Regular Paper

Regression Verification for C Functions with Recursive Data Structure — Using SAW —

Kozo Okano†, Rin Karashima†, Satoshi Harauchi‡, and Shinpei Ogata†
†Faculty of Engineering, Shinshu University, Japan
‡Mitsubishi Electric, Japan
okano@cs.shibshu-u.ac.jp, ogata@cs.shinshu-u.ac.jp

Abstract - Programs are usually revised to improve performance. In such cases, programmers have to ensure that the revised program preserves the behavior of the previous version of the program. Regression testing is performed to check whether both the revised version and the previous version have the same behavior. It, however, requires much time and large number of test-cases. Tools based on formal method might reduce the costs. They ensure that two given programs output the same results for the same inputs based on a logical analysis of their source code and they perform effective path search using SAT/SMT solvers. Software Analysis Workbench (SAW), a novel tool based on formal methods, can check whether two given functions in C act in the same behavior (conformance verification). SAW, however, has a limitation that it cannot check functions dealing with data structures. This paper proposes a new technique for conformance verification on C functions with data structures using SAW. The technique is based on a kind of bounded model checking. We limit the size of data structures which are generated by recursive definitions, in order to limit the space to search. This paper also reports results on performance evaluation that shows our proposed method works for standard data structures.

1 INTRODUCTION

C programs are widely used especially in embedded systems which have a limitation of resources and they are often revised to improve performance. In such cases, programmers have to ensure that the revised program preserves the behavior of the previous version of the program. Usually, regression testing is performed to check whether both of the revised version and the previous version have the same behavior. Thus, it is tedious work and requires large number of test-cases.

Formal technique approach might help to reduce such costs. Based on formal approaches several tools have been developed. Such techniques exhaustively check whether two given programs have always the same output for every same input. Thus, these tools will find potential bugs or confirm the conformance 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. Such a program sometimes suffers a halting problem in computability theory.

In this paper, we firstly propose a method for FVC for a program with recursive data structures. In order to avoid the halting problem, the method is based on the bounded model verification technique [1], [2]. We also perform experimental evaluations using Software Analysis Workbench (SAW) [3]. SAW is a recent formal verification tool. We use SeaHorn [4] to compare with SAW. SeaHorn is also a recent formal verification tool for the C language.

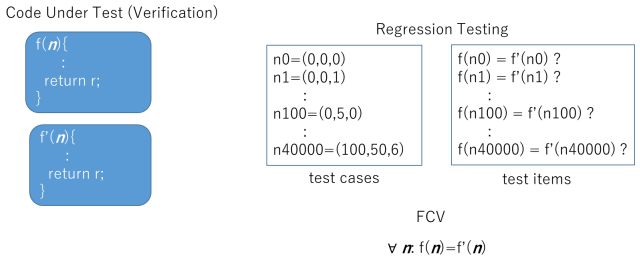

Figure 1: Regression Testing versus FCV

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

2 PRELIMINARY AND RELATED WORK

In general, testing is the historical and popular method for checking the quality of source code. For conformance testing, regression testing is widely used. Regression testing, however, has a high time cost and workload. It also has the disadvantage that the verification becomes incomplete in most cases.

Figure 1 shows the difference between the conventional regression testing and FCV approach. As a given Code Under Test (CUT), or CUV Code Under Verification, let us assume that two functions f and f' exist. We want to check that for any input $\boldsymbol{n}$, $f(\boldsymbol{n}) = f'(\boldsymbol{n})$ holds.

In regression testing, we have to prepare a sufficient number of test cases (in Fig. 1, we have 40000 cases) and check all cases by executing a test driver. As we can see, regression testing requires a substantial amount of time and it does not cover whole range of input cases.

2.1 Formal Conformance Verification

The following two theorems are well-known results in Computation Theory [5].

Theorem 1 (Termination Problem (Halting Problem))
The Termination Problem is undecidable.

In other words, $\forall f$ and $\forall \boldsymbol{n} \in \mathbb{Z}^{|\boldsymbol{n}|}$, whether $f(\boldsymbol{n})$ always terminates or not, is undecidable.

Theorem 2
The general conformance checking problem is undecidable.

In other words, $\forall f,\ f'$ and $\forall \boldsymbol{n} \in \mathbb{Z}^{|\boldsymbol{n}|}$, whether $f(\boldsymbol{n}) = f'(\boldsymbol{n})$ always holds or not, is undecidable.

If we restrict the condition, the general conformance checking problem can be decidable. Thus, the restricted conformance checking problem is decidable.

Theorem 3
The restricted conformance checking problem is decidable.

$\forall f,\ f'$ and $\forall \boldsymbol{n} \in \mathbb{Z}_t^{|\boldsymbol{n}|}$, whether $f(\boldsymbol{n}) = f'(\boldsymbol{n})$ always holds or not, is decidable, provided that both f and f' terminate for any $\boldsymbol{n} \in \mathbb{Z}_t^{|\boldsymbol{n}|}$, where $\mathbb{Z}_t$ is a whole set of t-bit integers for some fixed parameter t.

Proof 3
The size of $\mathbb{Z}_t$ is 2^t. Therefore, the size of $\mathbb{Z}_t^{|\boldsymbol{n}|}$ is at most $2^{t|\boldsymbol{n}|}$. The assumption guarantees that we think of only functions f and f' that always terminate for any $bmn \in \mathbb{Z}_t^{|\boldsymbol{n}|}$. Thus, we can compute the result of $f(\boldsymbol{n})$ and $f'(\boldsymbol{n})$ in finite steps $\alpha(\boldsymbol{n})$ and $\alpha'(\boldsymbol{n})$, respectively. In conclusion, we can decide if $f(\boldsymbol{n}) = f'(\boldsymbol{n})$ always holds, in finite steps $2^{t|\boldsymbol{n}|} \cdot \max(\alpha(\boldsymbol{n}), \alpha(\boldsymbol{n}))$. □

In FCV, we check the logical expression $\forall \boldsymbol{n} : f(\boldsymbol{n}) = f'(\boldsymbol{n})$, where functions f and f' are expressed in some logical clauses derived from CUV.

Note that $\boldsymbol{n}$ is usually a vector of bounded integers, such as a 32-bit integer, thus, the number of check cases is finite.

Form Theorem 3, if we suppose that functions f, and f' always terminate then the expression can be efficiently checked using SAT/SMT solvers[6]–[12].

SAW and SeaHorn [4] are tools appearing recently to efficiently check all inputs cases.

2.2 SAW

Software Analysis Workbench (SAW) [3] is developed by Galois inc. It is an open source software which verifies code written in C or Java using a compiler that generates LLVM, or JVM (Java Virtual Machine). Some recent formal verification techniques [13]–[15] use JVM and LLVM as their targets. An LLVM file is compiled from a C, C++, or Objective-C source file. LLVM is a virtual machine instruction set (intermediate representation) and usually used for code optimization in compilers. It, therefore, supports a three-address code scheme and the Static Single Assignment form, which facilitate static analysis for optimizing compiled code. LLVM has pointer types as well, which is mandatory for compilers of C-family languages.

SAW supports equivalence checking between two C functions given in the LLVM format. Both symbolic execution and equivalence checking functions are provided as commands

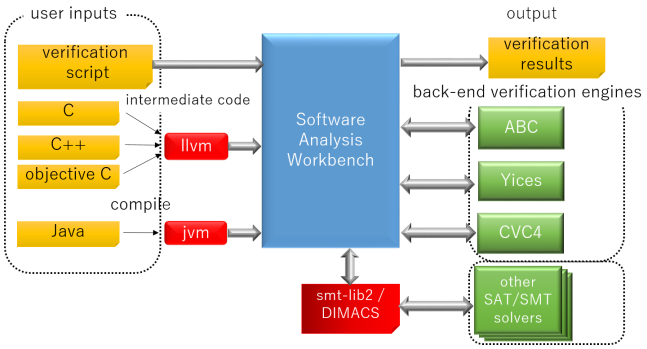

Figure 2: The architecture of SAW

of a script language used in SAW. SAW also supports property checking and has been successfully applied to security domains such as cryptographic protocol analysis.

Figure 2 shows the architecture of SAW.

We summarize the features of SAW.

- It uses its own verification script called SAWScript, which is a kind of functional programming languages.
- It has several verification packages that support specific fields:
  - llvm_extract
  - llvm_symexec
  - llvm_verify
  - crucible_llvm_verify
- It has a build-in solver, ABC [17], and can also use three SAT/SMT solvers, Z3 [6], Yices [7], and CVC4 [8].
- It can generate proof constraints with a form of AIG (And-Inverter Graphs) [16], and smtlib2 [18]. Using the format file, other external solvers can be available.

Package llvm_symexec is the original package used by SAW. Crucible_llvm_verify package is provided recently, and supports pointers and data structures in C.

2.2.1 Verification Examples using SAW

The following example shows the verification process for two functions that output a value twice of the input values (See Listing 1, 2, and 3).

Listing 1: Twice Program

```
// reference function
unsigned int reference_function(unsigned int x){
  return x * 2;
}

// implementation
unsigned int implementation_function(unsigned int x){
  return x << 1;
}
```

Listing 1 has two functions reference_function() and implementation_function(). Function reference_function() just outputs the value of the input multiplied by two, while implementation_function() outputs arithmetic left shift of the input value by 1 bit. Though the codes differ from each other, those functions output the same value for any value of the same input. Proof scripts Listing 2 and 3 prove by different approaches that the two functions are equivalent.

Listing 2: Verification script for twice program (llvm_symexec)

```
// llvm_symexec
// .bc is llvm format
load <- llvm_load_module "add.bc";

// reference_function
// variable x is defined as 32bit integer
x <- fresh_symbolic "x" {| [32] |};
// alloc is used when pointer is used
let alloc_ref  = [];
//
let input_ref  = [("x", x, 1)];
//
let output_ref = [("return", 1)];

t1 <-
  llvm_symexec load "reference_function"  alloc_ref
  input_ref output_ref true;

// implementation_function
let alloc_imp  = [];
let input_imp  = [("x", x, 1)];
let output_imp = [("return", 1)];

t2 <-
  llvm_symexec load "implementation_function" alloc_imp
  input_imp output_imp true;

// verification
thm <- abstract_symbolic {{ t1 == t2 }};
result <- prove z3 thm;
//
print result;
```

Listing 3: Verification script for twice program (crucible_llvm_verify)

```
// crucible_llvm_verify
// add.bc
load <- llvm_load_module "add.bc";

// reference_function
let add_setup = do {
//
    x <- crucible_fresh_var "x" (llvm_int 32);
//
    crucible_execute_func [crucible_term x];
//
    crucible_return (crucible_term {{ x << 2 : [32] }});
};

crucible_llvm_verify load "reference_function" [] false
  add_setup abc;
```

Listings 4 and 5 show the results, respectively.

Listing 4: Result (SAW:llvm_symexec)

```
$saw llvm_symexec.saw
Loading file "llvm_symexec.saw"
Running reference_function
Finished running reference_function
Running implementation_function
Finished running implementation_function
Valid
```

Listing 5: Result (SAW:crucible_llvm_verify)

```
$saw crucible_llvm_verify.saw
Loading file "crucible_llvm_verify.saw"
Proof succeeded! @reference_function
Running reference_function
```

Messages “Valid” and “Proof succeeded! @ reference_function” show that the two functions have the same behaviors, for llvm_symexec and crucible_llvm_verify, respectively.

Listings 6, 7, and 8 show the case that FCV outputs counter-examples.

Listing 6: Wrong implemented code

```
// Reference Function
unsigned int reference_function(unsigned int x){
  return x * 2;
}

// Implementation
// (llvm_symexec)
unsigned int implementation_function(unsigned int x){
  if(x == 10){
    return x * 3;
  }
  return x << 1;
}
```

Listing 7: Result (SAW:llvm_symexec)

```
$saw llvm_symexec.saw
Loading file "llvm_symexec.saw"
Running reference_function
Finished running reference_function
Running implementation_function
Finished running implementation_function
prove: 1 unsolved subgoal(s)
Invalid: [x = 10]
```

Listing 8: Result (SAW:crucible_llvm_verify)

```
$saw crucible_llvm_verify.saw
Loading file "crucible_llvm_verify.saw"
Subgoal failed: @reference_function safety assertion:
 literal equality postcondition
SolverStats {solverStatsSolvers = fromList ["ABC"],
 solverStatsGoalSize = 60}
----------Counterexample----------
("x",10)
user error ("crucible_llvm_verify"
(crucible_llvm_verify.saw:8:1-8:21):
Proof failed.)
```

For these cases, when x equals to 10, the behavior differs. SAW correctly shows the counter-examples. This is the most useful advantage of SAW.

2.2.2 ABC

A user of SAW can analyze the LLVM using symbolic execution. The result of the execution is stored in an AIG (And-Inverter Graphs) [16]. AIG data can be verified by a theorem prover called ABC [17]. ABC is especially suited for equivalence checking [19] between two functions represented in AIG. ABC is the default solver for SAW.

2.3 SeaHorn

SeaHorn [4] also verifies C program code using LLVM. SeaHorn has the following features.

- It is easy to use because a programmer can directly write assertions in the code. The notation is based on a simple notion of Design by Contract [20].
- Learning times for the tool are shorter than for other formal based tools.

2.3.1 Verification Example using SeaHorn

Listing 9 shows an example of verification using SeaHorn.

Listing 9: Verification script for twice program(seahorn)

```
#include "seahorn/seahorn.h"
extern int nd(void);

// code under verification
unsigned int implementation_function(unsigned int x){
  return x << 1;
}

int main(){
  int x, val;

  x = nd();
  val = nd();
  val = implementation_function(x);

  // assrtion
  sassert(val == x * 2 );
}
```

Here, function nd stands for non-deterministically, and returns an arbitral value. sassert(P) states that P is true.

Listing 10 shows the result of the verification.

Listing 10: Result (SeaHorn)

```
$sea pf double.c --show-invars
   -- omit --
unsat
Function: main
main@entry: true
main@entry.split: true
```

Note that SeaHorn usually checks unsatisfiability. In other words, unsat is printed if and only if the assertion holds in SeaHorn.

2.4 SAT/SMT solvers

Recently a number of efficient SAT (SATisfiability problem) solvers are emerging and these solvers prove many constraint based problems. A number of satisfiability problem is usually given as a set of clauses, where each clause is a logical disjunction of Boolean variables. The set of clauses is evaluated as a logical conjunction of clauses. Therefore, the set can be evaluated as satisfiable or unsatisfiable. Satisfiability problem is known as NP-complete. However, SAT solvers efficiently proves most of instances.

SMT (Satisfiability Modulo Theories) is an extension of SAT. Each Boolean variable is substituted with inequality expressions over integers or reals. Several classes are known for SMT. Some of these classes are decidable and there are tools which can efficiently proves the satisfiability of these expressions.

2.4.1 Z3

Z3 [6] is one of the famous SMT solvers developed by Microsoft Research. In SMT-COMP, an SMT solver competition, it has excellent results every year. It is one of the built-in SMT solvers by SAW.

2.4.2 CVC4

CVC4 [8] is one of the CVC (Cooperating Validity Checker) series used by the theorem proving system, SVC developed by Stanford University. At SMT-COMP 2017, it won in many divisions.

2.4.3 Yices

Yices [7] is an SMT solver developed at SRI and was upgraded as Yices 2 in 2009. It also had excellent results at SMT-COMP 2017. It is one of the built-in SMT solvers in SAW.

2.4.4 MathSAT

MathSAT [9] is an SMT solver which supports a wide range of theories, such as equality and uninterpreted functions, linear arithmetic, bit-vectors, and arrays. It also supports the computation of Craig interpolants, extractions of unsatisfiable cores, and the generation of models and proofs.

2.4.5 SMT-RAT

SMT-RAT [10] is an SMT solver that can perform parallel processing written in C ++. Since It is not built in as standard in SAW, it is necessary to output a file such as smtlib2 to use it.

2.4.6 minisat

Minisat[11] is one of the representative SAT solvers. It has the minimum set of functions as a SAT solver, and its source code is about 2000 lines. Because SAW does not built in as standard, it is necessary to apply minisat after outputting in Conjunctive Normal Form (CNF) or AIG [16] format file format.

3 OUR PROPOSED METHOD

Program codes with recursive data structures have a loop structure which has a termination condition. The termination condition depends on the recursive data structures, thus we cannot put a bound of the number of iterations to a fixed finite value. For this reason, verification on such a program code faces the so-called termination problem.

In other words, verification on program code with recursive data structures is essentially undecidable.

In order to overcome this problem, in practice, we usually approximate the problem by introducing the idea of bounded verification.

Our proposed method also uses bounded verification by bounding the size of recursive data structures.

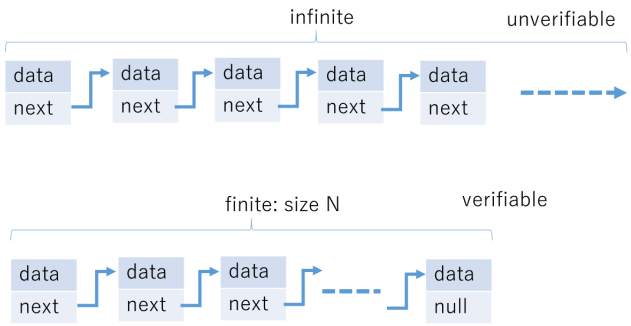

Figure 3: Bounded Model Checking

The upper half of Fig. 3 shows that verification does not end due to infinite size of the list, while the lower half of Fig. 3 shows that verification terminates due to the specifying of a limit of size.

3.1 The Concept

Bounded verification usually terminates iterations of a loop body by a fixed value. We limit the size of recursive data structures. This is essentially the same idea of the usual bounded model checking approaches.

For example, let us consider a linear list. We fixed the size of the list namely n. We produce a verification script for every pattern of the data structure with in size n (See Fig. 3).

Then we perform each formal verification for the produced scripts.

For example, if we choose 100 as n, then we perform formal verification with size 1 to 100 of the linear list. If all of the verification passed, we strongly assume that the program is valid for any size of a list.

The scheme has the advantage that we can choose any feasible value of n, but as we observe, the verification time becomes large as n becomes large.

For every pattern, we perform each formal verification for the produced script.

We use the above idea with the crucible_llvm_verify package for SAW, and evaluated the effectiveness of our proposed method.

In a similar way, for fixed value n, we produce every pattern of binary trees with size 1 to n, where n is the number of nodes in the binary tree. Figure 4 shows every pattern of the binary trees with a size of 3.

3.2 Verification Targets

We perform experiments for the following three data structures.

- Two-level nests
- Linear lists with recursive definition
- Binary trees with recursive definition

In this section, we show every program under verification.

3.2.1 Two-level nests

The program for calculating the summation of 32-bit integers in the parent and children nodes of a two-level nest is shown in Fig. 5.

Listing 11 shows the data structure.

Listing 11: Two-level nest

```
typedef struct s {
    int a;
} s_t;
//parent
typedef struct t {
    int x;
    s_t n;
} t_t;
//reference function
int f_ref(t_t *p) {
    return (p->n).a + p->x;
}
//implementation
int f_imp(t_t *p) {
    return  p->x + (p->n).a;
}
```

The difference between the reference function and the implementation is trivial. We simply change the left and the right terms.

3.2.2 Linear List

The program in Fig. 6 calculates the summation of 32-bit integers in every cell of a linear list.

Listing 12 shows the data structure of the program.

Listing 12: Linear List

```
struct NODE{
  uint32_t val;
  struct NODE* next;
};
typedef struct NODE* node_t;
// reference function
int linear1(node_t node){
  if(node->next == NULL){
    return node->val;
  }else{
    return node->val + linear1(node->next);
  }
}
// implementation
int linear2(node_t node){
  if(node->next != NULL){
    return node->val + linear2(node->next);
  }else{
    return node->val;
  }
}
```

The difference between the reference function and the implementation is in the form of the if statement.

3.2.3 Binary Trees

The program in Fig. 7 calculates summation of 32-bit integers in every node of a binary tree.

Listing 13 shows the data structure of the program.

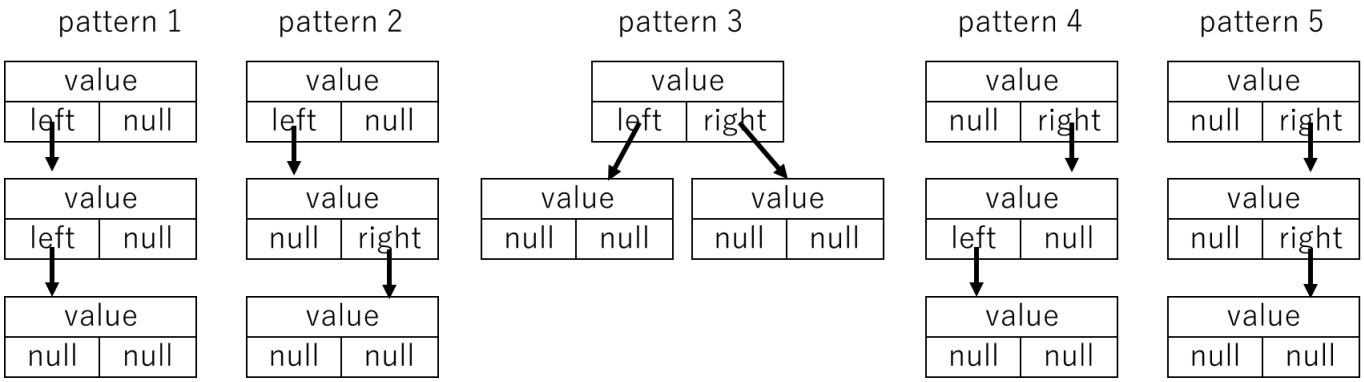

Figure 4: Patterns of Binary Trees

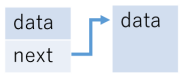

Figure 5: Two level nest

Figure 6: Linear List

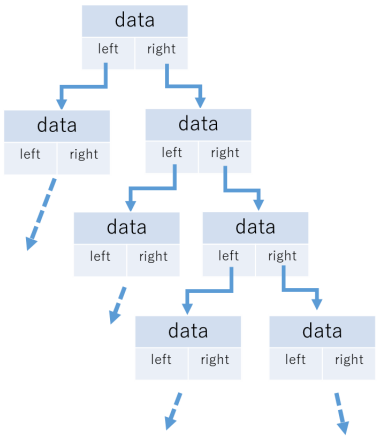

Figure 7: Binary Tree

Listing 13: Binary Tree

```
// node
struct BTREE {
    uint32_t val;
    struct BTREE* left;
    struct BTREE* right;
};
// reference function
uint32_t pre_order(struct BTREE* tree){
    if(tree == NULL){
        return 0;
    }
    return pre_order(tree->left) +
    pre_order(tree->right) + tree->val ;
}
// implementation
uint32_t in_order(struct BTREE* tree){
    if(tree == NULL){
        return 0;
    }
    return in_order(tree->left) +
    tree->val + in_order(tree->right);
}
```

The difference between the reference function and the implementation is the order of traversal. Thus, the difference is not trivial.

4 EXPERIMENTAL EVALUATION

The experiment environment is summarized as follows.

- OS: Windows 10 64-bit
- CPU: Intel Core i7-4500U CPU @ 1.80GHz 2.39GHz
- Memory: 8.00 GB
- Docker
  - version: 18.01.0-ce
  - Memory: 4096 MB
  - The number of CPUs: 2
- SAW: 0.2 (2018-01-31)
  - LLVM: 3.8.0
  - Z3: 4.5.0
  - Yices: 2.5.2
  - minisat: 2.2.0
  - SMT-RAT: 2.1.0
- SeaHorn: 0.1.0-rc3
  - LLVM: 3.6.0

4.1 Comparison of SMT Solvers: EXP 1

We evaluate the verification results and CPU execution times using the built-in function “Output a file for SAT/SMT solver”

of SAW. For ABC, we use a proof package named crucible_llvm_verify. For other SAT/SMT solvers, we use a proof package named llvm_symexec.

4.2 Comparison of SAW and SeaHorn: Exp 2

We evaluate the verification results and CPU execution times using SAW and SeaHorn. We use a proof package named crucible_llvm_verify.

4.3 THE RESULTS

For all results, T/O specifies that verification time is over 3,600 sec. The '–' symbol shows a failure of verification. The unit of time is second.

4.3.1 Comparison of SAT/SMT solvers

Table 1 summarizes the verification of Two-level nest with varying SAT/SMT solvers.

Table 2 summarizes the verification of Linear Lists of size 100 with varying SMT/SAT solvers.

Table 3 summarizes the verification of Binary Trees of depth 5 with varying SAT/SMT solvers.

4.3.2 Comparison with SAW and SeaHorn

Table 4 summarizes verification of Two-level nests using SAW and SeaHorn.

Table 5 summarizes the verification of Linear Lists of size 100 using SAW and SeaHorn.

Table 6 summarizes the verification of Binary Trees of depth 5 using SAW and SeaHorn.

Table 1: Results for Two-level nests

SMT/SAT	ABC	Z3	Yices	minisat	SMTRAT
CPU time	1.06	1.23	1.47	1.24	T/O

Table 2: Results for Linear List

SMT/SAT	ABC	CVC4	Z3	Yices	Mathsat
CPU time	6.981	1.132	59.110	307.307	–

Table 3: Results for Binary Tree

SMT/SAT	ABC	CVC4	Z3	Yices	Mathsat
pattern 1	0.685	0.502	0.484	0.471	0.571
pattern 2	0.682	0.530	0.483	0.467	0.575
pattern 3	0.673	0.491	0.478	0.469	0.604
pattern 4	0.649	0.490	0.465	0.469	0.580
pattern 5	0.646	0.489	0.461	0.501	0.570

Table 4: Results for Two-level nests (SAW and SeaHorn)

Verification Tool	SAW	SeaHorn
CPU Time	1.06	0.104

t!

Table 5: Results for Linear List (SAW and SeaHorn)

Verification Tool	SAW	SeaHorn
CPU time	1.47	–

Table 6: Results for Binary Tree (SAW and SeaHorn)

Verification Tool	SAW	SeaHorn
CPU time	0.51	–

5 DISCUSSION

We verified a variety of code structures by applying bounded verification to functions that deal with structures including recursion. We can verify two-level nest and linear list structures correctly using crucible_lvm_verify. SeaHorn can only verify two-level nests. It was not possible to verify the binary tree structure by all verification methods.

In Exp1, the verification succeeded when we used the "crucible_lvm_verify" package.

In Exp2, it was possible to verify Linear List structures with 100 elements. When we investigated the maximum number of elements that package could be handled, the number of elements was about 5000. In a realistic verification, since sufficient verification can be performed even with the list structure up to 1000 elements, the bounded verification method can be applied. For binary tree structures, we can also obtain good results.

On the other hand, verification using SeaHorn needs less verification time for Two-level nest about one tenth of SAW. Therefore, it is superior to the SAW in view of the verification time. However, when trying to verify a program dealing with recursive structure, recursive functions. it automatically skips the analysis of the structures. Thus, it is impossible to handle programs containing recursive structures. As a result, it can be said that SAW that can handle of recursive structures is superior to that of SeaHorn at the present.

6 CONCLUSION

This paper proposed a new method for Formal Conformance Verification based on bounded model checking for programs with recursive data structures. We also conducted an experimental evaluation using SAW. We showed that the proposed method works well for a simple program which deals with calculations over a linear list.

In future work, we want to improve the performance for binary trees and other complex data structures.

ACKNOWLEDGMENTS

Funding from Mitsubishi Electric Corp. is gratefully acknowledged. The authors wish to thank Mr. Maruyama for his work in the preliminary experiments. The research is being partially conducted as Grant-in- Aid for Scientific Research C (16K00094) and C (17K00111).

REFERENCES

[1] E. Clarke, A. Biere, R. Raimi, Y. Zhu: "Bounded Model Checking Using Satisfiability Solving," Formal methods in system design, Vol.19 Issue 1, pp.7-34 (2012).
[2] T. Liu, M. Nagel, and M. Taghdiri, "Bounded Program Verification using an SMT Solver: A Case Study," Proceedings of the 5th International Conference on Software Testing, Verification and Validation, pp.101-110 (2012).
[3] R. Dockins, A. Foltzer, J. Hendrix, B. Huffman, D. McNamee, and A.Tomb: "Constructing Semantic Models of Programs with the Software Analysis Workbench," Proceedings of VSTTE 2016 (2016).
[4] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas: "The SeaHorn Verification Framework," International Conference on Computer Aided Verification, pp.343-361 (2015).
[5] J. E. Hopcroft, R. Motwani, and J. D. Ullman: "Introduction to Automata Theory, Languages, and Computation (2nd Edition)," Addison Wesley (2000).
[6] L. de Moura and N. Bjørner: "Z3: An efficient SMT solver," Proceedings of TACAS 2008, LNCS Vol. 4963, pp.337-340 (2008).
[7] B. Dutertre: "Yices 2.2," Proceedings of CAV2014, LNCS Vol.8559, pp.737-744 (2014).
[8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli: "CVC4," Proceedings of the 23rd international conference on Computer aided verification (CAV'11) pp. 171-177 (2011).
[9] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani: "The MathSAT5 SMT Solver," Proceedings of TACAS 2013, LNCS Vol. 7795, pp.93-107 (2013).
[10] F. Corzilius, G. Kremer, S. Junges, S. Schupp, and E. Abraham: "SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving," Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT 2015) pp.360-368 (2015).
[11] N. Een, A. Mishchenko, and N. Sörensson: "Applying Logic Synthesis for Speeding Up SAT," Proceedings of the 10th International Conference on Theory and applications of satisfiability testing pp. 272-286 (2007).
[12] A. Biere, M. Heule, H. Van Maaren, and T. Walsh: "Handbook of Satisfiability," IOS press (2009).
[13] K. Okano, S. Harauchi, T. Sekizawa, S. Ogata, and S. Nakajima: "Equivalence Checking of Java Methods — Toward Ensuring IoT Dependability —," Proceedings of the 26th International Conference on Computer Communications and Networks, pp.1-6 (ICCCN 2017) (August 2017).
[14] C. Belo Lourenco, Si-Mohamed Lamraoui, S. Nakajima, and J. Sousa Pinto: "Studying Verification Conditions for Imperative Programs," Proceedings of 15th International Workshop on Automated Verification of Critical Systems, AVoCS'15 (2015).
[15] Si-Mohamed Lamraoui, S. Nakajima: "A Formula-based Approach for Automatic Fault Localization of Multi-fault Programs," Journal of Information Processing, Vol.24, No.1, pp.88-98 (2016).
[16] A. Darringer, W. H. Joyner, Jr., C. L. Berman, and L. Trevillyan: "Logic synthesis through local transformations," IBM Journal of Research and Development, Vol.25 (4), pp.272-280 (1981).
[17] R. Brayton and A. Mishchenko: "ABC: An Academic Industrial-Strength Verification Tool," LNCS Vol.6174, pp.24-40 (2010).
[18] C. Barrett, A. Stump and C. Tinelli: "The SMT-LIB Standard Version 2.0," (2010).
[19] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai: "Robust boolean reasoning for equivalence checking and functional property verification," IEEE Transaction on CAD, vol.21 (12), pp.1377-1394 (2002).
[20] B. Meyer: "Object-Oriented Software Construction, second edition," Prentice Hall (1997).

(Received April 4, 2019)

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 Computer Science and 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.

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

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.

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.