exception.

```
import java.util.Random;

public class NullTest
{
 public static void main(String [] args)
 {
  String[] tab = {"a",null};
  Random r = new Random();
  int v = r.nextInt((2-1)+1)+1;
  for (int i=0;i<v;i++)
  {
   System.out.println(tab[i].length());
  }
 }
}
```

Figure 3: Code with a Null Pointer Exception error

```
State 84 function java::java.lang.System.<clinit>:()V thread 0
------------------------------------------------------
 dynamic_object16={ .@class_identifier="java::java.io.PrintStream" } ({ ? })

State 91 file NullTest.java function NullTest.main(java.lang.String[]) line 12 thread 0
------------------------------------------------------
 this=&a
(00000000 00010001 00000000 00000000 00000000 00000000 00000000 00000000)

State 99 file NullTest.java function NullTest.main(java.lang.String[]) line 12 thread 0
------------------------------------------------------
 this=&dynamic_object16
 (00000000 00010011 00000000 00000000 00000000 00000000 00000000 00000000)

State 100 file NullTest.java function NullTest.main(java.lang.String[]) line 12 thread 0
------------------------------------------------------
 stub_ignored_arg1=1 (00000000 00000000 00000000 00000001)

State 102 file NullTest.java function NullTest.main(java.lang.String[]) line 10 thread 0
------------------------------------------------------
 anonlocal::4i=1 (00000000 00000000 00000000 00000001)

Violated property:
 file NullTest.java function NullTest.main(java.lang.String[]) line 12 thread 0
 Null pointer check
 !((struct java.lang.String *)((struct java::array[reference] *)anonlocal::1a)-
>data[anonlocal::4i] == null)
```

Figure 4: Trace of counterexample from JBMC

2.4 Counterexample

When the model checker finds a violated property in a program, a counterexample is given. This output helps developers localize the bug by giving the behavior which leads to the bug using such information as variable values. As mentioned before, JBMC uses CPROVER which produces counterexamples. The trace example in Fig. 4 is the output given by the model checker after the verification of the code shown in Fig. 3.

From the output, we can easily see which property is violated, but it is difficult to understand how the model checker found the violation and which path we should take to get the same behavior. Every state in the counterexample represents one instruction such as a change of variable value, but this instruction is given by the CPROVER framework which was initially made for C and C++ programs. The problem is that it is difficult to understand these instructions and see what the Java equivalence is.

3 PROPOSED METHOD

This section describes the method proposed in this work for translating counterexamples given by JBMC into Java code. Figure 5 shows the process of the whole system. Step 1 is simply a transformation of Java code into Java Bytecode before sending it to the model checker. Step 2 is the running of the model checker, and if a property is violated we go to Step 3. Step 3 is the treatment of the counterexample and this paper focuses on this work. The last step is the report of the output as a Java code program after the translation of the counterexample.

3.1 Method Objective

The main objective of our proposed method is to support developers who want to use model checking for verifying their software. Since it is an experimental method, the principal goal is to be sure that it is possible to get easily un-

Table 1: Experiment results

	Execution Time	New Assertion	New Instructions	Fusion
Null Pointer	1.03s	OK	OK	OK
Index Out of Bound	0.50s	OK	OK	OK
Division By 0	0.95s	OK	OK	OK
User Assertion	0.88s	NONE	OK	OK

derstandable code from a counterexample produced by the JBMC model checker. To achieve this goal, we take data from a counterexample and put it into the source code in a way that will be clear to the developer who wrote the source code. The main objective will be to use simple code and to analyze it with simple classes. It also has to be adaptive so that we can extend it for more complex software in the future.

3.2 Method Contribution

The contributions of our proposed method are to (1) provide a concrete method to build program code that explicitly shows the source of bugs; and (2) show the effectiveness of the proposed method through examples.

3.3 Step 1: Analysis of Counterexamples

The method proposed in this work is based on a transformation algorithm. The first part adds all new instructions resulting from the counterexample such as variable changes into the source code. The most important task is to understand what property was violated and where it occurred. The last lines of the counterexample give us the data related to the violated property. To help the user find the bug, we create a new line of Java code from this data with an assertion which will fail. Next we find and translate every state of the counterexample into lines of code. More than one state can have the same corresponding line number so in this case, we check if one of our newly created lines corresponds to this line number and update it instead of creating a new line.

3.4 Step 2: Transformation of Bytecode Source Files into Java

The second step of our proposed method is to transform the source file which is in Bytecode into Java code so that it can be used later for debugging. To do this, we use a Java library that transforms bytecode into Java. There are some good libraries such as Procyon, but for this work we chose CFR which does not optimize the code by techniques such as deleting dead code or using variable propagation. This will permit the user to find the code in its original form before the compilation in Bytecode and make it easier to localize the error.

3.5 Step 3: Addition of New Java Code Lines into the New Source File

The last step of the procedure is the fusion of the two previous ones. The goal is to produce Java code that reproduces the behavior of the software at the point where the model checker found the violation of a property. To accomplish this we read the new source file line by line and check to see if there is a new instruction that needs to be added from the counterexample. If we have a new instruction, we delete the old line and replace it with the corresponding one created earlier in Step 1. The variable names from the source file and the counterexample are different, so we have to be sure to preserve the original recognizable variable names and only change their values according to the information in the counterexample rather than simply replacing the whole line. When we reach the line of the violated property, we insert a new line above it with the new assertion created. Figure 6 represents the final result of applying the proposed method to the example in Fig. 3.

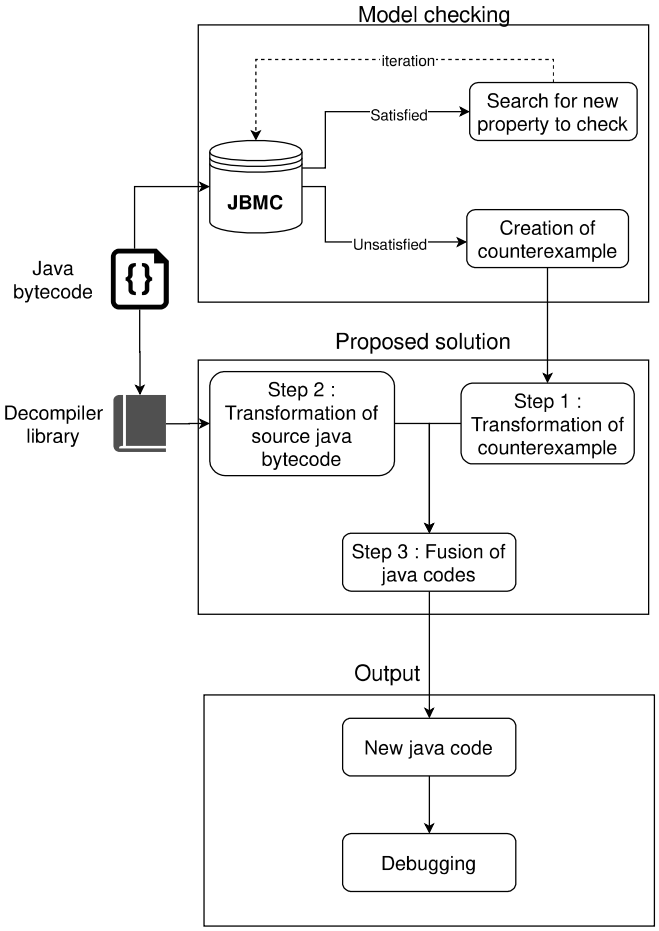

Figure 5: Proposed method process

```
import java.util.Random;

public class NullTest
{
  public static void main(String [] args
      )
  {
    String[] tab = {"a",null};
    Random r = new Random();
    int v = 2;
    for (int i=0;i<v;i++)
    {
      assert (tab[1] != null);
      System.out.println(tab[i].length()
          );
    }
  }
}
```

Figure 6: New code generated by the proposed method

4 EXPERIMENTS AND RESULTS

To check the efficiency of the proposed method, we tested it with a number of simple programs with different types of errors for which JBMC can find a counterexample. We made one program per error for Null Pointer Exception, Index Out Of Bounds Exception, Division By Zero Exception, and a user assertion. For this experiment we chose a bound of k = 5. Table 1 shows the experiment results for each program step by step. Figures 7 to 12 show all source codes and generated codes from the experiment data.

All of the cases were successful and this experiment shows that it is possible to translate a counterexample into Java code. New assertions and the new values of variables are useful in deriving from where an error originates. During this experiment, we realized that it is sometimes difficult to see where the code has changed so it would be a good idea to add comments before new lines of code generated in the output.

```
import java.util.Random;

public class IndexTest
{
  public static void main(String [] args
      )
  {
    int[] test = {1};
    Random r = new Random();
    int v = r.nextInt((2-1)+1)+1;
    for (int i=0;i<v;i++)
    {
      System.out.println(test[i]);
    }
  }
}
```

Figure 7: Source code of an Index Out of Bounds Exception

```
import java.util.Random;

public class IndexTest
{
  public static void main(String [] args
      )
  {
    int[] test = {1};
    Random r = new Random();
    int v = 2;
    for (int i=0;i<v;i++)
    {
      assert (test.length >1)
      System.out.println(test[i]);
    }
  }
}
```

Figure 8: New generated code of an Index Out of Bounds Exception

```
import java.util.Random;

public class DivisionTest
{
  public static void main(String [] args
      )
  {
    int[] test = {1,0};
    Random r = new Random();
    int v = r.nextInt((2-1)+1)+1;
    for (int i=0;i<v;i++)
    {
      System.out.println(10 / test[i]);
    }
  }
}
```

Figure 9: Source code of a Division by 0 Exception

This experiment was done with very simple programs, but it is a good start for the understanding of counterexamples. One Java code is cut into multiple instructions in a counterexample. Even the translation of CPROVER into Java code is complicated. Variable names are also not the same as the source file output. Without this new processing part, the fusion between new Java code instructions and the decompiled source Java Bytecode cannot be done. The next step will be to create an algorithm for finding and sorting all related instructions of a variable from the counterexample within more complex programs.

5 DISCUSSION

Even with our proposed method, the understanding of counterexamples from the Java model checker still seems to

```
import java.util.Random;

public class DivisionTest
{
  public static void main(String [] args
      )
  {
    int[] test = {1,0};
    Random r = new Random();
    int v = 2;
    for (int i=0;i<v;i++)
    {
      assert (test [1]!= 0)
      System.out.println(10 / test[i]);
    }
  }
}
```

Figure 10: New generated code of a Division by 0 Exception

```
import java.util.Random;

public class AssertTest
{
  public static void main(String [] args
      )
  {
    int[] test = {10,15};
    Random r = new Random();
    int v = r.nextInt((2-1)+1)+1;
    assert test[v] ==10;
  }
}
```

Figure 11: Source code of a user assertion

```
import java.util.Random;

public class AssertTest
{
  public static void main(String [] args
      )
  {
    int[] test = {10,15};
    Random r = new Random();
    int v = 2;
    assert test[v] ==10;
  }
}
```

Figure 12: New generated code of a user assertion

be difficult, but it has opened a way for researchers to think about model checking in Java and how to improve it. During this research, we saw that Java model checking does not have many reported solutions as compared to C model checking, but it is still one of the most used languages for software development and requires further studies.

The best future direction will be to have a model checker such as JBMC give developers instructions directly in Java language which can be directly used inside Java code. This step of making counterexamples more understandable is an important one to make model checkers more convenient and useful.

6 FUTURE WORK

As explained above, the method proposed in this paper works with small programs, but it has to be more efficient for use in larger production software. The next step will be to make this method more adaptive to easily add unexpected classes and cases. After that, we can add an implementation for the most used classes including List, Map, etc. This method also has to be adapted for translating software with more than one class which uses a lot of different files. To make the newly generated code more useful, we can directly generate code in a JUnit test which can be a more efficient way to find bugs.

Another innovative way to improve this solution can be to use AI techniques that can directly predict the kind of property violated and suggest ways to resolve it. Employing deep learning by using neural networks can be interesting, but to use this technique we need a large amount of dataset samples before starting.

7 RELATED WORK

The work of improving counterexample generation in model checking is not new ([6], [8], [15]–[19]). One of the first approaches for generating executable counterexamples was made by Dirk Beyer [20]. This paper was inspired by the work of [7] in which a counterexample simplification is done for ANSI-C software by using the ESBMC model checker which also uses the CPROVER framework. All of the understanding of model checking was based on the book Principles of Model Checking [1].

8 CONCLUSION

To conclude, the method proposed in this work was created to help Java developers who are not experts in verification methods to more easily use model checking techniques. It was tested with simple and small programs and has to improve its efficiency to be used with complex software. The new method can still be improved and can open doors for future work such as using AI techniques to make steps in the process automatic and efficient. Model checking, especially for Java programs, is still often underused because of the complexity of use. The method presented in this work can be the beginning for making model checking a common tool for software developers.

ACKNOWLEDGEMENT

The research is being partially conducted as Grant-in-Aid for Scientific Research A (18H04094) and C (21K11826).

REFERENCES

[1] C. Baier and J-P. Katoen: "Principles of Model Checking," MIT Press (2008).

[2] E. M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith: "Model Checking, second edition," MIT Press. (2008).

[3] R. Jhala and R. Majumdar, : "Software model checking," In: ACM Computing Surveys, Vol. 41, No. 4 pp.1-54 (2009).

[4] A. Groce, D. Kroening, and F. Lerda: "Understanding counterexamples with explain," In : International Conference on Computer Aided Verification. CAV 2004, Lecture Notes in Computer Science, Vol. 3114, pp.453-456 (2004).

[5] M. N. Seghir and D. Kroening: "A visual studio plug-in for CProver," In : 2013 3rd International Workshop on Developing Tools as Plug-Ins, TOPI, pp.43-48 (2013).

[6] J. Gennari, A. Gurfinkel, T. Kahsai, J. A. Navas, and E. J. Schwartz: "Executable counterexamples in software model checking," In : Working Conference on Verified Software: Theories, Tools, and Experiments, pp.17-37 (2018).

[7] H. Rocha, R. Barreto, L. Cordeiro, and A. Neto: "Understanding Programming Bugs in ANSI-C Software Using Bounded Model Checking Counter-Examples," In: International Conference on Integrated Formal Methods, IFM 2012, pp.128-142 (2012).

[8] L. Cordeiro, D. Kroening, and P. Schrammel: "JBMC: Bounded Model Checking for Java Bytecode," In: Beyer D., Huisman M., Kordon F., Steffen B. (eds) Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2019, Lecture Notes in Computer Science, Vol. 11429, pp.219-223 (2019).

[9] D. Beyer: "Software verification with validation of results - (report on SV-COMP 2017)," In : Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference. TACAS 2017, Lecture Notes in Computer Science, Vol. 10206, pp.331-349 (2017).

[10] D. Beyer and T. Lemberger: "Software Verification: Testing vs. Model Checking A Comparative Evaluation of the State of the Art," In : 13th International Haifa Verification Conference, HVC 2017, pp.99-114 (2017).

[11] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu: "Bounded model checking," In : Advances in computers, Vol. 58, No. 11, pp.117-148 (2003).

[12] O. Strichman: "Tuning SAT Checkers for Bounded Model Checking," In : Computer Aided Verification, CAV 2000, Lecture Notes in Computer Science, Vol. 1865, pp.480-494 (2000).

[13] E. M. Clarke, D. Kroening, and F. Lerda: "A tool for checking ANSI-c programs," In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2004, Lecture Notes in Computer Science, Vol. 2988, pp.168-176 (2004).

[14] D. Kroening and E.M. Clarke: "The CPROVER User Manual," https://www.cprover.org/cbmc/doc/manual.pdf (2001) (06 July 2021 accessed).

[15] N. Shankar and M. Sorea: "Counterexample-driven model checking," In : Technical Report SRI-CSL-03-04, SRI International Computer Science Laboratory (2003).

[16] T. Ball, M. Naik, and S. Rajamani: "From Symptom to Cause: Localizing Errors in Counterexample Traces," In : Conference Record of the Annual ACM Symposium on Principles of Programming Languages, pp.97-105 (2003).

[17] P. Müller and J. N. Ruskiewicz: "Using Debuggers to Understand Failed Verification Attempts," In : Formal Methods - 17th International Symposium on Formal Methods, FM 2011, pp.73–87 (2011).

[18] D. Kroening, A. Groce, and E. M. Clarke: "Counterexample Guided Abstraction Refinement Via Program Execution," In : Formal Methods and Software Engineering, ICFEM 2004, Lecture Notes in Computer Science, Vol. 3308, pp.224-238 (2004).

[19] K. Rustan, M. Leino, T. Millstein, and J. B. Saxe: "Generating error traces from verification-condition counterexamples," In : Science of Computer Programming, Vol. 55, pp.209-226 (2005).

[20] D. Bayer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar: "Generating Tests from Counterexamples," In : Proceedings of the 26th International Conference on Software Engineering, ICSE 04, IEEE Computer Society, pp.326–335 (2004).

(Received November 4, 2021)
(Accepted June 9, 2021)

Marwan Bernard Hassan Chellet received his associate degree in computer science from La Rochelle Institute of Technology, France in 2017. One year after, he received his BE in computer science from La Rochelle University, France. Currently, he is a second year student in the master's program at Shinshu Uniersity, Japan majoring in computer science. His current research interests are in verification methods and software model checking.

Shinpei Ogata is an Associate Professor of the Graduate School of Science and Technology in Shinshu University, Japan. He received a PhD from Shibaura Institute of Technology, Japan in 2012. His current research interests include model-driven engineering for information system development. He is a member of IEEE, ACM, IEICE, and IPSJ.

Kozo Okano received his BE, ME, and PhD degrees in Information and Computer Sciences from Osaka University in 1990, 1992, and 1995, respectively. From 2002 to 2015, he was an Associate Professor at the Graduate School of Information Science and Technology of Osaka University. In 2002 and 2003, he was a visiting researcher at the Department of Computer Science of the University of Kent in Canterbury, and a visiting lecturer at the School of Computer Science of the University of Birmingham, respectively. Since 2020, he has been a Professor at the Department of Electrical and Computer Engineering, Shinshu University. His current research interests include formal methods for software and information system design. He is a member of IEEE, IEICE, and IPSJ.