Published in final edited form as:

Int Conf Dependable Syst Netw Workshops. 2004 ; 2004: 399–404. doi:10.1109/DSN.2004.1311909.

Does Your Result Checker Really Check?

Lan Guo, Supratik Mukhopadhyay, and Bojan Cukic

Lane Department of CSEE, West Virginia University, Morgantown, WV 26506-6109, USA

Lan Guo: lan@csee.wvu.edu; Supratik Mukhopadhyay: supratik@csee.wvu.edu; Bojan Cukic: cukic@csee.wvu.edu

Abstract

A result checker is a program that checks the output of the computation of the observed program for correctness. Introduced originally by Blum, the result checking paradigm has provided a powerful platform assuring the reliability of software. However, constructing result checkers for most problems requires not only significant domain knowledge but also ingenuity and can be error prone. In this paper we present our experience in validating result checkers using formal methods. We have conducted several case studies in validating result checkers from the commercial LEDA system for combinatorial and geometric computing. In one of our case studies, we detected a logical error in a result checker for a program computing max flow of a graph.

1. Introduction

As software systems are increasingly being deployed in mission critical applications, it has become imperative that they operate reliably in accordance with the requirements. The past decade has witnessed a lot of research activity aimed at ensuring that software systems perform exactly the tasks for which they are designed. Research in software reliability has proceeded in four distinct directions: dynamic analysis based on testing, static analysis based on formal methods, fault tolerance and runtime monitoring based on result checking.

One of the most widely used techniques for ensuring software reliability is dynamic analysis based on software testing. In the paradigm of testing, software is debugged using *testing suites*: one runs a program on a variety of carefully selected inputs, identifying a bug whenever the program fails to perform correctly. Two important questions are left unanswered here. First, how does one know whether or not the program performs correctly? Usually this is accomplished by using an *oracle*. The oracle may be constructed from the program specification or may be written manually by the programmer. Both approaches require considerable amount of painstaking manual effort (although to some extent specification-based test oracle generation can be performed automatically) [21]. The second question concerns the completeness of the approach. Given that a test suite feeds a program only selected inputs out of the often enormous space of possibilities, how does one ensure that every bug in the code will be surfaced? Indeed, particular combinations of circumstances leading to a failure may well go undiscovered [14]. Besides, it is difficult for a testing suite to accurately simulate an input distribution which the program will encounter

in its lifetime. Thus, a supposedly debugged program may in fact fail quite frequently. Furthermore, while testing can show the presence of bugs, it cannot prove that high level logical properties required by the system have been satisfied.

One alternative to testing is formal methods based static analysis of software. Using this methodology, one can prove (once and for all) that a system implementation correctly meets its specification. Not only is constructing such a proof unexpectedly hard, but also there exists a gap between the abstract model that the proof works on and the physical system. While on one hand this gap makes formal methods effective and useful [23] by abstracting away unnecessary details, on the other hand the software components of an implementation can be much more complex and detailed than the corresponding components of the abstract model and can fail in unexpected ways that cannot be explained in the abstract model. In addition, it is common to make assumptions about the hardware in a software specification, which may not be always met in the implementation. Formal verification does not and can not address these sources of failure [18].

A third alternative for ensuring software assurance is fault tolerance. In this methodology, the reliability of software is enhanced by having several groups of programmers creating separate versions of the same software. At run time, all of the versions are executed and their outputs are compared. It is evident that such an approach is grossly inefficient not only in terms of programming manpower but also in terms of increased run time and hardware resources required.

In this paper we restrict ourselves to the fourth approach: runtime monitoring by result checking [22]. Result checking is a technique inspired by the field of error correcting codes to ensure the correct functioning of software by monitoring the functioning of the system at runtime. The system output is monitored at run time by embedding result checkers into the software itself. The checkers monitor the system output and each time the output is incorrect, warnings are issued and informative output is written to a file that will be periodically reviewed by software maintenance engineers. Such an automatic system for identifying bugs should in fact reduce the probability that bugs will be ignored or forgotten. Checkers can catch errors in program components whose effects may not be readily apparent to the user. Thus a checker might well identify a fault in a critical system before it goes on to cause a catastrophic failure. Checkers can be used to trigger self-correction: potentially a method for the creation of extremely reliable systems. Unlike the other approaches to software reliability, result checking can reveal incorrect output arising from any cause: software faults, hardware faults or transient run time errors.

There are some problems in result checking. Firstly, constructing result checkers for even simple programs is unexpectedly hard and can be error prone. A bug in a result checker can be more disastrous than one in the program it is checking, since the user is led to believe a buggy output of a program as correct when certified by a (buggy) result checker. Hence using result checkers in a software system indiscriminately without validating them might produce disastrous results. Secondly, most monitors proposed do not prove that the system is functioning correctly, since they test only necessary but not sufficient conditions for correct functioning [18]. However, the good news is that result checkers are typically much smaller

than the original program (which might consist of millions of lines of code) and hence might be more amenable to formal validation than the original program. Formally validating the result checkers in a software system amounts to providing a formal certification to the entire system at a “low cost”. Furthermore, verifying the correctness of result checkers increases the possibility of discovering bugs in a system at runtime.

In this paper, we combine the best of both the worlds of formal verification and result checking. More specifically, we use automatic proof tools from formal methods to validate result checkers. Our methodology roughly consists of first specifying both the result checkers and the properties they are supposed to satisfy in a common semantic framework: in our case the common semantic framework is a suitable monadic second order logic; a decision procedure for monadic second order logic can then be used to determine if the specification of a result checker entails the properties it is supposed to satisfy. The Mona [16] tool implements a decision procedure for monadic second order logic. However, the input language for Mona is too low level for expressing a high level algorithm. FMona [13] is a high level language for describing WS1S (weak monadic second order logic over one successor) formulas. We specify both the result checker and its properties in the FMona language. The FMona compiler translates programs (i.e., formulas) written in the FMona language to the Mona input language (FMona thus acts as a front end to Mona). The Mona tool can then be invoked to validate the formulas. If the specification of a result checker does not entail the properties it is supposed to satisfy, Mona comes up with a counterexample that can be used for debugging purposes.

We have conducted several case studies in validating result checkers for the commercial LEDA package [19]. Due to space constraints, we present only two of them which we find most instructive. In one case study, we detected a subtle logical error in the result checker for computing maximum flow of a graph, which has not been reported before.

The remainder of this paper is organized as follows: section 2 introduces related work. Section 3 provides preliminaries of our method and tools. Section 4 illustrates two case studies in validating result checkers. Section 5 presents the experimental results. Section 6 concludes the paper.

2. Related Work

Result checkers [4] are programs that check if program P under test has performed correctly on specific inputs at runtime. Result checkers can remain in a program’s code throughout its lifetime, providing a runtime warning of incorrect output arising from any cause. The field of result checking provides an efficient means for assuring the correctness of hardware/software computations [22]. For a program P implementing an algorithm that runs in $\mathcal{O}(f(n))$ in the worst case, its result checker must run in $\mathcal{O}(f(n))$ so that the asymptotic complexity of the program together with the result checker remains the same as that of the original program. Such constraints make the task of constructing result checkers an extremely daunting and error prone one that requires not only great ingenuity but also significant knowledge in the problem domain. Since its introduction by Blum, result checking has enjoyed immense success in a variety of applications ranging from software for geometric

computation [19], distributed protocols [2], computing cryptographic functions [12], program classification [9], to certifying correctness of memories [5]. Result checkers have been deployed extensively in commercial software packages like LEDA [19].

Mona is a low level language for describing WS1S/M2LStr (Monadic Second Order Logic over Strings) formulas [3, 16]. The Mona tool converts WS1S/M2LStr formulas into equivalent finite state automata and decides if those formulas are valid. Mona is considered as an automatic proof tool [8] and has been applied to verification of hardware circuits [17] as well as safety and liveness properties of a parameterized distributed system [13]. Its other applications include automatically tracing pointer usage [15], extending Yacc for specifying constraints in regular tree languages [10], and automatically verifying the safety of C programs [11, 20].

FMona is a high level language for describing WS1S (Weak monadic second order logic over one successor) [6]. It acts as a frontend to the Mona tool. The iterative and abstraction methods expressed in FMona were applied for the validation of infinite or parameterized space problems [7, 8]. The detailed syntax of FMona language was described in [8].

A combination of the automatic proof tool Mona and higher level aspects FMona makes the expression of validation methods easy. We choose this approach for our methodology instead of model checkers or theorem provers, because [1, 8]:

- theorem provers require considerable manual effort. They require the knowledge of the underlying type theory, the proof tactics, the tactic languages and the underlying decision procedures;
- model checkers are generally easy to use but can only deal with simple data structures. In particular model checkers cannot be directly used to validate parameterized systems [1] or systems involving unbounded data structures. In our case unbounded data structures can be expressed in the LEDA language in which the result checkers are written, e.g.,

```
forall_edges(e,G)
if ( f[e] < 0 && f[e] > cap[e] )
```

where the size of the graph G is unbounded. Hence model checkers are not suitable for validating LEDA result checkers;

while the Mona tool implements a decision procedure for monadic second order logic. Additionally, the FMona language allows definition of enumeration and record with update types and quantifying over them. Furthermore, it allows the definition of higher order macros parameterized by types and predicates that allows us to define parameterized systems with possibly unbounded data structures.

3. Preliminaries

Result checkers are embedded in a software product to ensure the correctness of its output at runtime. However, as discussed above, constructing result checkers for even simple programs is a notoriously hard and error prone job. A bug in a result checker can be more disastrous than one in the program it is checking, since the user is led to believe a buggy output of a program as correct when certified by a (buggy) result checker. In order to ensure the correctness of result checkers, we provide a framework based on tools from formal methods to validate them. Specifically, we use a combination of the FMona language and the Mona tool to validate result checkers.

3.1 Result Checking

The result checkers we validated are from the LEDA system [19]. LEDA (Library of Efficient Data Types and Algorithms) is a C++ based library of combinatorial and geometric data types and algorithms. From a user's point of view, LEDA is a platform for combinatorial and geometric computing. It provides *algorithmic intelligence* for a wide range of applications. It eases a programmer's life by providing powerful and easy-to-use data types and algorithms which can be used as building blocks for larger systems. Stacks, queues, graphs etc. are built-in data types in LEDA. LEDA has been used in diverse areas such as code optimization, VLSI design, robot motion planning, traffic scheduling, machine learning and computational biology. The LEDA system is currently a commercial package marketed by *Algorithmic Solutions GmbH (AS)* and is currently installed at more than 1500 sites worldwide. It provides a result checker for each major algorithm in combinatorial and geometric computing. We validated the result checkers for several graph algorithms from the LEDA book [19]. Due to space constraints, we summarize our experience in validating two of them which we find most instructive: the result checkers for computing *Maximum Bipartite Cardinality Matching* and *Maximum Flow*.

In order to describe the result checker for computing maximum bipartite matching, we will provide the definition of a bipartite graph first. For a bipartite graph $G = (V, E)$, there is a partition $V = A \cup B$ of the nodes of G that every edge of G has one endpoint in A and one endpoint in B . A *matching* M is a subset of the edges no two of which share an endpoint. The cardinality $|M|$ of a matching M is the number of edges in M . A *node cover* is a set U of nodes such that for every edge (v, w) of G at least one of the endpoints is in U . For a general matching, $|M| \leq |U|$. If $|M| = |U|$, then M is a maximum cardinality matching. This equation is the basis for the checker of a program computing maximum matching of a bipartite graph. The checker first takes a set M of edges and a set NC of nodes, and then checks that M is a matching (no two edges share an endpoint), NC is a node cover (each edge has at least one endpoint covered), and that the cardinality of M is equal to the cardinality of NC [19].

For checking maximum flow, let $G = (V, E)$ be a directed graph, let s and t be distinct vertices in G called the *source* and the *sink*. For an edge e , $cap(e)$ is called the *capacity* of e . A *flow* function f satisfies the capacity constraints and the flow conservation constraints. The capacity constraints state that the flow across any edge is bounded by the capacity of the edge; and the flow conservation constraints state that for every node other than s and t , the total flow out of the node is equal to the total flow into the node. For a node v , the *excess* of

v is defined as $excess(v) = \sum_{e, target(e)=v} f(e) - \sum_{e, source(e)=v} f(e)$. Flow conservation states that all nodes except for s and t have zero excess.

Based on those constraints, the checker examines the capacity condition for each edge and computes the excess of all nodes. All nodes but s and t must have excess equal to zero. It then uses breadth-first search to compute the set of nodes reachable from s in the residual graph; t must not be reachable. The checker from [19] for capacity constraints is shown in Figure 1.

3.2 FMona

FMona [6] is a high level interface to Mona. It provides high level aspects for expressing validation methods, and then translates the high level description into low level Mona syntax. Therefore, the expression of validation methods becomes easy. It is possible to define enumeration and record with update types and quantify over them in FMona syntax. Furthermore, it allows the definition of higher order macros parameterized by types and predicates.

The FMona compiler translates the methods and properties into Mona formulas and Mona acts as an automatic proof tool to validate them. In our methodology, we write the result checkers and properties in FMona, and the FMona compiler translates the programs into Mona language for validation.

3.3 Mona

Mona is a logic-based programming language and a tool that decides if programs (formulas) are valid in WS1S. The decision procedure is based on translating WS1S formulas to finite automata. Although the complexity of the decision procedure for WS1S implemented in Mona is nonelementary, Mona is known to perform well in practice [16]. It deals with the state explosion problem by using BDDs (Binary Decision Diagrams) to represent the state space symbolically.

We validate a result checker by checking if its specification entails the properties it is supposed to satisfy; i.e., if $Result\ Checker \wedge \neg Property$ is unsatisfiable. The counter example generated by Mona can also help to trace the error.

4. Validating Result Checkers

In this section, we present two of the case studies in validating result checkers that we conducted. In both cases, the result checkers are taken from the commercial LEDA package.

4.1 Validating the Checker for Computing Maximum Bipartite Cardinality Matching

The checker for computing maximum bipartite matching first takes a set M of edges and a set NC of nodes, and then checks that M is a matching (no two edges share an endpoint), NC is a node cover (each edge has at least one endpoint covered), and that the cardinality of M is equal to the cardinality of NC [19]. We translate the checker algorithm for maximum bipartite matching [19] into a FMona program. *Graph*, *NodeCover*, as well as *Matching* are each defined as a record type, respectively. A graph G , a node cover NC , and a matching M

are each defined as a variable of the corresponding record type. The syntax of FMona programs is very similar to C language.

Each negation of the condition for maximum bipartite cardinality matching is also expressed in FMona: M is not a matching; NC is not a node cover; and $|M| \neq |NC|$. Each negation of condition is conjuncted with the checker properties respectively. FMona compiler translates the programs into Mona formulas and the Mona tool is invoked at the same time to validate them. Three formulas of the conjunction are all invalid, which proves that the checker is correct.

4.2 Validating the Checker for Computing Maximum Flow

The checker for a program computing maximum flow of a graph examines the capacity condition for each edge and computes the excess of all nodes. All nodes but the source s and the sink t must have excess equal to zero. It then uses breadth-first search to compute the set of nodes reachable from s in the residual graph; t must not be reachable. A FMona program is written according to the checker algorithm for maximum flow in [19]. The FMona program is separated into two files, because the automata generated from one file is too large for the Mona tool. The capacity condition checking and the excess condition checking are put into one file, while the queue implementation and the reachability checking are put into another file.

There is one problem to implement the excess checking in FMona. The general sum arithmetic operation of two variables cannot be expressed. Therefore, we first express the sum of a variable with a constant in FMona, and use the FMona tool to generate Mona code. We then change the Mona code to add and compare two secondary order variables by using the Pressburger arithmetic predicates provided in Mona. A part of the modified Mona program is shown as follows.

```
pred check_flow(var2 G$n$node,
var2 G$n$excess, var0 G$n$reached,
var2 G$s$node, var2 G$s$excess,
var0 G$s$reached, var2 G$t$node,
var2 G$t$excess, var0 G$t$reached,
var2 G$e$source$node, var2
G$e$source$excess, var0
G$e$source$reached, var2
G$e$target$node, var2 G$e$target$excess,
var0 G$e$target$reached,
var2 G$e$cap, var2 G$e$f) =
~(((ex2 X: (X=pconst(0)&less(G$e$f,X)))
& less(G$e$cap,G$e$f) ));
```

When we validate the conjunction of the checker translated from the LEDA book with negation of the flow condition, the Mona output does not give us “*The formula is invalid*”.

Therefore, there is an error in the original algorithm for checking the flow condition. We found the error that the conjunction operation (&) should be a disjunction operation (||) (Please see Figure 1 for reference). This error results in checking only necessary conditions, instead of sufficient conditions, for the correct flow function. After we corrected the code, the formula for the conjunction of the checker with negation of the flow condition is invalid. Hence, the algorithm for checking the flow condition is right after the correction.

5. Experimental Results

We validated the checker for computing maximum bipartite cardinality matching and proved that it is correct. During the validation of the maximum flow checker, we detected a bug in the flow condition checking, and proved that the excess checking is correct. However, there is a state explosion problem in validating the reachability checking by breadth-first search. The BDD is too large (>16777216 nodes). It aborts with 69% of automaton completed. Generally, the validation process is time efficient. It takes less than a couple of minutes to generate Mona programs and complete the validation. The detailed execution time of the whole validation process, the size of the automata, and the number of the BDD nodes generated are listed in Table 1. All experiments were conducted in a Linux machine with a 1.2GHz AMD Athlon processor, 36.2GB of hardware drive size, and 1GB of RAM.

6. Conclusions

In this paper, we present our experience in validating result checkers by automatic proof tools. Our method can detect logical errors in result checkers. In particular, it can prove that result checkers test sufficient conditions, instead of only necessary conditions, for correct functioning. To our knowledge, this is the first attempt to combine result checking with automatic proof tools. We have conducted several case studies in validating result checkers for the commercial LEDA system. Our technique has been able to detect a subtle logical error in one of these result checkers, which has not been unveiled previously. The experimental results indicate that our approach is viable and efficient for combining formal verification with checking to achieve higher confidence in the correctness of software systems. We believe that a combination of result checking along with formal methods can provide a robust framework for ensuring software reliability.

Acknowledgements

Jean-Paul Bodeveix provided valuable guidance in our study of FMona.

References

1. Apt KR, Kozen D. Limits for Automatic Verification of Finite-State Concurrent Systems. *Inf. Process. Lett.* 1986; 22(6):307–309.
2. Awerbuch, B.; Varghese, G. Distributed Program Checking: A Paradigm for Building Self Stabilizing Distributed Protocols; FOCS'91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science; 1991. p. 258-267.
3. Biehl M, Klarlund N, Rauhe T. Mona: decidable arithmetic in practice. *FTSRTS'96*. 1996
4. Blum, M.; Kannan, S. Designing programs that check their work; *Proc. 21st ACM Symposium on Theory of Computing*; 1989.

5. Blum M, Evans WS, Gemmell P, Kannan S, Naor M. Checking the Correctness of Memories. IEEE Symposium on Foundations of Computer Science. 1991:90–99.
6. Bodeveix J-P, Finali M. The FMONA tool. 1999 May. <http://www.irit.fr/ACTIVITIES/EQ/COS/MF/FMONA, IRIT>.
7. Bodeveix, J-P.; Finali, M. A tool for expressing validation techniques over infinite state systems; Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00); 2000.
8. Bodeveix J-P, Finali M. Experimenting acceleration methods for the validation of infinite state systems. Workshop on Distributed System Validation and Verification (ICDCS'00). 2000:23–30.
9. Collberg C, Proebstring T. Program Classification using Program Checking. Fun with Algorithms. 2001
10. Damgaard N, Klarlund N, Schwartzbach MI. Yakyak: Parsing with logical side constraints. DLT'99. 1999
11. Elgaard J, Møller A, Schwartzbach MI. Compile-time debugging of c programs working on trees. ESOP'2000. 2000
12. Frankel Y, Gemmell P, Yung M. Witness-based Cryptographic Program Checking and Robust Function Sharing. Proceedings of STOC'96. 1996
13. Henriksen JG, Jensen J, Jorgensen M, Klarlund N, Paige R, Rauhe T, Sandholm A. Mona: Monadic second-order logic in practice. TACAS'95. 1995
14. Huang JC. An Approach to Program Testing. ACM Computing Surveys. 1975 Sept.8(3):113–128.
15. Jensen JL, Jorgensen ME, Schwartzbach MI. Automatic verification of pointer programs using monadic second-order logic. PLDI'97. 1997
16. Klarlund N, Møller A. Mona version 1.4 user manual. 2003 <http://www.brics.dk/mona/papers.html>.
17. Klarlund N, Nielsen M, Sunesen K. Automata based symbolic reasoning in hardware verification. Formal Methods in System Design. 1998; 13:255–288.
18. Lee I. Formal Verification, Testing and Checking of Real-Time Systems. ACM Computing Surveys. 1996 Dec.28(4es):a182.
19. Mehlhorn, K.; Näher, S. LEDA A Platform for Combinatorial and Geometric Computing. Cambridge University Press; 1999. <http://www.mpi-sb.mpg.de/mehlhorn/LEDAbook.html>
20. Møller A, Schwartzbach MI. The pointer assertion logic engine. PLDI'2001. 2001
21. Richardson DJ, Leif Aha S, Malley TO. Specification-based Oracles for Reactive Systems. Proceedings of ICSE'92. 1992
22. Rubinfeld R. Designing Checkers for Programs that Run in Parallel. Algorithmica. 1996; 15(4): 287–301.
23. Rushby JM, von Henke F. Formal Verification of Algorithms for Critical Systems. IEEE Trans. Software Eng. 1993 Jan.19(1):13–23.

```
<max flow check>:
...
bool CHECK_MAX_FLOW_T(const graph& G,
    node s, node t, const edge_array<NT>&
    cap, const edge_array<NT>& f)
{node v; edge e;
  forall_edges(e, G)
    if ( f[e] < 0 && f[e] > cap[e] )
      return False_MF("illegal flow value");
... }
```

Figure 1.
The LEDA result checker for computing maximum flow

Table 1

Experimental Results

Formula validated	Execution time	Num. of states	Size of BDD
(mcb_matching checker) $\wedge$ (M not a matching)	00:01:18.35	16,385	131,072
(mcb_matching checker) $\wedge$ (NC not a node cover)	00:01:23.81	16,385	131,072
(mcb_matching checker) $\wedge$ ($ M = NC $)	00:01:21.45	16,385	131,072
(original maximum_flow checker) $\wedge$ ($\neg$ flow_condition)	00:00:01.57	60	162,233
(corrected maximum_flow checker) $\wedge$ ($\neg$ flow_condition)	00:0:01.05	1	1
(corrected maximum_flow checker) $\wedge$ ($\neg$ excess_condition)	00:00:01.06	1	1