Software Model Checking and The BLAST Toolkit

Raivo Laanemets
rlaanemt@ut.ee

Programming Languages Research Seminar MTAT.03.204
Institute of Computer Science
University of Tartu

October 17, 2009

Abstract

Model checking is a common technique for verifying computer hardware but it can be used also for software verification. This report gives a gentle introduction to model checking and introduces the BLAST analyzer. BLAST stands for Berkeley Lazy Abstraction Software Verification Tool and uses model checking algorithm that is specialized for efficient and scalable software verification. It can be used for proving certain properties of computer programs written in C programming language.

1 Introduction

Software verification and program analysis in general is a very important field of research in computer science. A high number of lines of code are written every day. Not all programs will do what they are meant to do because they contain defective code. Bugs in the code can lead to high financial or even human life losses. Therefore it is important to reduce the number of code defects as much as possible or prove the absence of bugs and the code conformance to the specification. This is where software model checking techniques can help.

Model checking is especially suitable for checking program concurrency properties (e.g. presence of deadlocks) and for checking assertion validity, but also it can be used for detecting dead code, removing unnecessary null pointer checks, and for checking many other interesting properties.

This report gives a brief introduction into model checking. Only the basic concepts are described. A closer view is given to the one of the software model checking toolkits, known as BLAST. This is followed by a short overview of the related work.

2 Principles of Model Checking

Model checking is about testing whether the given model of hardware or software meets a given specification. For this task to be carried out algorithmically both the model and the specification have to be formulated in a precise mathematical language. There are three essential steps in the model checking process [1]:

Modeling. Firstly, the design of a system must be translated into a mathematical form. For hardware systems a suitable model could be given as a labeled finite state transition system (finite state machine). In the modeling step, some irrelevant details of the system might be removed.

Specification. Secondly, we need to specify the interesting properties of the system that we would like to check. A popular choice for specification language is some kind of logic formalism. It is common to use a variant of temporal logics. This will be discussed more in the report.

Verification. Finally, the model is checked for the specified properties. If a logic formalism was used for the specification then this step could mean that an algorithm or a decision procedure for logical inference is applied. When the verification fails the algorithm should provide insight (a counterexample, an error trace) why the model fails to satisfy the specification.

By no means these three steps have to be completed as separated tasks. Moreover, especially in software model checkers, these steps should be tightly integrated in order to guarantee maximum achievable performance [2].

2.1 Kripke structure

The structure suitable for model checking is called a Kripke structure. It is a four-tuple $M = (S, S_0, R, L)$ that consists of: a finite set of states S; a set of initial states S_0 ($S_0 \subset S$); a transition relation $R \subset S \times S$; and a labeling of states $L : S \to 2^{AP}$ where AP is the set of atomic propositions (atomic logical formulas) that are true in that state [1].

A Kripke structure describes labeled finite state automaton. A state of such automaton is labeled by the predicates which hold in the given state. For example, states of a microwave oven could contain the following predicates: *heat*, *closed* and *started*. The predicates represent the operation of the oven: *heat* - the oven is heated up or not; *closed* - whether the door of the oven is closed or not; *started* - the oven is turned on or off. A possible property which we would like to verify for the oven is that it is safe to use, i.e. it does not heat up until the door is closed [16]. A possible starting state would be $\{\neg closed, \neg started, \neg heat\}$.

2.2 Temporal Logics

Model checking with temporal logics is an automatic verification technique commonly employed for finite state concurrent systems [1]. It is mainly used for verification of

computer hardware which can be described as a finite state system. The first algorithms for temporal-logics model checking were developed at the start of 1980s.

Temporal logics differs from usual propositional logics by the additional logical operators and quantifiers that can be used for representing and reasoning about propositions in terms of time. Some examples (operators of CTL* logics) [1, 9]:

Temporal Operators:		Path Quantifiers:	
$X\,p$	"p holds next time"	A	"for every path"
$F\,p$	"p holds sometimes in the future"	E	"there exists a path"
$G\,p$	"p holds globally in the future"		
$p\,U\,q$	"p holds until q holds"		
$p\,R\,q$	"q is released when p becomes false"		

The semantics of the operators above can be defined in terms of a Kripke structure. A formal description is found in [1, 9].

There are various version of temporal logic systems. Most popular ones are Interval Temporal Logics (ITL), μ-calculus, Hennessy-Milner Logics (HML), Computational Tree Logic (CTL), Linear Temporal Logics (LTL). Some of them are more expressive, some are less. An overview is given in [1].

A simple and concise example how to apply CTL (which is CTL* with some minor restrictions) for a model checking problem can be found in [16]. The example is about the same microwave oven as described above.

2.3 Symbolic Model Checking

Explicit Kripke structure construction is not computationally feasible for larger state spaces. In symbolic model checking, the transition relation of the Kripke structure is not explicitly represented. The state space is constructed by boolean functions instead. There exists a very efficient way to manipulate with boolean functions in the form of BDD's *(Binary Decision Diagrams)*. This has allowed to push the tractable state count up to 10^{120} states [9]. With BDD-based approach, a temporal logics that can be implemented in terms of BDD operations is preferred. There is a suitable variant of CTL* (ACTL*) described in [9]. A compherensive guide into different types of BDD's and their operations can be found in [17].

2.4 State Space Abstraction

The algorithm or proof procedure for hardware verification can be stated as an exhaustive search of a given state space [1]. However, for larger state spaces this kind of algorithm might be computationally too inefficient [2]. Furthermore, it is completely useless for software verification since most of the software cannot be described as a finite state system. Some details have to be eliminated.

The abstraction of infinite state system can give us finite state abstract system [15, 19]. The dangers of abstraction are the loss of analysis precision and the possibility of false

positives/negatives. Therefore it is important to carefully choose the abstraction method. Most common technique is to use homomorphic transforms which partition the concrete state space into abstract state space where one abstract state describes one or more (possibly infinite) of concrete states.

2.5 Abstraction refinement

A very interesting method is not to use a fixed abstraction but to refine it during the verification procedure [9, 18]. It enables the possibility of exploring the state space only as much as is needed for proving the interesting property. An algorithm for verification with abstraction should obey the following characteristics: when the abstract program is proved to meet the given specification then the concrete program also meets the specification. Then the verification procedure can be terminated as soon as enough detailed abstraction is found to satisfy the specification.

On the other hand, the abstraction might be too coarse and the verification might fail although the program is correct. In this case a counterexample is provided by the model checking algorithm. The counterexample can be then used to refine the abstraction. This technique is called *counterexample-guided abstraction refinement* (CEGAR) [9].

2.6 Model Checking vs. Static Analysis

Model checking is usually not considered to be a static analysis method. Indeed, in the software case it can be more viewed as the (abstract) symbolic execution of the program. In general, static analysis is considered to be easier and to find shallow bugs while model checking is more expensive and gives better results. There is a comparasion between model checking and static analysis in [31]. Interestingly, the paper puts SLAM, a well-known model checker, in the category of static analysis tools.

3 The BLAST Toolkit

BLAST *(Berkeley Lazy Abstraction Software Verification Tool)* is a software model checker for C language programs [2, 11, 12]. It employs state space abstraction and uses extremely efficient abstraction refinement which tries iteratively refine only the parts of the model which are necessary for verification of the given property. Unlike other software model checkers BLAST is scalable and has been tested on programs with size up to 30000 lines of code. The BLAST project contains compherensive list of components: a scalable and efficient core system; a query language for specifications [13]; a graphical user interface [2]; an Eclipse plugin [14].

BLAST is written in OCaml and uses CIL library to parse and preprocess the input source code. CIL stands for C Intermediate Language and is also used in various other program analysis toolkits that are developed in OCaml [21].

The algorithm[1] contains steps of counterexample-guided-abstraction which are combined with lazy abstraction. The same basic algorithm layout (abstract-verify-refine) is also used in MAGIC and SLAM [2].

3.1 Program Representation

CFA. The program code is stored as a set of Control Flow Automata (CFA). There is one such automaton for each program function. A CFA is a directed graph whose nodes are program locations and edges correspond to operations between those locations. The function starting point (entry) is the root of the graph. Operations can be basic block*s*, assume predicates, function calls or return instructions.

Basic blocks are assignments in the form *lval=exp*, where *lval* is a variable, structure field or pointer dereference and *exp* is an arithmetic expression.

Assume predicates contain boolean expressions involving local variables. These represent true statements derived from the conditional expressions (e.g. from *if*-expression condition part).

Function calls are transformed to call-by-value form. All C language programs can be converted into this representation [2, 21]. Figure 1 shows an example CFA for a function that calculates the minimum values of two given arguments.

```
int min(int a, int b) {
L1:  int m = 0;
L2:  if (a < b) {
L3:     m = a;
     } else {
L4:     m = b;
     }
L5:  return m;
}
```

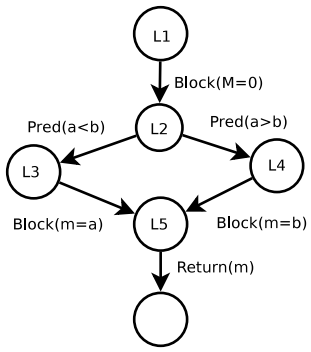

Figure 1: **Control Flow Automaton (CFA) for min function**

ART. An ART *(Abstract Reachability Tree)* is a labeled tree. It can be thought of as a program execution tree, except that variable values are not concrete but represented by predicates over them. Each node of the tree contains the following components: a CFA node (a program code location), current call stack and an overapproximation of the set of data states reachable from this node. The call stack of a node is a sequence of CFA nodes representing return addresses and the approximation of data states, i.e. possible variable values are described as predicates over those variables. The edges of the tree are the operations in CFAs of the program and each path in an ART corresponds to a program execution [2, 13, 27].

[1]Surprisingly, BLAST does not use temporal logics. The root ideas of the approach (turning the analysis into a graph search) are from [29].

```
int main() {
L6:  int a = readInt();
L7:  int b = readInt();
L8:  r = min(a, b);
L9:  if (r > a) {
L10:    ERROR;
     }
L11: return r;
}
```

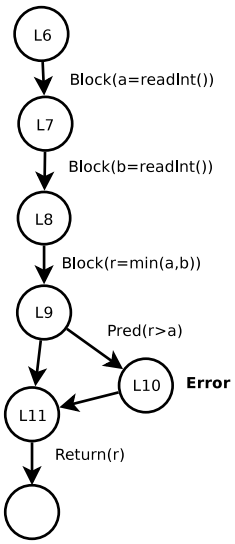

Figure 2: **Control Flow Automaton (CFA) for main function**

Branches in an ART are formed by *if*-expressions of the program. The reachable data states are constrained by assuming the *if*-expression condition to be true in one branch and false in the other. The ART tree is built and refined during the verification process.

Figure 3 shows complete tree for the program in figure 2 . The program reads two integer numbers using the function *readInt*. The details and return values of these functions are completely irrelevant for this example. This is consistent with BLAST which assumes that libabry functions can return any value (if not specified otherwise). The program calculates the minimum of these numbers using the *min* function. Although this example is simple, we would still like to check the return value of *min* for sanity (it is not greater than the first argument). The complete verfication tree contains predicates for each node in the bold fontface. It can be seen that *ERROR* label (*L10*) is not reachable. Instead of *if*-statement we could also have used *assert* (see *Assertion Checking* below). Also, the specification (in the form of *if*-statement) could have checked that the return values of min is not greater than both of the arguments. The tree would be almost same.

3.2 Abstract-Check-Refine Loop

The central part of the BLAST algorithm is a loop that implements CEGAR principles. The steps are outlined here in a quite informal way. The actual implementation is much more complex and out of scope of this report. Furthermore, it is not at all clear which techniques are implemented in the tool, and which are not and how all the implemented methods play together. Articles [2, 13, 14] and the other articles available on the homepage [12] currently focus on general concepts or specific use cases. The manual [11] lists possible command line options but the description of those options is not informative enough. Also, the source code obtained from the homepage contains a minimal amount of comments.

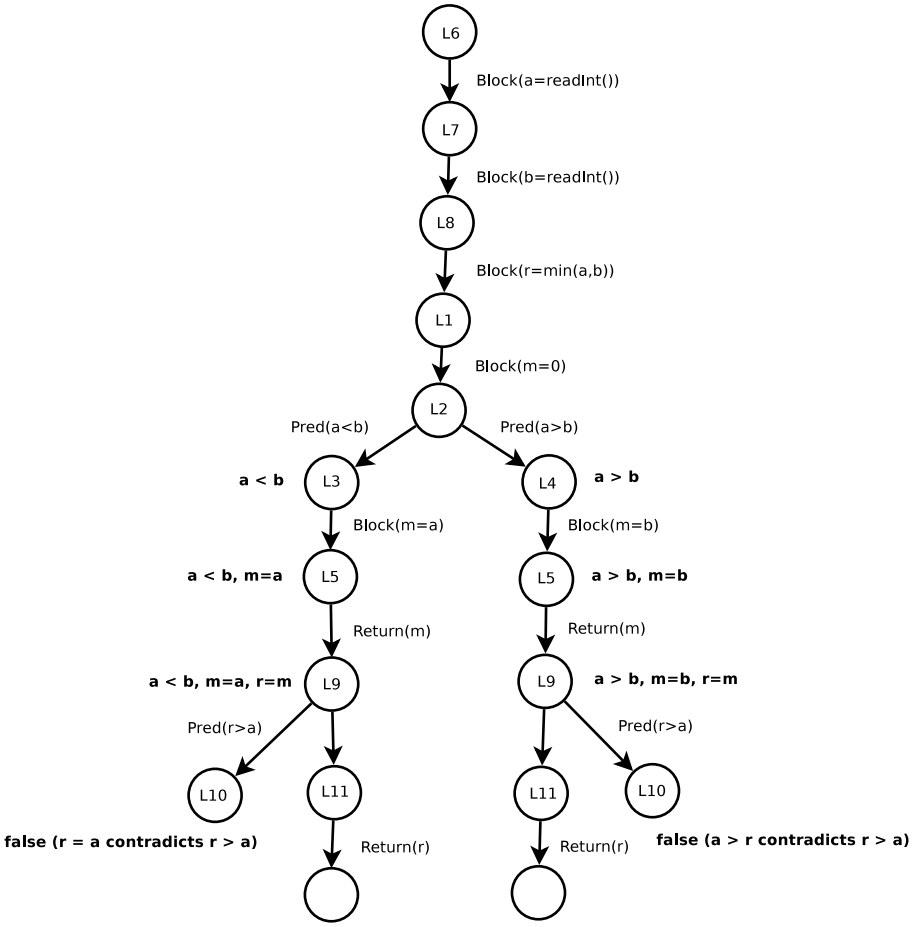

Figure 3: **Complete Abstract Reachability Tree**

1. **Abstraction.** In the first steps of the algorithm a set of predicates from CFAs is chosen to abstract the concrete program. This is done by running a depth-first search (although breath-first search can be used, too [11]). As a result, the push-down automaton in the form of ART is built. This constitutes the abstract model of the program [2, 22, 28]. The ART building process is also called a *forward search* phase.

2. **Verification.** During the forward search, it can be automatically checked whether some specified error label is visited. When no error labels are visited during the search, it can be concluded that the given property is verified. See the next section of the report how to express analysis problems as reachability. The algorithm will end with the result "The system is safe".
   When the error label is visited, it might be that a program error is found or the abstraction was not precise enough. The path in the ART that leads to the error label is checked with a theorem prover that such execution trace is indeed possible.

Early versions of BLAST used Simplify [30] theorem prover [12]. In the case of a true error the loop will terminate with the result "The system is unsafe".

3. **Refinement**. When the theorem prover fails to prove the ART path to error being a possible execution path, the abstraction of the program is refined *(backward search)*. For greater efficiency only necessary parts of the model are considered. This concept is called a *lazy abstraction* [28]. As a result new predicates are added into the set of predicates that are used in step 1. The process of adding new predicates is called an *interpolation-based predicate discovery* [18].

The presented algorithm is correct which means there are no false positives, i.e. the algorithm terminates with "The system is safe" but the program contains an error in the context of the checked property. On the other hand, it might fail to prove some properties because in a some case it might require solving a halting problem [2].

4 Program Analysis with BLAST

The BLAST toolkit can be used for many different analyses. The most basic one is reachability checking. All other analyses can be reduced to it [2, 11, 27].

4.1 Reachability Checking

Reachability checking verifies whether the special label `ERROR` is reachable in the source code. A rather primitive example code to demonstrate it is in the figure 4.

```
int main() {
  int x,y;
  if (x > y) {
    x = x - y;
    if (x <= 0) {
      ERROR: goto ERROR;
    }
  }
}
```

Figure 4: **Reachability Checking Example**

The example can be checked using the command *path/to/pblast.opt example.c*. The output is "No error found. The system is safe :-)". Notice that knowing the values of x and y is not necessary to prove that the *ERROR* label is never reached. When the condition in the second *if*-expression is replaced with $x >= 0$, the tool will output "Error found! The system is unsafe :-(".

4.2 Assertion Checking

Reachability analysis can be used for checking assertions [26] statically in the source code. Assertions are usually checked at runtime during testing, however, they can be verified with BLAST by using *assert.h* header that comes with the toolkit. The definition of the assert function is shown in the figure 5.

```
void assert(int b) {
  if (!b) {
    ERROR: goto ERROR;
  }
}
```

Figure 5: **Assertion Checking as Reachability**

4.3 Correct Locking

For concurrent programs it is important to have shared resources protected from non-synchronized access. That is, when the thread will access a shared resource it must first acquire a lock. After finishing with the resources the thread must release the lock. Thus locking and unlocking actions should be done in an alternating sequence.

Locking problem can be also expressed using program reachability by replacing locking and unlocking with the following function definitions [22] in the figure 6.

```
lock() {
  if (LOCK == 0) {
    LOCK = 1;
  } else {
    ERROR: goto ERROR;
  }
}
```

```
unlock() {
  if (LOCK == 1) {
    LOCK = 0;
  } else {
    ERROR: goto ERROR;
  }
}
```

Figure 6: **Locking as Reachability**

Here *LOCK* is a global variable for tracking the state of the lock. For example, if the lock is already held the value of *LOCK* is 1 and trying to acquire lock second time without unlocking first will lead to the *ERROR* label. To check correct locking it is therefore sufficient to check whether those *ERROR* labels are not reachable. The program can be rewritten to use these definitions manually or using the BLAST query language [13]. There is a practical study of applying the toolkit to the Linux kernel source in [24].

5 Issues with BLAST

Although [2, 11, 13, 14] are touting the BLAST project as a huge success in software model checking, there have been some reported difficulties in applying it to practical

problems.

Report [22] applies BLAST to a simple linked-list library and concludes that its pointer analysis capabilities are still not sophisticated enough. Pointer analysis (alias analysis) is a rather expensive component of static analysis [25]. The BLAST version was 1.0 and it is likely that things have improved this far. The current version of BLAST is 2.5.

Another study [24] checks memory safety and correct locking behavior of Linux device drivers with BLAST 2.0. Similarly to the previous study, pointer analysis capabilities are lacking for proving safety in many cases. The biggest problem seems to be with function pointers.

Both articles reported problems with usability and documentation. The analysis process failed many times with cryptic uninformative error messages. For the second study this required manual rewriting and simplification of the input source code until the tool accepted it.

The second study also suggested that BLAST should accept the same arguments as the compiler and should automate the required preprocessing steps of the source code as this would make it easier for a developer to apply the tool.

6 Related work

A number of software model checkers other than BLAST have been developed in the last decade. Some of them will be described in brief details. Although the list of software described here is more-less random and certainly does not represent all software model checkers available, it illustrates the usage of the ideas introduced above.

6.1 Groove

Groove is a model checker for object-oriented systems (Java). It is uses similar methods as described above and uses CT logics directly. It does not employ state space abstraction. The state space transition system is represented as a call graph and explicit graph transformation rules on it. The representation is later turned into a Kripke structure and the standard CTL algorithm is applied. Groove does not scale well to larger programs [3]. Abstraction (with respect to graph transitions) can be still implemented as described in [4].

6.2 SPIN

SPIN *(Simple Promela Interpreter)* is a general verification system that is specialized for the design and verification of asynchronous process systems. It accepts system specifications written in PROMELA *(Process Meta Language)* verification language. SPIN uses LT logics (by translating it to Büchi automaton [1, p. 122], then applying optimized depth-first graph search) and is most suitable for verification of highly concurrent software like distributed operating systems. SPIN can be easily extended to construct more

advanced/specific tools [7]. Instead of state space abstraction it uses a partial order reduction method to reduce the number of states that must be explored to complete a verification. Partial order reduction is an alternative method to speed up the model checking process [8].

6.3 PathFinder

PathFinder is a model checker for Java programs. It translates Java program into a PROMELA model which is input to the previously described SPIN model checker. The state space of the generated PROMELA model corresponds directly to the state space of the translated Java program. The model is checked for deadlocks and assertion violations. PathFinder is developed by NASA. Previously in NASA, the SPIN system had been successfully used in verification of multithreaded operating system of Deep Space 1 spacecraft (written in Common Lisp instead of Java). However, PathFinder scalability is limited by the SPIN model checker and it has been only applied to programs with up to 2000 lines [10].

6.4 MAGIC

The MAGIC (Modular Analysis of Programs in C) approaches verification by decomposing large programs into modular parts that reflect the modularity in the software design. The verification procedure is defined as weak simulation between the specification and model, i.e. the specification is defined using state charts and the model is a finite state machine derived from the code using predicate extraction in combination with abstraction [20]. The predicate abstraction in MAGIC is similar as described in [9, 19]. The checking of simulation relation between the specification and the model is reduced to boolean satisfiability and thus highly efficient SAT solvers can be used.

6.5 SLAM

SLAM *(Software, Languages, Analysis, and Modeling)* is an analysis engine from Microsoft Research. The main goal of the project was to create a tool for automatic checking that a C program correctly uses the interface to an external library. SLAM was the first model checker to implement counterexample-guided predicate abstraction refinement (an implementation of more general CEGAR principle) [5, 6]. The ideas used in the SLAM project were later largely improved by the BLAST tool [2]. SLAM is internally used in the development of device drivers in Microsoft and is available as a part of Windows Driver Development Kit for third-party driver developers [6].

More detailed overview, including the direct comparison between SLAM and BLAST can be found in [23].

7 Conclusions

Although model checking as a verification method is commonly used for computer hardware it can be used for software verification as well. The techniques of model checking are cleverly employed in the BLAST program analyzer which is a scalable and efficient model checker for C language programs. The described analyzer can be used for proving various interesting properties of computer programs. Current problems with the toolkit are the lack of user documentation and shortcomings in pointer analysis and general usability.

References

[1] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. Model Checking. The MIT Press 2000

[2] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, Rupak Majundar. The Software Model Checker BLAST: Applications to Software Engineering. Int. Journal on Software Tools for Technology Transfer 2007

[3] Harmen Kastenberg, Arend Rensink. CTL Model Checking in Groove. Department of Computer Science, University of Twente.

[4] Arend Rensink, Dino Distefano. Abstract Graph Transformation. In International Workshop on Software Verification and Validation 2005

[5] Thomas Ball, Byron Cook, Vladimir Levin, Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. Integrated Formal Methods 2004

[6] The homepage of SLAM. *http://research.microsoft.com/en-us/projects/slam/*. Microsoft Research 2009

[7] Gerard J. Holzmann. The Model Checker SPIN. IEEE Transactions on Software Engineering, 23(5):279–295, May 1997

[8] Gerard J. Holzmann, Doron Peled, An Improvment in Formal Verification, Proceedings in Seventh FORTE Conference on Formal Description Techniques, p. 177-194, Bern, Switzerland, October 1994.

[9] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith, Counterexample-guided abstraction refinement. In Proc CAV., LNCS 1855, p. 154-169, Springer 2000.

[10] Klaus Havelund, Thomas Pressburger, Model Checking Java Programs Using Java PathFinder, International Journal on Software Tools for Technology Transfer, Vol. 2, No. 4. (2000), pp. 366-381.

[11] The BLAST team, BLAST User Manual, From the homepage of BLAST: *http://mtc.epfl.ch/software-tools/blast/*.

[12] The homepage of BLAST, http://mtc.epfl.ch/software-tools/blast/, The BLAST team 2009.

[13] Dirk Beyer, Adam J. Chlipala, Thomas Henzinger, Ranjit Jhala, Rupak Majumdar. The Blast Query Language for Software Verification. Proceedings of 11th Static Analysis Symposium (SAS'04), Lecture Notes in Computer Science 3148, Springer-Verlag. August 2004

[14] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar, An Eclipse plug-in for model checking, Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC 2004), pages 251-255, IEEE Computer Society Press, 2004.

[15] Randal E. Bryant, Sriram K. Rajamani, Verifying Properties of Hardware and Software by Predicate Abstraction and Model Checking, Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design, p. 437 - 438, IEEE Computer Society 2004.

[16] Yuguo Sun, Modelchecking: Eine Einführung, Seminar Sicherheitskritische Systeme, *http://pi.informatik.uni-siegen.de/niere/lehre/SS04/SeminarFinal/6_sun/Ausarbeitung.pdf*

[17] Henrik Reif Andersen. An Introduction to Binary Decision Diagrams (lecture notes). IT University of Copenhagen, 1999. *http://www.itu.dk/~hra/notes-index.html*

[18] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar and Kenneth L. McMillan. Abstractions from Proofs. In ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages, 2004.

[19] Susanne Graf, Hassen Saïdi. Construction of Abstract State Graphs with PVS. Proceedings of the 9th International Conference on Computer Aided Verification p. 72-83, 1997

[20] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, Helmut Veith. Modular Verification of Software Components in C. An ACM-SIGSOFT Distinguished Paper in the 25th International Conference on Software Engineering (ICSE) 2003, pages 385-395, 2003

[21] George C. Necula, Scott McPeak, S.P. Rahul and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In Proceedings of Conference on Compilier Construction, 2002.

[22] Wai Sum Mong. Lazy Abstraction on Software Model Checking. Project report, *http://www.cs.toronto.edu/~arie/csc2108conf/mong.pdf*

[23] Elisabeth A. Strunk, M. Anthony Aiello, John C. Knight, Eds. A Survey of Tools for Model Checking and Model-Based Development. Technical Report CS-2006-17, Department of Computer Science, University of Virginia, June 2006

[24] Jan T. Mühlberg, Gerald Lüttgen. BLASTing Linux Code. Formal Methods: Applications and Technology, pages 211-226, 2007.

[25] Derek Rayside. Points-To Analysis (lecture notes). Massachusetts Institute of Technology, 2005, *http://groups.csail.mit.edu/pag/6.883/lectures/points-to.pdf*

[26] The Single UNIX® Specification. assert function. *http://www.opengroup.org/onlinepubs/9699919799/functions/assert.html.* The IEEE and The Open Group, 2008

[27] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Checking memory safety with BLAST. Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005), LNCS 3442, pages 2-18, Springer-Verlag, 2005.

[28] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar and Gregoire Sutre. Lazy Abstraction. In ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages, pages 58-70, 2002

[29] Thomas Reps, Susan Horwitz, Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 49-61, 1995.

[30] David Detlefs, Greg Nelson, James B. Saxe. Simplify: a theorem prover for program checking. Journal of the ACM, Volume 52 , Issue 3, pages 365 - 473, 2005

[31] Dawson Engler, Madanlal Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In VMCAI, pages 191-210, 2004