Regular Paper

Proposal and Evaluation for A Method to Verify Equivalence of Specifications of C and Java Functions with Recursive Data Structures by SAW: Case Studies of Linear Structures and Binary Trees

Rin Karashima[†], Kozo Okano[‡], Shinpei Ogata[‡], Satoshi Harauchi[§], and Toshifusa Sekizawa*

[†]Faculty of Engineering, Shinshu University, Japan
[§] Mitsubishi Electric Corporation Advanced Technology R&D Center
* Faculty of Informatics, Nihon University, Japan
19w2036c@shinshu-u.ac.jp[†]
{okano, ogata}@cs.shinshu-u.ac.jp[‡]
Harauchi.Satoshi@bc.MitsubishiElectric.co.jp[§]
sekizawa@cs.ce.nihon-u.ac.jp*

Abstract - A process to improve the internal structure of a software system while keeping its external behavior is called refactoring. When performing refactoring, programmers need to verify equivalence of the new and old programs by checking that the functions of both satisfy the given specifications to avoid unexpected bugs. Generally, programmers perform unit testing by using test cases like JUnit, but it has the problems of lack of completeness and high time costs. In the Software Analysis Workbench (SAW) verification tool, programmers use the SAWScript scripting language to define verification properties that target functions must satisfy and formal verification is performed by using SMT solvers and symbolic execution. With the symbolic execution of SAW, programmers can perform unit testing completely and effectively without test cases.

This paper proposes a method for verifying the specifications of target functions by generating a helper function and calling a target function as its return value. By using this method, programmers can define the values of class objects that the target function receives in a helper function and make SAW verification possible. As a case study, we performed SAW verification for Java functions that receive recursive class objects as arguments and showed that property verification and equivalence checking is actually possible.

Keywords: SAW, verification, equivalence, refactoring, unit test, recursive data structures

1 INTRODUCTION

Both Java and C programs are widely used for modern information systems. Sometimes, changes of the working environment force engineers to develop new software code with the same functions as the old codes. For example, C programs working in an old environment are sometimes changed to new Java programs that must work in a new environment. In these situations, programmers have to ensure that the revised Java programs preserve the behavior of the old C versions.

As another example, a programmer might have to develop new C programs for an embedded system, where the CPU and memory resources are limited, based on an existing Java program which runs in a newer environment with rich resources. In such a case, programmers have to ensure that the revised C programs preserve the behavior of the old Java programs. The process of improving the internal structure of a software system while maintaining its external behavior is called refactoring [1].

Usually, regression testing is performed to check whether different versions of a program have the same behavior. In the above situation, however, we usually cannot use the same regression test-suites because they use different programming languages. Software Analysis Workbench (SAW) [2], [3] provides the ability to verify programs written in C and Java by symbolic execution.

Formal approach techniques might help in such a situation. These techniques will find potential bugs by checking the conformance to program specifications with adequate efficiency. We call this kind of verification formal conformance verification (FCV). Recent tools, however, do not fully support programs dealing with dynamic data structures especially recursive data structures. For example, our previous research [4] uncovered the possibility of FCV for C programs with recursive data structures.

In this work, we attempt to resolve this problem for Java programs by using the idea of bounded model checking. However, for Java programs, we do not know the possibility of applying FCV. The reasons are summarized as follows.

- SAW retrieves information for verification from the byte codes of C or Java and the difference of the codes affects the checking algorithm.
- Java is based on the Java Virtual Machine while C has no reference machine to interpret.

In order to avoid the halting problem (e.g., if verification is performed without defining the end of a linear list structure, the verification may fail without completing the recursion and a state explosion could occur), our proposed method is based on the bounded model verification technique [5], [6]. In this work, we also perform experimental evaluation using SAW, a recent formal verification tool.

From the experimental results, we find that it is partially possible to perform verification of functions dealing with recursive data structures in Java. This observation supports the possibility of using FCV between C and Java with recursive data structures.

The rest of this paper is organized as follows. Section 2 gives the preliminaries and Section 3 describes the proposed method. Sections 4 and 5 describe the experimental evaluation and results. Section 6 discusses the results. Finally, Section 7 summarizes this paper.

2 PRELIMINARIES

2.1 Equivalence Between Two Functions

Refactoring is defined as the process of improving the internal structure of a software system while keeping its external behavior [1]. Thus, when we perform refactoring on a program function, we need to check that the external behaviors of the new and old functions are equivalent.

In this research, we defined external behaviors as the inputs and outputs of program functions. When two functions f and g are equivalent, expression (1) holds.

$$\forall x \in A : f(x) = g(x) \tag{1}$$

We say these functions are equivalent for any x that satisfies A, where A specifies the range and type of the variable x. For example, if x belongs to an integer type, then $A = \mathbb{Z}_{32}$ holds, where $\mathbb{Z}_{32}$ represents the set of signed integers with 32-bit width. In this paper, we use the following notations to show the integer types $\mathbb{Z}_{16}$, $\mathbb{Z}_{32}$, $\mathbb{Z}_{64}$, $\mathbb{Z}_{8^u}$, $\mathbb{Z}_{16^u}$, $\mathbb{Z}_{32^u}$, and $\mathbb{Z}_{64^u}$ which correspond to the signed integers of 16-, 32-, and 64-bit width, and the unsigned integers of 8-, 16-, 32-, and 64-bit width, respectively.

In this work, we consider inputs and outputs as the requirements of equivalence and do not consider program size, algorithm, execution time, type of programming language, etc.

The parameter x can be easily extended to a parameter vector with the same signatures. Here, a signature is a list of types corresponding to each element of the parameters.

The programming languages for f and g are not fixed, but in this paper, we use the case where f is implemented in Java and g is implemented in C.

2.2 SAW

The unit testing framework xUnit[7][8] is the most popular method for verifying the specifications of a program. However, verifying programs by using test cases has the difficuly of having to create all possible cases and the time cost problem of generation. Checking specifications of a program automatically is an important theme and there exists much research about it especially concerning SAT and SMT solvers [9]-[11].

The Software Analysis Workbench (SAW) is an open-source formal verification tool developed by Galois [3]. The SAW tool automatically generates a formal model from the byte code of a target function and by using symbolic execution, it can verify whether the model satisfies a verification property for any set of inputs and finds a counterexample if it does not.

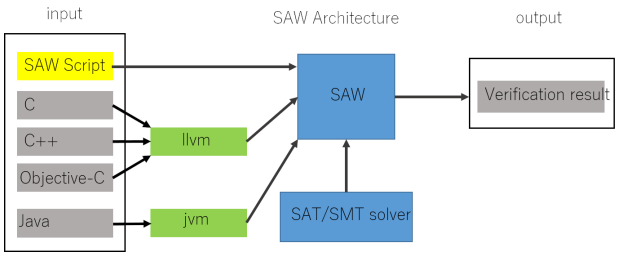

Figure 1: SAW Architecture

Users can define verification properties by using the special SAWScript scripting language which we explain in the next section.

The SAW architecture is shown in Fig. 1. SAW can analyze Low Level Virtual Machine (LLVM) [12], Java Virtual Machine (JVM) [13] byte code, and Cryptol [14], [15]. High level languages like C or Java must be compiled before performing SAW verification.

2.3 SAWScript

According to [2], "the process of model generation and transformation in SAW, and the interaction with third-party proof tools, is coordinated by a scripting language called SAWScript" and users can specify a .saw file to control or command SAW execution, e.g., extract target functions from .bc or .class files, define verification properties, generate SMT expressions and prove them by solvers.

We explain the SAW verification procedure with a simple example. We prove the function `oldJavafunc` in Listing 1 returns $2x$ when it receives x as any integer type. The SAWScript for this verification is shown in Listing 2. First, we describe explicitly the verification properties and which SMT solver to use in the JavaSetup block. In this case, the java_var command in the second line generates integer type symbolic inputs to correspond to the Java variable x as $java\_input$. The java_return command in the third line defines the expected return value of the target function as $2java\_input$. The java_verify_tactic command in the fourth line specifies the SMT solver to use. Next, the load .class file operation is indicated by the java_load_class command in the sixth line as $java\_mul$. Finally, we verify whether or not `oldJavafunc` of the Mul class satisfies the verification properties by the java_verify command in the seventh line. If the verification is valid, z3 proves that `oldJavafunc` the verification property described in the JavaSetup block and `newJavafunc` fulfill expression (2). Thus, `oldJavafunc` always returns $2 * x$ for any input x when x is an integer type.

$$\begin{aligned} &\forall java\_input : oldJavafunc(java\_input) \\ &\quad = 2java\_input \\ &\wedge java\_input \in \mathbb{Z}_{32}, java\_input = x \\ &\vdash \forall x \in \mathbb{Z}_{32} : oldJavafunc(x) = 2x \end{aligned} \tag{2}$$

Listing 1: Java Mul Class

```
class Mul{
  int oldJavafunc(int x){
    return x + x;
  }
}
```

Listing 2: SAWScript for Java Mul Class

```
let java_spec : JavaSetup () = do {
    java_input <- java_var "x" java_int;
    java_return {{java_input*2}};
    java_verify_tactic z3;
};
java_mul <- java_load_class "Mul";
java_verify java_mul "oldJavafunc" [] java_spec
    ;
```

2.4 SAT/SMT Solvers

A SATisfiability problem (SAT) searches for all combinations of boolean variables that make a propositional formula true [16]. In particular, programs that solve CNF propositional formulas automatically are called SAT solvers.

We explain the process of proving propositions using SAT solvers. If a propositional statement is a tautology, it is satisfiable for any status of its boolean variables. That is, the negation of the propositional logic is UNSAT for any combination of its boolean variables. Thus, proving that the negation of a propositional logic leads to UNSAT means that the propositional logic is a tautology.

There are many proposed SAT solvers for quickly solving propositional logic. However, SAT solvers handle only propositional logic and do not have the definitions of background knowledge like in predicate logic [17]. Satisfiability Modulo Theories (SMT) solvers have been proposed to resolve this problem. In addition to the definitions of predicate logic, SMT solvers can define integer types, arrays, etc. that are used in recent computer science and allow users to describe propositions abstractly and effectively.

SAW supports ABC [18], Boolector [19], CVC4 [20], MathSAT [21], Yices [22], and Z3 [23]. In this work, we used all of the above SMT solvers except for Boolector which does not run on Windows.

2.5 Problems with Handling Class Objects

The SAW uses the prove command to verify equivalence between two functions by comparing the formal models of the semantics of the two functions that are extracted by the java_extract command. This method can easily verify equivalence of the two functions from intermediate code. However, since this method does not perform assertion checks, we cannot find a counterexample regarding the two functions that has the same bugs. Thus we need to devise a specification based method to check equivalence. Another problem of this method is that the java_extract command cannot model functions that have array type arguments and we need to devise a new method to do this.

The SAW tool uses the java_verify and llvm_verify commands to prove equivalence between a C or Java function and a specification. The java_verify command is used for proving the equivalence between JVM code and a specification that users describe in the JavaSetup block of a SAWScript. The llvm_verify command and LLVMSetup block are for LLVM code. In the JavaSetup block, users can create variables that show the entire set of values of inputs by the java_var command and use it as a verification property. The java_var command can handle array type variables, however when a variable is declared, the concrete size of the array must be given by a parameter. Thus, SAW verification is valid only for an array specified with a given size in SAWScript. Due to the static symbolic execution of SAW, increasing of the size of an array leads to a state explosion.

In this work, we want to verify C and Java functions that handle data structures as arguments. In C, data structures are defined by struct, thus, users need to define struct variables as a verification property in SAWScript. In this case, verification can be performed by allocating memory spaces for data structures by the llvm_alloc command and assigning their heap values by the llvm_points_to command. However, SAW currently has no command to assign symbolic variables for fields of class objects that are allocated by the jvm_alloc_object command. Thus, we need to devise a method to assign symbolic variables and to perform SAW verification on Java functions.

3 PROPOSED METHOD

3.1 Equivalence Checking by Using SAW

In this section, we first explain how to perform an equivalence verification with the prove command using the two functions shown in Listing 3. The SAWScript is shown in Listing 4. The java_extract command in the second and third lines extract Java functions from the .class file and create formal models that show these semantics. The fourth line defines a verification property that states the return values of the two formal models are equivalent for any argument x. The fifth line verifies the property with the Z3 solver by the prove command which verifies the given assertion and generates a counterexample if it is invalid. In this case, SAW does not find any counterexamples. Thus the two functions are equivalent.

The prove command provides an easy method for checking the equivalence of two functions. However, the java_extract command cannot model functions that have indefinite size arguments like array types because this command generates a formal model by performing symbolic simulation. When a function that has an array type argument is assigned to the java_extract command, SAW will terminate. This method has a demerit that the prove command will only check the equivalence of two functions regardless of whether or not these functions satisfy the required specification.

We introduce a method to verify equivalence of functions that have array type arguments by employing an assertion check with the verify command. The conceptual diagram is shown in Fig. 2. In this method, users describe appropriate properties that target functions need to satisfy in the SAWScript and perform verification with the java_verify or llvm_verify commands for each function. When two functions satisfy the same properties, in other words, SAW does

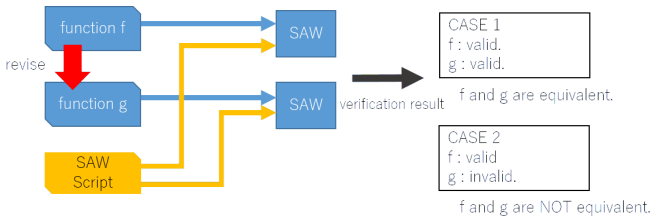

Figure 2: Equivalence checking by SAW

not find a counterexample, the two functions are judged as equivalent.

Listing 3: Revised Java Mul Class

```
1  class Mul{
2    int oldJavafunc(int x){
3      return x + x;
4    }
5
6    int newJavafunc(int x){
7      return x << 1;
8    }
9  }
```

Listing 4: Equivalence Verification with the prove Command

```
1  java_mul <- java_load_class "Mul";
2  old <- java_extract java_mul "oldJavafunc"
       java_pure;
3  new <- java_extract java_mul "newJavafunc"
       java_pure;
4  let thm = {{ \x -> old x == new x }};
5  result <- prove z3 thm;
6  print result;
```

We show a concrete verification example using the target code shown in Listing 3 and the SAWScript is shown in Listing 5. The JavaSetup block of the SAWScript is the same as in Listing 2. Line 7 verifies `oldJavaFunc` and the Line 8 verifies `newJavaFunc`. Both verifications are valid, thus these functions are equivalent meaning that they have the same property of returning $2x$ for any integer type argument x.

The merits of this method are shown below. By using the verify command, users can verify not only equivalence between functions but also equivalence between target functions and specifications. The verify command can handle functions that have array type arguments by defining their sizes in SAWScript as well as reuse verified specifications for other verifications. This is effective in shortening verification times for complex functions.

Listing 5: SAWScript for Revised Java Mul Class

```
1  let java_spec : JavaSetup () = do {
2      java_input <- java_var "x" java_int;
3      java_return {{java_input*2}};
4      java_verify_tactic z3;
5  };
6  java_mul <- java_load_class "Mul";
7  java_verify java_mul "oldJavafunc" [] java_spec
       ;
8  java_verify java_mul "newJavafunc" [] java_spec
       ;
```

If the programming languages that implement the functions are different, the method for proving equivalence requires an additional step since we also need to prove that the verification properties of both program functions are equivalent. We created a C program shown in Listing 6 by revising Listing 1 and the SAWScript for verifying it is shown in Listing 7. In the Java verification, SAW uses the java_int command when creating an integer type symbolic input for Java verification, llvm_int for C. C verification uses LLVMSetup to describe a verification property, whereas Java uses JavaSetup. This is due to the difference of the compiling methods between LLVM and JVM. Therefore, we cannot use the verification property shown in Listing 5 for functions written in a language other than Java and we need to check that the verification properties have the same inputs/outputs in each range. In this case, we defined $\mathbb{Z}_{32}$ that the two functions receive as expression (3).

$$\begin{aligned} \forall x \in \mathbb{Z} \land -2147483648 \leqq x \leqq 2147483647 \\ \Leftrightarrow x \in \mathbb{Z}_{32} \end{aligned} \tag{3}$$

It is apparent that ints of both languages are integer types from their specifications [24], [25]. The range of values changes according to the number of bits of the OS or environment of the compiler. The min/max values of the integer type can be checked by using limits.h for C and Integer.MAX_VALUE for Java. The environment of the experiment is explained in Section 5. As a result, the ranges of int of both languages satisfy expression (3). Thus, we consider that the int of Java and C are the same.

As a result of SAW verification, we confirmed that `newCfunc` satisfies the verification property of Listing 7. Therefore, expression (4) holds and the two functions satisfy the definition of equivalence.

$$\begin{aligned} \forall java\_input : oldJavafunc(java\_input) = 2java\_input \\ \land \forall c\_input : newCfunc(c\_input) = 2c\_input \\ \land java\_input = c\_input \in \mathbb{Z}_{32} \\ \vdash \forall x \in \mathbb{Z}_{32} : \\ oldJavafunc(x) = newCfunc(x) = 2x \end{aligned} \tag{4}$$

Listing 6: C Mul Function

```
1  #include <stdint.h>
2
3  int newCfunc(int x){
4    return x << 1;
5  }
```

Listing 7: SAWScript for C Mul Function

```
1  let c_spec : LLVMSetup () = do{
2    c_input <- llvm_var "x" (llvm_int 32);
3      llvm_return {{ c_input*2 : [32] }};
4      llvm_verify_tactic z3;
5  };
6  c_mul <- llvm_load_module "mul.bc";
7  llvm_verify c_mul "newCfunc" [] c_spec;
```

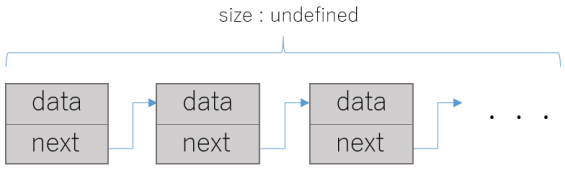

Figure 3: Linear List with Unspecified Size

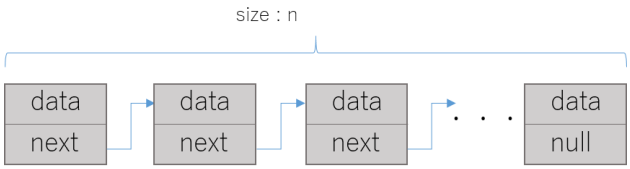

Figure 4: Linear List with Specified Size

3.2 Bounded Model Checking

The java_var command can handle array type variables by using the java_arrray command. However, when users declare array type variables in SAWScript, the concrete size of an array must be given by a parameter. This means there is no method to verify the properties of target functions valid with any array size. Owing to the static symbolic execution of the SAW, increasing the size of data leads to the verification failing without completing the recursion and ending in a state explosion.

In this paper, in order to solve this problem, we verify the data of such structures using the bounded verification method. The bounded verification method verifies that the specified verification property is satisfied for all states obtained by state transitions from the initial state to a certain given number n, using the assumption that the execution of a program is regarded as a state transition [5]. In general, bounded verification methods are realized by using an unfolding technique for a finite number of application steps such as loops. In this paper, we define n to be the size of data structure elements to be given to the function to be verified as shown in Fig. 4.

3.3 Introduction of Helper Functions to SAWScript

The SAW generates symbolic states from intermediate code and looks for combinations of input and output that can disable assertions in SAWScript. Thus, to verify functions that have properties that are difficult to describe in SAWScript, users need to devise alternative methods. We consider for example Java functions that handle linear lists that are defined by class objects. The SAW 0.2 does not have a method to assign symbolic variables to fields of allocated class objects. Thus, the SAW tool cannot directly verify Java functions that handle class object type arguments.

To solve this problem, we devised a method of using helper functions. Helper functions are defined as wrapper functions that call the target functions. Helper functions absorb differences between properties of the target functions and assertions that can be described in SAWScript and make verifying possible. Introducing helper functions to SAW has a merit of being able to perform verifications without changing any code of the target functions.

A simple example of verifying with a helper function is shown in Listings 8 and 9. The target function returns the value of a field x of a class object that receives it as an argument. The argument of the target function is a class object, thus we create a helper function called `testme` which receives an integer type argument and creates an instance for the target function. The helper function will return the return value of the target function, if the target function satisfies the property shown in Listing 9. In this case, the helper function resolves the problem of SAWScript mentioned above by allocating values in it.

Listing 8: Example of Helper Function

```
1  public class Helper {
2    int x;
3
4    Helper(int x){
5      this.x = x;
6    }
7
8    static int target(Helper foo){
9      return foo.x;
10   }
11
12   int testme(int x){
13     Helper help = new Helper(x);
14
15     return target(help);
16   }
17 }
```

Listing 9: SAWScript for Helper Function

```
1  let java_spec : JavaSetup () = do {
2    java_input <- java_var "x" java_int;
3    java_return {{java_input}};
4    java_verify_tactic z3;
5  };
6
7  java_mul <- java_load_class "Helper";
8  java_verify java_mul "testme" [] java_spec;
```

4 EXPERIMENTS

The following two research questions were established for conducting the evaluation experiments.

RQ1 With the proposed method, can we use SAW to evaluate the behavior of Java functions that handle recursive structures?

RQ2 How does the verification time change depending on the solver used and the size of input data?

In order to investigate these questions, we implemented two programs that deal with linear data list structures and binary tree structures, which are typical for handling recursive data structures. Unlike C, Java generally uses classes to implement data structures.

Due to the limitation of SAW, we use a single Java class to implement the data structures.

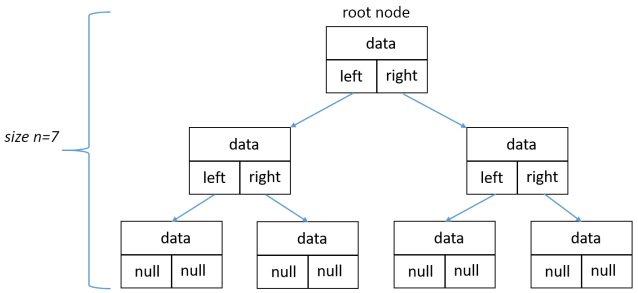

Figure 5: Perfect Binary Tree with size $n = 7$

4.1 Binary Trees

We performed SAW verification for Java functions that can receive binary trees and return the sum of all of the nodes. A binary tree is a data structure that has nodes defined recursively. The example in Fig. 5 shows a perfect binary tree with size $n = 7$.

In this paper, all nodes of a binary tree are Node objects with an integer type variable and two Node objects as their fields. For convenience, we call the nodes "right" and "left" and the size of a binary tree "n".

First, we created a definition of a binary tree shown in Listing 10 by using the Alloy Analyzer [26]. Only one root node exists and it is defined as a subset of Node. All the nodes have lone relationships with other nodes to "right" and "left" and a relationship with Value. The Value signature is $\mathbb{Z}_{32}$. The Alloy Analyzer has an Int signature of integers, however, its range is from -32 to +31. We consider that each node in a binary tree has a value, thus we defined this abstractly by using a Value signature. If data structure S satisfies the constraint below, it is called $S \in Binary\ Tree$.

- Each Node has only one variable.
- All of the Nodes and the Values are able to be referred to from the Root node.
- Each Node has only one route from the Root node.
- Nodes do not loop.

Listing 10: Alloy Code for Binary Trees

```
1  sig Node {
2    val: one Value,
3    left: lone Node,
4    right: lone Node
5  }
6  one sig Root extends Node {}
7  sig Value {}
8
9  fact {
10   #Node = #Value
11   all v: Value, r: Root | v in r.*(right + left
         ).data
12   no n: Node, n': n.right.*(right + left), n'': 
          n.left.*(right + left) | n' = n''
13   all n: Node | n not in n.^(right + left)
14 }
```

The Java program that handles a binary tree is shown in Listing 11. The Node and Value signature of the Alloy script

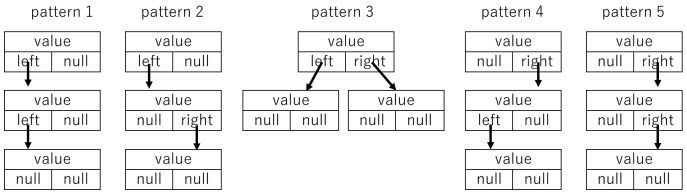

Figure 6: All Patterns of Binary Trees with size $n = 3$

corresponds with the objects of this class and the var in the second line. Generally, a node class is defined in another class, however, we use a single class to simplify the SAW verification in this case.

We consider the case in which we refactored `oldFunc` to create `newFunc` and we need to prove that these functions are equivalent.

When each of the functions receives the top node of a binary tree, they return the sum of all of the values in it, but their algorithms are different. Their return values are integer types, however, the arguments of both are defined as class objects. Since it is difficult to define a class object in SAWScript, we created a helper function based on the proposed method and verified it.

We explain the behavior of the helper function `testme`. When it receives an integer type array, it generates three class objects of `JavaBTree`.

The node2 object is assigned to node1.left and node3 is assigned to node1.right. The data structure generated by the `testme` function is a perfect binary tree with size $n = 3$ that satisfies the definition of a binary tree shown in Listing 10. The `helperFunc` calls either of the target functions as its return value.

The possible binary tree structures with size $n = 3$ are the five patterns shown in Fig. 6. The structure of the binary tree generated by `testme` equals Pattern 3. To say that the target function satisfies the verification property with size $n = 3$ we need to check for all patterns. Therefore, we created and proved helper functions corresponding to each of the binary tree structures. Since implementation is possible by simply switching the node indicated by the pointers, we omit the source code here.

We explain the SAWScript shown in Listing 12. Since the size of the array that `testme` assumes is 3, we defined an integer type array with size $n = 3$ as input and the sum of its elements as output in the SAWScript. The same SAWScript is used when the size of the binary tree is 3. The size of the array increases or decreases based on the size of the binary trees that `testme` assumes.

Listing 11: Java Program for Pattern 3

```
1  public class JavaBTree{
2    int val;
3    JavaBTree left;
4    JavaBTree right;
5
6    JavaBTree(int val){
7        this.val = val;
8        left = null;
9        right = null;
```

```
10    }
11
12    int oldFunc(JavaBTree now){
13      if(now != null && now.right != null && now.
            left != null){
14        return now.val + oldJavaBTreeFunc(now.
              right) + oldJavaBTreeFunc(now.left);
15      }else if(now != null && now.right == null
            && now.left != null){
16        return now.val + oldJavaBTreeFunc(now.
              left);
17      }else if(now != null && now.right != null
            && now.left == null){
18        return now.val + oldJavaBTreeFunc(now.
              right);
19      }else if(now != null && now.right == null
            && now.left == null){
20        return now.val;
21      }else{
22        return 0;
23      }
24    }
25
26    int newFunc(JavaBTree now){
27      if(now == null){
28        return 0;
29      }
30      return now.val + newJavaBTreeFunc(now.right
            ) + newJavaBTreeFunc(now.left);
31    }
32
33    int testme(int[] x){
34      JavaBTree node1 = new JavaBTree(x[0]);
35      JavaBTree node2 = new JavaBTree(x[1]);
36      JavaBTree node3 = new JavaBTree(x[2]);
37      node1.left = node2;
38      node1.right = node3;
39
40      return oldFunc(node1);
41      //return newFunc(node1);
42    }
43  }
```

Listing 12: SAWScript for Java BTree Program

```
1  let linear_spec : JavaSetup () = do {
2    ary <- java_var "x" (java_array 3 java_int);
3    java_return {{ ary@0 + ary@1 + ary@2 }};
4    java_verify_tactic yices;
5  };
6  linear_java <- java_load_class "JavaBTree";
7  java_verify linear_java "
       helperFunc_size3_pattern3" [] linear_spec;
```

We performed the following two experiments.

- SAW verification for all binary tree structures with size $1 \leq n \leq 3$. The numbers of possible binary tree structures are 1 when size $n = 1$, 2 when size $n = 2$, and 5 when size $n = 3$. The verification result is shown in Section 6.1.1.
- SAW verification for all perfect binary trees with size $1 \leq n \leq 63$ to investigate the gains of verification time by size. The verification result is shown in Section 6.1.2.

4.2 Linear Lists

We performed SAW verification for C and Java functions that receive linear lists and return the sum of all of the nodes.

A linear list is a data structure that has nodes defined recursively. The example in Fig. 7 shows a linear list with size $n = 3$. In this paper, all nodes of linear lists are Node objects and they have an integer type variable and one Node object as their fields. For convenience, we call nodes indicated by the pointer as "next" and the size of a linear list as "n".

As with the binary trees, we created a definition of a linear list shown in Listing 13 by using the Alloy Analyzer. Only one top node exists and it is defined as a subset of Node. All nodes have lone relationships with other nodes as "next" and a relationship with Value. The Value signature is taken as $\mathbb{Z}_{32}$ and if the data structure S satisfies the constraint below, it is called $S \in Linear\ List$.

- Each Node has only one variable.
- All of the Nodes and the Values are able to be referred to from the Top node.
- Nodes do not loop.

Listing 13: Alloy Code for Linear Lists

```
1  sig Node {
2    val: one Value,
3    next: lone Node
4  }
5    one sig Top extends Node {}
6    sig Value {
7  }
8
9  fact {
10   #Node = #Value
11   all v: Value, t: Top | v in t.*(next).val
12   no n: Node | n in n.^next
13 }
```

4.2.1 Java Linear List

A Java program for defining a linear list class is shown in Listing 14. We performed SAW verification for all linear list structures with size $1 \leq n \leq 10$. The verification result is shown in Section 6.2.1. As described above, we implemented the program in a single Java class for simplicity since we are focusing on a single function.

We assume that we need to prove the property of the function `javaLinearFunc`. When this function receives the top node of a linear list, it returns the sum of its data by calling itself recursively. The data type of the argument of the target function is a class object of `JavaLinear`, which is difficult to define in SAWScript as input.

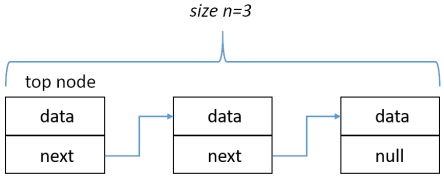

Figure 7: Linear List Structure

Thus, we create `helperFunc` based on the proposed method. This function generates a linear list with size $n = 3$ that satisfies expression (5). Since the function structure is the same as Listing 11, we omit the explanation.

$$now = node1(x[0], node2(x[1], node3(x[2], null))) \quad (5)$$

As a result, all possible linear list patterns that can be constructed with size $n = 3$ can be completed with one helper function. This is a benefit of symbolic execution. The SAWScript is shown in Listing 15.

Listing 14: Java Linear class

```
1  public class JavaLinear{
2    int val;
3      JavaLinear next;
4
5      JavaLinear(int val){
6        this.val = val;
7        next = null;
8      }
9
10   int javaLinearFunc(JavaLinear now){
11     if(now.next != null){
12       return now.val + javaLinearFunc(now.next)
             ;
13     }else{
14       return now.val;
15     }
16   }
17
18   int helperFunc(int[] x){
19     JavaLinear node1 = new JavaLinear(x[0]);
20     JavaLinear node2 = new JavaLinear(x[1]);
21     JavaLinear node3 = new JavaLinear(x[2]);
22     node1.next = node2;
23     node2.next = node3;
24     return javaLinearFunc(node1);
25   }
26 }
```

Listing 15: SAWScript for Java Linear Program

```
1  let java_spec : JavaSetup () = do {
2    ary <- java_var "x" (java_array 3 java_int);
3    java_return {{ary@0 + ary@1 + ary@2}};
4    java_verify_tactic mathsat;
5  };
6  linear_java <- java_load_class "JavaLinear";
7  java_verify linear_java "helperFunc" []
       java_spec;
```

4.2.2 C Linear List

We revised the previous Java program of Listing 14 to a C program shown in Listing 16. We performed SAW verification for all linear list structures with size $1 \leq n \leq 10$. The verification result is shown in Section 6.2.2. A node is defined by using struct, has an integer type variable and a pointer to the next node. The target function is `cLinearFunc` and its algorithm is the same as `javaLinearFunc` in Listing 14.

The SAWScript is shown in Listing 17. Defining a struct of C in SAWScript is easily possible by using the crucible_alloc and crucible_points_to commands.

The crucible_alloc command allows SAW to reserve memory space. In this experiment, we need to perform SAW verification on a linear structure with size $n = 3$. The description from Lines 6 to 8 let SAW reserve memory space of struct named node1, 2, and 3.

The crucible_points_to allows SAW to specify the destination of a pointer in the memory space. By using this, users can create data structures that the target function receives in SAWScript. The description from Lines 10 to 15 determines the specification of node1: its field is val1, next node is node2.

The crucible_execute_func in Line 29 allows SAW to determine the argument that the target function receives. In this case, the target function needs to receive the top node of a linear list.

Listing 16: C Linear Program

```
1  #include <stdint.h>
2  typedef struct NODE{
3    int val;
4    struct NODE *next;
5  }node_t;
6
7  int cLinearFunc(node_t *now){
8    if(now->next != '\0'){
9      return now->val +cLinearFunc(now->next);
10   }else{
11     return now->val;
12   }
13 }
```

Listing 17: SAWScript for C Linear Program

```
1  let linear_spec = do{
2    val1<-crucible_fresh_var "val1" (llvm_int 32)
         ;
3    val2<-crucible_fresh_var "val2" (llvm_int 32)
         ;
4    val3<-crucible_fresh_var "val3" (llvm_int 32)
         ;
5
6    node1<-crucible_alloc (llvm_struct "struct.
         NODE");
7    node2<-crucible_alloc (llvm_struct "struct.
         NODE");
8    node3<-crucible_alloc (llvm_struct "struct.
         NODE");
9
10   crucible_points_to node1 (
11     crucible_struct [
12     crucible_term val1,
13     node2
14     ]
15   );
16   crucible_points_to node2 (
17     crucible_struct [
18     crucible_term val2,
19     node3
20     ]
21   );
22   crucible_points_to node3 (
23     crucible_struct [
24     crucible_term val3,
25     crucible_null
26     ]
27   );
28
29   crucible_execute_func [node1];
30   crucible_return(crucible_term{{val1+val2+val3
         }});
```

```
31  };
32
33  print "llvm_load start";
34  linear_c <- llvm_load_module "liner.bc";
35  crucible_llvm_verify linear_c "cLinearFunc" []
        false linear_spec abc;
36  print "Done.";
```

4.2.3 Increases of Verification Time

As shown in Tables 1 and 2, timeout (T/O) verification times greater than 600 sec. did not occur with the verifications by CVC4, Yices, and Z3. Therefore, we performed additional tests with these solvers to investigate the increases in verification time with size. We generated test cases of linear lists with size $100 \leq n \leq 2000$ in 50 node increments and performed SAW verification. The results are shown in Section 6.2.3.

5 ENVIRONMENT OF THE EXPERIMENTS

The following summarizes the specifications of the PC used for the experiments.

- PC : TOSHIBA Dynabook T55/76MG
- OS : Windows 10 64-bit
- CPU : Intel Core i7 4510U
- RAM : 8GB DDR3

The versions of the tools used in the experiments are as follows.

- SAW : 0.2
- Clang : 3.7.1
- Java : 1.8.0_211
- ABC : 1.0.1
- CVC4 : 1.5
- Z3 : 4.6.0
- Yices : 2.5.4
- MathSAT : 5.5.1

We used the older LLVM 3.7.1, due to the SAW limitation.

6 EXPERIMENT RESULTS

In this section, we show the results of experiments for the various solvers. The numbers in the tables represent seconds. Verification times greater than 600 seconds were judged as timeouts (T/O).

6.1 Verification Results for Binary Trees

6.1.1 Verification Result with Size $1 \leq n \leq 3$

We performed SAW verification for all binary tree structures with size $1 \leq n \leq 3$. As a result, SAW verification for both functions was successful for all possible binary tree patterns in this range.

6.1.2 Verification Times Changing the Number of Elements

We performed SAW verification for all perfect binary trees with size $1 \leq n \leq 63$. The verification times in seconds are shown in Table 1.

6.2 Verification Results for Linear Lists

6.2.1 Verification Times for Java Linear List

We performed SAW verification for the `javaLinearFunc` function in Listing 14 by changing the size of the linear list. The verification times in seconds are shown in Table 2.

6.2.2 Verification Times of C Linear List Structures

The result of SAW verification for Listing 16 is shown below. The value of the counterexample is 1 more than INT_MAX.

> SolverStats solverStatsSolvers = fromList [”ABC”], solverStatsGoalSize = 45
>
> ————-Counterexample————-
>
> (“val2”,2147483648)
>
> (“val3”,2147483648)

When we changed the type of variable and function from int to uint32_t, SAW verification was successful. Thus, we performed a modified SAW verification for that program and the results are shown in Table 3.

6.2.3 Increases of Verification Times

The results of the experiment in Section 4.2.3 are shown in Figs. 6.2.3 and 6.2.3. Figure 6.2.3 shows the change in verification time of the Java program of Listing 14. No timeouts occured with any of the three SMT solvers and SAW verifications were successful, taking less than 30 seconds with size $n = 2000$.

Table 1: Verification Times for Binary Trees

size n	ABC	CVC4	MathSAT	Yices	Z3
1	0.203	0.340	0.319	0.358	0.315
3	0.212	0.454	0.415	0.526	0.423
7	133.626	0.462	T/O	0.504	0.431
15	T/O	0.494	T/O	0.536	0.469
31	T/O	0.635	T/O	0.616	0.571
63	T/O	0.634	T/O	0.706	0.571

Figure 6.2.3 shows the change of verification time of the C program of Listing 16. To perform SAW verification, we changed the type of variable and function from int to uint32_t. timeouts occur with CVC4 when size $n = 1400$, Yices and Z3 when size $n = 1350$.

7 DISCUSSION

7.1 Property Proving for Java Binary Tree Programs

When function `helperFunc` of Listing 11 satisfies the verification property shown in Listing 12, expression (6) holds.

$$\begin{aligned} &\forall x \in \mathbb{Z}_{32}\ Array, size = 3 : \\ &helperFunc(x) = \sum x = x[0] + x[1] + x[2] \end{aligned} \tag{6}$$

This means that when `helperFunc` receives any integer type array with size $n = 3$, it returns the sum of its elements. Property proving for the target function must be given by users.

According to the result of the experiment, when the return value of `helperFunc` is either target function, it satisfies the verification property. Thus, the return value of `helperFunc` is the same as the return value of `oldJavaBTreeFunc(node1)` and `newJavaBTreeFunc(node1)`. Also, the argument node1 that the target functions receive is the root node of any binary tree with $size\ n = 3$ and $Pattern = 3$ in Fig. 6. Thus, expression (7) holds.

$$\begin{aligned} &\forall x \in \mathbb{Z}_{32}\ Array \wedge size = 3 \\ &\wedge \forall node1 \in BinaryTree \wedge size = 3 \wedge Pattern = 3 : \\ &helperFunc(x) = oldJavaBTreeFunc(node1) \\ &= newJavaBTreeFunc(node1) \\ &= \sum x \end{aligned} \tag{7}$$

When instance creation occurs in `helperFunc`, an integer value is given for the val of each node as data by a constructor. From the description of Listing 11, expression (8) holds.

Table 2: Verification Times for Java Linear List Structures

size n	ABC	CVC4	MathSAT	Yices	Z3
1	0.186	0.205	0.312	0.202	0.221
2	0.081	0.206	0.177	0.224	0.214
3	0.278	0.205	0.301	0.208	0.211
4	1.184	0.208	6.573	0.216	0.219
5	5.978	0.212	114.327	0.219	0.213
6	54.713	0.214	332.668	0.214	0.214
7	144.602	0.211	T/O	0.215	0.228
8	T/O	0.221	T/O	0.214	0.228
9	T/O	0.252	T/O	0.238	0.266
10	T/O	0.224	T/O	0.225	0.231

Table 3: Verification Times for C Linear List Stuctures

size n	ABC	CVC4	MathSAT	Yices	Z3
1	0.512	0.462	0.447	0.463	0.459
2	0.568	0.508	0.539	0.540	0.505
3	0.766	0.506	0.617	0.494	0.505
4	1.823	0.514	6.478	0.495	0.606
5	7.234	0.513	105.291	0.502	0.502
6	58.827	0.521	315.102	0.505	0.509
7	134.956	0.514	T/O	0.507	0.525
8	256.111	0.526	T/O	0.508	0.516
9	T/O	0.523	T/O	0.509	0.529
10	T/O	0.525	T/O	0.512	0.526

$$\begin{aligned} &node1.val = x[0] \wedge node2.val = x[1] \wedge node3.val = x[2] \\ &\vdash \sum x = \sum_{i=1}^{n} node_n.val \end{aligned} \tag{8}$$

From expressions (7) and (8), it is proved that when both target functions receive the root node of any binary tree with $size\ n = 3$ and $Pattern = 3$, they return the sum of all node values.

According to the result of the experiment, SAW verifications succeeded for all possible binary tree structures with size $1 \leq n \leq 3$. The proof for the other structure patterns is possible in the same way, therefore expression (9) holds.

$$\begin{aligned} &\forall now \in BinaryTree \wedge 1 \leq n \leq 3 : \\ &oldJavaBTreeFunc(now) \\ &= newJavaBTreeFunc(now) \\ &= \sum_{i=1}^{n} node_n.val \end{aligned} \tag{9}$$

In other words, two functions being equivalent means that they return the sum of all elements of a binary tree when they receive any binary tree with size $1 \leq n \leq 3$.

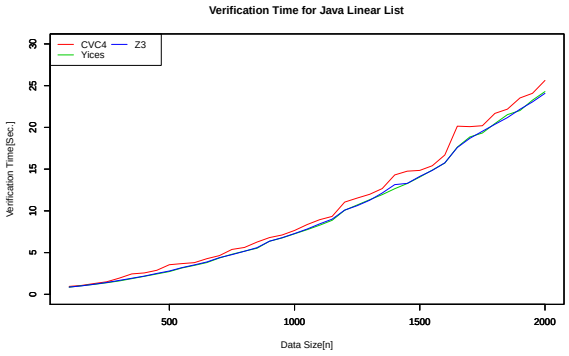

Figure 8: Verification Time for Java Linear List Program

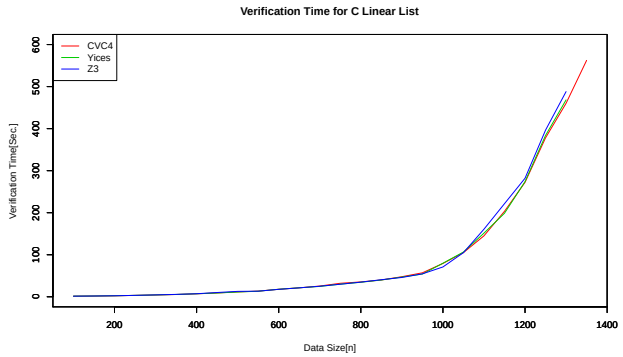

Figure 9: Verification Time for C Linear List Program

7.2 Property Proving for Java and C Linear List Programs

Since the number of possible linear list structure orderings does not increase with additional nodes, verification is easy compared with binary tree structures.

7.2.1 Java Linear List

The function `javaLinearFunc` of Listing 14 satisfies expression (10).

$$\begin{aligned} &\forall now \in LinearList \wedge 1 \leq size\ n \leq 10 : \\ &= javaLinearFunc(now) = \sum_{i=1}^{n} node_n.val \end{aligned} \tag{10}$$

In other words, when `javaLinearFunc` receives any linear list with size $1 \leq n \leq 10$, it returns the sum of its elements. The proof is similar to that in Section 7.1 and we omit it here.

7.2.2 C Linear List

As shown in Section 6.2.2, we changed the type of variable and function from int to uint_32 to enable a SAW verification. Thus, the definition of a linear list was changed to expression (11).

$$\begin{aligned} \forall f(x,n) : x \in \mathbb{Z}_{32^u} \wedge n ::= f(x,n)|null \\ \Leftrightarrow f(x,n) \in C\ Linear\ List \end{aligned} \tag{11}$$

It is possible for SAWScript to define a struct of C for C verification. Thus, expression (12) holds.

$$\begin{aligned} &\forall now \in C\ Linear\ List \wedge 1 \leq size\ n \leq 10 : \\ &= cLinearFunc(now) = \sum_{i=1}^{n} node_n.val \end{aligned} \tag{12}$$

7.2.3 Equivalence between Java and C Functions

The value of the counterexample shown in Section 6.2.2 was 1 more than INT_MAX and SAW verification was successful when we changed the type of variable and function from int to uint32_t. Thus, we assume that llvm_int 32 of SAWScript means uint32_t of C and we implemented an intentional bug of Listing 6 so that it returns x when x==0xFFFFFFFF. The type of its variable and function were changed into uint32_t. The result of SAW verification is shown below.

> Proof of return value failed.
> ————-Counterexample————-
> %x: 4294967295
> return value
> Encountered: 4294967295
> Expected: 4294967294
> Proof failed.

From this result, we determined that llvm_int 32 defines the symbolic variable of uint32_t. However, it is not clear why the SAW verification of the original program of Listing 6 was successful in spite of the types of the target function and its argument being int. Currently, we consider this behavior of SAW as a bug and we need to investigate further and clarify the specification of SAW.

The types of the target functions and data values of linear lists are $\mathbb{Z}_{32}$ in Java, $\mathbb{Z}_{32^u}$ in C. These differences make it difficult to prove the two functions are equivalent in a strict sense. However, it can at least be said that the two functions meet the same specification in that they return the sum of a linear list when they receive the top node of it.

7.3 Verification Time

In this section, we discuss the verification times. SAW automatically generates formal models based on target functions and descriptions of SAWScript and then solves them using SMT solvers. Thus, the time or ability of calculation is strongly influenced by the performances of the solvers.

There are pros and cons of the various SMT solvers. For example, in Table 2, the verification time with size $n = 5$ of Yices is over 100 sec. less than that of MathSAT. It is known as a demerit of SMT verification that estimating the verification time before performing verification is difficult due to the complex implementation of SMT solvers [27]. However, we can change which SMT solver to use in SAW just by editing the description of SAWScript and users can investigate the differences of verification time easily.

Due to the symbolic execution of SAW, we reduced the number of required test cases for verifying functions that handle recursive data structures to the number of possible patterns of data structures.

The number of possible binary tree structures with size n is shown in expression (13) and is called a Catalan Number[28].

$$P(n) = \frac{(2n)!}{(n+1)!n!}(n \geq 0) \tag{13}$$

This shows that the time complexity of verifying a binary tree function is $O(n!)$. Table 1 shows that Z3 verifies a perfect binary tree with size $n = 15$ in 0.469 sec., however, the

total number of possible binary tree patterns with size $n = 15$ is about 9.7 million. In spite of using symbolic execution, verifying a function with recursive data structures still needs enormous time. Thus, we need to devise a method that verifies more effectively.

7.4 Threats to Validity

7.4.1 Proposed Method

In this paper, we proposed a method to perform SAW verification for functions that receive class objects by using helper functions. This method is useful in defining arguments of the target function that cannot be defined in SAWScript. However, SAW can only verify the property of the helper functions, which means users need to prove properties of the target function manually like in Section 7.1. This dependency is a threat to internal validity and a way of generalizing the method is needed.

7.4.2 Definition of Equivalence

It is obvious that there are various requirements for the definition of equivalence [29], e.g., time complexity, readability, requests of memory space, and power consumption. However, in this paper, we adopted the description in [1] as the definition of refactoring and expression (1) as the definition of equivalence between two program functions. The reason for this is due to the specification of SAW. SAW automatically generates formal models of target functions and solves them by using SMT solvers. However, in SAWScript, users can define inputs and outputs as verification properties of the target function by this method, and we used SAW as a black-box unit testing tool. To evaluate the requirements shown above, SAW is not an adequate tool and users need to adopt other approaches.

8 CONCLUSION

In this work, we applied our proposed method to Java programs dealing with two types of recursive data structures and verified them using SAW to show that verification is actually possible. For the verification of linear lists, a comparison with the verification for C in our previous research was discussed.

We also proposed a method for simplifying the description of SAWScript and performing verification inductively by creating a helper function and defining a structure piece by piece in it. We confirmed that equivalence verification by SAW can be performed in the sense that two functions written in C and Java satisfy the same verification property.

In our future work, we will conduct further evaluation experiments and devise a method to verify the equivalence of the behavior of functions which have more complex algorithms like sorting algorithms. In addition, we would like to investigate the influence of language and algorithm on the solvers, and consider a method to find the optimal solver in equivalence verification.

ACKNOWLEDGEMENT

Funding from Mitsubishi Electric Corp. is gratefully acknowledged.

The research is being partially conducted as Grant-in- Aid for Scientific Research C (16K00094), C (17K00111) and A (18H04094).

REFERENCES

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, and Gamma E, *"Refactoring: Improving the Design of Existing Code"*. Addison-Wesley Professional, July 8, (1999).

[2] Dock A., J. Hendrix, B. Huffman, D. McNamee, and A. Tomb, "Constructing semantic models of programs with the software analysis workbench". In *11th Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 56–72, (2016).

[3] K. Carter, A. Foltzer, J. Hendrix, B. Huffman, and A. Tomb, "Saw: The software analysis workbench". In *Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '13, pages 15–18, (2013).

[4] R. Karashima, S. Harauchi, K. Okano, and S. Ogata, "Proposal and evaluation for equivalence checking for program with recursive data using saw". In *25th Workshop of Fundamentals of Software Engineering*, pages 91–96, (2018).

[5] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving". *Formal Methods in System Design*, 19(1):7–34, (2001).

[6] L. Tianhai, N. Michael, and T. Mana, "Bounded program verification using an smt solver: A case study". In *5th International Conference on Software Testing, Verification and Validation*, pages 101–110, (2012).

[7] G. Meszaros, *"xUnit Test Patterns: Refactoring Test Code"*. Addison-Wesley Signature Series. Addison-Wesley.

[8] S. Gulati and R. Sharma, *"Java Unit Testing with JUnit 5: Test Driven Development with JUnit 5"*. Apress, (2017).

[9] S. Gupta, A. Saxena, A. Mahajan, and S. Bansal, "Effective use of smt solvers for program equivalence checking through invariant-sketching and query-decomposition". In *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 365–382. Springer International Publishing, (2018).

[10] Y. Sasaki, K. Okano, and S. Kusumoto, "A design of analyzer for java program using smt solver". *Foundation of Software Engineering XIX*, pages 33–38, Dec. (2012).

[11] D. R. Cok and J. R. Kiniry, "Esc/java2: Uniting esc/java and jml". In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 108–128.

[12] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation". In *Proceedings of the International Symposium on Code*

Generation and Optimization: Feedback-directed and Runtime Optimization, pages 75–88, (2004).
[13] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *"The Java Virtual Machine Specification, Java SE 8 Edition"*. Addison-Wesley Professional, 1st edition, (2014).
[14] R. L. Jeffrey and A. M. Brad, "Cryptol: high assurance, retargetable crypto development and validation". *IEEE Military Communications Conference*, Vol.2:820–825, (2003).
[15] L. Erkök and J. Matthews, "High assurance programming in cryptol". In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, CSIIRW '09, pages 60:1–60:2, (2009).
[16] A. Biere, *"Handbook of Satisfiability*. IOS Press, (2009).
[17] K. Iwanuma and H. Nabeshima., "Smt: Satisfiability modulo theories". *Journal of Japanese Society for Artificial Intelligence*, 25(1):86–95, Jan. (2010).
[18] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool". In *Computer Aided Verification*, pages 24–40, (2010).
[19] N. Aina, P. Mathias, and B. Armin, "Boolector 2.0". *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, (2014).
[20] C. Barrett, Christopher L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4". In *Computer Aided Verification*, pages 171–177, (2011).
[21] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 smt solver". In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, (2013).
[22] B. Dutertre, "Yices 2.2". In *Computer Aided Verification*, pages 737–744, (2014).
[23] L. de Moura and Nikolaj Bjørner, "Z3: An efficient smt solver". In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, (2008).
[24] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *"The Java Language Specification, Java SE 8 Edition"*. Addison-Wesley Professional, 1st edition, (2014).
[25] ISO. *"ISO/IEC 9899:2011 Information technology — Programming languages — C"*. International Organization for Standardization, Geneva, Switzerland.
[26] D. Jackson, "Alloy: A lightweight object modelling notation". *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, (2002).
[27] Y. Takano, H. Sakaji, and S. Sato, "Toward automatic selection and automatic tuning of smt solver using machine learning". In *"35th Japan Society for Software Science and Technology Annual Conference*, Aug. (2018).
[28] R. P. Stanley, *"Catalan Numbers"*. Cambridge University Press, (2015).
[29] K. Maruyama, S. Hayashi, N. Yoshida, and E. Choi, "Frame-based behavior preservation in refactoring". In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 573–574, Feb. (2017).

(Received December 10, 2019)
(Revised July 30, 2020)

Rin Karashima is a graduate student of Shinshu University. His areas of interest include formal verification and STAMP/STPA.

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 2015, he has been an Associate 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.

Shinpei Ogata is an Assistant 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.

Satoshi Harauchi received BE and ME degrees in Information Sciences from Kyoto University in 1996 and 1998, respectively. Since 1998, he has been at the Advanced Technology R&D Center of Mitsubishi Electric Corporation and is currently interested in software engineering for social infrastructure systems. He is a member of IEICE and JSASS.

Toshifumi Sekizawa received his MSc degree in physics from Gakushuin University in 1998, and Ph.D. in information science and technology from Osaka University in 2009. He previously worked at Nihon Unisys Ltd., Japan Science and Technology Agency, National Institute of Advanced Industrial Science and Technology, and Osaka Gakuin University. He is currently working at College of Engineering, Nihon University. His research interests include model checking and its applications.